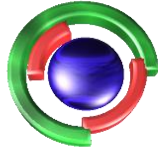
FUTURE VISION BIE

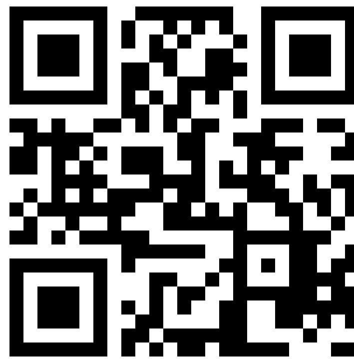
One Stop for All Study Materials

& Lab Programs



By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

Table of Contents

1. Computer Graphics Hardware
Donald D. Hearn/M. Pauline Baker, Warren Carithers 1

Computer Graphics Hardware Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers 27

2. Computer Graphics Software
Donald D. Hearn/M. Pauline Baker, Warren Carithers 29

3. Graphics Output Primitives
Donald D. Hearn/M. Pauline Baker, Warren Carithers 45

4. Attributes of Graphics Primitives
Donald D. Hearn/M. Pauline Baker, Warren Carithers 99

5. Implementation Algorithms for Graphics Primitives and Attributes
Donald D. Hearn/M. Pauline Baker, Warren Carithers 131

6. Two-Dimensional Geometric Transformations
Donald D. Hearn/M. Pauline Baker, Warren Carithers 189

7. Two-Dimensional Viewing
Donald D. Hearn/M. Pauline Baker, Warren Carithers 227

8. Three-Dimensional Geometric Transformations
Donald D. Hearn/M. Pauline Baker, Warren Carithers 273

9. Three-Dimensional Viewing
Donald D. Hearn/M. Pauline Baker, Warren Carithers 301

Three-Dimensional Viewing Color Plate
Donald D. Hearn/M. Pauline Baker, Warren Carithers 353

10. Hierarchical Modeling
Donald D. Hearn/M. Pauline Baker, Warren Carithers 355

11. Computer Animation
Donald D. Hearn/M. Pauline Baker, Warren Carithers 365

https://hemanthrajhemu.github.io

12. Three-Dimensional Object Representations
Donald D. Hearn/M. Pauline Baker, Warren Carithers **389**

Three-Dimensional Object Representations Color Plate
Donald D. Hearn/M. Pauline Baker, Warren Carithers **407**

13. Spline Representations
Donald D. Hearn/M. Pauline Baker, Warren Carithers **409**

14. Visible-Surface Detection Methods
Donald D. Hearn/M. Pauline Baker, Warren Carithers **465**

15. Illumination Models and Surface-Rendering Methods
Donald D. Hearn/M. Pauline Baker, Warren Carithers **493**

Illumination Models and Surface-Rendering Methods Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **541**

16. Texturing and Surface-Detail Methods
Donald D. Hearn/M. Pauline Baker, Warren Carithers **543**

Texturing and Surface-Detail Methods Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **567**

17. Color Models and Color Applications
Donald D. Hearn/M. Pauline Baker, Warren Carithers **569**

Color Models and Color Applications Color Plate
Donald D. Hearn/M. Pauline Baker, Warren Carithers **589**

18. Interactive Input Methods and Graphical User Interfaces
Donald D. Hearn/M. Pauline Baker, Warren Carithers **591**

Interactive Input Methods and Graphical User Interfaces Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **631**

19. Global Illumination
Donald D. Hearn/M. Pauline Baker, Warren Carithers **633**

Global Illumination Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **659**

20. Programmable Shaders
Donald D. Hearn/M. Pauline Baker, Warren Carithers **663**

Programmable Shaders Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **693**

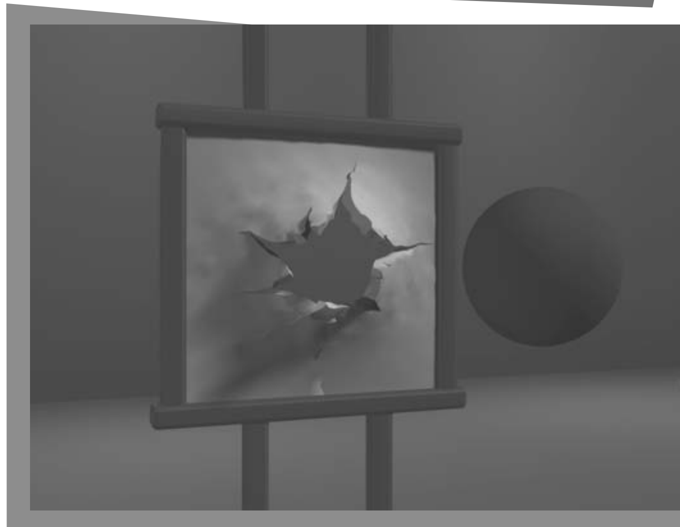
21. Algorithmic Modeling
Donald D. Hearn/M. Pauline Baker, Warren Carithers **695**

Algorithmic Modeling Color Plates
Donald D. Hearn/M. Pauline Baker, Warren Carithers **725**

https://hemanthrajhemu.github.io

Two-Dimensional Viewing

1 The Two-Dimensional Viewing Pipeline
2 The Clipping Window
3 Normalization and Viewport Transformations
4 OpenGL Two-Dimensional Viewing Functions
5 Clipping Algorithms
6 Two-Dimensional Point Clipping
7 Two-Dimensional Line Clipping
8 Polygon Fill-Area Clipping
9 Curve Clipping
10 Text Clipping
11 Summary



We now examine in more detail the procedures for displaying views of a two-dimensional picture on an output device. Typically, a graphics package allows a user to specify which part of a defined picture is to be displayed and where that part is to be placed on the display device. Any convenient Cartesian coordinate system, referred to as the world-coordinate reference frame, can be used to define the picture. For a two-dimensional picture, a view is selected by specifying a region of the xy plane that contains the total picture or any part of it. A user can select a single area for display, or several areas could be selected for simultaneous display or for an animated panning sequence across a scene. The picture parts within the selected areas are then mapped onto specified areas of the device coordinates. When multiple view areas are selected, these areas can be placed in separate display locations, or some areas could be inserted into other, larger display areas.

From Chapter 8 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.
Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

https://hemanthrajhemu.github.io

Two-dimensional viewing transformations from world to device coordinates involve translation, rotation, and scaling operations, as well as procedures for deleting those parts of the picture that are outside the limits of a selected scene area.

1 The Two-Dimensional Viewing Pipeline

A section of a two-dimensional scene that is selected for display is called a **clipping window** because all parts of the scene outside the selected section are "clipped" off. The only part of the scene that shows up on the screen is what is inside the clipping window. Sometimes the clipping window is alluded to as the *world window* or the *viewing window.* And, at one time, graphics systems referred to the clipping window simply as "the window," but there are now so many windows in use on computers that we need to distinguish between them. For example, a window-management system can create and manipulate several areas on a video screen, each of which is called "a window," for the display of graphics and text. So we will always use the term *clipping window* to refer to a selected section of a scene that is eventually converted to pixel patterns within a display window on the video monitor. Graphics packages allow us also to control the placement within the display window using another "window" called the **viewport.** Objects inside the clipping window are mapped to the viewport, and it is the viewport that is then positioned within the display window. The clipping window selects *what* we want to see; the viewport indicates *where* it is to be viewed on the output device.

By changing the position of a viewport, we can view objects at different positions on the display area of an output device. Multiple viewports can be used to display different sections of a scene at different screen positions. Also, by varying the size of viewports, we can change the size and proportions of displayed objects. We achieve zooming effects by successively mapping different-sized clipping windows onto a fixed-size viewport. As the clipping windows are made smaller, we zoom in on some part of a scene to view details that are not shown with the larger clipping windows. Similarly, more overview is obtained by zooming out from a section of a scene with successively larger clipping windows. And panning effects are achieved by moving a fixed-size clipping window across the various objects in a scene.

Usually, clipping windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes. Other window or viewport geometries, such as general polygon shapes and circles, are used in some applications, but these shapes take longer to process. We first consider only rectangular viewports and clipping windows, as illustrated in Figure 1.

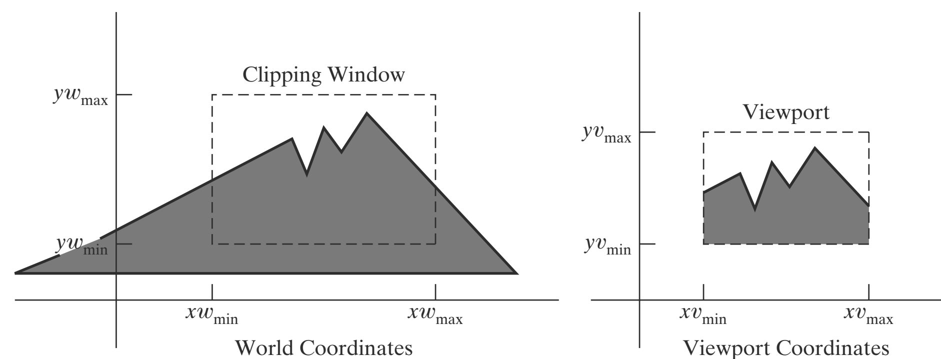


FIGURE 1
A clipping window and associated viewport, specified as rectangles aligned with the coordinate axes.

https://hemanthrajhemu.github.io

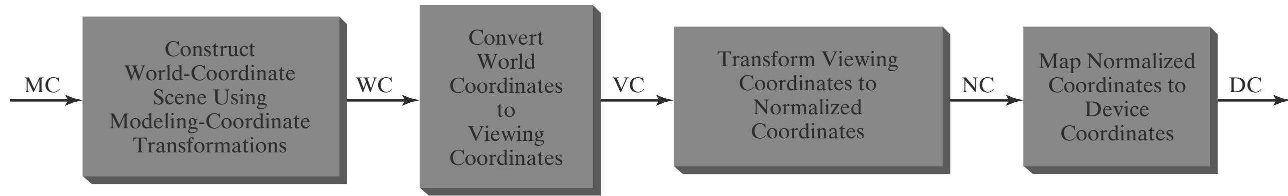


FIGURE 2
Two-dimensional viewing-transformation pipeline.

The mapping of a two-dimensional, world-coordinate scene description to device coordinates is called a **two-dimensional viewing transformation.** Sometimes this transformation is simply referred to as the *window-to-viewport transformation* or the *windowing transformation.* But, in general, viewing involves more than just the transformation from clipping-window coordinates to viewport coordinates. In analogy with three-dimensional viewing, we can describe the steps for two-dimensional viewing as indicated in Figure 2. Once a world-coordinate scene has been constructed, we could set up a separate two-dimensional, **viewing-coordinate reference frame** for specifying the clipping window. But the clipping window is often just defined in world coordinates, so viewing coordinates for two-dimensional applications are the same as world coordinates. (For a three-dimensional scene, however, we need a separate viewing frame to specify the parameters for the viewing position, direction, and orientation.)

To make the viewing process independent of the requirements of any output device, graphics systems convert object descriptions to normalized coordinates and apply the clipping routines. Some systems use normalized coordinates in the range from 0 to 1, and others use a normalized range from −1 to 1. Depending upon the graphics library in use, the viewport is defined either in normalized coordinates or in screen coordinates after the normalization process. At the final step of the viewing transformation, the contents of the viewport are transferred to positions within the display window.

Clipping is usually performed in normalized coordinates. This allows us to reduce computations by first concatenating the various transformation matrices. Clipping procedures are of fundamental importance in computer graphics. They are used not only in viewing transformations, but also in window-manager systems, in painting and drawing packages to erase picture sections, and in many other applications.

2 The Clipping Window

To achieve a particular viewing effect in an application program, we could design our own clipping window with any shape, size, and orientation we choose. For example, we might like to use a star pattern, an ellipse, or a figure with spline boundaries as a clipping window. But clipping a scene using a concave polygon or a clipping window with nonlinear boundaries requires more processing than clipping against a rectangle. We need to perform more computations to determine where an object intersects a circle than to find out where it intersects a straight line. The simplest window edges to clip against are straight lines that are parallel to the coordinate axes. Therefore, graphics packages commonly allow only rectangular clipping windows aligned with the x and y axes.

If we want some other shape for a clipping window, then we must implement our own clipping and coordinate-transformation algorithms, or we could just edit

https://hemanthrajhemu.github.io

the picture to produce a certain shape for the display frame around the scene. For example, we could trim the edges of a picture with any desired pattern by overlaying polygons that are filled with the background color. In this way, we could generate any desired border effects or even put interior holes in the picture.

Rectangular clipping windows in standard position are easily defined by giving the coordinates of two opposite corners of each rectangle. If we would like to get a rotated view of a scene, we could either define a rectangular clipping window in a rotated viewing-coordinate frame or, equivalently, we could rotate the world-coordinate scene. Some systems provide options for selecting a rotated, two-dimensional viewing frame, but usually the clipping window must be specified in world coordinates.

Viewing-Coordinate Clipping Window

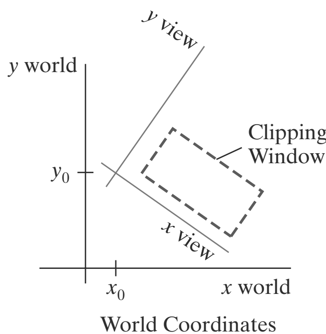


FIGURE 3
A rotated clipping window defined in viewing coordinates.

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 in Figure 3. To obtain a view of the world-coordinate scene as determined by the clipping window of Figure 3, we just need to transfer the scene description to viewing coordinates. Although many graphics packages do not provide functions for specifying a clipping window in a two-dimensional viewing-coordinate system, this is the standard approach for defining a clipping region for a three-dimensional scene.

We choose an origin for a two-dimensional viewing-coordinate frame at some world position $\mathbf{P}_0 = (x_0, y_0)$, and we can establish the orientation using a world vector $\mathbf{V}$ that defines the y_{view} direction. Vector $\mathbf{V}$ is called the two-dimensional **view up vector.** An alternative method for specifying the orientation of the viewing frame is to give a rotation angle relative to either the x or y axis in the world frame. From this rotation angle, we can then obtain the view up vector. Once we have established the parameters that define the viewing-coordinate frame, we transform the scene description to the viewing system. This involves a sequence of transformations equivalent to superimposing the viewing frame on the world frame.

The first step in the transformation sequence is to translate the viewing origin to the world origin. Next, we rotate the viewing system to align it with the world frame. Given the orientation vector $\mathbf{V}$, we can calculate the components of unit vectors $\mathbf{v} = (v_x, v_y)$ and $\mathbf{u} = (u_x, u_y)$ for the y_{view} and x_{view} axes, respectively. These unit vectors are used to form the first and second rows of the rotation matrix $\mathbf{R}$ that aligns the viewing $x_{\text{view}}y_{\text{view}}$ axes with the world $x_w y_w$ axes.

Object positions in world coordinates are then converted to viewing coordinates with the composite two-dimensional transformation matrix

$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T} \tag{1}$$

where $\mathbf{T}$ is the translation matrix that takes the viewing origin $\mathbf{P}_0$ to the world origin, and $\mathbf{R}$ is the rotation matrix that rotates the viewing frame of reference into coincidence with the world-coordinate system. Figure 4 illustrates the steps in this coordinate transformation.

World-Coordinate Clipping Window

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

https://hemanthrajhemu.github.io

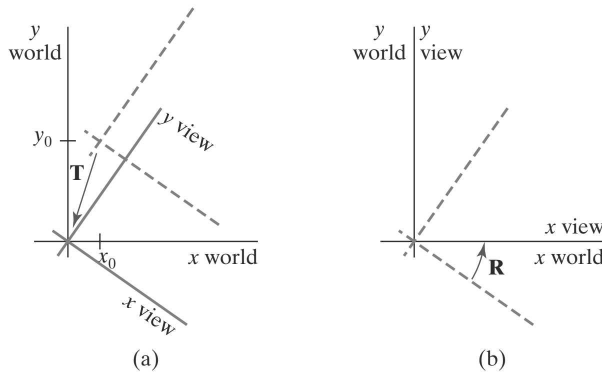


FIGURE 4
A viewing-coordinate frame is moved into coincidence with the world frame by (a) applying a translation matrix **T** to move the viewing origin to the world origin, then (b) applying a rotation matrix **R** to align the axes of the two systems.

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.

If we want to obtain a rotated view of a two-dimensional scene, as discussed in the previous section, we perform exactly the same steps as described there, but without considering a viewing frame of reference. Thus, we simply rotate (and possibly translate) objects to a desired position and set up the clipping window—all in world coordinates. For example, we could display a rotated view of the triangle in Figure 5(a) by rotating it into the position we want and setting up a standard clipping rectangle. In analogy with the coordinate transformation described in the previous section, we could also translate the triangle to the world origin and define a clipping window around the triangle. In that case, we define an orientation vector and choose a reference point such as the triangle's centroid. Then we translate the reference point to the world origin and rotate the orientation vector onto the yworld axis using transformation matrix 1. With the triangle in the desired orientation, we can use a standard clipping window in world coordinates to capture the view of the rotated triangle. The transformed position of the triangle and the selected clipping window are shown in Figure 5(b).

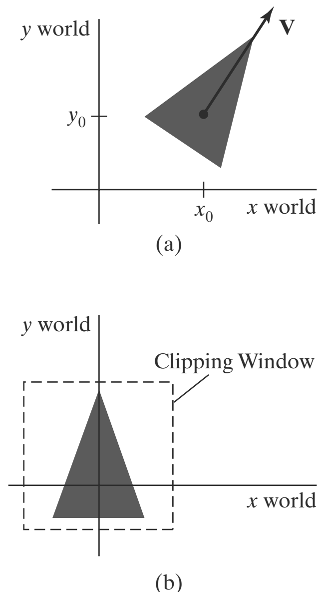


FIGURE 5
A triangle (a), with a selected reference point and orientation vector, is translated and rotated to position (b) within a clipping window.

3 Normalization and Viewport Transformations

With some graphics packages, the normalization and window-to-viewport transformations are combined into one operation. In this case, the viewport coordinates are often given in the range from 0 to 1 so that the viewport is positioned within a unit square. After clipping, the unit square containing the viewport is mapped to the output display device. In other systems, the normalization and clipping routines are applied before the viewport transformation. For these systems, the viewport boundaries are specified in screen coordinates relative to the display-window position.

Mapping the Clipping Window into a Normalized Viewport

To illustrate the general procedures for the normalization and viewport transformations, we first consider a viewport defined with normalized coordinate values between 0 and 1. Object descriptions are transferred to this normalized space using a transformation that maintains the same relative placement of a point in

https://hemanthrajhemu.github.io

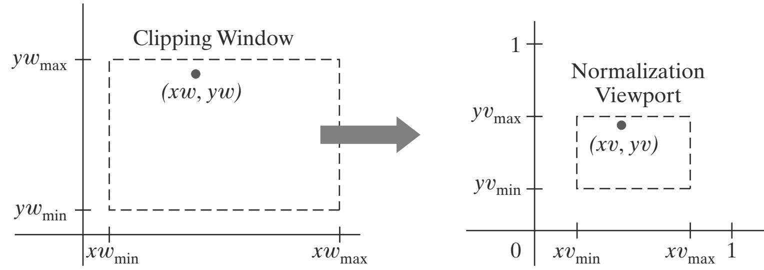


FIGURE 6
A point (xw, yw) in a world-coordinate clipping window is mapped to viewport coordinates (xv, yv), within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

the viewport as it had in the clipping window. If a coordinate position is at the center of the clipping window, for instance, it would be mapped to the center of the viewport. Figure 6 illustrates this window-to-viewport mapping. Position (xw, yw) in the clipping window is mapped to position (xv, yv) in the associated viewport.

To transform the world-coordinate point into the same relative position within the viewport, we require that

$$\frac{xv - xv_{min}}{xv_{max} - xv_{min}} = \frac{xw - xw_{min}}{xw_{max} - xw_{min}}$$
$$\frac{yv - yv_{min}}{yv_{max} - yv_{min}} = \frac{yw - yw_{min}}{yw_{max} - yw_{min}} \tag{2}$$

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

$$xv = s_x xw + t_x$$
$$yv = s_y yw + t_y \tag{3}$$

where the scaling factors are

$$s_x = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$$
$$s_y = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}} \tag{4}$$

and the translation factors are

$$t_x = \frac{xw_{max}xv_{min} - xw_{min}xv_{max}}{xw_{max} - xw_{min}}$$
$$t_y = \frac{yw_{max}yv_{min} - yw_{min}yv_{max}}{yw_{max} - yw_{min}} \tag{5}$$

Because we are simply mapping world-coordinate positions into a viewport that is positioned near the world origin, we can also derive Equations 3 using 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 following sequence:

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

https://hemanthrajhemu.github.io

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

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & xw_{\min}(1 - s_x) \\ 0 & s_y & yw_{\min}(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

where s_x and s_y are the same as in Equations 4. The two-dimensional matrix representation for the translation of the lower-left corner of the clipping window to the lower-left viewport corner is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & xv_{\min} - xw_{\min} \\ 0 & 1 & yv_{\min} - yw_{\min} \\ 0 & 0 & 1 \end{bmatrix} \tag{7}$$

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

$$\mathbf{M}_{\text{window, normviewp}} = \mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{8}$$

which gives us the same result as in Equations 3. Any other clipping-window reference point, such as the top-right corner or the window center, could be used for the scale–translate operations. Alternatively, we could first translate any clipping-window position to the corresponding location in the viewport, and then scale relative to that viewport location.

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, on the other hand, are maintained only if the aspect ratio of the viewport is the same as the aspect ratio of the clipping window. In other words, we keep the same object proportions if the scaling factors sx and sy are the same. Otherwise, world objects will be stretched or contracted in either the x or y directions (or both) when displayed on the output device.

The clipping routines can be applied using either the clipping-window boundaries or the viewport boundaries. After clipping, the normalized coordinates are transformed into device coordinates. And the unit square can be mapped onto the output device using the same procedures as in the window-to-viewport transformation, with the area inside the unit square transferred to the total display area of the output device.

Mapping the Clipping Window into a Normalized Square

Another approach to two-dimensional viewing is to transform the clipping window into a normalized square, clip in normalized coordinates, and then transfer the scene description to a viewport specified in screen coordinates. This transformation is illustrated in Figure 7 with normalized coordinates in the range from -1 to 1. The clipping algorithms in this transformation sequence are now standardized so that objects outside the boundaries $x = \pm 1$ and $y = \pm 1$ are detected and removed from the scene description. At the final step of the viewing transformation, the objects in the viewport are positioned within the display window.

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 obtained from Equation 8 by substituting -1 for $xv_{\min}$ and $yv_{\min}$ and substituting +1 for $xv_{\max}$ and $yv_{\max}$.

https://hemanthrajhemu.github.io

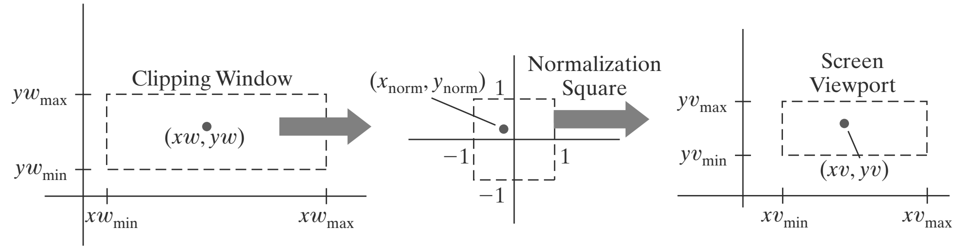


FIGURE 7
A point (xw, yw) in a clipping window is mapped to a normalized coordinate position (x_{norm}, y_{norm}), then to a screen-coordinate position (xv, yv) in a viewport. Objects are clipped against the normalization square before the transformation to viewport coordinates occurs.

Making these substitutions in the expressions for t_x, t_y, s_x, and s_y, we have

$$\mathbf{M}_{\text{window, normsquare}} = \begin{bmatrix} \dfrac{2}{xw_{max} - xw_{min}} & 0 & -\dfrac{xw_{max} + xw_{min}}{xw_{max} - xw_{min}} \\ 0 & \dfrac{2}{yw_{max} - yw_{min}} & -\dfrac{yw_{max} + yw_{min}}{yw_{max} - yw_{min}} \\ 0 & 0 & 1 \end{bmatrix} \tag{9}$$

Similarly, after the clipping algorithms have been applied, the normalized square with edge length equal to 2 is transformed into a specified viewport. This time, we get the transformation matrix from Equation 8 by substituting −1 for xw_{min} and yw_{min} and substituting +1 for xw_{max} and yw_{max}:

$$\mathbf{M}_{\text{normsquare, viewport}} = \begin{bmatrix} \dfrac{xv_{max} - xv_{min}}{2} & 0 & \dfrac{xv_{max} + xv_{min}}{2} \\ 0 & \dfrac{yv_{max} - yv_{min}}{2} & \dfrac{yv_{max} + yv_{min}}{2} \\ 0 & 0 & 1 \end{bmatrix} \tag{10}$$

The last step in the viewing process is to position the viewport area in the display window. Typically, the lower-left corner of the viewport is placed at a coordinate position specified relative to the lower-left corner of the display window. Figure 8 demonstrates the positioning of a viewport within a display window.

As before, we maintain the initial proportions of objects by choosing the aspect ratio of the viewport to be the same as the clipping window. Otherwise, objects

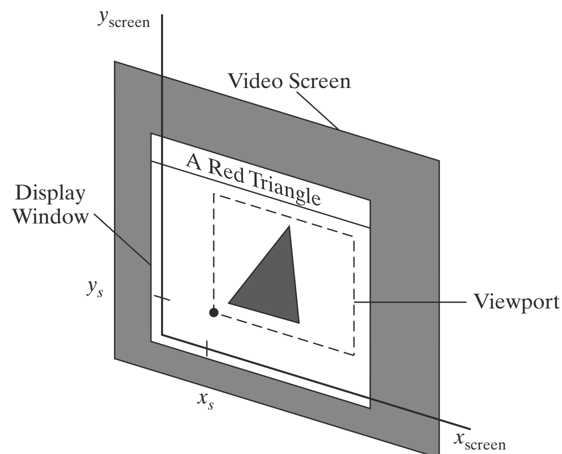


FIGURE 8
A viewport at coordinate position (x_s, y_s) within a display window.

https://hemanthrajhemu.github.io

will be stretched or contracted in the x or y directions. Also, the aspect ratio of the display window can affect the proportions of objects. If the viewport is mapped to the entire area of the display window and the size of the display window is changed, objects may be distorted unless the aspect ratio of the viewport is also adjusted.

Display of Character Strings

Character strings can be handled in one of two ways when they are mapped through the viewing pipeline to a viewport. The simplest mapping maintains a constant character size. This method could be employed with bitmap character patterns. But outline fonts could be transformed the same as other primitives; we just need to transform the defining positions for the line segments in the outline character shapes. Algorithms for determining the pixel patterns for the transformed characters are then applied when the other primitives in the scene are processed.

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 scene. And 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. In addition, we could list other information or menus in a third viewport.

It is also possible that two or more output devices could be operating concurrently on a particular system, and we can set up a clipping-window/viewport pair for each output device. A mapping to a selected output device is sometimes referred to as a **workstation transformation.** In this case, viewports could be specified in the coordinates of a particular display device, or each viewport could be specified within a unit square, which is then mapped to a chosen output device. Some graphics systems provide a pair of workstation functions for this purpose. One function is used to designate a clipping window for a selected output device, identified by a *workstation number*, and the other function is used to set the associated viewport for that device.

4 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

https://hemanthrajhemu.github.io

separate two-dimensional viewing-coordinate system as in Figure 3, 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 9 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 $(xw_{\min}, yw_{\min}) = (-1.0, -1.0)$ and $(xw_{\max}, yw_{\max}) = (1.0, 1.0)$. 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 lower-left 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`

https://hemanthrajhemu.github.io

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 using Matrix 10. Coordinates for the upper-right corner of the viewport are calculated for this transformation matrix in terms of the viewport width and height:

$$xv_{\max} = xv_{\min} + vpWidth, \qquad yv_{\max} = yv_{\min} + vpHeight \tag{11}$$

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 (see Section 3). 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

https://hemanthrajhemu.github.io

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);
```


https://hemanthrajhemu.github.io

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 change and objects may be distorted from their original shapes. We can adjust for a change in display-window dimensions using the statement

```
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");
```


https://hemanthrajhemu.github.io

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);
```


https://hemanthrajhemu.github.io

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.

https://hemanthrajhemu.github.io

Other GLUT Functions

The GLUT library provides a wide variety of routines to handle processes that are system dependent and to add features to the basic OpenGL library. For example, this library contains functions for generating bitmap and outline characters, and it provides functions for loading values into a color table. In addition, some GLUT functions are available for displaying three-dimensional objects, either as solids or in a wireframe representation. These objects include a sphere, a torus, and the five regular polyhedra (cube, tetrahedron, octahedron, dodecahedron, and icosahedron).

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.

We also have GLUT functions for obtaining and processing interactive input and for creating and managing menus. Individual routines are provided by GLUT for input devices such as a mouse, keyboard, graphics tablet, and spaceball.

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

As a demonstration of the use of the OpenGL viewport function, we use a split-screen effect to show two views of a triangle in the xy plane with its centroid at the world-coordinate origin. First, a viewport is defined in the left half of the display window, and the original triangle is displayed there in blue. Using the same clipping window, we then define another viewport for the right half of the display window, and the fill color is changed to red. The triangle is then rotated about its centroid and displayed in the second viewport.

```
#include <GL/glut.h>

class wcPt2D {
   public:
      GLfloat x, y;
};
```


https://hemanthrajhemu.github.io

```
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 ( );
}
```


https://hemanthrajhemu.github.io

5 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 multiwindow 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 $xw_{\min}$, $xw_{\max}$, $yw_{\min}$, and $yw_{\max}$. 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.

6 Two-Dimensional Point Clipping

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

$$\begin{aligned} xw_{\min} \le x \le xw_{\max} \\ yw_{\min} \le y \le yw_{\max} \end{aligned} \tag{12}$$

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

https://hemanthrajhemu.github.io

clouds, sea foam, smoke, or explosions that are modeled with "particles," such as the center coordinates for small circles or spheres.

7 Two-Dimensional Line Clipping

Figure 9 illustrates possible positions for straight-line segments in relationship to a standard 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 $\mathbf{P}_1$ to $\mathbf{P}_2$ 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 $\overline{\mathbf{P}_3\mathbf{P}_4}$ 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 (x_0, y_0) and $(x_{\text{end}}, y_{\text{end}})$ designate the two line endpoints:

$$\begin{aligned} x &= x_0 + u(x_{\text{end}} - x_0) \\ y &= y_0 + u(y_{\text{end}} - y_0) \qquad 0 \le u \le 1 \end{aligned} \tag{13}$$

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 xw_{min}, 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.

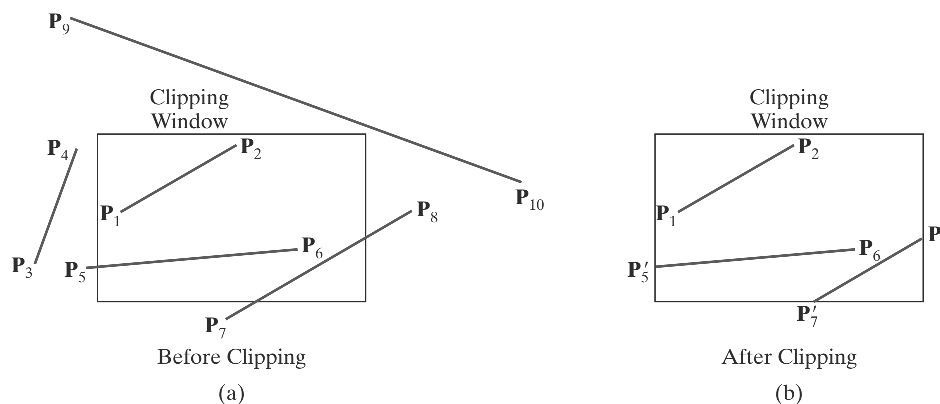


FIGURE 9
Clipping straight-line segments using a standard rectangular clipping window.

https://hemanthrajhemu.github.io

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

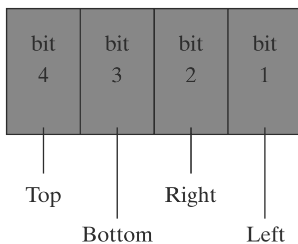


FIGURE 10
A possible ordering for the clipping-window boundaries corresponding to the bit positions in the Cohen-Sutherland endpoint region code.

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. 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. 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 $x < xw_{\min}$, and

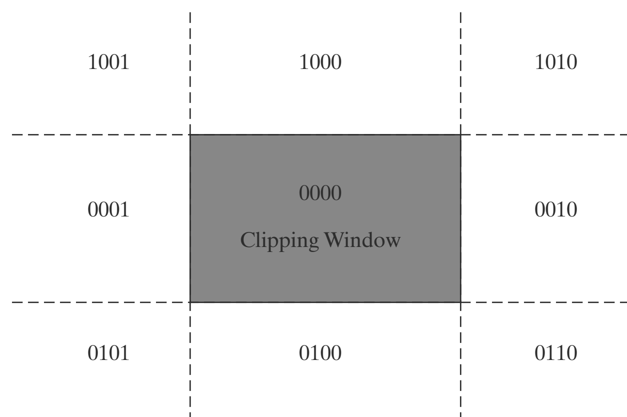


FIGURE 11
The nine binary region codes for identifying the position of a line endpoint, relative to the clipping-window boundaries.

https://hemanthrajhemu.github.io

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 $x - xw_{\text{min}}$; bit 2 is the sign bit of $xw_{\text{max}} - x$; bit 3 is the sign bit of $y - yw_{\text{min}}$; and bit 4 is the sign bit of $yw_{\text{max}} - y$.

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. 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. For the following discussion, we 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.

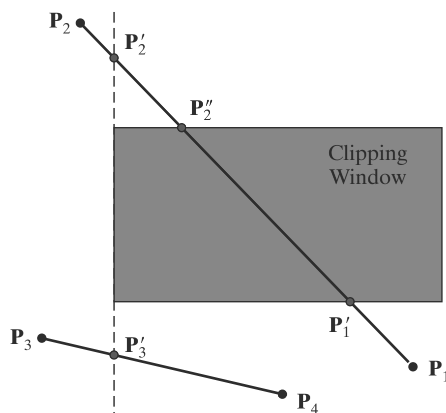


FIGURE 12
Lines extending from one clipping-window region to another may cross into the clipping window, or they could intersect one or more clipping boundaries without entering the window.

https://hemanthrajhemu.github.io

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 $\mathbf{P}_1$ to $\mathbf{P}_2$ are 0100 and 1001. Thus, $\mathbf{P}_1$ is inside the left clipping boundary and $\mathbf{P}_2$ is outside that boundary. We then calculate the intersection position $\mathbf{P}'_2$, and we clip off the line section from $\mathbf{P}_2$ 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 $\mathbf{P}_1$ 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 $\mathbf{P}_1$ 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 $\mathbf{P}_3$ is outside the left boundary and $\mathbf{P}_4$ is inside. Thus, we calculate the intersection position $\mathbf{P}'_3$ and eliminate the line section from $\mathbf{P}_3$ to $\mathbf{P}'_3$. By checking region codes for the endpoints $\mathbf{P}'_3$ and $\mathbf{P}_4$, 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.

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 $(x_{\text{end}}, y_{\text{end}})$, the y coordinate of the intersection point with a vertical clipping border line can be obtained with the calculation

$$y = y_0 + m(x - x_0) \tag{14}$$

where the x value is set to either xw_{min} or xw_{max}, and the slope of the line is calculated as $m = (y_{\text{end}} - y_0)/(x_{\text{end}} - x_0)$. Similarly, if we are looking for the intersection with a horizontal border, the x coordinate can be calculated as

$$x = x_0 + \frac{y - y_0}{m} \tag{15}$$

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

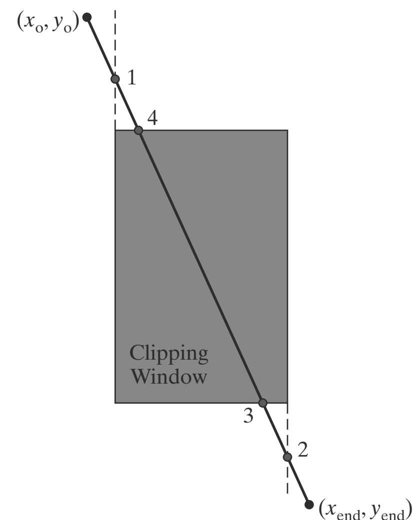


FIGURE 13
Four intersection positions (labeled from 1 to 4) for a line segment that is clipped against the window boundaries in the order left, right, bottom, top.

https://hemanthrajhemu.github.io

An implementation of the two-dimensional, Cohen-Sutherland line-clipping algorithm is given in the following procedures.

```
class wcPt2D {
   public:
      GLfloat x, y;
};

inline GLint round (const GLfloat a)  { return GLint (a + 0.5); }

/*  Define a four-bit code for each of the outside regions of a
 *  rectangular clipping window.
 */
const GLint winLeftBitCode = 0x1;
const GLint winRightBitCode = 0x2;
const GLint winBottomBitCode = 0x4;
const GLint winTopBitCode = 0x8;

/*  A bit-mask region code is also assigned to each endpoint of an input
 *  line segment, according to its position relative to the four edges of
 *  an input rectangular clip window.
 *
 *  An endpoint with a region-code value of 0000 is inside the clipping
 *  window, otherwise it is outside at least one clipping boundary.  If
 *  the 'or' operation for the two endpoint codes produces a value of
 *  false, the entire line defined by these two endpoints is saved
 *  (accepted).  If the 'and' operation between two endpoint codes is
 *  true, the line is completely outside the clipping window, and it is
 *  eliminated (rejected) from further processing.
 */
inline GLint inside (GLint code) { return GLint (!code); }
inline GLint reject (GLint code1, GLint code2)
                        { return GLint (code1 & code2); }
inline GLint accept (GLint code1, GLint code2)
                        { return GLint (!(code1 | code2)); }

GLubyte encode (wcPt2D pt, wcPt2D winMin, wcPt2D winMax)
{
  GLubyte code = 0x00;

  if (pt.x < winMin.x)
    code = code | winLeftBitCode;
  if (pt.x > winMax.x)
    code = code | winRightBitCode;
  if (pt.y < winMin.y)
    code = code | winBottomBitCode;
  if (pt.y > winMax.y)
    code = code | winTopBitCode;
  return (code);
}
```


https://hemanthrajhemu.github.io

```
void swapPts (wcPt2D * p1, wcPt2D * p2)
{
  wcPt2D tmp;

  tmp = *p1; *p1 = *p2; *p2 = tmp;
}

void swapCodes (GLubyte * c1, GLubyte * c2)
{
  GLubyte tmp;

  tmp = *c1; *c1 = *c2; *c2 = tmp;
}

void lineClipCohSuth (wcPt2D winMin, wcPt2D winMax, wcPt2D p1, wcPt2D p2)
{
  GLubyte code1, code2;
  GLint done = false, plotLine = false;
  GLfloat m;

  while (!done) {
    code1 = encode (p1, winMin, winMax);
    code2 = encode (p2, winMin, winMax);
    if (accept (code1, code2)) {
      done = true;
      plotLine = true;
    }
    else
      if (reject (code1, code2))
        done = true;
      else {
        /*  Label the endpoint outside the display window as p1. */
        if (inside (code1)) {
          swapPts (&p1, &p2);
          swapCodes (&code1, &code2);
        }
        /*  Use slope m to find line-clipEdge intersection.  */
        if (p2.x != p1.x)
          m = (p2.y - p1.y) / (p2.x - p1.x);
        if (code1 & winLeftBitCode) {
          p1.y += (winMin.x - p1.x) * m;
          p1.x = winMin.x;
        }
        else
          if (code1 & winRightBitCode) {
            p1.y += (winMax.x - p1.x) * m;
            p1.x = winMax.x;
          }
          else
            if (code1 & winBottomBitCode) {
              /*  Need to update p1.x for nonvertical lines only.  */
              if (p2.x != p1.x)
                p1.x += (winMin.y - p1.y) / m;
              p1.y = winMin.y;
            }
```


https://hemanthrajhemu.github.io

```
            else
              if (code1 & winTopBitCode) {
                if (p2.x != p1.x)
                  p1.x += (winMax.y - p1.y) / m;
                p1.y = winMax.y;
              }
      }
   }
   if (plotLine)
     lineBres (round (p1.x), round (p1.y), round (p2.x), round (p2.y));
}
```

Liang-Barsky Line Clipping

Faster line-clipping algorithms have been developed that do more line testing before proceeding to the intersection calculations. One of the earliest efforts in this direction is an algorithm developed by Cyrus and Beck, which is based on analysis of the parametric line equations. Later, Liang and Barsky independently devised an even faster form of the parametric line-clipping algorithm.

For a line segment with endpoints (x_0, y_0) and $(x_{\text{end}}, y_{\text{end}})$, we can describe the line with the parametric form

$$\begin{aligned} x &= x_0 + u\Delta x \\ y &= y_0 + u\Delta y \qquad 0 \le u \le 1 \end{aligned} \tag{16}$$

where $\Delta x = x_{\text{end}} - x_0$ and $\Delta y = y_{\text{end}} - y_0$. In the Liang-Barsky algorithm, the parametric line equations are combined with the point-clipping conditions 12 to obtain the inequalities

$$\begin{aligned} xw_{\text{min}} \le x_0 + u\Delta x \le xw_{\text{max}} \\ yw_{\text{min}} \le y_0 + u\Delta y \le yw_{\text{max}} \end{aligned} \tag{17}$$

which can be expressed as

$$u\, p_k \le q_k, \qquad k = 1, 2, 3, 4 \tag{18}$$

where parameters p and q are defined as

$$\begin{aligned} p_1 &= -\Delta x, & q_1 &= x_0 - xw_{\text{min}} \\ p_2 &= \Delta x, & q_2 &= xw_{\text{max}} - x_0 \\ p_3 &= -\Delta y, & q_3 &= y_0 - yw_{\text{min}} \\ p_4 &= \Delta y, & q_4 &= yw_{\text{max}} - y_0 \end{aligned} \tag{19}$$

Any line that is parallel to one of the clipping-window edges has $p_k = 0$ for the value of k corresponding to that boundary, where k = 1, 2, 3, and 4 correspond to the left, right, bottom, and top boundaries, respectively. If, for that value of k, we also find $q_k < 0$, then the line is completely outside the boundary and can be eliminated from further consideration. If $q_k \ge 0$, the line is inside the parallel clipping border.

When $p_k < 0$, the infinite extension of the line proceeds from the outside to the inside of the infinite extension of this particular clipping-window edge. If $p_k > 0$, the line proceeds from the inside to the outside. For a nonzero value of p_k, we can calculate the value of u that corresponds to the point where the infinitely

https://hemanthrajhemu.github.io

extended line intersects the extension of window edge k as

$$u = \frac{q_k}{p_k} \tag{20}$$

For each line, we can calculate values for parameters u_1 and u_2 that define that part of the line that lies within the clip rectangle. The value of u_1 is determined by looking at the rectangle edges for which the line proceeds from the outside to the inside ($p < 0$). For these edges, we calculate $r_k = q_k/p_k$. The value of u_1 is taken as the largest of the set consisting of 0 and the various values of r. Conversely, the value of u_2 is determined by examining the boundaries for which the line proceeds from inside to outside ($p > 0$). A value of r_k is calculated for each of these boundaries, and the value of u_2 is the minimum of the set consisting of 1 and the calculated r values. If $u_1 > u_2$, the line is completely outside the clip window and it can be rejected. Otherwise, the endpoints of the clipped line are calculated from the two values of parameter u.

This algorithm is implemented in the following code sections. Line intersection parameters are initialized to the values $u_1 = 0$ and $u_2 = 1$. For each clipping boundary, the appropriate values for p and q are calculated and used by the function `clipTest` to determine whether the line can be rejected or whether the intersection parameters are to be adjusted. When $p < 0$, parameter r is used to update u_1; when $p > 0$, parameter r is used to update u_2. If updating u_1 or u_2 results in $u_1 > u_2$, we reject the line. Otherwise, we update the appropriate u parameter only if the new value results in a shortening of the line. When $p = 0$ and $q < 0$, we can eliminate the line because it is parallel to and outside this boundary. If the line has not been rejected after all four values of p and q have been tested, the endpoints of the clipped line are determined from values of u_1 and u_2.

```
   class wcPt2D
   {
      private:
         GLfloat x, y;

      public:
      /*  Default Constructor: initialize position as (0.0, 0.0).  */
      wcPt3D ( ) {
         x = y = 0.0;
      }

      setCoords (GLfloat xCoord, GLfloat yCoord) {
         x = xCoord;
         y = yCoord;
      }

      GLfloat getx ( ) const {
         return x;
      }

      GLfloat gety ( ) const {
         return y;
      }
   };

inline GLint round (const GLfloat a)  { return GLint (a + 0.5); }
```


https://hemanthrajhemu.github.io

```
GLint clipTest (GLfloat p, GLfloat q, GLfloat * u1, GLfloat * u2)
{
  GLfloat r;
  GLint returnValue = true;

  if (p < 0.0) {
    r = q / p;
    if (r > *u2)
      returnValue = false;
    else
      if (r > *u1)
        *u1 = r;
  }
  else
    if (p > 0.0) {
      r = q / p;
      if (r < *u1)
        returnValue = false;
      else if (r < *u2)
        *u2 = r;
    }
    else
      /*  Thus p = 0 and line is parallel to clipping boundary.  */
      if (q < 0.0)
        /*  Line is outside clipping boundary.  */
        returnValue = false;

  return (returnValue);
}

void lineClipLiangBarsk (wcPt2D winMin, wcPt2D winMax, wcPt2D p1, wcPt2D p2)
{
  GLfloat u1 = 0.0, u2 = 1.0, dx = p2.getx ( ) - p1.getx ( ), dy;

  if (clipTest (-dx, p1.getx ( ) - winMin.getx ( ), &u1, &u2))
    if (clipTest (dx, winMax.getx ( ) - p1.getx ( ), &u1, &u2)) {
      dy = p2.gety ( ) - p1.gety ( );
      if (clipTest (-dy, p1.gety ( ) - winMin.gety ( ), &u1, &u2))
        if (clipTest (dy, winMax.gety ( ) - p1.gety ( ), &u1, &u2)) {
          if (u2 < 1.0) {
            p2.setCoords (p1.getx ( ) + u2 * dx, p1.gety ( ) + u2 * dy);
          }
          if (u1 > 0.0) {
            p1.setCoords (p1.getx ( ) + u1 * dx, p1.gety ( ) + u1 * dy);
          }
          lineBres (round (p1.getx ( )), round (p1.gety ( )),
                    round (p2.getx ( )), round (p2.gety ( )));
        }
    }
}
```

In general, the Liang-Barsky algorithm is more efficient than the Cohen-Sutherland line-clipping algorithm. Each update of parameters u_1 and u_2 requires only one division; and window intersections of the line are computed only once, when the final values of u_1 and u_2 have been computed. In contrast, the Cohen

https://hemanthrajhemu.github.io

and Sutherland algorithm can calculate intersections repeatedly along a line path, even though the line may be completely outside the clip window. In addition, each Cohen-Sutherland intersection calculation requires both a division and a multiplication. The two-dimensional Liang-Barsky algorithm can be extended to clip three-dimensional lines.

Nicholl-Lee-Nicholl Line Clipping

By creating more regions around the clipping window, the Nicholl-Lee-Nicholl (NLN) algorithm avoids multiple line-intersection calculations. In the Cohen-Sutherland method, for example, multiple intersections could be calculated along the path of a line segment before an intersection on the clipping rectangle is located or the line is completely rejected. These extra intersection calculations are eliminated in the NLN algorithm by carrying out more region testing before intersection positions are calculated. Compared to both the Cohen-Sutherland and the Liang-Barsky algorithms, the Nicholl-Lee-Nicholl algorithm performs fewer comparisons and divisions. The trade-off is that the NLN algorithm can be applied only to two-dimensional clipping, whereas both the Liang-Barsky and the Cohen-Sutherland methods are easily extended to three-dimensional scenes.

Initial testing to determine whether a line segment is completely inside the clipping window or outside the window limits can be accomplished with region-code tests, as in the previous two algorithms. If a trivial acceptance or rejection of the line is not possible, the NLN algorithm proceeds to set up additional clipping regions.

For a line with endpoints $\mathbf{P}_0$ and $\mathbf{P}_{\text{end}}$, we first determine the position of point $\mathbf{P}_0$ for the nine possible regions relative to the clipping window. Only the three regions shown in Figure 14 need be considered. If $\mathbf{P}_0$ lies in any one of the other six regions, we can move it to one of the three regions in Figure 14 using a symmetry transformation. For example, the region directly above the clip window can be transformed to the region left of the window using a reflection about the line $y = -x$, or we could use a 90° counterclockwise rotation.

Assuming that $\mathbf{P}_0$ and $\mathbf{P}_{\text{end}}$ are not both inside the clipping window, we next determine the position of $\mathbf{P}_{\text{end}}$ relative to $\mathbf{P}_0$. To do this, we create some new regions in the plane, depending on the location of $\mathbf{P}_0$. Boundaries of the new regions are semi-infinite line segments that start at the position of $\mathbf{P}_0$ and pass through the clipping-window corners. If $\mathbf{P}_0$ is inside the clipping window, we set up the four regions shown in Figure 15. Then, depending on which one of the four regions (L, T, R, or B) contains $\mathbf{P}_{\text{end}}$, we compute the line-intersection position with the corresponding window boundary.

If $\mathbf{P}_0$ is in the region to the left of the window, we set up the four regions labeled L, LT, LR, and LB in Figure 16. These four regions again determine a unique

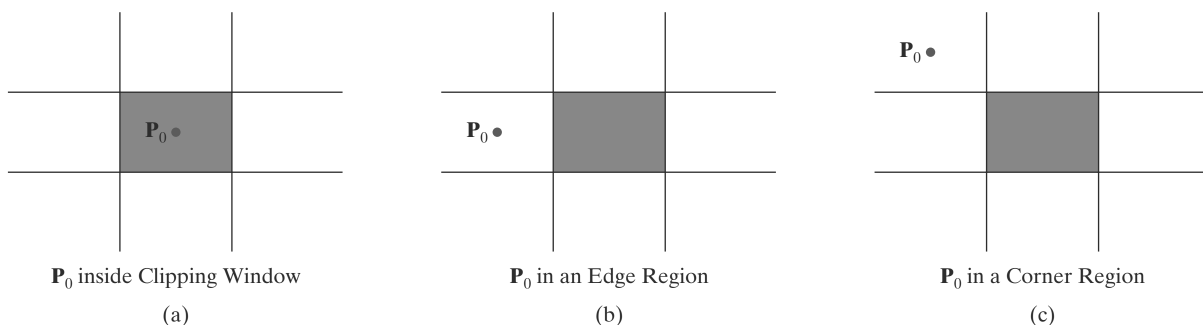


FIGURE 14
Three possible positions for a line endpoint $\mathbf{P}_0$ in the NLN line-clipping algorithm.

https://hemanthrajhemu.github.io

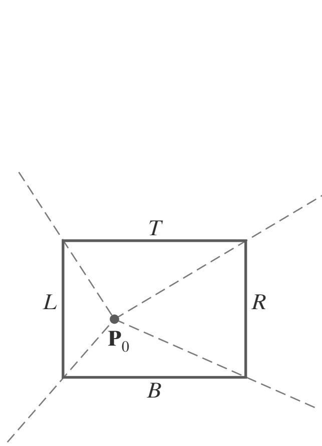


FIGURE 15 The four regions used in the NLN algorithm when $\mathbf{P_0}$ is inside the clipping window and $\mathbf{P_{end}}$ is outside.

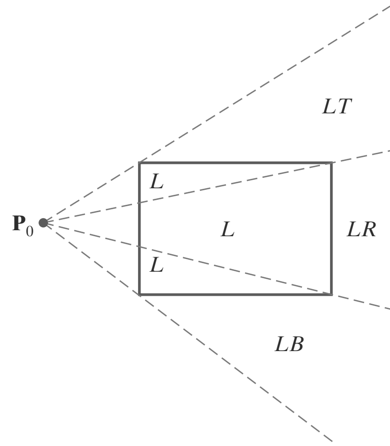


FIGURE 16 The four clipping regions used in the NLN algorithm when $\mathbf{P_0}$ is directly to the left of the clip window.

clipping-window edge for the line segment, relative to the position of $\mathbf{P}_{\text{end}}$. For instance, if $\mathbf{P}_{\text{end}}$ is in any one of the three regions labeled L, we clip the line at the left window border and save the line segment from this intersection point to $\mathbf{P}_{\text{end}}$. If $\mathbf{P}_{\text{end}}$ is in region LT, we save the line segment from the left window boundary to the top boundary. Similar processing is carried out for regions LR and LB. However, if $\mathbf{P}_{\text{end}}$ is not in any of the four regions L, LT, LR, or LB, the entire line is clipped.

For the third case, when $\mathbf{P}_0$ is to the left and above the clipping window, we use the regions in Figure 17. In this case, we have the two possibilities shown, depending on the position of $\mathbf{P}_0$ within the top-left corner of the clipping window. When $\mathbf{P}_0$ is closer to the left clipping boundary of the window, we use the regions in (a) of this figure. Otherwise, when $\mathbf{P}_0$ is closer to the top clipping boundary of the window, we use the regions in (b). If $\mathbf{P}_{\text{end}}$ is in one of the regions T, L, TR, TB, LR, or LB, this determines a unique clipping-window border for the intersection calculations. Otherwise, the entire line is rejected.

To determine the region in which $\mathbf{P}_{\text{end}}$ is located, we compare the slope of the line segment to the slopes of the boundaries of the NLN regions. For example, if $\mathbf{P}_0$ is left of the clipping window (Figure 16), then $\mathbf{P}_{\text{end}}$ is in region LT if

$$\text{slope}\overline{\mathbf{P}_0\mathbf{P}_{TR}} < \text{slope}\overline{\mathbf{P}_0\mathbf{P}_{\text{end}}} < \text{slope}\overline{\mathbf{P}_0\mathbf{P}_{TL}} \tag{21}$$

or

$$\frac{y_T - y_0}{x_R - x_0} < \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0} < \frac{y_T - y_0}{x_L - x_0} \tag{22}$$

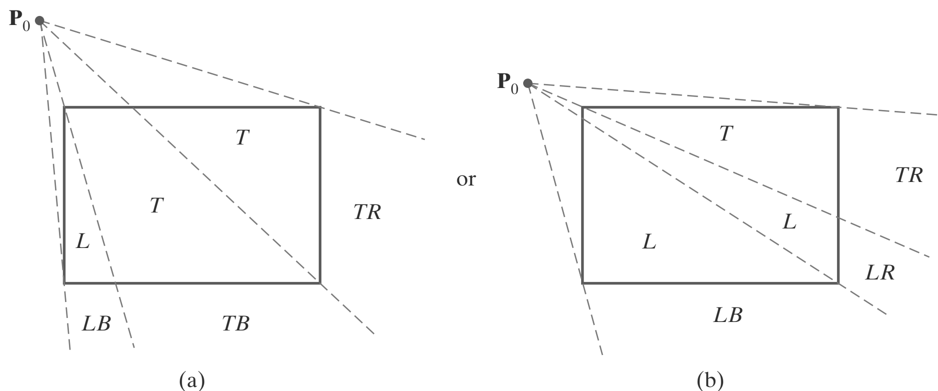


FIGURE 17 The two possible sets of clipping regions used in the NLN algorithm when $\mathbf{P_0}$ is above and to the left of the clipping window.

https://hemanthrajhemu.github.io

We clip the entire line if

$$(y_T - y_0)(x_{\text{end}} - x_0) < (x_L - x_0)(y_{\text{end}} - y_0) \tag{23}$$

The coordinate-difference calculations and product calculations used in the slope tests are saved and also used in the intersection calculations. From the parametric equations

$$x = x_0 + (x_{\text{end}} - x_0)u$$
$$y = y_0 + (y_{\text{end}} - y_0)u$$

we calculate an x-intersection position on the left window boundary as $x = x_L$, with $u = (x_L - x_0)/(x_{\text{end}} - x_0)$, so that the y-intersection position is

$$y = y_0 + \frac{y_{\text{end}} - y_0}{x_{\text{end}} - x_0}(x_L - x_0) \tag{24}$$

An intersection position on the top boundary has $y = y_T$ and $u = (y_T - y_0)/(y_{\text{end}} - y_0)$, with

$$x = x_0 + \frac{x_{\text{end}} - x_0}{y_{\text{end}} - y_0}(y_T - y_0) \tag{25}$$

Line Clipping Using Nonrectangular Polygon Clip Windows

In some applications, it may be desirable to clip lines against arbitrarily shaped polygons. Methods based on parametric line equations, such as either the Cyrus-Beck algorithm or the Liang-Barsky algorithm, can be readily extended to clip lines against convex polygon windows. We do this by modifying the algorithm to include the parametric equations for the boundaries of the clipping region. Preliminary screening of line segments can be accomplished by processing lines against the coordinate extents of the clipping polygon.

For concave-polygon clipping regions, we could still apply these parametric clipping procedures if we first split the concave polygon into a set of convex polygons. Another approach is simply to add one or more edges to the concave clipping area so that it is modified to a convex-polygon shape. Then a series of clipping operations can be applied using the modified convex polygon components, as illustrated in Figure 18. The line segment $\overline{\mathbf{P}_1\mathbf{P}_2}$ in (a) of this figure is to be clipped by the concave window with vertices $\mathbf{V}_1$, $\mathbf{V}_2$, $\mathbf{V}_3$, $\mathbf{V}_4$, and $\mathbf{V}_5$. Two convex clipping regions are obtained, in this case, by adding a line segment from $\mathbf{V}_4$ to $\mathbf{V}_1$. Then the line is clipped in two passes: (1) Line $\overline{\mathbf{P}_1\mathbf{P}_2}$ is clipped by the convex polygon with vertices $\mathbf{V}_1$, $\mathbf{V}_2$, $\mathbf{V}_3$, and $\mathbf{V}_4$ to yield the clipped segment $\overline{\mathbf{P}'_1\mathbf{P}'_2}$ [Figure 18(b)]. (2) The internal line segment $\overline{\mathbf{P}'_1\mathbf{P}'_2}$ is clipped off using the convex polygon with vertices $\mathbf{V}_1$, $\mathbf{V}_5$, and $\mathbf{V}_4$ [Figure 18(c)] to yield the final clipped line segment $\overline{\mathbf{P}''_1\mathbf{P}'_2}$.

Line Clipping Using Nonlinear Clipping-Window Boundaries

Circles or other curved-boundary clipping regions are also possible, but they require more processing because the intersection calculations involve nonlinear equations. At the first step, lines could be clipped against the bounding rectangle (coordinate extents) of the curved clipping region. Lines that are outside the coordinate extents are eliminated. To identify lines that are inside a circle, for instance, we could calculate the distance of the line endpoints from the circle center. If the square of this distance for both endpoints of a line is less than or equal to the radius squared, we can save the entire line. The remaining lines are then processed through the intersection calculations, which must solve simultaneous circle-line equations.

https://hemanthrajhemu.github.io

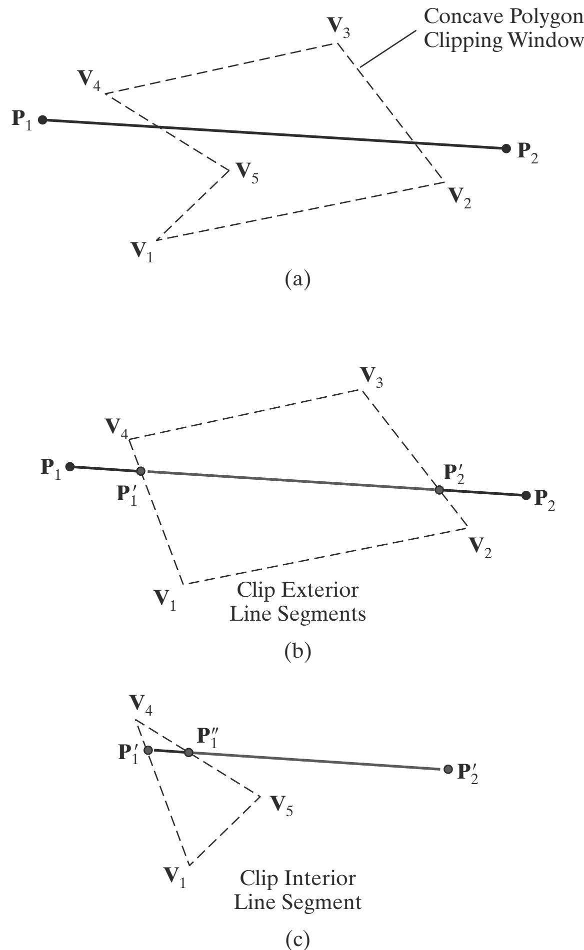


FIGURE 18
A concave- polygon clipping window (a), with vertex list (V_1, V_2, V_3, V_4, V_5), is modified to the convex polygon (V_1, V_2, V_3, V_4) in (b). The external segments of line $\overline{P_1P_2}$ are then snipped off using this convex clipping window. The resulting line segment, $\overline{P'_1P'_2}$, is next processed against the triangle (V_1, V_5, V_4) (c) to clip off the internal line segment $\overline{P'_1P''_1}$ to produce the final clipped line $\overline{P''_1P'_2}$.

8 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 form a 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. 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.

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.

https://hemanthrajhemu.github.io

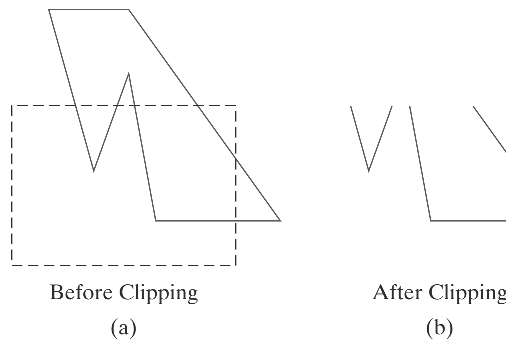


FIGURE 19
A line-clipping algorithm applied to the line segments of the polygon boundary in (a) generates the unconnected set of lines in (b).

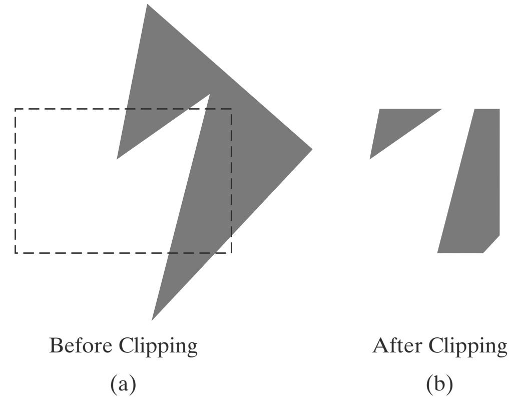


FIGURE 20
Display of a correctly clipped polygon fill area.

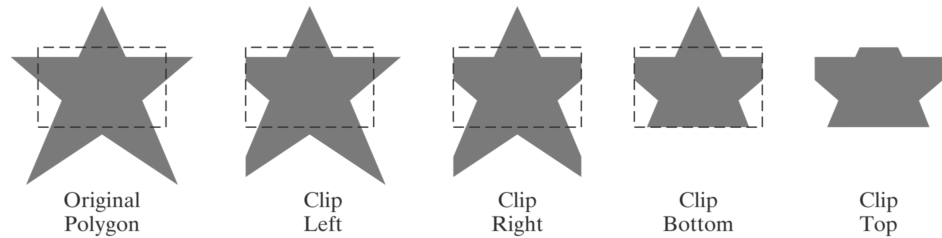


FIGURE 21
Processing a polygon fill area against successive clipping-window boundaries.

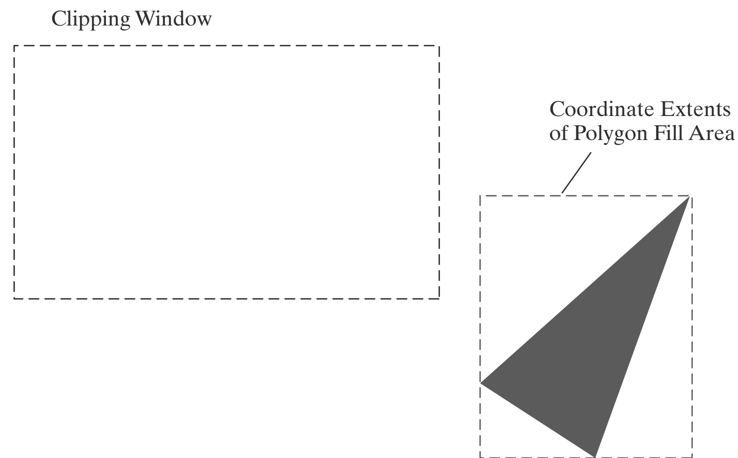


FIGURE 22
A polygon fill area with coordinate extents outside the right clipping boundary.

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).

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

https://hemanthrajhemu.github.io

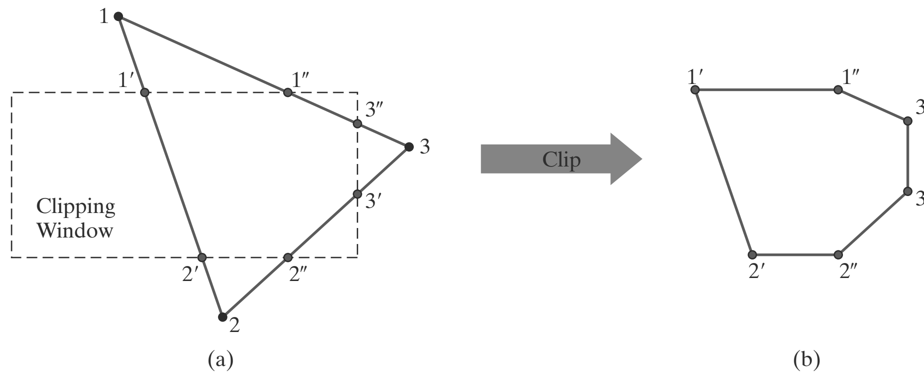


FIGURE 23
A convex-polygon fill area (a), defined with the vertex list {1, 2, 3}, is clipped to produce the fill-area shape shown in (b), which is defined with the output vertex list {1′, 2′, 2″, 3′, 3″, 1″}.

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.

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.

Because the Sutherland-Hodgman algorithm produces only one list of output vertices, it cannot correctly generate the two output polygons in Figure 20(b) that are the result of clipping the concave polygon shown in Figure 20(a). However, more processing steps can be added to the algorithm to allow it to produce multiple output vertex lists, so that general concave-polygon clipping could be accomodated. And 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. 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.

https://hemanthrajhemu.github.io

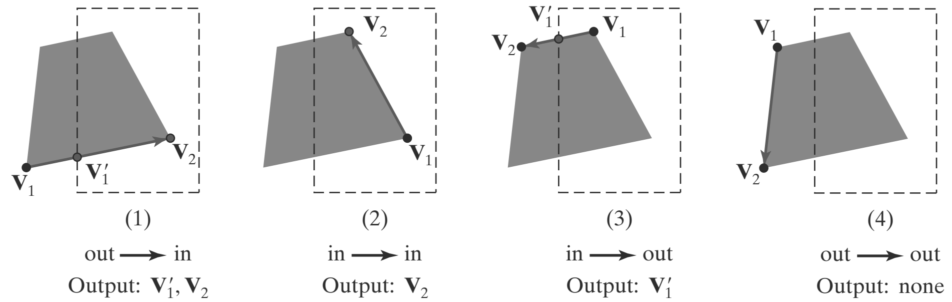
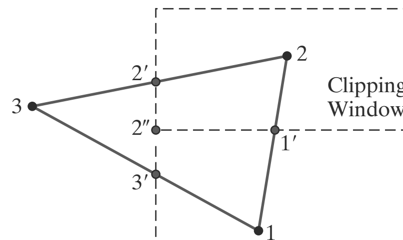


FIGURE 24
The four possible outputs generated by the left clipper, depending on the position of a pair of endpoints relative to the left boundary of the clipping window.



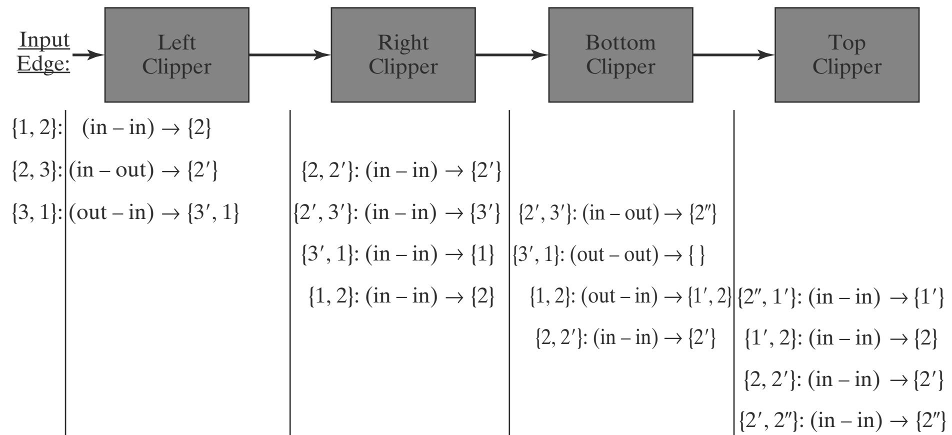


FIGURE 25
Processing a set of polygon vertices, {1, 2, 3}, through the boundary clippers using the Sutherland-Hodgman algorithm. The final set of clipped vertices is {1′, 2, 2′, 2″}.

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 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

https://hemanthrajhemu.github.io

```
  return (iPt);
}

void clipPoint (wcPt2D p, Boundary winEdge, wcPt2D wMin, wcPt2D wMax,
                wcPt2D * pOut, int * cnt, wcPt2D * first[], wcPt2D * s)
{
  wcPt2D iPt;

  /* If no previous point exists for this clipping boundary,
   * save this point.
   */
  if (!first[winEdge])
    first[winEdge] = &p;
  else
    /*  Previous point exists.  If p and previous point cross
     *  this clipping boundary, find intersection.  Clip against
     *  next boundary, if any.  If no more clip boundaries, add
     *  intersection to output list.
     */
    if (cross (p, s[winEdge], winEdge, wMin, wMax)) {
      iPt = intersect (p, s[winEdge], winEdge, wMin, wMax);
      if (winEdge < Top)
        clipPoint (iPt, b+1, wMin, wMax, pOut, cnt, first, s);
      else {
        pOut[*cnt] = iPt;  (*cnt)++;
      }
    }

  /*  Save p as most recent point for this clip boundary.  */
  s[winEdge] = p;

  /*  For all, if point inside, proceed to next boundary, if any.  */
  if (inside (p, winEdge, wMin, wMax))
    if (winEdge < Top)
      clipPoint (p, winEdge + 1, wMin, wMax, pOut, cnt, first, s);
    else {
      pOut[*cnt] = p;  (*cnt)++;
    }
}

void closeClip (wcPt2D wMin, wcPt2D wMax, wcPt2D * pOut,
                GLint * cnt, wcPt2D * first [ ], wcPt2D * s)
{
  wcPt2D pt;
  Boundary winEdge;

  for (winEdge = Left; winEdge <= Top; winEdge++) {
    if (cross (s[winEdge], *first[winEdge], winEdge, wMin, wMax)) {
      pt = intersect (s[winEdge], *first[winEdge], winEdge, wMin, wMax);
      if (winEdge < Top)
        clipPoint (pt, winEdge + 1, wMin, wMax, pOut, cnt, first, s);
      else {
        pOut[*cnt] = pt;  (*cnt)++;
      }
    }
  }
}
```


https://hemanthrajhemu.github.io

top clipper is the set of vertices defining the clipped fill area. For this example, the output vertex list is $\{1', 2, 2', 2''\}$.

A sequential implementation of the Sutherland-Hodgman polygon-clipping algorithm is demonstrated in the following set of procedures. An input set of vertices is converted to an output vertex list by clipping it against the four edges of the axis-aligned rectangular clipping region.

```
typedef enum { Left, Right, Bottom, Top } Boundary;
const GLint nClip = 4;

GLint inside (wcPt2D p, Boundary b, wcPt2D wMin, wcPt2D wMax)
{
  switch (b) {
  case Left:   if (p.x < wMin.x) return (false); break;
  case Right:  if (p.x > wMax.x) return (false); break;
  case Bottom: if (p.y < wMin.y) return (false); break;
  case Top:    if (p.y > wMax.y) return (false); break;
  }
  return (true);
}

GLint cross (wcPt2D p1, wcPt2D p2, Boundary winEdge, wcPt2D wMin, wcPt2D wMax)
{
  if (inside (p1, winEdge, wMin, wMax) == inside (p2, winEdge, wMin, wMax))
    return (false);
  else return (true);
}

wcPt2D intersect (wcPt2D p1, wcPt2D p2, Boundary winEdge,
                                           wcPt2D wMin, wcPt2D wMax)
{
  wcPt2D iPt;
  GLfloat m;

  if (p1.x != p2.x) m = (p1.y - p2.y) / (p1.x - p2.x);
  switch (winEdge) {
  case Left:
    iPt.x = wMin.x;
    iPt.y = p2.y + (wMin.x - p2.x) * m;
    break;
  case Right:
    iPt.x = wMax.x;
    iPt.y = p2.y + (wMax.x - p2.x) * m;
    break;
  case Bottom:
    iPt.y = wMin.y;
    if (p1.x != p2.x) iPt.x = p2.x + (wMin.y - p2.y) / m;
    else iPt.x = p2.x;
    break;
  case Top:
    iPt.y = wMax.y;
    if (p1.x != p2.x) iPt.x = p2.x + (wMax.y - p2.y) / m;
    else iPt.x = p2.x;
    break;
  }
```


https://hemanthrajhemu.github.io

```
GLint polygonClipSuthHodg (wcPt2D wMin, wcPt2D wMax, GLint n, wcPt2D * pIn, wcPt2D * pOut)
{
  /*  Parameter "first" holds pointer to first point processed for
   *  a boundary; "s" holds most recent point processed for boundary.
   */
  wcPt2D * first[nClip] = { 0, 0, 0, 0 }, s[nClip];
  GLint k, cnt = 0;

  for (k = 0; k < n; k++)
    clipPoint (pIn[k], Left, wMin, wMax, pOut, &cnt, first, s);

  closeClip (wMin, wMax, pOut, &cnt, first, s);
  return (cnt);
}
```

When a concave polygon is clipped with the Sutherland-Hodgman algorithm, extraneous lines may be displayed. An example of this effect is demonstrated in Figure 26. This occurs when the clipped polygon should have two or more separate sections. But since there is only one output vertex list, the last vertex in the list is always joined to the first vertex.

There are several things we can do to display clipped concave polygons correctly. For one, we could split a concave polygon into two or more convex polygons and process each convex polygon separately using the Sutherland-Hodgman algorithm. Another possibility is to modify the Sutherland- Hodgman method so that the final vertex list is checked for multiple intersection points along any clipping-window boundary. If we find more than two vertex positions along any clipping boundary, we can separate the list of vertices into two or more lists that correctly identify the separate sections of the clipped fill area. This may require extensive analysis to determine whether some points along the clipping boundary should be paired or whether they represent single vertex points that have been clipped.Athird possibility is to use a more general polygon clipper that has been designed to process concave polygons correctly.

Weiler-Atherton Polygon Clipping

This algorithm provides a general polygon-clipping approach that can be used to clip a fill area that is either a convex polygon or a concave polygon. Moreover, the method was developed as a means for identifying visible surfaces in a three-dimensional scene. Therefore, we could also use this approach to clip any polygon fill area against a clipping window with any polygon shape.

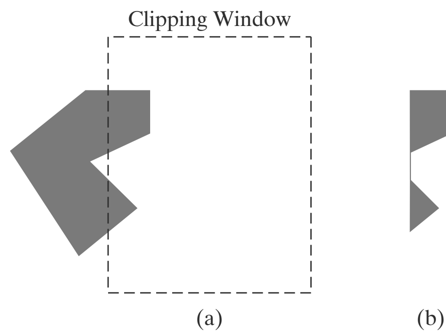


FIGURE 26
Clipping the concave polygon in (a) using the Sutherland-Hodgman algorithm produces the two connected areas in (b).

https://hemanthrajhemu.github.io

Instead of simply clipping the fill-area edges as in the Sutherland-Hodgman method, the Weiler-Atherton algorithm traces around the perimeter of the fill polygon searching for the borders that enclose a clipped fill region. In this way, multiple fill regions, as in Figure 26(b), can be identified and displayed as separate, unconnected polygons. To find the edges for a clipped fill area, we follow a path (either counterclockwise or clockwise) around the fill area that detours along a clipping-window boundary whenever a polygon edge crosses to the outside of that boundary. The direction of a detour at a clipping-window border is the same as the processing direction for the polygon edges.

We can usually determine whether the processing direction is counterclockwise or clockwise from the ordering of the vertex list that defines a polygon fill area. In most cases, the vertex list is specified in a counterclockwise order as a means for defining the front face of the polygon. Thus, the cross-product of two successive edge vectors that form a convex angle determines the direction for the normal vector, which is in the direction from the back face to the front face of the polygon. If we do not know the vertex ordering, we could calculate the normal vector, or we can locate the interior of the fill area from any reference position. Then, if we sequentially process the edges so that the polygon interior is always on our left, we obtain a counterclockwise traversal. Otherwise, with the interior to our right, we have a clockwise traversal.

For a counterclockwise traversal of the polygon fill-area vertices, we apply the following Weiler-Atherton procedures:

1. Process the edges of the polygon fill area in a counterclockwise order until an inside-outside pair of vertices is encountered for one of the clipping boundaries; that is, the first vertex of the polygon edge is inside the clip region and the second vertex is outside the clip region.
2. Follow the window boundaries in a counterclockwise direction from the exit-intersection point to another intersection point with the polygon. If this is a previously processed point, proceed to the next step. If this is a new intersection point, continue processing polygon edges in a counterclockwise order until a previously processed vertex is encountered.
3. Form the vertex list for this section of the clipped fill area.
4. Return to the exit-intersection point and continue processing the polygon edges in a counterclockwise order.

Figure 27 illustrates the Weiler-Atherton clipping of a concave polygon against a standard, rectangular clipping window for a counterclockwise traversal of the polygon edges. For a clockwise edge traversal, we would use a clockwise clipping-window traversal.

Starting from the vertex labeled 1 in Figure 27(a), the next polygon vertex to process in a counterclockwise order is labeled 2. Thus, this edge exits the clipping window at the top boundary. We calculate this intersection position (point $1'$) and make a left turn there to process the window borders in a counterclockwise direction. Proceeding along the top border of the clipping window, we do not intersect a polygon edge before reaching the left window boundary. Therefore, we label this position as vertex $1''$ and follow the left boundary to the intersection position $1'''$. We then follow this polygon edge in a counterclockwise direction, which returns us to vertex 1. This completes a circuit of the window boundaries and identifies the vertex list $\{1, 1', 1'', 1'''\}$ as a clipped region of the original fill area. Processing of the polygon edges is then resumed at point $1'$. The edge defined by points 2 and 3 crosses to the outside of the left boundary, but points 2 and $2'$

https://hemanthrajhemu.github.io

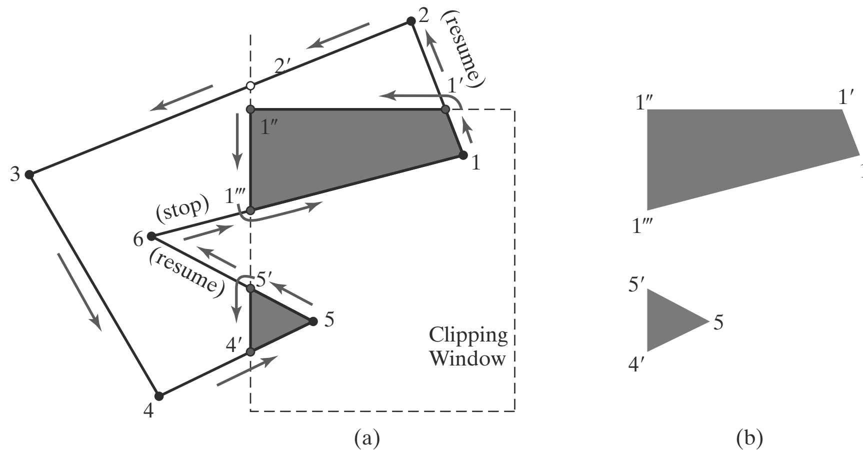


FIGURE 27
A concave polygon (a), defined with the vertex list {1, 2, 3, 4, 5, 6}, is clipped using the Weiler-Atherton algorithm to generate the two lists {1, 1′, 1″, 1‴} and {4′, 5, 5′}, which represent the separate polygon fill areas shown in (b).

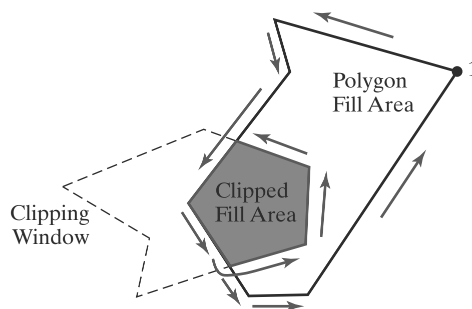


FIGURE 28
Clipping a polygon fill area against a concave-polygon clipping window using the Weiler-Atherton algorithm.

are above the top clipping-window border and points 2′ and 3 are to the left of the clipping region. Also, the edge with endpoints 3 and 4 is outside the left clipping boundary, but the next edge (from endpoint 4 to endpoint 5) reenters the clipping region and we pick up intersection point 4′. The edge with endpoints 5 and 6 exits the window at intersection position 5′, so we detour down the left clipping boundary to obtain the closed vertex list {4′, 5, 5′}. We resume the polygon edge processing at position 5′, which returns us to the previously processed point 1‴. At this point, all polygon vertices and edges have been processed, so the fill area is completely clipped.

Polygon Clipping Using Nonrectangular Polygon Clip Windows

The Liang-Barsky algorithm and other parametric line-clipping methods are particularly well suited for processing polygon fill areas against convex-polygon clipping windows. In this approach, we use a parametric representation for the edges of both the fill area and the clipping window, and both polygons are represented with a vertex list. We first compare the positions of the bounding rectangles for the fill area and the clipping polygon. If we cannot identity the fill area as completely outside the clipping polygon, we can use inside-outside tests to process the parametric edge equations. After completing all the region tests, we solve pairs of simultaneous parametric line equations to determine the window intersection positions.

We can also process any polygon fill area against any polygon-shaped clipping window (convex or concave), as in Figure 28, using the edge-traversal approach of the Weiler-Atherton algorithm. In this case, we need to maintain a vertex list for the clipping window as well as for the fill area, with both lists arranged in a counterclockwise (or clockwise) order. In addition, we need to apply

https://hemanthrajhemu.github.io

inside-outside tests to determine whether a fill-area vertex is inside or outside a particular clipping-window boundary. As in the previous examples, we follow the window boundaries whenever a fill-area edge exits a clipping boundary. This clipping method can also be used when either the fill area or the clipping window contains holes that are defined with polygon borders. In addition, we can use this basic approach in constructive solid-geometry applications to identify the result of a union, intersection, or difference operation on two polygons. In fact, locating the clipped region of a fill area is equivalent to determining the intersection of two planar areas.

Polygon Clipping Using Nonlinear Clipping-Window Boundaries

One method for processing a clipping window with curved boundaries is to approximate the boundaries with straight-line sections and use one of the algorithms for clipping against a general polygon-shaped clipping window. Alternatively, we could use the same general procedures that we discussed for line segments. First, we can compare the coordinate extents of the fill area to the coordinate extents of the clipping window. Depending on the shape of the clipping window, we may also be able to perform some other region tests based on symmetric considerations. For fill areas that cannot be identified as completely inside or completely outside the clipping window, we ultimately need to calculate the window intersection positions with the fill area.

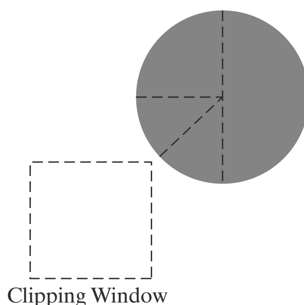


FIGURE 29
A circle fill area, showing the quadrant and octant sections that are outside the clipping-window boundaries.

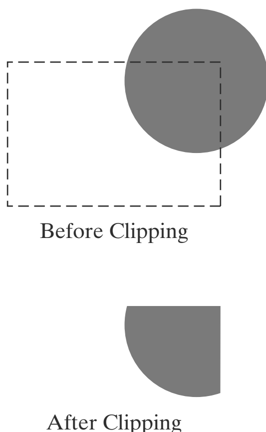


FIGURE 30
Clipping a circle fill area.

9 Curve Clipping

Areas with curved boundaries can be clipped with methods similar to those discussed in the previous sections. If the objects are approximated with straight-line boundary sections, we use a polygon-clipping method. Otherwise, the clipping procedures involve nonlinear equations, and this requires more processing than for objects with linear boundaries.

We can first test the coordinate extents of an object against the clipping boundaries to determine whether it is possible to accept or reject the entire object trivially. If not, we could check for object symmetries that we might be able to exploit in the initial accept/reject tests. For example, circles have symmetries between quadrants and octants, so we could check the coordinate extents of these individual circle regions. We cannot reject the complete circular fill area in Figure 29 just by checking its overall coordinate extents. But half of the circle is outside the right clipping border (or outside the top border), the upper-left quadrant is above the top clipping border, and the remaining two octants can be similarly eliminated.

An intersection calculation involves substituting a clipping-boundary position (xw_{min}, xw_{max}, yw_{min}, or yw_{max}) in the nonlinear equation for the object boundary and solving for the other coordinate value. Once all intersection positions have been evaluated, the defining positions for the object can be stored for later use by the scan-line fill procedures. Figure 30 illustrates circle clipping against a rectangular window. For this example, the circle radius and the endpoints of the clipped arc can be used to fill the clipped region, by invoking the circle algorithm to locate positions along the arc between the intersection endpoints.

Similar procedures can be applied when clipping a curved object against a general polygon clipping region. On the first pass, we could compare the bounding rectangle of the object with the bounding rectangle of the clipping region. If this does not save or eliminate the entire object, we next solve the simultaneous line-curve equations to determine the clipping intersection points.

https://hemanthrajhemu.github.io

10 Text Clipping

Several techniques can be used to provide text clipping in a graphics package. In a particular application, the choice of clipping method depends on how characters are generated and what requirements we have for displaying character strings.

The simplest method for processing character strings relative to the limits of a clipping window is to use the *all-or-none string-clipping* strategy shown in Figure 31. If all of the string is inside the clipping window, we display the entire string. Otherwise, the entire string is eliminated. This procedure is implemented by examining the coordinate extents of the text string. If the coordinate limits of this bounding rectangle are not completely within the clipping window, the string is rejected.

An alternative is to use the *all-or-none character-clipping* strategy. Here we eliminate only those characters that are not completely inside the clipping window (Figure 32). In this case, the coordinate extents of individual characters are compared to the window boundaries. Any character that is not completely within the clipping-window boundary is eliminated.

A third approach to text clipping is to clip the components of individual characters. This provides the most accurate display of clipped character strings, but it requires the most processing. We now treat characters in much the same way that we treated lines or polygons. If an individual character overlaps a clipping window, we clip off only the parts of the character that are outside the window (Figure 33). Outline character fonts defined with line segments are processed in this way using a polygon-clipping algorithm. Characters defined with bit maps are clipped by comparing the relative position of the individual pixels in the character grid patterns to the borders of the clipping region.

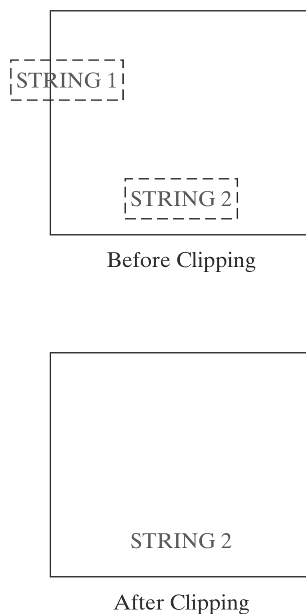


FIGURE 31
Text clipping using the coordinate extents for an entire string.

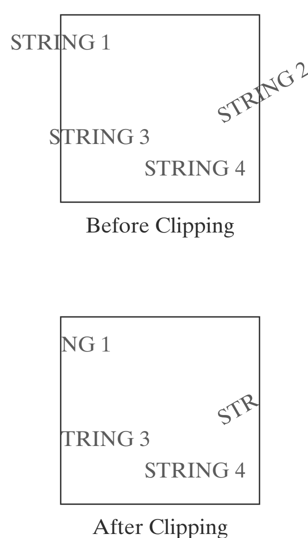


FIGURE 32
Text clipping using the bounding rectangle for individual characters in a string.

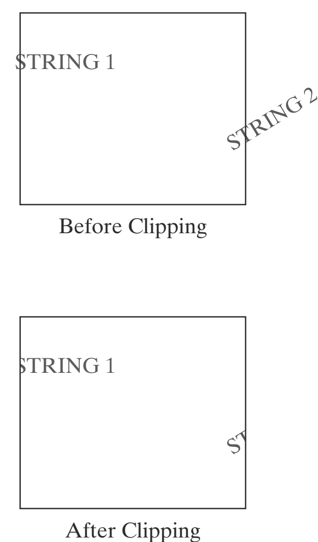


FIGURE 33
Text clipping performed on the components of individual characters.

https://hemanthrajhemu.github.io

11 Summary

The two-dimensional viewing-transformation pipeline is a series of operations that result in the display of a world-coordinate picture that has been defined in the xy plane. After we construct the scene, it can be mapped to a viewing-coordinate reference frame, then to a normalized coordinate system where clipping routines can be applied. Finally, the scene is transferred to device coordinates for display. Normalized coordinates can be specified in the range from 0 to 1 or in the range from −1 to 1, and they are used to make graphics packages independent of the output-device requirements.

We select part of a scene for display on an output device using a clipping window, which can be described in the world-coordinate system or in a viewing-coordinate frame defined relative to world coordinates. The contents of the clipping window are transferred to a viewport for display on an output device. In some systems, a viewport is specified within normalized coordinates. Other systems specify the viewport in device coordinates. Typically, the clipping window and viewport are rectangles whose edges are parallel to the coordinate axes. An object is mapped to the viewport so that it has the same relative position in the viewport as it has in the clipping window. To maintain the relative proportions of an object, the viewport must have the same aspect ratio as the corresponding clipping window. In addition, we can set up any number of clipping windows and viewports for a scene.

Clipping algorithms are usually implemented in normalized coordinates, so that all geometric transformations and viewing operations that are independent of device coordinates can be concatenated into one transformation matrix. With the viewport specified in device coordinates, we can clip a two-dimensional scene against a normalized, symmetric square, with normalized coordinates varying from −1 to 1, before transferring the contents of the normalized, symmetric square to the viewport.

All graphics packages include routines for clipping straight-line segments and polygon fill areas. Packages that contain functions for specifying single point positions or text strings also include clipping routines for those graphics primitives. Because the clipping calculations are time-consuming, the development of improved clipping algorithms continues to be an area of major concern in computer graphics. Cohen and Sutherland developed a line-clipping algorithm that uses a region code to identify the position of a line endpoint relative to the clipping-window boundaries. Endpoint region codes are used to identify quickly those lines that are completely inside the clipping window and some of the lines that are completely outside. For the remaining lines, intersection positions at the window boundaries must be calculated. Liang and Barsky developed a faster line-clipping algorithm that represents line segments with parametric equations, similar to the Cyrus-Beck algorithm. This approach allows more testing to be accomplished before proceeding to the intersection calculations. The Nicholl-Lee-Nicholl (NLN) algorithm further reduces intersection calculations by using more region testing in the xy plane. Parametric line-clipping methods are extended easily to convex clipping windows and to three-dimensional scenes. However, the NLN approach applies only to two-dimensional line segments.

Algorithms for clipping straight-line segments against concave-polygon clipping windows have also been developed. One approach is to split a concave clipping window into a set of convex polygons and apply the parametric line-clipping methods. Another approach is to add edges to the concave window to modify it to a convex shape. Then a series of exterior and interior clipping operations can be performed to obtain the clipped line segment.

https://hemanthrajhemu.github.io

TABLE 1

Summary of OpenGL Two-Dimensional Viewing Functions

Function	Description
gluOrtho2D	Specifies clipping-window coordinates as parameters for a two-dimensional orthogonal projection.
glViewport	Specifies screen-coordinate parameters for a viewport.
glGetIntegerv	Uses arguments GL_VIEWPORT and vpArray to obtain parameters for the currently active viewport.
glutInit	Initializes the GLUT library.
glutInitWindowPosition	Specifies coordinates for the top-left corner of a display window.
glutInitWindowSize	Specifies width and height for a display window.
glutCreateWindow	Creates a display window (which is assigned an integer identifier) and specify a display-window title.
glutInitDisplayMode	Selects parameters such as buffering and color mode for a display window.
glClearColor	Specifies a background RGB color for a display window.
glClearIndex	Specifies a background color for a display window using color-index mode.
glutDestroyWindow	Specifies an identifier number for a display window that is to be deleted.
glutSetWindow	Specifies the identifier number for a display window that is to be the current display window.
glutPositionWindow	Resets the screen location for the current display window.
glutReshapeWindow	Resets the width and height for the current display window.
glutFullScreen	Sets current display window to the size of the video screen.
glutReshapeFunc	Specifies a function that is to be invoked when display-window size is changed.
glutIconifyWindow	Converts the current display window to an icon.
glutSetIconTitle	Specifies a label for a display-window icon.
glutSetWindowTitle	Specifies new title for the current display window.
glutPopWindow	Moves current display window to the "top"; i.e., in front of all other windows.
glutPushWindow	Moves current display window to the "bottom"; i.e., behind all other windows.
glutShowWindow	Returns the current display window to the screen.
glutCreateSubWindow	Creates a second-level window within a display window.
glutSetCursor	Selects a shape for the screen cursor.
glutDisplayFunc	Invokes a function to create a picture within the current display window.
glutPostRedisplay	Renews the contents of the current display window.
glutMainLoop	Executes the computer-graphics program.
glutIdleFunc	Specifies a function to execute when the system is idle.
glutGet	Queries the system about a specified state parameter.

https://hemanthrajhemu.github.io

Although clipping windows with curved boundaries are rarely used, we can apply similar line-clipping methods. However, intersection calculations now involve nonlinear equations.

A polygon fill area is defined with a vertex list, and polygon-clipping procedures must retain information about how the clipped edges are to be connected as the polygon proceeds through the various processing stages. In the Sutherland-Hodgman algorithm, pairs of fill-area vertices are processed by each boundary clipper in turn, and clipping information for that edge is passed immediately to the next clipper, which allows the four clipping routines (left, right, bottom, and top) to be operating in parallel. This algorithm provides an efficient method for clipping convex-polygon fill areas. However, when a clipped concave polygon contains disjoint sections, the Sutherland-Hodgman algorithm produces extraneous connecting line segments. Extensions of parametric line clippers, such as the Liang-Barsky method, can also be used to clip convex polygon fill areas. Both convex and concave fill areas can be clipped correctly with the Weiler-Atherton algorithm, which uses a boundary-traversal approach.

Fill areas can be clipped against convex clipping windows using an extension of the parametric line-representation approach. And the Weiler-Atherton method can clip any polygon fill area using any polygon-shaped clipping window. Fill areas can be clipped against windows with nonlinear boundaries by using a polygon approximation for the window or by processing the fill area against the curved window boundaries.

The fastest text-clipping method is the all-or-none strategy, which completely clips a text string if any part of the string is outside any clipping-window boundary. Alternatively, we could clip a text string by eliminating only those characters in the string that are not completely inside the clipping window. And the most accurate text-clipping method is to apply either point, line, polygon, or curve clipping to the individual characters in a string, depending on whether characters are defined as point grids or outline fonts.

Although OpenGL is designed for three-dimensional applications, a two-dimensional GLU function is provided for specifying a standard, rectangular clipping window in world coordinates. In OpenGL, the clipping-window coordinates are parameters for the projection transformation. Therefore, we first need to invoke the projection matrix mode. Next we can specify the viewport, using a function in the basic OpenGL library, and a display window, using GLUT functions. A wide range of GLUT functions are available for setting various display-window parameters. Table 1 summarizes the OpenGL two-dimensional viewing functions. In addition, the table lists some viewing-related functions.

REFERENCES

Line-clipping algorithms are discussed in Sproull and Sutherland (1968), Cyrus and Beck (1978), Liang and Barsky (1984), and Nicholl, Lee, and Nicholl (1987). Methods for improving the speed of the Cohen-Sutherland line-clipping algorithm are given in Duvanenko (1990).

Basic polygon-clipping methods are presented in Sutherland and Hodgman (1974) and in Liang and Barsky (1983). General techniques for clipping arbitrarily shaped polygons against each other are given in Weiler and Atherton (1977) and in Weiler (1980).

Viewing operations in OpenGL are discussed in Woo, et al. (1999). Display-window GLUT routines are discussed in Kilgard (1996), and additional information on GLUT can be obtained online at http://reality.sgi.com/opengl/glut3/glut3.html.

https://hemanthrajhemu.github.io

EXERCISES

1 Write a procedure to calculate the elements of matrix 1 for transforming two-dimensional world coordinates to viewing coordinates, given the viewing coordinate origin $\mathbf{P}_0$ and the view up vector **V.**

2 Derive matrix 8 for transferring the contents of a clipping window to a viewport by first scaling the window to the size of the viewport, then translating the scaled window to the viewport position. Use the center of the clipping window as the reference point for the scaling and translation operations.

3 Write a procedure to calculate the elements of matrix 9 for transforming a clipping window to the symmetric normalized square.

4 Write a set of procedures to implement the two-dimensional viewing pipeline without clipping operations. Your program should allow a scene to be constructed with modeling-coordinate transformations, a specified viewing system, and a transformation to the symmetric normalized square. As an option, a viewing table could be implemented to store different sets of viewing transformation parameters.

5 Write a complete program to implement the Cohen-Sutherland line-clipping algorithm.

6 Modify the program in the previous exercise to produce an animation of a single line whose length is longer than the diagonal length of the viewing window. The midpoint of the line should be placed at the center of the viewing window and the line should rotate clockwise by a small amount in each frame. The clipping algorithm implemented in the previous exercise should clip the line appropriately in each frame.

7 Carefully discuss the rationale behind the various tests and methods for calculating the intersection parameters u_1 and u_2 in the Liang-Barsky line-clipping algorithm.

8 Compare the number of arithmetic operations performed in the Cohen-Sutherland and the Liang-Barsky line-clipping algorithms for several different line orientations relative to a clipping window.

9 Write a complete program to implement the Liang-Barsky line-clipping algorithm.

10 Modify the program in the previous exercise to produce an animation similar to the one described in Exercise 6. The clipping algorithm implemented in the previous exercise should clip the line appropriately in each frame.

11 Devise symmetry transformations for mapping the intersection calculations for the three regions in Figure 14 to the other six regions of the xy plane.

12 Set up a detailed algorithm for the Nicholl-Lee-Nicholl approach to line clipping for any input pair of line endpoints.

13 Compare the number of arithmetic operations performed in the NLN algorithm to both the Cohen-Sutherland and Liang-Barsky line-clipping algorithms, for several different line orientations relative to a clipping window.

14 Adapt the Liang-Barsky line-clipping algorithm to polygon clipping.

15 Use the implementation of Liang-Barsky polygon clipping developed in the previous exercise to write a program that displays an animation of a moving hexagon in the display window. The hexagon should be displayed as moving into the window from the top-left corner of the window diagonally towards the bottom-right corner and off the screen. Once the hexagon has exited the window completely the animation should repeat.

16 Set up a detailed algorithm for Weiler-Atherton polygon clipping, assuming that the clipping window is a rectangle in standard position.

17 Use the implementation of Weiler-Atherton polygon clipping developed in the previous exercise to write a program that produces a similar animation to the one described in Exercise 14.

18 Devise an algorithm for Weiler-Atherton polygon clipping, where the clipping window can be any convex polygon.

19 Devise an algorithm for Weiler-Atherton polygon clipping, where the clipping window can be any specified polygon (convex or concave).

20 Write a routine to clip an ellipse in standard position against a rectangular window.

21 Assuming that all characters in a text string have the same width, develop a text-clipping algorithm that clips a string according to the all-or-none character-clipping strategy.

22 Use the implementation of text clipping developed in the previous exercise to write a program that displays an animation of a moving marquee in the display window. That is, a sequence of characters should be displayed as moving into the window from the left side, across the window horizontally, and out of the window on the right side. Once all of the characters have exited the viewport on the right side the animation should repeat.

23 Develop a text-clipping algorithm that clips individual characters, assuming that the characters are defined in a pixel grid of a specified size.

24 Use the implementation of text clipping developed in the previous exercise to write a program that performs the same behavior as that in Exercise 21.

https://hemanthrajhemu.github.io

IN MORE DEPTH

1 Implement both the Sutherland-Hodgman and Weiler-Atherton polygon-clipping algorithms in two separate routines. Use them to clip the objects in the current snapshot of your scene against a sub-rectangle of the full scene extents. Compare the performance of the two algorithms. Make any modifications necessary to handle clipping of concave polygons in your scene using the Sutherland-Hodgman algorithm. The routines should take in the position and size of a rectangular clipping window and clip the objects in the scene against it.

2 Use the GLUT commands discussed in this chapter to set up a display window in which you will display a portion of the animated scene that you developed. More specifically, define a rectangle whose size is moderately smaller than the coordinate extents of all the objects in your scene. This rectangle will act as a clipping window against which you will employ the clipping algorithms you implemented in the previous exercise. The animation should be run continuously, but the objects in the scene should be clipped in each frame against the clipping window, and only this portion of the scene displayed in the display window. Additionally, add the ability to move the clipping window around the scene via keyboard input by using the directional arrows. Each keystroke should move the clipping window by a small amount in the appropriate direction. Run the animation using both the Sutherland-Hodgman and Weiler-Atherton algorithms and note any differences in performance.

https://hemanthrajhemu.github.io

Three-Dimensional Geometric Transformations

1 Three-Dimensional Translation
2 Three-Dimensional Rotation
3 Three-Dimensional Scaling
4 Composite Three-Dimensional Transformations
5 Other Three-Dimensional Transformations
6 Transformations between Three-Dimensional Coordinate Systems
7 Affine Transformations
8 OpenGL Geometric-Transformation Functions
9 OpenGL Three-Dimensional Geometric- Transformation Programming Examples
10 Summary

Methods for geometric transformations in three dimensions are extended from two-dimensional methods by including considerations for the z coordinate. In most cases, this extension is relatively straightforward. However, in some cases—particularly, rotation—the extension to three dimensions is less obvious.

When we discussed two-dimensional rotations in the xy plane, we needed to consider only rotations about axes that were perpendicular to the xy plane. In three-dimensional space, we can now select any spatial orientation for the rotation axis. Some graphics packages handle three-dimensional rotation as a composite of three rotations, one for each of the three Cartesian axes. Alternatively, we can set up general rotation equations, given the orientation of a rotation axis and the required rotation angle.

A three-dimensional position, expressed in homogeneous coordinates, is represented as a four-element column vector. Thus, each geometric transformation operator is now a 4×4 matrix, which

From Chapter 9 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers. Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

https://hemanthrajhemu.github.io

premultiplies a coordinate column vector. In addition, as in two dimensions, any sequence of transformations is represented as a single matrix, formed by concatenating the matrices for the individual transformations in the sequence. Each successive matrix in a transformation sequence is concatenated to the left of previous transformation matrices.

1 Three-Dimensional Translation

A position $\mathbf{P} = (x, y, z)$ in three-dimensional space is translated to a location $\mathbf{P}' = (x', y', z')$ by adding translation distances t_x, t_y, and t_z to the Cartesian coordinates of $\mathbf{P}$:

$$x' = x + t_x, \qquad y' = y + t_y, \qquad z' = z + t_z \tag{1}$$

Figure 1 illustrates three-dimensional point translation.

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

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2}$$

or

$$\mathbf{P}' = \mathbf{T} \cdot \mathbf{P} \tag{3}$$

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.

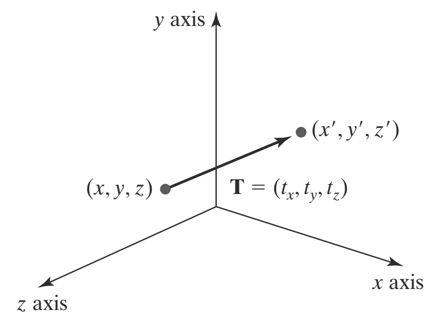


FIGURE 1
Moving a coordinate position with translation vector $\mathbf{T} = (t_x, t_y, t_z)$.

https://hemanthrajhemu.github.io

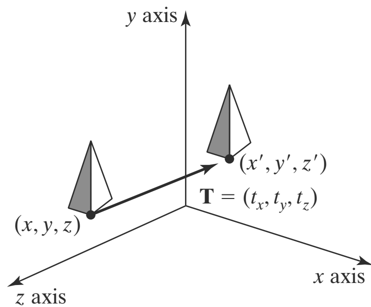


FIGURE 2
Shifting the position of a three-dimensional object using translation vector **T**.

```
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.

2 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. Therefore, we first consider the operations involved in coordinate-axis rotations, then we discuss the calculations needed for other rotation axes.

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 (Figure 3). This agrees with our earlier discussion of

https://hemanthrajhemu.github.io

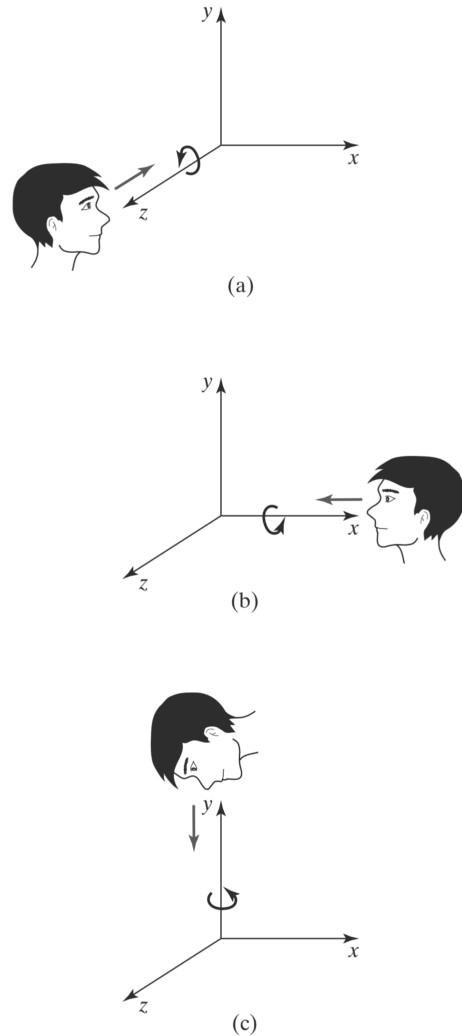


FIGURE 3
Positive rotations about a coordinate axis are counterclockwise, when looking along the positive half of the axis toward the origin.

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:

$$\begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \\ z' &= z \end{aligned} \tag{4}$$

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

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{5}$$

https://hemanthrajhemu.github.io

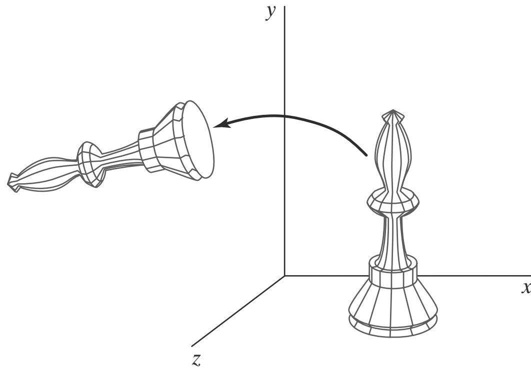


FIGURE 4
Rotation of an object about the z axis.

which we can write more compactly as

$$\mathbf{P}' = \mathbf{R}_z(\theta) \cdot \mathbf{P} \tag{6}$$

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 in Equations 4:

$$x \rightarrow y \rightarrow z \rightarrow x \tag{7}$$

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.

Substituting permutations 7 into Equations 4, we get the equations for an ***x*-axis rotation**:

$$\begin{aligned} y' &= y\cos\theta - z\sin\theta \\ z' &= y\sin\theta + z\cos\theta \\ x' &= x \end{aligned} \tag{8}$$

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

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

$$\begin{aligned} z' &= z\cos\theta - x\sin\theta \\ x' &= z\sin\theta + x\cos\theta \\ y' &= y \end{aligned} \tag{9}$$

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 $-\theta$.

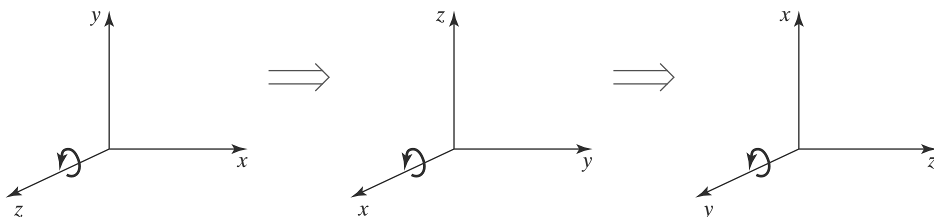


FIGURE 5
Cyclic permutation of the Cartesian-coordinate axes to produce the three sets of coordinate-axis rotation equations.

https://hemanthrajhemu.github.io

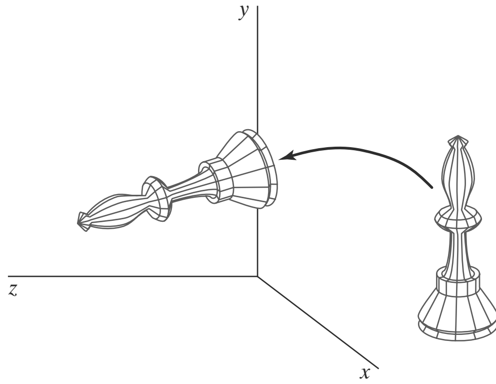


FIGURE 6
Rotation of an object about the x axis.

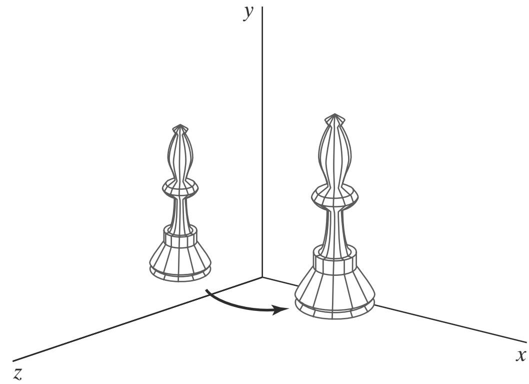


FIGURE 7
Rotation of an object about the y axis.

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

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

$$\mathbf{P}' = \mathbf{T}^{-1} \cdot \mathbf{R}_x(\theta) \cdot \mathbf{T} \cdot \mathbf{P} \tag{10}$$

where the composite rotation matrix for the transformation is

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_x(\theta) \cdot \mathbf{T} \tag{11}$$

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

https://hemanthrajhemu.github.io

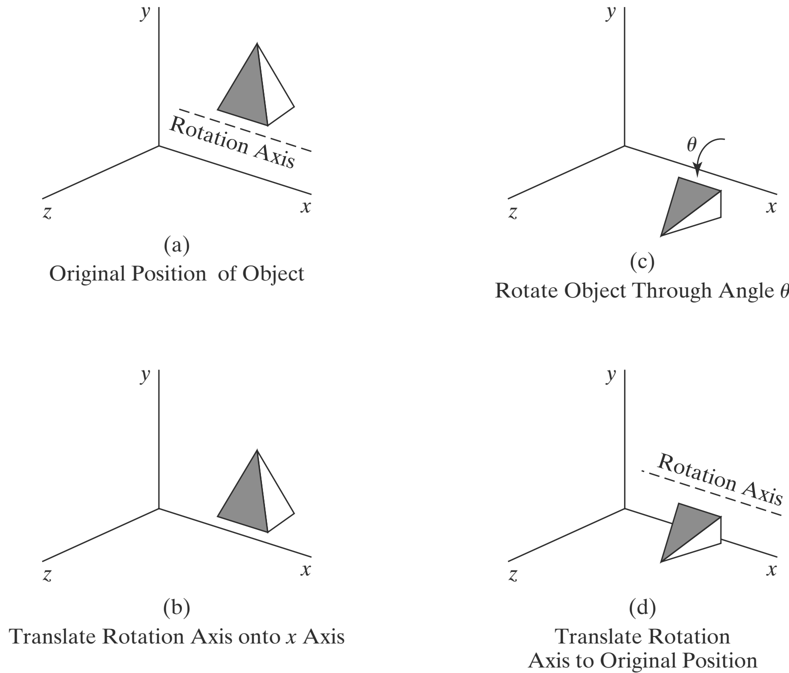


FIGURE 8
Sequence of transformations for rotating an object about an axis that is parallel to the x axis.

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).

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 $\mathbf{P}_2$ to $\mathbf{P}_1$. The components of the rotation-axis vector are then computed as

$$\begin{aligned}\mathbf{V} &= \mathbf{P}_2 - \mathbf{P}_1 \\ &= (x_2 - x_1, y_2 - y_1, z_2 - z_1)\end{aligned} \tag{12}$$

The unit rotation-axis vector $\mathbf{u}$ is

$$\mathbf{u} = \frac{\mathbf{V}}{|\mathbf{V}|} = (a, b, c) \tag{13}$$

https://hemanthrajhemu.github.io

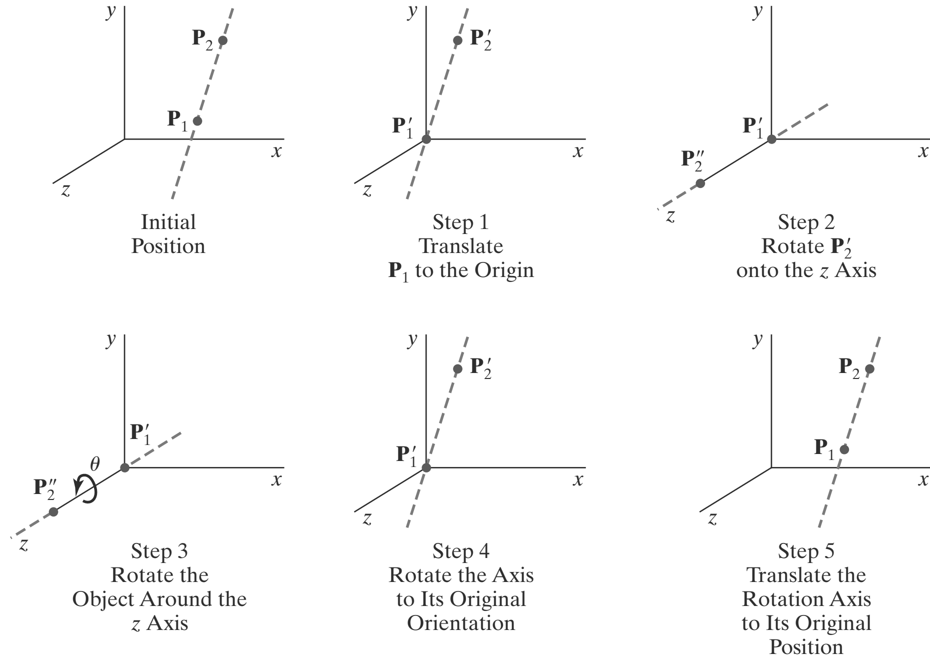


FIGURE 9
Five transformation steps for obtaining a composite matrix for rotation about an arbitrary axis, with the rotation axis projected onto the *z* axis.

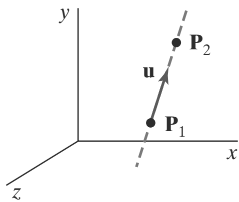


FIGURE 10
An axis of rotation (dashed line) defined with points $\mathbf{P}_1$ and $\mathbf{P}_2$. The direction for the unit axis vector **u** is determined by the specified rotation direction.

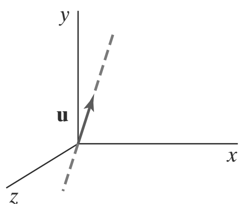


FIGURE 11
Translation of the rotation axis to the coordinate origin.

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 $\mathbf{P}_2$ to $\mathbf{P}_1$), then we would reverse axis vector **V** and unit vector **u** so that they point in the direction from $\mathbf{P}_2$ to $\mathbf{P}_1$.

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 $\mathbf{P}_2$ to $\mathbf{P}_1$ (Figure 10), we move the point $\mathbf{P}_1$ to the origin. (If the rotation had been specified in the opposite direction, we would move $\mathbf{P}_2$ to the origin.) This translation matrix is

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{15}$$

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

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**.

Because rotation calculations involve sine and cosine functions, we can use standard vector operations to obtain elements of the two rotation matrices. A vector dot product can be used to determine the cosine term, and a vector cross product can be used to calculate the sine term.

https://hemanthrajhemu.github.io

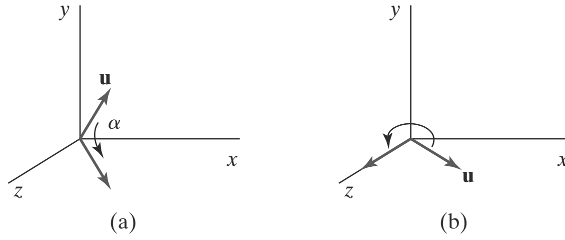


FIGURE 12
Unit vector **u** is rotated about the x axis to bring it into the xz plane (a), then it is rotated around the y axis to align it with the z axis (b).

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

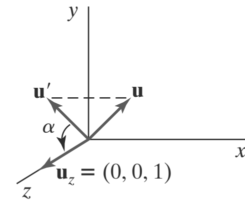


FIGURE 13
Rotation of **u** around the x axis into the xz plane is accomplished by rotating $\mathbf{u}'$ (the projection of **u** in the yz plane) through angle α onto the z axis.

$$\cos\alpha = \frac{\mathbf{u}' \cdot \mathbf{u}_z}{|\mathbf{u}'|\,|\mathbf{u}_z|} = \frac{c}{d} \tag{16}$$

where d is the magnitude of $\mathbf{u}'$:

$$d = \sqrt{b^2 + c^2} \tag{17}$$

Similarly, we can determine the sine of α from the cross-product of $\mathbf{u}'$ and $\mathbf{u}_z$. The coordinate-independent form of this cross-product is

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_x\,|\mathbf{u}'|\,|\mathbf{u}_z|\sin\alpha \tag{18}$$

and the Cartesian form for the cross-product gives us

$$\mathbf{u}' \times \mathbf{u}_z = \mathbf{u}_x \cdot b \tag{19}$$

Equating the right sides of Equations 18 and 19, and noting that $|\mathbf{u}_z| = 1$ and $|\mathbf{u}'| = d$, we have

$$d\sin\alpha = b$$

or

$$\sin\alpha = \frac{b}{d} \tag{20}$$

Now that we have determined the values for $\cos\alpha$ and $\sin\alpha$ 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:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & -\frac{b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{21}$$

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

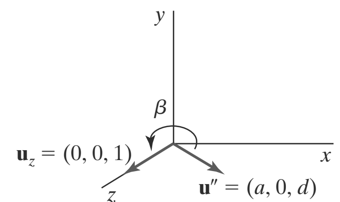


FIGURE 14
Rotation of unit vector $\mathbf{u}''$ (vector **u** after rotation into the xz plane) about the y axis. Positive rotation angle β aligns $\mathbf{u}''$ with vector $\mathbf{u}_z$.

https://hemanthrajhemu.github.io

the unit vector in the xz plane, resulting from the rotation about the x axis. This vector, labeled $\mathbf{u}''$, 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 $\mathbf{u}'$), because vector $\mathbf{u}'$ has been rotated onto the z axis. Also, the y component of $\mathbf{u}''$ 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 $\mathbf{u}''$ and $\mathbf{u}_z$. Thus,

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

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

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

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

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

With transformation matrices 15, 21, and 26, we have aligned the rotation axis with the positive z axis. The specified rotation angle θ can now be applied as a rotation about the z axis as follows:

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{27}$$

To complete the required rotation about the given axis, we need to transform the rotation axis back to its original position. This is done by applying the inverse of transformations 15, 21, and 26. The transformation matrix for rotation about an arbitrary axis can then be expressed as the composition of these seven individual transformations:

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_x^{-1}(\alpha) \cdot \mathbf{R}_y^{-1}(\beta) \cdot \mathbf{R}_z(\theta) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha) \cdot \mathbf{T} \tag{28}$$

A somewhat quicker, but perhaps less intuitive, method for obtaining the composite rotation matrix $\mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$ is to use the fact that the composite matrix for any sequence of three-dimensional rotations is of the form

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

The upper-left 3×3 submatrix of this matrix is orthogonal. This means that the rows (or the columns) of this submatrix form a set of orthogonal unit vectors that

https://hemanthrajhemu.github.io

are rotated by matrix **R** onto the x, y, and z axes, respectively:

$$\mathbf{R} \cdot \begin{bmatrix} r_{11} \\ r_{12} \\ r_{13} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \qquad \mathbf{R} \cdot \begin{bmatrix} r_{21} \\ r_{22} \\ r_{23} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}, \qquad \mathbf{R} \cdot \begin{bmatrix} r_{31} \\ r_{32} \\ r_{33} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \tag{30}$$

Therefore, we can set up a local coordinate system with one of its axes aligned on the rotation axis. Then the unit vectors for the three coordinate axes are used to construct the columns of the rotation matrix. Assuming that the rotation axis is not parallel to any coordinate axis, we could form the following set of local unit vectors (Figure 15).

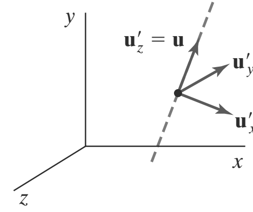


FIGURE 15
Local coordinate system for a rotation axis defined by unit vector **u**.

$$\begin{aligned} \mathbf{u}'_z &= \mathbf{u} \\ \mathbf{u}'_y &= \frac{\mathbf{u} \times \mathbf{u}_x}{|\mathbf{u} \times \mathbf{u}_x|} \\ \mathbf{u}'_x &= \mathbf{u}'_y \times \mathbf{u}'_z \end{aligned} \tag{31}$$

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

$$\begin{aligned} \mathbf{u}'_x &= (u'_{x1}, u'_{x2}, u'_{x3}) \\ \mathbf{u}'_y &= (u'_{y1}, u'_{y2}, u'_{y3}) \\ \mathbf{u}'_z &= (u'_{z1}, u'_{z2}, u'_{z3}) \end{aligned} \tag{32}$$

then the required composite matrix, which is equal to the product $\mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$, is

$$\mathbf{R} = \begin{bmatrix} u'_{x1} & u'_{x2} & u'_{x3} & 0 \\ u'_{y1} & u'_{y2} & u'_{y3} & 0 \\ u'_{z1} & u'_{z2} & u'_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{33}$$

This matrix transforms the unit vectors $\mathbf{u}'_x$, $\mathbf{u}'_y$, and $\mathbf{u}'_z$ onto the x, y, and z axes, respectively. This aligns the rotation axis with the z axis, because $\mathbf{u}'_z = \mathbf{u}$.

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:

$$s = \cos\frac{\theta}{2}, \qquad \mathbf{v} = \mathbf{u}\sin\frac{\theta}{2} \tag{34}$$

https://hemanthrajhemu.github.io

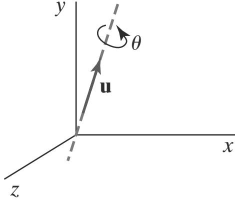


FIGURE 16
Unit quaternion parameters θ and **u** for rotation about a specified axis.

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 $\mathbf{p} = (x, y, z)$. The rotation of the point is then carried out with the quaternion operation

$$\mathbf{P}' = q\mathbf{P}q^{-1} \tag{35}$$

where $q^{-1} = (s, -\mathbf{v})$ is the inverse of the unit quaternion q with the scalar and vector parts given in Equations 34. This transformation produces the following new quaternion:

$$\mathbf{P}' = (0, \mathbf{p}') \tag{36}$$

The second term in this ordered pair is the rotated point position $\mathbf{p}'$, which is evaluated with vector dot and cross-products as

$$\mathbf{p}' = s^2\mathbf{p} + \mathbf{v}(\mathbf{p} \cdot \mathbf{v}) + 2s(\mathbf{v} \times \mathbf{p}) + \mathbf{v} \times (\mathbf{v} \times \mathbf{p}) \tag{37}$$

Values for parameters s and $\mathbf{v}$ are obtained from the expressions in 34. Many computer graphics systems use efficient hardware implementations of these vector calculations to perform rapid three-dimensional object rotations.

Transformation 35 is equivalent to rotation about an axis that passes through the coordinate origin. This is the same as the sequence of rotation transformations in Equation 28 that aligns the rotation axis with the z axis, rotates about z, and then returns the rotation axis to its original orientation at the coordinate origin.

We can evaluate the terms in Equation 37 using the definition for quaternion multiplication. Also, designating the components of the vector part of q as $\mathbf{v} = (a, b, c)$, we obtain the elements for the composite rotation matrix $\mathbf{R}_x^{-1}(\alpha) \cdot \mathbf{R}_y^{-1}(\beta) \cdot \mathbf{R}_z(\theta) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$ in a 3×3 form as

$$\mathbf{M}_R(\theta) = \begin{bmatrix} 1 - 2b^2 - 2c^2 & 2ab - 2sc & 2ac + 2sb \\ 2ab + 2sc & 1 - 2a^2 - 2c^2 & 2bc - 2sa \\ 2ac - 2sb & 2bc + 2sa & 1 - 2a^2 - 2b^2 \end{bmatrix} \tag{38}$$

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 38 as

$$\mathbf{M}_R(\theta) = \begin{bmatrix} u_x^2(1-\cos\theta) + \cos\theta & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & u_y^2(1-\cos\theta) + \cos\theta & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & u_z^2(1-\cos\theta) + \cos\theta \end{bmatrix} \tag{39}$$

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, corresponding to Equation 28, is

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{M}_R \cdot \mathbf{T} \tag{40}$$

https://hemanthrajhemu.github.io

For example, we can perform a rotation about the z axis by setting rotation-axis vector $\mathbf{u}$ to the unit z-axis vector (0, 0, 1). Substituting the components of this vector into Matrix 39, we get the 3×3 version of the z-axis rotation matrix $\mathbf{R}_z(\theta)$ in Equation 5. Similarly, substituting the unit-quaternion rotation values into Equation 35 produces the rotated coordinate values in Equations 4.

In the following code, we give examples of procedures that could be used to construct a three-dimensional rotation matrix. The quaternion representation in Equation 40 is used to set up the matrix elements for a general three-dimensional rotation.

```
class wcPt3D {
   public:
      GLfloat x, y, z;
};
typedef float Matrix4x4 [4][4];

Matrix4x4 matRot;

/* 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);
}

/* Premultiply matrix m1 by matrix m2, store result in m2. */
void matrix4x4PreMultiply (Matrix4x4 m1, Matrix4x4 m2)
{
   GLint row, col;
   Matrix4x4 matTemp;

   for (row = 0; row < 4; row++)
      for (col = 0; col < 4 ; col++)
         matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *
                            m2 [1][col] + m1 [row][2] * m2 [2][col] +
                            m1 [row][3] * m2 [3][col];
   for (row = 0; row < 4; row++)
      for (col = 0; col < 4; col++)
         m2 [row][col] = matTemp [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;
```


https://hemanthrajhemu.github.io

```
   /*  Concatenate translation matrix with matRot.  */
   matrix4x4PreMultiply (matTransl3D, matRot);
}

void rotate3D (wcPt3D p1, wcPt3D p2, GLfloat radianAngle)
{
   Matrix4x4 matQuaternionRot;

   GLfloat axisVectLength = sqrt ((p2.x - p1.x) * (p2.x - p1.x) +
                        (p2.y - p1.y) * (p2.y - p1.y) +
                        (p2.z - p1.z) * (p2.z - p1.z));
   GLfloat cosA = cos (radianAngle);
   GLfloat oneC = 1 - cosA;
   GLfloat sinA = sin (radianAngle);
   GLfloat ux = (p2.x - p1.x) / axisVectLength;
   GLfloat uy = (p2.y - p1.y) / axisVectLength;
   GLfloat uz = (p2.z - p1.z) / axisVectLength;

   /*  Set up translation matrix for moving p1 to origin.  */
   translate3D (-p1.x, -p1.y, -p1.z);

   /*  Initialize matQuaternionRot to identity matrix.  */
   matrix4x4SetIdentity (matQuaternionRot);

   matQuaternionRot [0][0] = ux*ux*oneC + cosA;
   matQuaternionRot [0][1] = ux*uy*oneC - uz*sinA;
   matQuaternionRot [0][2] = ux*uz*oneC + uy*sinA;
   matQuaternionRot [1][0] = uy*ux*oneC + uz*sinA;
   matQuaternionRot [1][1] = uy*uy*oneC + cosA;
   matQuaternionRot [1][2] = uy*uz*oneC - ux*sinA;
   matQuaternionRot [2][0] = uz*ux*oneC - uy*sinA;
   matQuaternionRot [2][1] = uz*uy*oneC + ux*sinA;
   matQuaternionRot [2][2] = uz*uz*oneC + cosA;

   /*  Combine matQuaternionRot with translation matrix.  */
   matrix4x4PreMultiply (matQuaternionRot, matRot);

   /*  Set up inverse matTransl3D and concatenate with
    *  product of previous two matrices.
    */
   translate3D (p1.x, p1.y, p1.z);
}

void displayFcn (void)
{
   /*  Input rotation parameters.  */

   /*  Initialize matRot to identity matrix:  */
   matrix4x4SetIdentity (matRot);

   /*  Pass rotation parameters to procedure rotate3D.  */

   /*  Display rotated object.  */
}
```


https://hemanthrajhemu.github.io

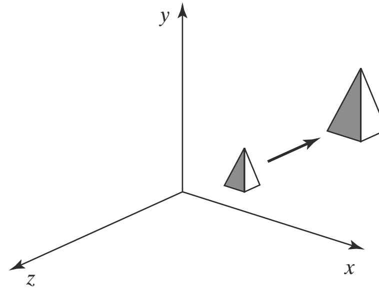


FIGURE 17
Doubling the size of an object with transformation 41 also moves the object farther from the origin.

3 Three-Dimensional Scaling

The matrix expression for the three-dimensional scaling transformation of a position $\mathbf{P} = (x, y, z)$ 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:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{41}$$

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

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P} \tag{42}$$

where scaling parameters s_x, s_y, and s_z are assigned any positive values. Explicit expressions for the scaling transformation relative to the origin are

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y, \qquad z' = z \cdot s_z \tag{43}$$

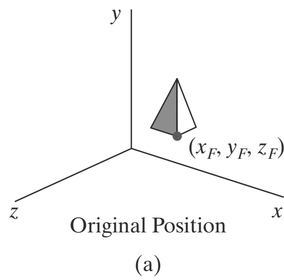
Scaling an object with transformation 41 changes the position of the object relative to the coordinate origin. A parameter value greater than 1 moves a point farther from the origin in the corresponding coordinate direction. Similarly, a parameter value less than 1 moves a point closer to the origin 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.

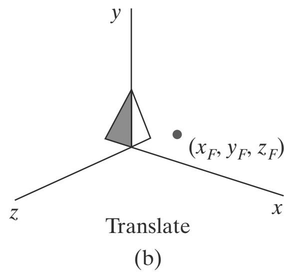
Because some graphics packages provide only a routine that scales relative to the coordinate origin, we can always construct a scaling transformation with respect to any selected *fixed position* (x_f, y_f, z_f) using the following transformation sequence:

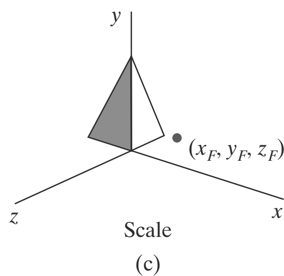
1. Translate the fixed point to the origin.
2. Apply the scaling transformation relative to the coordinate origin using Equation 41.
3. Translate the fixed point back to its original position.

This sequence of transformations is demonstrated in Figure 18. The matrix representation for an arbitrary fixed-point scaling can then be expressed as the

https://hemanthrajhemu.github.io







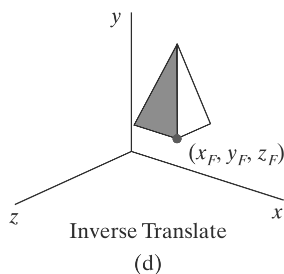


FIGURE 18
A sequence of transformations for scaling an object relative to a selected fixed point, using Equation 41.

concatenation of these translate-scale-translate transformations:

$$\mathbf{T}(x_f, y_f, z_f) \cdot \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{T}(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (44)$$

We can set up programming procedures for constructing a three-dimensional scaling matrix using either a translate-scale-translate sequence or a direct incorporation of the fixed-point coordinates. In the following code example, we demonstrate a direct construction of a three-dimensional scaling matrix relative to a selected fixed point using the calculations in Equation 44:

```
class wcPt3D
{
   private:
      GLfloat x, y, z;

   public:
   /*  Default Constructor:
    *  Initialize position as (0.0, 0.0, 0.0).
    */
   wcPt3D ( ) {
      x = y = z = 0.0;
   }

  setCoords (GLfloat xCoord, GLfloat yCoord, GLfloat zCoord) {
      x = xCoord;
      y = yCoord;
      z = zCoord;
   }

   GLfloat getx ( ) const {
      return x;
   }

   GLfloat gety ( ) const {
      return y;
   }

   GLfloat getz ( ) const {
      return z;
   }
};

typedef float Matrix4x4 [4][4];

void scale3D (GLfloat sx, GLfloat sy, GLfloat sz, wcPt3D fixedPt)
{
   Matrix4x4 matScale3D;

   /*  Initialize scaling matrix to identity.  */
   matrix4x4SetIdentity (matScale3D);
```


https://hemanthrajhemu.github.io

```
    matScale3D [0][0] = sx;
    matScale3D [0][3] = (1 - sx) * fixedPt.getx ( );
    matScale3D [1][1] = sy;
    matScale3D [1][3] = (1 - sy) * fixedPt.gety ( );
    matScale3D [2][2] = sz;
    matScale3D [2][3] = (1 - sz) * fixedPt.getz ( );
}
```

An inverse, three-dimensional scaling matrix is set up for either Equation 41 or Equation 44 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.

4 Composite Three-Dimensional Transformations

As with two-dimensional transformations, we form a composite three-dimensional transformation by multiplying the matrix representations for the individual operations in the transformation sequence. Any of the two-dimensional transformation sequences, such as scaling in noncoordinate directions, can be carried out in three-dimensional space.

We can implement a transformation sequence by concatenating the individual matrices from right to left or from left to right, depending on the order in which the matrix representations are specified. Of course, the rightmost term in a matrix product is always the first transformation to be applied to an object and the leftmost term is always the last transformation. We need to use this ordering for the matrix product because coordinate positions are represented as four-element column vectors, which are premultiplied by the composite 4×4 transformation matrix.

The following program provides example routines for constructing a three-dimensional composite transformation matrix. The three basic geometric transformations are combined in a selected order to produce a single composite matrix, which is initialized to the identity matrix. For this example, we first rotate, then scale, then translate. We choose a left-to-right evaluation of the composite matrix so that the transformations are called in the order that they are to be applied. Thus, as each matrix is constructed, it is concatenated on the left of the current composite matrix to form the updated product matrix.

```
class wcPt3D {
   public:
      GLfloat x, y, z;
};
typedef GLfloat Matrix4x4 [4][4];

Matrix4x4 matComposite;

/* Construct the 4 x 4 identity matrix. */
void matrix4x4SetIdentity (Matrix4x4 matIdent4x4)
```


https://hemanthrajhemu.github.io

```
{
   GLint row, col;

   for (row = 0; row < 4; row++)
      for (col = 0; col < 4 ; col++)
         matIdent4x4 [row][col] = (row == col);
}

/* Premultiply matrix m1 by matrix m2, store result in m2. */
void matrix4x4PreMultiply (Matrix4x4 m1, Matrix4x4 m2)
{
   GLint row, col;
   Matrix4x4 matTemp;

   for (row = 0; row < 4; row++)
      for (col = 0; col < 4 ; col++)
         matTemp [row][col] = m1 [row][0] * m2 [0][col] + m1 [row][1] *
                              m2 [1][col] + m1 [row][2] * m2 [2][col] +
                              m1 [row][3] * m2 [3][col];
   for (row = 0; row < 4; row++)
      for (col = 0; col < 4; col++)
         m2 [row][col] = matTemp [row][col];
}

/*  Procedure for generating 3-D translation matrix.  */
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;

   /*  Concatenate matTransl3D with composite matrix.  */
   matrix4x4PreMultiply (matTransl3D, matComposite);
}

/*  Procedure for generating a quaternion rotation matrix.  */
void rotate3D (wcPt3D p1, wcPt3D p2, GLfloat radianAngle)
{
   Matrix4x4 matQuatRot;

   float axisVectLength = sqrt ((p2.x - p1.x) * (p2.x - p1.x) +
                         (p2.y - p1.y) * (p2.y - p1.y) +
                         (p2.z - p1.z) * (p2.z - p1.z));
   float cosA = cosf (radianAngle);
   float oneC = 1 - cosA;
   float sinA = sinf (radianAngle);
   float ux = (p2.x - p1.x) / axisVectLength;
   float uy = (p2.y - p1.y) / axisVectLength;
   float uz = (p2.z - p1.z) / axisVectLength;

   /*  Set up translation matrix for moving p1 to origin,
```


https://hemanthrajhemu.github.io

```
     *  and concatenate translation matrix with matComposite.
     */
   translate3D (-p1.x, -p1.y, -p1.z);

   /*  Initialize matQuatRot to identity matrix.  */
   matrix4x4SetIdentity (matQuatRot);

   matQuatRot [0][0] = ux*ux*oneC + cosA;
   matQuatRot [0][1] = ux*uy*oneC - uz*sinA;
   matQuatRot [0][2] = ux*uz*oneC + uy*sinA;
   matQuatRot [1][0] = uy*ux*oneC + uz*sinA;
   matQuatRot [1][1] = uy*uy*oneC + cosA;
   matQuatRot [1][2] = uy*uz*oneC - ux*sinA;
   matQuatRot [2][0] = uz*ux*oneC - uy*sinA;
   matQuatRot [2][1] = uz*uy*oneC + ux*sinA;
   matQuatRot [2][2] = uz*uz*oneC + cosA;

   /*  Concatenate matQuatRot with composite matrix.  */
   matrix4x4PreMultiply (matQuatRot, matComposite);

   /*  Construct inverse translation matrix for p1 and
    *  concatenate with composite matrix.
    */
   translate3D (p1.x, p1.y, p1.z);
}

/*  Procedure for generating a 3-D scaling matrix.  */
void scale3D (Gfloat sx, GLfloat sy, GLfloat sz, wcPt3D fixedPt)
{
   Matrix4x4 matScale3D;

   /*  Initialize scaling matrix to identity.  */
   matrix4x4SetIdentity (matScale3D);

   matScale3D [0][0] = sx;
   matScale3D [0][3] = (1 - sx) * fixedPt.x;
   matScale3D [1][1] = sy;
   matScale3D [1][3] = (1 - sy) * fixedPt.y;
   matScale3D [2][2] = sz;
   matScale3D [2][3] = (1 - sz) * fixedPt.z;

   /*  Concatenate matScale3D with composite matrix.  */
   matrix4x4PreMultiply (matScale3D, matComposite);
}

void displayFcn (void)
{
   /*  Input object description.  */
   /*  Input translation, rotation, and scaling parameters.  */

   /* Set up 3-D viewing-transformation routines. */

   /*  Initialize matComposite to identity matrix:  */
   matrix4x4SetIdentity (matComposite);

   /*  Invoke transformation routines in the order they
```


https://hemanthrajhemu.github.io

```
    *  are to be applied:
    */
   rotate3D (p1, p2, radianAngle);  //  First transformation: Rotate.
   scale3D (sx, sy, sz, fixedPt);   //  Second transformation: Scale.
   translate3D (tx, ty, tz);        //  Final transformation: Translate.

   /*  Call routines for displaying transformed objects.  */
}
```

5 Other Three-Dimensional Transformations

In addition to translation, rotation, and scaling, the other transformations discussed for two-dimensional applications are also useful in many three-dimensional situations. These additional transformations include reflection, shear, and transformations between coordinate-reference frames.

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. 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

$$M_{z\text{reflect}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{45}$$

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.

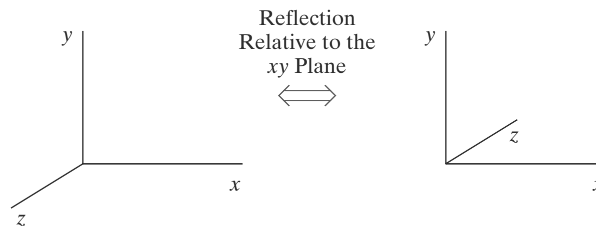


FIGURE 19
Conversion of coordinate specifications between a right-handed and a left-handed system can be carried out with the reflection transformation 45.

https://hemanthrajhemu.github.io

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 applications, 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:

$$M_{z\text{shear}} = \begin{bmatrix} 1 & 0 & \text{sh}_{zx} & -\text{sh}_{zx} \cdot z_{\text{ref}} \\ 0 & 1 & \text{sh}_{zy} & -\text{sh}_{zy} \cdot z_{\text{ref}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{46}$$

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 z_{ref}, while leaving the z coordinate unchanged. Plane areas that are perpendicular to the z axis are thus shifted by an amount equal to $z - z_{\text{ref}}$. An example of the effect of this shearing matrix on a unit cube is shown in Figure 20 for shearing values $\text{sh}_{zx} = \text{sh}_{zy} = 1$ and a reference position $z_{\text{ref}} = 0$. 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.

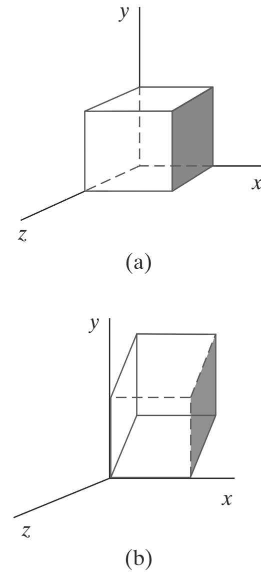


FIGURE 20
A unit cube (a) is sheared relative to the origin (b) by Matrix 46, with $\text{sh}_{zx} = \text{sh}_{zy} = 1$.

6 Transformations between Three-Dimensional Coordinate Systems

Coordinate-system transformations are employed in computer-graphics packages to construct (model) scenes and to implement viewing routines for both two-dimensional and three-dimensional applications. A transformation matrix for transferring a two-dimensional scene description from one coordinate system to another is constructed with operations for superimposing the coordinate axes of the two systems. The same procedures apply to three-dimensional scene transfers.

We again consider only Cartesian reference frames, and we assume that an $x'y'z'$ system is defined with respect to an xyz system. To transfer the xyz coordinate descriptions to the $x'y'z'$ system, we first set up a translation that brings the $x'y'z'$ coordinate origin to the position of the xyz origin. This is followed by a sequence of rotations that align corresponding coordinate axes. If different scales are used in the two coordinate systems, a scaling transformation may also be necessary to compensate for the differences in coordinate intervals.

Figure 21 shows an $x'y'z'$ coordinate system with origin (x_0, y_0, z_0) and unit axis vectors defined relative to an xyz reference frame. The coordinate origin of

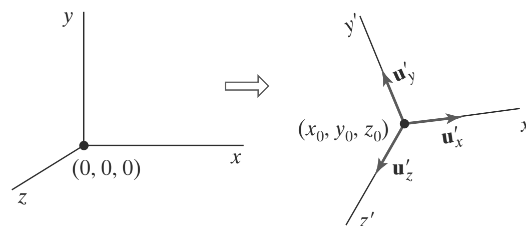


FIGURE 21
An $x'y'z'$ coordinate system defined within an xyz system. A scene description is transferred to the new coordinate reference using a transformation sequence that superimposes the $x'y'z'$ frame on the xyz axes.

https://hemanthrajhemu.github.io

the $x'y'z'$ system is brought into coincidence with the xyz origin using the translation matrix $\mathbf{T}(-x_0, -y_0, -z_0)$. Also, we can use the unit axis vectors to form the coordinate-axis rotation matrix

$$\mathbf{R} = \begin{bmatrix} u'_{x1} & u'_{x2} & u'_{x3} & 0 \\ u'_{y1} & u'_{y2} & u'_{y3} & 0 \\ u'_{z1} & u'_{z2} & u'_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{47}$$

which transforms unit vectors $\mathbf{u}'_x$, $\mathbf{u}'_y$, and $\mathbf{u}'_z$ onto the x, y, and z axes, respectively. The complete coordinate transformation sequence is then given by the composite matrix $\mathbf{R} \cdot \mathbf{T}$. This matrix correctly transforms coordinate descriptions from one Cartesian system to another, even if one system is left-handed and the other is right-handed.

7 Affine Transformations

A coordinate transformation of the form

$$\begin{aligned} x' &= a_{xx}x + a_{xy}y + a_{xz}z + b_x \\ y' &= a_{yx}x + a_{yy}y + a_{yz}z + b_y \\ z' &= a_{zx}x + a_{zy}y + a_{zz}z + b_z \end{aligned} \tag{48}$$

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 a_{ij} and b_k 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.

8 OpenGL Geometric-Transformation Functions

The basic OpenGL functions for performing geometric transformations are the same functions used to perform transformations in three dimensions. For convenience, those functions are listed in Table 1 at the end of the chapter.

OpenGL Matrix Stacks

You are already familiar with the OpenGL modelview mode. This mode is selected with the `glMatrixMode` routine and is used to select the modelview

https://hemanthrajhemu.github.io

composite transformation matrix as the target of subsequent OpenGL transformation calls.

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 4×4 composite matrix that combines the viewing transformations and the various geometric transformations that we want to apply to a scene. In some cases, we may want to create multiple views and transformation sequences, and then save the composite matrix for each. Therefore, OpenGL supports a modelview stack depth of at least 32, and some implementations may allow more than 32 matrices to be saved on the modelview stack. 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`. The other three matrix modes have a minimum stack depth of 2, and we can determine the maximum available depth of each for a particular implementation using one of the following OpenGL symbolic constants: `GL_MAX_PROJECTION_STACK_DEPTH`, `GL_MAX_TEXTURE_STACK_DEPTH`, or `GL_MAX_COLOR_STACK_DEPTH`.

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

```
glGetIntegerv (GL_MODELVIEW_STACK_DEPTH, numMats);
```

Initially, the modelview stack contains only the identity matrix, so the value 1 is returned by this function if we issue the query before any stack processing has occurred. Similar symbolic constants are available for determining the number of matrices currently in the other three stacks.

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, particularly when the stack functions are implemented in hardware. For example, a hardware implementation can copy multiple matrix elements simultaneously. And we can maintain an identity matrix on the stack, so that initializations of the current matrix can be performed faster than by using repeated calls to `glLoadIdentity`.

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

https://hemanthrajhemu.github.io

9 OpenGL Three-Dimensional Geometric-Transformation Programming Examples

Usually, it is more efficient to use the stack-processing functions than to use the matrix-manipulation functions. This is particularly true when we want to make several changes in the viewing or geometric transformations. In the following code, we perform rectangle transformations using stack processing instead of the `glLoadIdentity` function:

```
glMatrixMode (GL_MODELVIEW);

glColor3f (0.0, 0.0, 1.0);          // Set current color to blue.
glRecti (50, 100, 200, 150);        // Display blue rectangle.

glPushMatrix ( );                   // Make copy of identity (top) matrix.
glColor3f (1.0, 0.0, 0.0);          // Set current color to red.

glTranslatef (-200.0, -50.0, 0.0); // Set translation parameters.
glRecti (50, 100, 200, 150);        // Display red, translated rectangle.

glPopMatrix ( );                    // Throw away the translation matrix.
glPushMatrix ( );                   // Make copy of identity (top) matrix.

glRotatef (90.0, 0.0, 0.0, 1.0);    // Set 90-deg. rotation about z axis.
glRecti (50, 100, 200, 150);        // Display red, rotated rectangle.

glPopMatrix ( );                    // Throw away the rotation matrix.
glScalef (-0.5, 1.0, 1.0);          // Set scale-reflection parameters.
glRecti (50, 100, 200, 150);        // Display red, transformed rectangle.
```

For our next geometric-transformation programming example, we give an OpenGL version of the three-dimensional, composite-transformation code in Section 4. Because OpenGL postmultiplies transformation matrices as they are called, we must now invoke the transformations in the opposite order from which they are to be applied. Thus, each subsequent transformation call concatenates the designated transformation matrix on the right of the composite matrix. Because we have not yet explored the three-dimensional OpenGL viewing routines, this program could be completed using two-dimensional OpenGL viewing operations and applying the geometric transformations to objects in the xy plane.

```
class wcPt3D {
   public:
      GLfloat x, y, z;
};

/* Procedure for generating a matrix for rotation about
```


https://hemanthrajhemu.github.io

```
 * an axis defined with points p1 and p2.
 */
void rotate3D (wcPt3D p1, wcPt3D p2, GLfloat thetaDegrees)
{
   /*  Set up components for rotation-axis vector.  */
   float vx = (p2.x - p1.x);
   float vy = (p2.y - p1.y);
   float vz = (p2.z - p1.z);

   /* Specify translate-rotate-translate sequence in reverse order: */
   glTranslatef (p1.x, p1.y, p1.z); // Move p1 back to original position.
   /*  Rotate about axis through origin:  */
   glRotatef (thetaDegrees, vx, vy, vz);
   glTranslatef (-p1.x, -p1.y, -p1.z);  // Translate p1 to origin.
}

/*  Procedure for generating a matrix for a scaling
 *  transformation with respect to an arbitrary fixed point.
 */
void scale3D (GLfloat sx, GLfloat sy, GLfloat sz, wcPt3D fixedPt)
{
   /* Specify translate-scale-translate sequence in reverse order: */
   /* (3) Translate fixed point back to original position: */
   glTranslatef (fixedPt.x, fixedPt.y, fixedPt.z);
   glScalef (sx, sy, sz);      // (2) Scale with respect to origin.
   /* (1) Translate fixed point to coordinate origin: */
   glTranslatef (-fixedPt.x, -fixedPt.y, -fixedPt.z);
}

void displayFcn (void)
{
   /*  Input object description.  */
   /*  Set up 3D viewing-transformation routines. */
   /*  Display object.  */

   glMatrixMode (GL_MODELVIEW);

   /* Input translation parameters tx, ty, tz. */
   /* Input the defining points, p1 and p2, for the rotation axis. */
   /* Input rotation angle in degrees. */
   /* Input scaling parameters: sx, sy, sz, and fixedPt. */

   /*  Invoke geometric transformations in reverse order:  */
   glTranslatef (tx, ty, tz);     // Final transformation: Translate.
   scale3D (sx, sy, sz, fixedPt);    // Second transformation: Scale.
   rotate3D (p1, p2, thetaDegrees);  // First transformation: Rotate.

   /*  Call routines for displaying transformed objects.  */
}
```

10 Summary

We can express three-dimensional transformations as 4×4 matrix operators, so that sequences of transformations can be concatenated into a single composite

https://hemanthrajhemu.github.io

TABLE 1

Summary of OpenGL Geometric Transformation Functions

Function	Description
`glTranslate*`	Specifies translation parameters.
`glRotate*`	Specifies parameters for rotation about any axis through the origin.
`glScale*`	Specifies scaling parameters with respect to coordinate origin.
`glMatrixMode`	Specifies current matrix for geometric-viewing transformations, projection transformations, texture transformations, or color transformations.
`glLoadIdentity`	Sets current matrix to identity.
`glLoadMatrix*` (elems);	Sets elements of current matrix.
`glMultMatrix*` (elems);	Postmultiplies the current matrix by the specified matrix.
`glGetIntegerv`	Gets max stack depth or current number of matrices in the stack for the selected matrix mode.
`glPushMatrix`	Copies the top matrix in the stack and store copy in the second stack position.
`glPopMatrix`	Erases the top matrix in the stack and moves the second matrix to the top of the stack.
`glPixelZoom`	Specifies two-dimensional scaling parameters for raster operations.

matrix to allow efficient application of multiple transformations. We use a four-element column matrix (vector) representation for three-dimensional coordinate points, representing them using a homogeneous coordinate representation.

We can create composite transformations through matrix multiplications of translation, rotation, scaling, and other transformations. In general, matrix multiplications are not commutative. The upper-left 3×3 submatrix of a rigid-body transformation is an orthogonal matrix. Thus, rotation matrices can be formed by setting the upper-left, 3×3 submatrix equal to the elements of two orthogonal unit vectors. When the angle is small, we can reduce rotation computations by using first-order approximations for the sine and cosine functions. Over many rotational steps, however, the approximation error can accumulate to a significant value.

Transformations between Cartesian coordinate systems in three dimensions are accomplished with a sequence of translate-rotate transformations that brings the two systems into coincidence. We specify the coordinate origin and axis vectors for one reference frame relative to the original coordinate reference frame. The transfer of object descriptions from the original coordinate system to the second system is calculated as the matrix product of a translation that moves the new origin to the old coordinate origin and a rotation to align the two sets of axes. The rotation needed to align the two frames can be obtained from the orthonormal set of axis vectors for the new system.

https://hemanthrajhemu.github.io

The OpenGL library provides functions for applying individual translate, rotate, and scale transformations to coordinate positions. Each function generates a matrix that is premultiplied by the modelview matrix. Transformation matrices are applied to subsequently defined objects. In addition to accumulating transformation sequences in the modelview matrix, we can set this matrix to the identity or some other matrix, and can also form products with the modelview matrix and any specified matrices. All matrices are stored in stacks, and OpenGL maintains four stacks for the various types of transformations that we use in graphics applications. We can use an OpenGL query function to determine the current stack size or the maximum allowable stack depth for a system. Two stack-processing routines are available: one for copying the top matrix in a stack to the second position, and one for removing the top matrix.

Table 1 summarizes the OpenGL geometric-transformation functions and matrix routines discussed in this chapter.

REFERENCES

For additional techniques involving matrices and geometric transformations, see Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995). Discussions of homogeneous coordinates in computer graphics can be found in Blinn and Newell (1978) and in Blinn (1993, 1996, and 1998).

Additional programming examples using OpenGL geometric-transformation functions are given in Woo, et al. (1999). Programming examples for the OpenGL geometric-transformation functions are also available at Nate Robins's tutorial website: http://www.xmission.com/~nate/opengl.html. Finally, a complete listing of OpenGL geometric-transformation functions is provided in Shreiner (2000).

EXERCISES

1 Show that rotation matrix 33 is equal to the composite matrix $\mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha)$.

2 By evaluating the terms in Equation 37, derive the elements for the general rotation matrix given in Equation 38.

3 Prove that the quaternion rotation matrix 38 reduces to the matrix representation in Equation 5 when the rotation axis is the coordinate z axis.

4 Prove that Equation 40 is equivalent to the general rotation transformation given in Equation 28.

5 Using trigonometric identities, derive the elements of the quaternion-rotation matrix 39 from 38.

6 Develop a procedure for animating a three-dimensional object by incrementally rotating it about any specified axis. Use appropriate approximations to the trigonometric equations to speed up the calculations, and reset the object to its initial position after each complete revolution about the axis.

7 Derive the three-dimensional transformation matrix for scaling an object by a scaling factor s in a direction defined by the direction cosines α, β, and γ.

8 Develop a routine to reflect a three-dimensional object about an arbitrarily selected plane.

9 Write a procedure to shear a three-dimensional object with respect to any specified axis, using input values for the shearing parameters.

10 Develop a procedure for converting an object definition in one three-dimensional coordinate reference to any other coordinate system defined relative to the first system.

11 Develop a routine to scale an object by a given factor in each dimension relative to a given point contained within the object.

https://hemanthrajhemu.github.io

12 Write a program to perform a series of transformations on a 30×30 square whose centroid lies at $(-20, -20, 0)$ and that is contained in the xy plane. Use three-dimensional OpenGL matrix operations to perform the transformations. The square should first be reflected in the x axis, then rotated counterclockwise by $45°$ about its center, then sheared in the x direction by a value of 2.

13 Modify the program from the previous exercise so that the transformation sequence can be applied to any two-dimensional polygon, with vertices specified as user input.

14 Modify the example program in the previous exercise so that the order of the geometric transformation sequence can be specified as user input.

15 Modify the example program from the previous exercise so that the geometric transformation parameters are specified as user input.

IN MORE DEPTH

1 You have not yet been exposed to the material necessary to construct three-dimensional representations of the objects in your application, so you can instead embed the two-dimensional polygonal approximations to those objects in a three dimensional scene and perform three-dimensional transformations on those approximations using the techniques in this chapter. In this exercise, you will set up a set of transformations to produce an animation. Define the three-dimensional transformation matrices to do this using homogeneous coordinate representations. If two or more objects act as a single "unit" in certain behaviors that are easier to model in terms of relative positions, you can use the techniques in Section 6 to convert the local transformations of the objects relative to each other (in their own coordinate frame) into transformations in the world coordinate frame.

2 Use the matrices you designed in the previous exercise to produce an animation. You should employ the OpenGL matrix operations for three-dimensional transformations and have the matrices produce small changes in position for each of the objects in the scene. Since you haven t yet covered the material necessary for generating views of a threedimensional scene, simply display the animation using a two-dimensional orthogonal projection, with all of the polygons in the scene being contained in the xy plane. The transformations themselves, however, are still three-dimensional.

https://hemanthrajhemu.github.io

Three-Dimensional Viewing

1 Overview of Three-Dimensional Viewing Concepts
2 The Three-Dimensional Viewing Pipeline
3 Three-Dimensional Viewing-Coordinate Parameters
4 Transformation from World to Viewing Coordinates
5 Projection Transformations
6 Orthogonal Projections
7 Oblique Parallel Projections
8 Perspective Projections
9 The Viewport Transformation and Three-Dimensional Screen Coordinates
10 OpenGL Three-Dimensional Viewing Functions
11 Three-Dimensional Clipping Algorithms
12 OpenGL Optional Clipping Planes
13 Summary

For two-dimensional graphics applications, viewing operations transfer positions from the world-coordinate plane to pixel positions in the plane of the output device. Using the rectangular boundaries for the clipping window and the viewport, a two-dimensional package clips a scene and maps it to device coordinates. Three-dimensional viewing operations, however, are more involved, because we now have many more choices as to how we can construct a scene and how we can generate views of the scene on an output device.

From Chapter 10 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.
Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

https://hemanthrajhemu.github.io

1 Overview of Three-Dimensional Viewing Concepts

When we model a three-dimensional scene, each object in the scene is typically defined with a set of surfaces that form a closed boundary around the object interior. And, for some applications, we may need also to specify information about the interior structure of an object. In addition to procedures that generate views of the surface features of an object, graphics packages sometimes provide routines for displaying internal components or cross-sectional views of a solid object. Viewing functions process the object descriptions through a set of procedures that ultimately project a specified view of the objects onto the surface of a display device. Many processes in three-dimensional viewing, such as the clipping routines, are similar to those in the two-dimensional viewing pipeline. But three-dimensional viewing involves some tasks that are not present in two-dimensional viewing. For example, projection routines are needed to transfer the scene to a view on a planar surface, visible parts of a scene must be identified, and, for a realistic display, lighting effects and surface characteristics must be taken into account.

Viewing a Three-Dimensional Scene

To obtain a display of a three-dimensional world-coordinate scene, we first set up a coordinate reference for the viewing, or “camera,” parameters. This coordinate reference defines the position and orientation for a *view plane* (or *projection plane*) that corresponds to a camera film plane (Figure 1). Object descriptions are then transferred to the viewing reference coordinates and projected onto the view plane. We can generate a view of an object on the output device in wireframe (outline) form, or we can apply lighting and surface-rendering techniques to obtain a realistic shading of the visible surfaces.

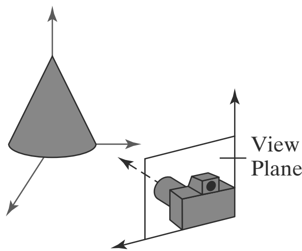


FIGURE 1
Coordinate reference for obtaining a selected view of a three-dimensional scene.

Projections

Unlike a camera picture, we can choose different methods for projecting a scene onto the view plane. One method for getting the description of a solid object onto a view plane is to project points on the object surface along parallel lines. This technique, called *parallel projection,* is used in engineering and architectural drawings to represent an object with a set of views that show accurate dimensions of the object, as in Figure 2.

Another method for generating a view of a three-dimensional scene is to project points to the view plane along converging paths. This process, called a *perspective projection,* causes objects farther from the viewing position to be displayed smaller than objects of the same size that are nearer to the viewing position. A scene that is generated using a perspective projection appears more realistic, because this is the way that our eyes and a camera lens form images. Parallel lines along the viewing direction appear to converge to a distant point in the background, and objects in the background appear to be smaller than objects in the foreground.

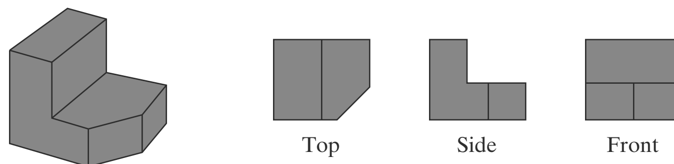


FIGURE 2
Three parallel-projection views of an object, showing relative proportions from different viewing positions.

https://hemanthrajhemu.github.io

Depth Cueing

With few exceptions, depth information is important in a three-dimensional scene so that we can easily identify, for a particular viewing direction, which is the front and which is the back of each displayed object. Figure 3 illustrates the ambiguity that can result when a wire-frame object is displayed without depth information. There are several ways in which we can include depth information in the two-dimensional representation of solid objects.

A simple method for indicating depth with wire-frame displays is to vary the brightness of line segments according to their distances from the viewing position. Figure 4 shows a wire-frame object displayed with *depth cueing*. The lines closest to the viewing position are displayed with the highest intensity, and lines farther away are displayed with decreasing intensities. Depth cueing is applied by choosing a maximum and a minimum intensity value and a range of distances over which the intensity is to vary.

Another application of depth cuing is modeling the effect of the atmosphere on the perceived intensity of objects. More distant objects appear dimmer to us than nearer objects due to light scattering by dust particles, haze, and smoke. Some atmospheric effects can even change the perceived color of an object, and we can model these effects with depth cueing.

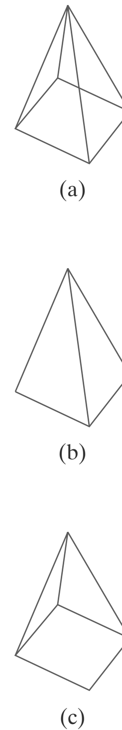


FIGURE 3
The wire-frame representation of the pyramid in (a) contains no depth information to indicate whether the viewing direction is (b) downward from a position above the apex or (c) upward from a position below the base.

Identifying Visible Lines and Surfaces

We can also clarify depth relationships in a wire-frame display using techniques other than depth cueing. One approach is simply to highlight the visible lines or to display them in a different color. Another technique, commonly used for engineering drawings, is to display the nonvisible lines as dashed lines. Or we could remove the nonvisible lines from the display, as in Figures 3(b) and 3(c). But removing the hidden lines also removes information about the shape of the back surfaces of an object, and wire-frame representations are generally used to get an indication of an object's overall appearance, front and back.

When a realistic view of a scene is to be produced, back parts of the objects are completely eliminated so that only the visible surfaces are displayed. In this case, surface-rendering procedures are applied so that screen pixels contain only the color patterns for the front surfaces.

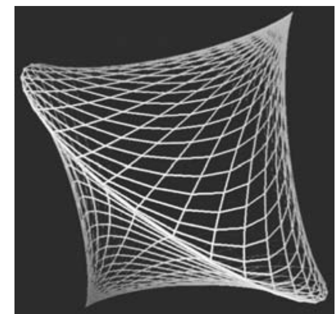


FIGURE 4
A wire-frame object displayed with depth cueing, so that the brightness of lines decreases from the front of the object to the back.

Surface Rendering

Added realism is attained in displays by rendering object surfaces using the lighting conditions in the scene and the assigned surface characteristics. We set the lighting conditions by specifying the color and location of the light sources, and we can also set background illumination effects. Surface properties of objects include whether a surface is transparent or opaque and whether the surface is smooth or rough. We set values for parameters to model surfaces such as glass, plastic, wood-grain patterns, and the bumpy appearance of an orange. In Color Plate 9 surface-rendering methods are combined with perspective and visible-surface identification to generate a degree of realism in a displayed scene.

Exploded and Cutaway Views

Many graphics packages allow objects to be defined as hierarchical structures, so that internal details can be stored. Exploded and cutaway views of such objects can then be used to show the internal structure and relationship of the object parts. An alternative to exploding an object into its component parts is a cutaway view, which removes part of the visible surfaces to show internal structure.

https://hemanthrajhemu.github.io

Three-Dimensional and Stereoscopic Viewing

Other methods for adding a sense of realism to a computer-generated scene include three-dimensional displays and stereoscopic views. Three-dimensional views can be obtained by reflecting a raster image from a vibrating, flexible mirror. The vibrations of the mirror are synchronized with the display of the scene on the cathode ray tube (CRT). As the mirror vibrates, the focal length varies so that each point in the scene is reflected to a spatial position corresponding to its depth.

Stereoscopic devices present two views of a scene: one for the left eye and the other for the right eye. The viewing positions correspond to the eye positions of the viewer. These two views are typically displayed on alternate refresh cycles of a raster monitor. When we view the monitor through special glasses that alternately darken first one lens and then the other, in synchronization with the monitor refresh cycles, we see the scene displayed with a three-dimensional effect.

2 The Three-Dimensional Viewing Pipeline

Procedures for generating a computer-graphics view of a three-dimensional scene are somewhat analogous to the processes involved in taking a photograph. First of all, we need to choose a viewing position corresponding to where we would place a camera. We choose the viewing position according to whether we want to display a front, back, side, top, or bottom view of the scene. We could also pick a position in the middle of a group of objects or even inside a single object, such as a building or a molecule. Then we must decide on the camera orientation (Figure 5). Which way do we want to point the camera from the viewing position, and how should we rotate it around the line of sight to set the "up" direction for the picture? Finally, when we snap the shutter, the scene is cropped to the size of a selected clipping window, which corresponds to the aperture or lens type of a camera, and light from the visible surfaces is projected onto the camera film.

We need to keep in mind, however, that the camera analogy can be carried only so far, because we have more flexibility and many more options for generating views of a scene with a computer-graphics program than we do with a real camera. We can choose to use either a parallel projection or a perspective projection, we can selectively eliminate parts of a scene along the line of sight, we can move the projection plane away from the "camera" position, and we can even get a picture of objects in back of our synthetic camera.

Some of the viewing operations for a three-dimensional scene are the same as, or similar to, those used in the two-dimensional viewing pipeline. A two-dimensional viewport is used to position a projected view of the threedimensional scene on the output device, and a two-dimensional clipping window is used to

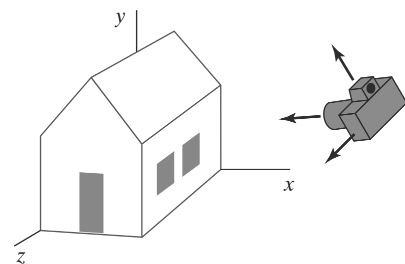


FIGURE 5
Photographing a scene involves selection of the camera position and orientation.

https://hemanthrajhemu.github.io

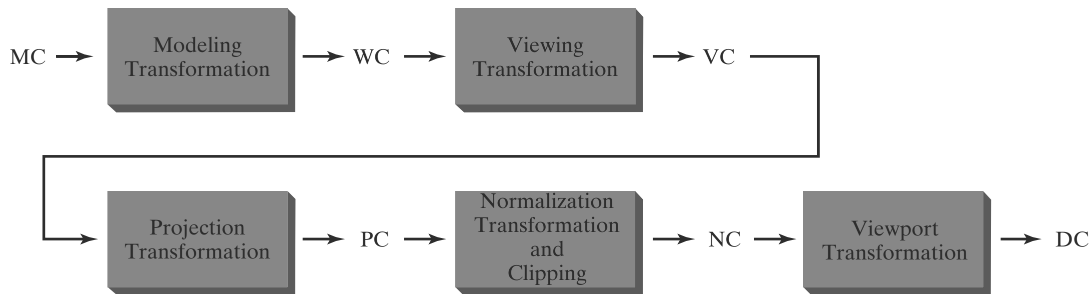


FIGURE 10-6
General three-dimensional transformation pipeline, from modeling coordinates (MC) to world coordinates (WC) to viewing coordinates (VC) to projection coordinates (PC) to normalized coordinates (NC) and, ultimately, to device coordinates (DC).

select a view that is to be mapped to the viewport. In addition, we set up a display window in screen coordinates, just as we do in a two-dimensional application. Clipping windows, viewports, and display windows are usually specified as rectangles with their edges parallel to the coordinate axes. In three-dimensional viewing, however, the clipping window is positioned on a selected view plane, and scenes are clipped against an enclosing volume of space, which is defined by a set of *clipping planes*. The viewing position, view plane, clipping window, and clipping planes are all specified within the viewing-coordinate reference frame.

Figure 6 shows the general processing steps for creating and transforming a three-dimensional scene to device coordinates. Once the scene has been modeled in world coordinates, a viewing-coordinate system is selected and the description of the scene is converted to viewing coordinates. The viewing coordinate system defines the viewing parameters, including the position and orientation of the projection plane (view plane), which we can think of as the camera film plane. A two-dimensional clipping window, corresponding to a selected camera lens, is defined on the projection plane, and a three-dimensional clipping region is established. This clipping region is called the **view volume,** and its shape and size depends on the dimensions of the clipping window, the type of projection we choose, and the selected limiting positions along the viewing direction. Projection operations are performed to convert the viewing-coordinate description of the scene to coordinate positions on the projection plane. Objects are mapped to normalized coordinates, and all parts of the scene outside the view volume are clipped off. The clipping operations can be applied after all device-independent coordinate transformations (from world coordinates to normalized coordinates) are completed. In this way, the coordinate transformations can be concatenated for maximum efficiency.

As in two-dimensional viewing, the viewport limits could be given in normalized coordinates or in device coordinates. In developing the viewing algorithms, we will assume that the viewport is to be specified in device coordinates and that normalized coordinates are transferred to viewport coordinates, following the clipping operations. There are also a few other tasks that must be performed, such as identifying visible surfaces and applying the surface-rendering procedures. The final step is to map viewport coordinates to device coordinates within a selected display window. Scene descriptions in device coordinates are sometimes expressed in a left-handed reference frame so that positive distances from the display screen can be used to measure depth values in the scene.

https://hemanthrajhemu.github.io

3 Three-Dimensional Viewing-Coordinate Parameters

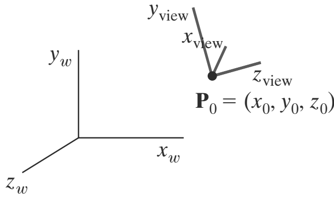


FIGURE 7
A right-handed viewing-coordinate system, with axes x_{view}, y_{view}, and z_{view}, relative to a right-handed world-coordinate frame.

Establishing a three-dimensional viewing reference frame is similar to setting up the two-dimensional viewing reference frame. We first select a world-coordinate position $\mathbf{P}_0 = (x_0, y_0, z_0)$ for the viewing origin, which is called the **view point** or **viewing position**. (Sometimes the view point is also referred to as the *eye position* or the *camera position*.) And we specify a **view-up vector V,** which defines the y_{view} direction. For three-dimensional space, we also need to assign a direction for one of the remaining two coordinate axes. This is typically accomplished with a second vector that defines the z_{view} axis, with the viewing direction along this axis. Figure 7 illustrates the positioning of a three-dimensional viewing-coordinate frame within a world system.

The View-Plane Normal Vector

Because the viewing direction is usually along the z_{view} axis, the **view plane,** also called the **projection plane,** is normally assumed to be perpendicular to this axis. Thus, the orientation of the view plane, as well as the direction for the positive z_{view} axis, can be defined with a **view-plane normal vector N**, as shown in Figure 8.

An additional scalar parameter is used to set the position of the view plane at some coordinate value z_{vp} along the z_{view} axis, as illustrated in Figure 9. This parameter value is usually specified as a distance from the viewing origin along the direction of viewing, which is often taken to be in the negative z_{view} direction. Thus, the view plane is always parallel to the $x_{\text{view}}y_{\text{view}}$ plane, and the projection of objects to the view plane corresponds to the view of the scene that will be displayed on the output device.

Vector **N** can be specified in various ways. In some graphics systems, the direction for **N** is defined to be along the line from the world-coordinate origin to a selected point position. Other systems take **N** to be in the direction from a reference point $\mathbf{P}_{\text{ref}}$ to the viewing origin $\mathbf{P}_0$, as in Figure 10. In this case, the reference point is often referred to as a *look-at point* within the scene, with the viewing direction opposite to the direction of **N**.

We could also define the view-plane normal vector, and other vector directions, using *direction angles*. These are the three angles, α, β, and γ, that a spatial line makes with the x, y, and z axes, respectively. But it is usually much easier to specify a vector direction with two point positions in a scene than with direction angles.

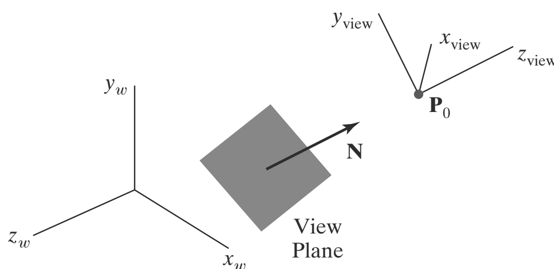


FIGURE 8
Orientation of the view plane and view-plane normal vector **N**.

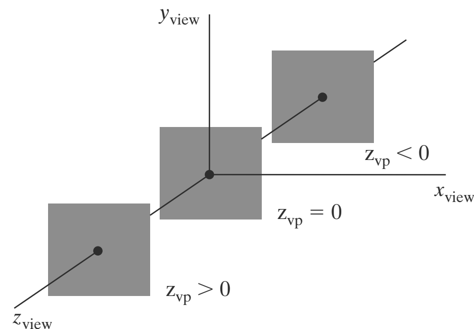


FIGURE 9
Three possible positions for the view plane along the z_{view} axis.

https://hemanthrajhemu.github.io

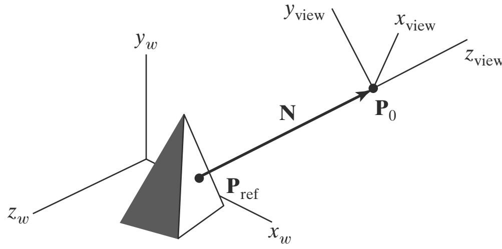


FIGURE 10
Specifying the view-plane normal vector **N** as the direction from a selected reference point $\mathbf{P}_{\text{ref}}$ to the viewing-coordinate origin $\mathbf{P}_0$.

The View-Up Vector

Once we have chosen a view-plane normal vector **N**, we can set the direction for the view-up vector **V**. This vector is used to establish the positive direction for the y_{view} axis.

Usually, **V** is defined by selecting a position relative to the world-coordinate origin, so that the direction for the view-up vector is from the world origin to this selected position. Because the view-plane normal vector **N** defines the direction for the z_{view} axis, vector **V** should be perpendicular to **N**. But, in general, it can be difficult to determine a direction for **V** that is precisely perpendicular to **N**. Therefore, viewing routines typically adjust the user-defined orientation of vector **V**, as shown in Figure 11, so that **V** is projected onto a plane that is perpendicular to the view-plane normal vector.

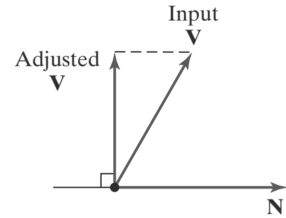


FIGURE 11
Adjusting the input direction of the view-up vector **V** to an orientation perpendicular to the view-plane normal vector **N**.

We can choose any direction for the view-up vector **V**, so long as it is not parallel to **N**. A convenient choice is often in a direction parallel to the world y_w axis; that is, we could set $\mathbf{V} = (0, 1, 0)$.

The uvn Viewing-Coordinate Reference Frame

Left-handed viewing coordinates are sometimes used in graphics packages, with the viewing direction in the positive z_{view} direction. With a left-handed system, increasing z_{view} values are interpreted as being farther from the viewing position along the line of sight. But right-handed viewing systems are more common, because they have the same orientation as the world-reference frame. This allows a graphics package to deal with only one coordinate orientation for both world and viewing references. Although some early graphics packages defined viewing coordinates within a left-handed frame, right-handed viewing coordinates are now used by the graphics standards. However, left-handed coordinate references are often used to represent screen coordinates and for the normalization transformation.

Because the view-plane normal **N** defines the direction for the z_{view} axis and the view-up vector **V** is used to obtain the direction for the y_{view} axis, we need only determine the direction for the x_{view} axis. Using the input values for **N** and **V**, we can compute a third vector, **U**, that is perpendicular to both **N** and **V**. Vector **U** then defines the direction for the positive x_{view} axis. We determine the correct direction for **U** by taking the vector cross product of **V** and **N** so as to form a right-handed viewing frame. The vector cross product of **N** and **U** also produces the adjusted value for **V**, perpendicular to both **N** and **U**, along the positive y_{view} axis. Following these procedures, we obtain the following set of unit axis vectors for a right-handed viewing coordinate system.

$$
\begin{aligned}
\mathbf{n} &= \frac{\mathbf{N}}{|\mathbf{N}|} = (n_x,\ n_y,\ n_z) \\
\mathbf{u} &= \frac{\mathbf{V} \times \mathbf{n}}{|\mathbf{V} \times \mathbf{n}|} = (u_x,\ u_y,\ u_z) \qquad (1) \\
\mathbf{v} &= \mathbf{n} \times \mathbf{u} = (v_x,\ v_y,\ v_z)
\end{aligned}
$$

https://hemanthrajhemu.github.io

Illumination Models and Surface-Rendering Methods

1 Light Sources
2 Surface Lighting Effects
3 Basic Illumination Models
4 Transparent Surfaces
5 Atmospheric Effects
6 Shadows
7 Camera Parameters
8 Displaying Light Intensities
9 Halftone Patterns and Dithering Techniques
10 Polygon Rendering Methods
11 OpenGL Illumination and Surface-Rendering Functions
12 Summary

Realistic displays of a scene are obtained by generating perspective projections of objects and applying natural lighting effects to the visible surfaces. 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. A **surface-rendering method** uses the color calculations from an illumination model to determine the pixel colors for all projected positions in a scene. The illumination model can be applied to every projection position, or the surface rendering can be accomplished by interpolating colors on the surfaces using a small set of illumination-model calculations. Scan-line, image-space algorithms typically use interpolation schemes. Sometimes, a surface-rendering procedure is called a shading method that calculates surface colors using a shading model, but this can lead to some confusion between the two terms. To avoid possible misinterpretations due to the use of similar terminology, we refer to the model for calculating the light intensity at a single surface point

From Chapter 17 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.
Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

https://hemanthrajhemu.github.io

as an *illumination model* or a *lighting model*, and we use the term *surface rendering* to mean a procedure for applying a lighting model to obtain pixel colors for all projected surface positions.

Among other things, photorealism in computer graphics involves two elements: accurate representations of surface properties and good physical descriptions of the lighting effects in a scene. These surface lighting effects include light reflections, transparency, surface texture, and shadows.

In general, modeling the lighting effects that we see on an object is a complex process, involving principles of both physics and psychology. Fundamentally, lighting effects are described with models that consider the interaction of electromagnetic energy with the object surfaces in a scene. Once light reaches our eyes, it triggers perception processes that determine what we actually "see." Physical illumination models involve a number of factors, such as material properties, object position relative to light sources and other objects, and the features of the light sources. Objects can be composed of opaque materials, or they can be more or less transparent. In addition, they can have shiny or dull surfaces, and they can have a variety of surface-texture patterns. Light sources of varying shapes, colors, and positions can be used to provide the illumination for a scene. Given the parameters for the optical properties of surfaces, the relative positions of the surfaces in a scene, the color and positions of the light sources, the characteristics of the light sources, and the position and orientation of the viewing plane, illumination models calculate the light intensity projected from a particular surface position in a specified viewing direction.

Illumination models in computer graphics are often approximations of the physical laws that describe surface-lighting effects. To reduce computations, most packages use empirical models based on simplified photometric calculations. In the following sections, we take a look at the basic lighting models often used in computer-graphics systems, and we explore the various surface-rendering algorithms for applying the lighting models to obtain effective displays of natural scenes.

1 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.

https://hemanthrajhemu.github.io

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

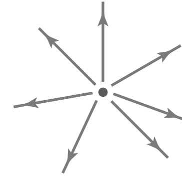


FIGURE 1
Diverging ray paths from a point light source.

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. 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.

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 $1/d_l^2$. This means that a surface close to the light source receives a higher incident light

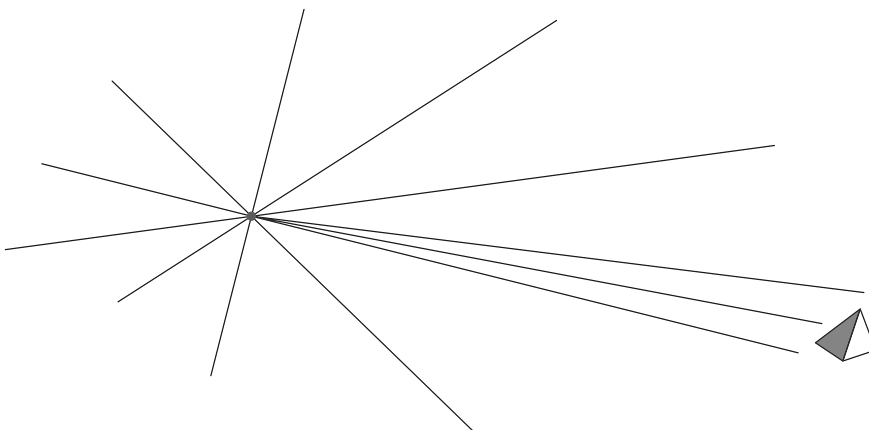


FIGURE 2
Light rays from an infinitely distant light source illuminate an object along nearly parallel light paths.

https://hemanthrajhemu.github.io

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 $1/d_l^2$ with a point source does not always produce realistic pictures. The factor $1/d_l^2$ tends to produce too much intensity variation for objects that are close to the light source, and very little variation when d_l is large. 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 d_l that includes a linear term:

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

The numerical values for the coefficients, a_0, a_1, and a_2, can then be adjusted to produce optimal attenuation effects. For instance, we can assign a large value to a_0 when d_l is very small to prevent $f_{\text{radatten}}(d_l)$ 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 intensity-attenuation calculation 1 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

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

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). A multicolor point light source could be modeled in this way using multiple direction vectors and a different emission color for each direction.

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

$$\mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} = \cos\alpha \tag{3}$$

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

https://hemanthrajhemu.github.io

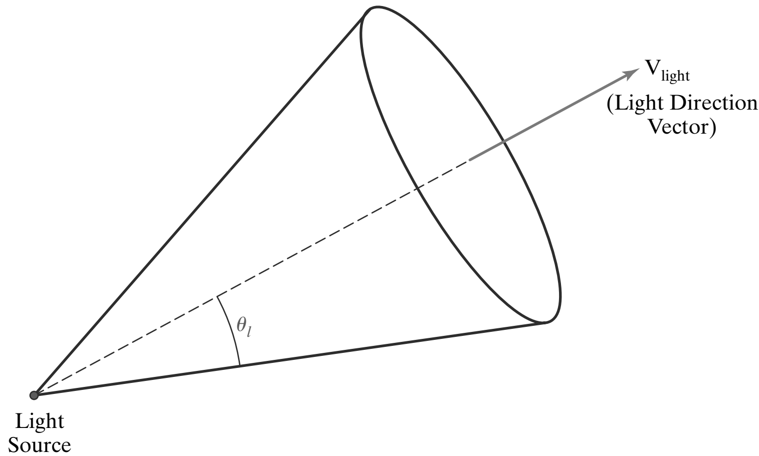


FIGURE 3
A directional point light source. The unit light-direction vector defines the axis of a light cone, and angle θ_l defines the angular extent of the circular cone.

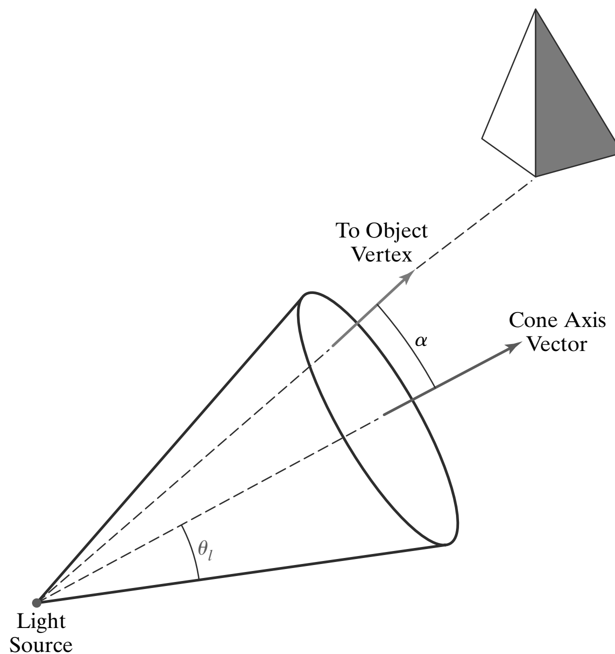


FIGURE 4
An object illuminated by a directional point light source.

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

$$f_{\text{angatten}}(\phi) = \cos^{a_l} \phi, \qquad 0^\circ \leq \phi \leq \theta \tag{4}$$

where the attenuation exponent a_l is assigned some positive value and angle ϕ is measured from the cone axis. Along the cone axis, $\phi = 0^\circ$ and $f_{\text{angatten}}(\phi) = 1.0$. The greater the value for the attenuation exponent a_l, the smaller the value of the angular intensity-attenuation function for a given value of angle $\phi > 0^\circ$.

https://hemanthrajhemu.github.io

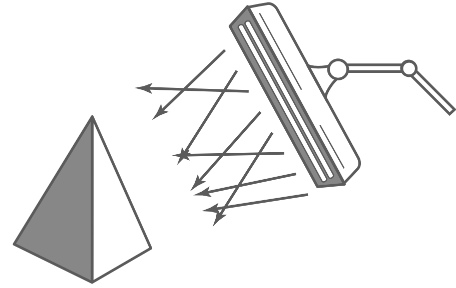


FIGURE 5
An object illuminated by a large nearby light source.

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 Equation 3. We designate $\mathbf{V}_{\text{light}}$ as the unit vector in the light-source direction (along the cone axis) and $\mathbf{V}_{\text{obj}}$ as the unit vector in the direction from the light source to an object position. Using these two unit vectors and assuming that $0^\circ < \theta_l \leq 90^\circ$, we can express the general equation for angular attenuation as

$$f_{l,\text{angatten}} = \begin{cases} 1.0, & \text{if source is not a spotlight} \\ 0.0, & \text{if } \mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} = \cos\alpha < \cos\theta_l \\ & \quad \text{(object is outside the spotlight cone)} \\ (\mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}})^{a_l}, & \text{otherwise} \end{cases} \tag{5}$$

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. 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.

2 Surface Lighting Effects

An illumination model computes the lighting effects for a surface using the various optical properties that have been assigned to that surface. These properties include degree of transparency, color reflectance coefficients, and various surface-texture parameters.

When light is incident on an opaque surface, part of it is reflected and part is absorbed. The amount of incident light reflected by the surface depends on

https://hemanthrajhemu.github.io

the type of material. Shiny materials reflect more of the incident light, and dull surfaces absorb more of the incident light. For a transparent surface, some of the incident light is also transmitted through the material.

Surfaces that are rough or grainy tend to scatter the reflected light in all directions. This scattered light is called **diffuse reflection.** A very rough, matte surface produces primarily diffuse reflections, so the surface appears equally bright from any viewing angle. Figure 6 illustrates diffuse light scattering from a surface. What we call the color of an object is the color of the diffuse reflection when the object is illuminated with white light, which is composed of a combination of all colors. A blue object, for example, reflects the blue component of the white light and absorbs all the other color components. If the blue object is viewed under a red light, it appears black because all the incident light is absorbed.

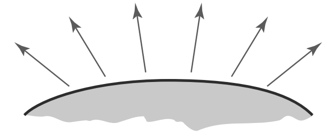


FIGURE 6
Diffuse reflections from a surface.

In addition to diffuse light scattering, some of the reflected light is concentrated into a highlight, or bright spot, called **specular reflection.** This highlighting effect is more pronounced on shiny surfaces than on dull surfaces, and we can see the specular reflection when we look at an illuminated shiny surface, such as polished metal, an apple, or a person's forehead, only when we view the surface from a particular direction. A representation of specular reflection is shown in Figure 7.

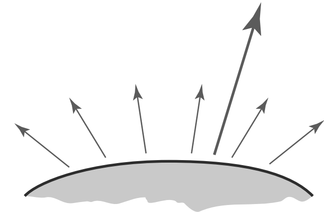


FIGURE 7
Specular reflection superimposed on diffuse reflection vectors.

Another factor that must be considered in an illumination model is the **background light** or **ambient light** in a scene. A surface that is not directly exposed to a light source may still be visible due to the reflected light from nearby objects that are illuminated. Thus, the ambient light for a scene is the illumination effect produced by the reflected light from the various surfaces in the scene. Figure 8 illustrates this background lighting effect. The total reflected light from a surface is the sum of the contributions from light sources and from the light reflected by other illuminated objects.

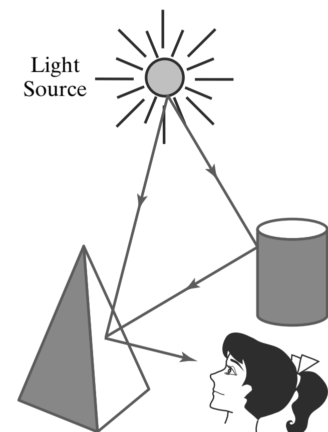


FIGURE 8
Surface lighting effects are produced by a combination of illumination from light sources and reflections from other surfaces.

3 Basic Illumination Models

Accurate surface lighting models compute the results of interactions between incident radiant energy and the material composition of an object. To simplify the surface-illumination calculations, we can use approximate representations for the physical processes that produce the lighting effects discussed in the previous section. The empirical model described in this section produces reasonably good results, and it is implemented in most graphics systems.

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 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

https://hemanthrajhemu.github.io

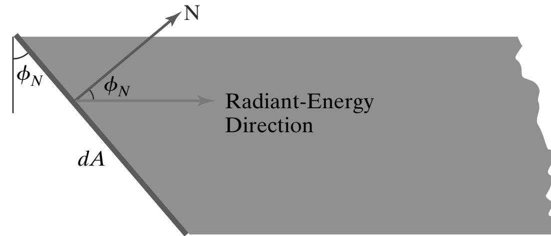


FIGURE 9
Radiant energy from a surface area element dA in direction ϕ_N relative to the surface normal direction is proportional to $\cos \phi_N$.

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 ϕ_N relative to the surface normal is proportional to $\cos \phi_N$ (Figure 9). The intensity of light in this direction can be computed as the ratio of the magnitude of the radiant energy per unit time divided by the projection of the surface area in the radiation direction:

$$\begin{aligned}\text{Intensity} &= \frac{\text{radiant energy per unit time}}{\text{projected area}} \\ &\propto \frac{\cos \phi_N}{dA \cos \phi_N} \\ &= \text{constant}\end{aligned} \tag{6}$$

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 I_a that we assigned to the scene. Therefore, the ambient contribution to the diffuse reflection at any point on a surface is simply

$$I_{\text{ambdiff}} = k_d I_a \tag{7}$$

Ambient light alone, however, produces a flat uninteresting shading for a surface (Color Plate 12), so scenes are rarely rendered using only ambient light. At least

https://hemanthrajhemu.github.io

one light source is included in a scene, often as a point source at the viewing position.

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).

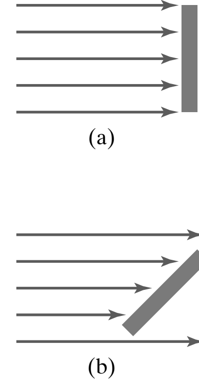


FIGURE 10
A surface that is perpendicular to the direction of the incident light (a) is more illuminated than an equal-sized surface at an oblique angle (b) to the incoming light direction.

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 $\cos\theta$. Therefore, we can model the amount of incident light on a surface from a source with intensity I_l as

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

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, $\theta = 90^\circ$ and $I_{l,\text{diff}} = k_d I_l$. 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° ($\cos\theta$ is in the interval from 0.0 to 1.0). When $\cos\theta < 0.0$, the light source is behind the surface.

At any surface position, we can denote the unit normal vector as $\mathbf{N}$ and the unit direction vector to a point source as $\mathbf{L}$, as in Figure 12. Then, $\cos\theta = \mathbf{N} \cdot \mathbf{L}$ and the diffuse reflection equation for single point-source illumination at a surface position can be expressed in the form

$$I_{l,\text{diff}} = \begin{cases} k_d I_l (\mathbf{N} \cdot \mathbf{L}), & \text{if } \mathbf{N} \cdot \mathbf{L} > 0 \\ 0.0, & \text{if } \mathbf{N} \cdot \mathbf{L} \le 0 \end{cases} \tag{10}$$

The unit direction vector $\mathbf{L}$ to a nearby point light source is calculated using the surface position and the light-source position:

$$\mathbf{L} = \frac{\mathbf{P}_{\text{source}} - \mathbf{P}_{\text{surf}}}{|\mathbf{P}_{\text{source}} - \mathbf{P}_{\text{surf}}|} \tag{11}$$

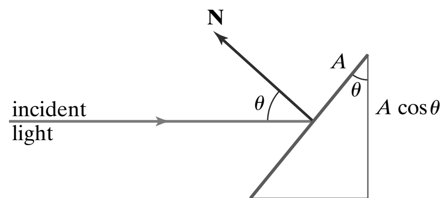


FIGURE 11
An illuminated area A projected perpendicular to the path of incoming light rays. This perpendicular projection has an area equal to $A \cos\theta$.

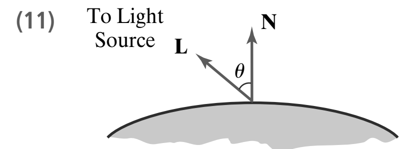


FIGURE 12
Angle of incidence θ between the unit light-source direction vector $\mathbf{L}$ and the unit normal vector $\mathbf{N}$ at a surface position.

https://hemanthrajhemu.github.io

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**.

Color Plate 13 illustrates the application of Equation 10 to positions over the surface of a sphere, using selected values for parameter k_d between 0 and 1. At $k_d = 0$, no light is reflected and the object surface appears black. Increasing values for k_d increase the intensity of the diffuse reflections, producing lighter shades of gray. Each projected pixel position for the surface is assigned an intensity value as calculated by the diffuse reflection equation. The surface renderings in this figure illustrate single point-source lighting with no other lighting effects. This is what we might expect to see if we shined a very small flashlight, such as a penlight, on the object in a completely darkened room. For general scenes, however, we expect some surface reflections due to the ambient light in addition to the illumination effects produced by a light source.

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

$$I_{\text{diff}} = \begin{cases} k_a I_a + k_d I_l (\mathbf{N} \cdot \mathbf{L}), & \text{if } \mathbf{N} \cdot \mathbf{L} > 0 \\ k_a I_a, & \text{if } \mathbf{N} \cdot \mathbf{L} \le 0 \end{cases} \tag{12}$$

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

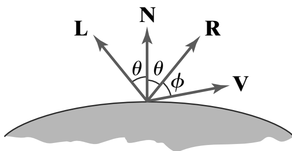


FIGURE 13
Specular reflection angle equals angle of incidence θ.

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**. 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 ($\phi = 0$).

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 $\cos^{n_s} \phi$. Angle ϕ can be assigned values in the range 0° to 90°, so that $\cos \phi$ varies from 0 to 1.0. The value assigned to the **specular-reflection exponent** n_s is determined by the type of surface that we want to display. A very shiny surface is modeled with a large value for n_s (say, 100 or more), and smaller values (down to 1) are used for duller surfaces. For a perfect reflector, n_s is infinite. For a rough surface, such as chalk or cinderblock, n_s is assigned a value near 1. Figures 14 and 15

https://hemanthrajhemu.github.io

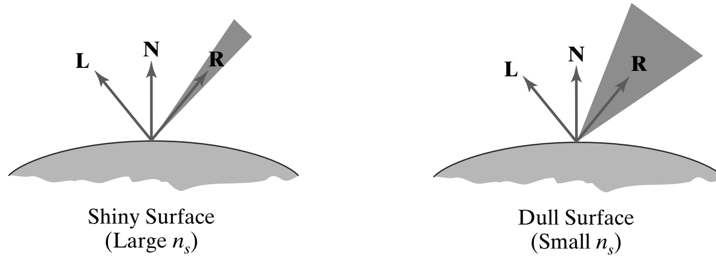


FIGURE 14
Modeling specular reflections (shaded area) with parameter n_s.

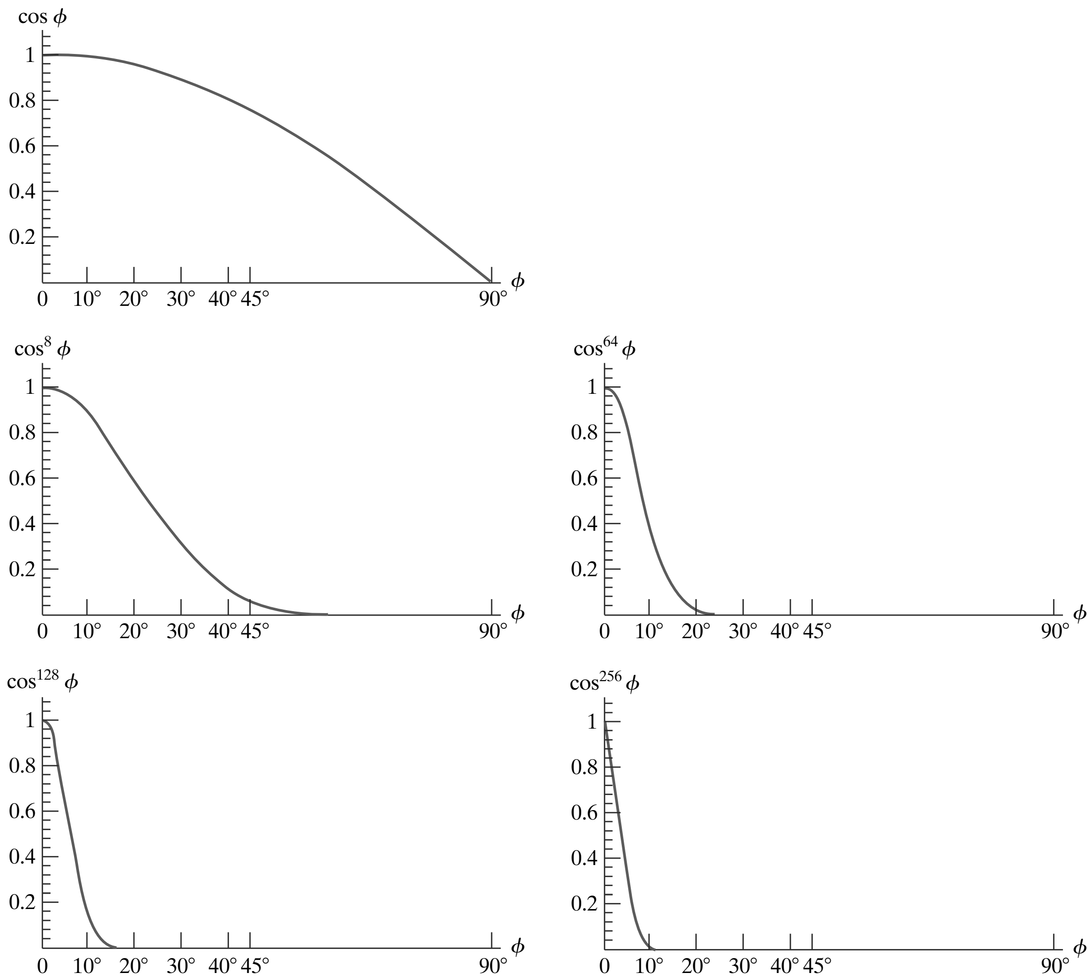


FIGURE 15
Plots of $\cos^{n_s} \phi$ using five different values for the specular exponent n_s.

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

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,** $W(\theta)$, for each surface. Figure 16 shows the general variation of $W(\theta)$ over the range $\theta = 0°$ to $\theta = 90°$ for a few materials. In general, $W(\theta)$ tends to increase as the angle of incidence increases. At $\theta = 90°$, all the incident light is reflected ($W(\theta) = 1$). The

https://hemanthrajhemu.github.io

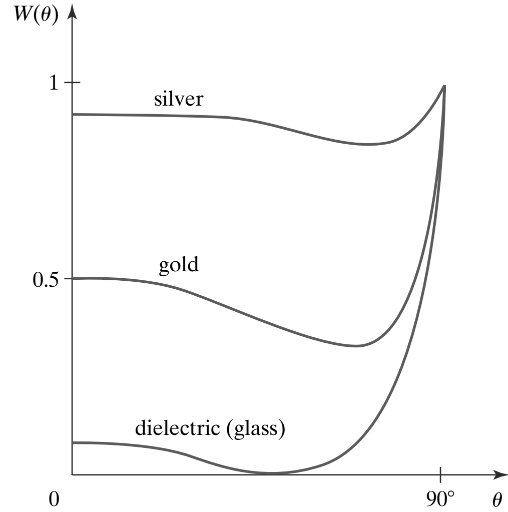


FIGURE 16
Approximate variation of the specular-reflection coefficient for different materials, as a function of the angle of incidence.

variation of specular intensity with angle of incidence is described by *Fresnel's Laws of Reflection*. Using the spectral-reflection function $W(\theta)$, we can write the Phong specular-reflection model as

$$I_{l,\text{spec}} = W(\theta) I_l \cos^{n_s} \phi \quad (13)$$

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

As seen in Figure 16, transparent materials, such as glass, exhibit appreciable specular reflections only as θ approaches 90°. At $\theta = 0^\circ$, about 4 percent of the incident light on a glass surface is reflected, and for most of the range of θ, the reflected intensity is less than 10 percent of the incident intensity. However, for many opaque materials, specular reflection is nearly constant for all incidence angles. In this case, we can reasonably model the specular effects by replacing $W(\theta)$ with a constant specular-reflection coefficient k_s. We then simply set 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 $\cos \phi$ with the dot product $\mathbf{V} \cdot \mathbf{R}$. 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

$$I_{l,\text{spec}} = \begin{cases} k_s I_l (\mathbf{V} \cdot \mathbf{R})^{n_s}, & \text{if } \mathbf{V} \cdot \mathbf{R} > 0 \quad \text{and} \quad \mathbf{N} \cdot \mathbf{L} > 0 \\ 0.0, & \text{if } \mathbf{V} \cdot \mathbf{R} \le 0 \quad \text{or} \quad \mathbf{N} \cdot \mathbf{L} \le 0 \end{cases} \quad (14)$$

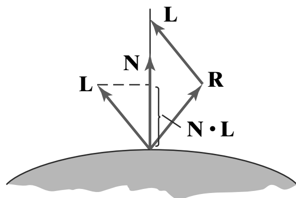


FIGURE 17
The projection of either **L** or **R** onto the direction of the normal vector **N** has a magnitude equal to **N • L**.

The direction for **R**, the reflection vector, can be computed from the directions for vectors **L** and **N**. As seen in Figure 17, the projection of **L** onto the direction of the normal vector has a magnitude equal to the dot product $\mathbf{N} \cdot \mathbf{L}$, which is also equal to the magnitude of the projection of unit vector **R** onto the direction of **N**. Therefore, from this diagram, we see that

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

and the specular-reflection vector is obtained as

$$\mathbf{R} = (2\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L} \quad (15)$$

We calculate **V** using the surface position and the viewing position, in same way that we obtained the unit vector **L** (Eq. 11). But if a fixed viewing direction

https://hemanthrajhemu.github.io

is to be used for all positions in a scene, we can set $\mathbf{V} = (0.0, 0.0, 1.0)$, which is a unit vector in the positive z direction. Specular calculations take less time to calculate using a constant $\mathbf{V}$, but the displays are not as realistic.

A somewhat simplified Phong model is obtained using the **halfway vector H** between $\mathbf{L}$ and $\mathbf{V}$ to calculate the range of specular reflections. If we replace $\mathbf{V} \cdot \mathbf{R}$ in the Phong model with the dot product $\mathbf{N} \cdot \mathbf{H}$, this simply replaces the empirical $\cos\phi$ calculation with the empirical $\cos\alpha$ calculation (Figure 18). The halfway vector is obtained as

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|} \tag{16}$$

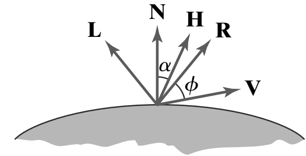


FIGURE 18
Halfway vector **H** along the bisector of the angle between **L** and **V**.

For nonplanar surfaces, $\mathbf{N} \cdot \mathbf{H}$ requires less computation than $\mathbf{V} \cdot \mathbf{R}$ because the calculation of $\mathbf{R}$ at each surface point involves the variable vector $\mathbf{N}$. Also, if both the viewer and the light source are sufficiently far from the surface, vectors $\mathbf{V}$ and $\mathbf{L}$ are each constants, and thus $\mathbf{H}$ is also constant for all surface points. If the angle between $\mathbf{H}$ and $\mathbf{N}$ is greater than 90°, $\mathbf{N} \cdot \mathbf{H}$ is negative and we set the specular-reflection contribution to 0.0.

Vector $\mathbf{H}$ 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, $\mathbf{H}$ is sometimes referred to as the surface orientation direction for maximum highlights. Also, if vector $\mathbf{V}$ is coplanar with vectors $\mathbf{L}$ and $\mathbf{R}$ (and thus $\mathbf{N}$), angle α has the value $\phi/2$. When $\mathbf{V}$, $\mathbf{L}$, and $\mathbf{N}$ are not coplanar, $\alpha > \phi/2$, 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 specular reflections from a position on an illuminated surface as

$$\begin{aligned} I &= I_{\text{diff}} + I_{\text{spec}} \\ &= k_a I_a + k_d I_l (\mathbf{N} \cdot \mathbf{L}) + k_s I_l (\mathbf{N} \cdot \mathbf{H})^{n_s} \end{aligned} \tag{17}$$

The surface is illuminated only with ambient light if the light source is behind the surface, and there are no specular effects if $\mathbf{V}$ and $\mathbf{L}$ are on the same side of the normal vector $\mathbf{N}$. Color Plate 12 illustrates surface lighting effects produced by the various terms in Equation 17.

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:

$$\begin{aligned} I &= I_{\text{ambdiff}} + \sum_{l=1}^{n} [I_{l,\text{diff}} + I_{l,\text{spec}}] \\ &= k_a I_a + \sum_{l=1}^{n} I_l [k_d (\mathbf{N} \cdot \mathbf{L}) + k_s (\mathbf{N} \cdot \mathbf{H})^{n_s}] \end{aligned} \tag{18}$$

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 including an emission term $I_{\text{surfemission}}$ in the illumination model in the same way that we

https://hemanthrajhemu.github.io

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

$$I = I_{\text{surfemission}} + I_{\text{ambdiff}} + \sum_{l=1}^{n} f_{l,\text{radatten}} f_{l,\text{angatten}} (I_{l,\text{diff}} + I_{l,\text{spec}}) \tag{19}$$

The radial attenuation function $f_{l,\text{radatten}}$ is evaluated using Equation 2, and the angular attenuation function is evaluated using Equation 5. For each light source, we calculate the diffuse reflection from a surface point as

$$I_{l,\text{diff}} = \begin{cases} 0.0, & \text{if } \mathbf{N} \cdot \mathbf{L}_l \leq 0.0 \text{ (light source behind object)} \\ k_d I_l (\mathbf{N} \cdot \mathbf{L}_l), & \text{otherwise} \end{cases} \tag{20}$$

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

$$I_{l,\text{spec}} = \begin{cases} 0.0, & \text{if } \mathbf{N} \cdot \mathbf{L}_l \leq 0.0 \\ & \quad \text{(light source behind object)} \\ k_s I_l \max\{0.0, (\mathbf{N} \cdot \mathbf{H}_l)^{n_s}\}, & \text{otherwise} \end{cases} \tag{21}$$

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.

https://hemanthrajhemu.github.io

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, $I_l = (I_{lR}, I_{lG}, I_{lB})$. Similarly, the reflection coefficients are also specified with RGB components: $k_a = (k_{aR}, k_{aG}, k_{aB})$, $k_d = (k_{dR}, k_{dG}, k_{dB})$, and $k_s = (k_{sR}, k_{sG}, k_{sB})$. 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 from modified expressions 20 and 21 as

$$I_{lB,\mathrm{diff}} = k_{dB} I_{lB} (\mathbf{N} \cdot \mathbf{L}_l) \tag{22}$$

and

$$I_{lB,\mathrm{spec}} = k_{sB} I_{lB} \max\{0.0, (\mathbf{N} \cdot \mathbf{H}_l)^{n_s}\} \tag{23}$$

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, k_{dB}, while the red and green reflectivity components are set to zero ($k_{dR} = k_{dG} = 0.0$). 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 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, as in Equation 23. Color Plate 14 illustrates color reflections from a matte surface, and Color Plates 15 and 16 show color reflections from metal surfaces.

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 (Eq. 22) is then calculated as

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

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.

https://hemanthrajhemu.github.io

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

$$I_{l\lambda,\text{diff}} = k_d S_{d\lambda} I_{l\lambda} (\mathbf{N} \cdot \mathbf{L}_l) \tag{25}$$

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

$$\text{luminance} = \int_{\text{visible} f} p(f)\, I(f)\, df \tag{26}$$

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

$$\text{luminance} = 0.299R + 0.587G + 0.114B \tag{27}$$

Sometimes, better lighting effects are achieved by increasing the contribution for the green component of each RGB color. One recommendation for this calculation is $0.2125R + 0.7154G + 0.0721B$. The luminance parameter is most often represented with the symbol Y, which corresponds to the Y component in the XYZ color model.

4 Transparent Surfaces

We describe an object, such as a glass windowpane, as *transparent* if we can see things that are behind that object. Similarly, if we cannot see things that are behind an object, it is *opaque*. In addition, some transparent objects, such as frosted glass and certain plastic materials, are **translucent** so that the transmitted light is diffused in all directions. Objects viewed through translucent materials appear blurred and are often not clearly identifiable.

https://hemanthrajhemu.github.io

A transparent surface, in general, produces both reflected and transmitted light. The light transmitted through the surface is the result of emissions and reflections from the objects and sources behind the transparent object. Figure 19 illustrates the intensity contributions to the surface lighting for a transparent object that is in front of an opaque object.

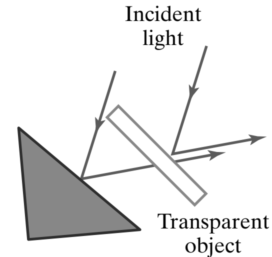


FIGURE 19
Light emission from a transparent surface is in general a combination of reflected and transmitted light.

Translucent Materials

Both diffuse and specular transmission can take place at the surfaces of a transparent object. Diffuse effects are important when translucent materials are to be modeled. Light passing through a translucent material is scattered so that background objects are seen as blurred images. We can simulate diffuse transmissions by distributing intensity contributions from background objects over a finite area, or we can use ray-tracing methods to simulate translucency. These manipulations are time-consuming, and basic illumination models ordinarily compute only specular-transparency effects.

Light Refraction

Realistic displays of a transparent material are obtained by modeling the **refraction** path of a ray of light through the material. When a light beam is incident upon a transparent surface, part of it is reflected and part is transmitted through the material as refracted light, as shown in Figure 20. Because the speed of light is different in different materials, the path of the refracted light is different from that of the incident light. The direction of the refracted light, specified by the **angle of refraction** with respect to the surface normal vector, is a function of the **index of refraction** of the material and the incoming direction of the incident light. Index of refraction is defined as the ratio of the speed of light in a vacuum to the speed of light in the material. Angle of refraction θ_r is calculated from **Snell's law** as

$$\sin\theta_r = \frac{\eta_i}{\eta_r}\sin\theta_i \tag{28}$$

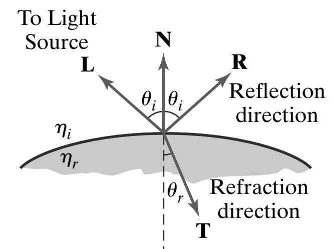


FIGURE 20
Reflection direction **R** and refraction (transmission) direction **T** for a ray of light incident upon a surface with index of refraction η_r.

where θ_i is the angle of incidence, η_i is the index of refraction for the incident material, and η_r is the index of refraction for the refracting material.

Actually, the index of refraction also depends on other factors, such as the temperature of the material and the wavelength of the incident light. Thus, the various color components of incident white light, for example, are refracted at different angles, which vary with temperature. Furthermore, within anisotropic materials such as crystalline quartz, the speed of light depends on direction, and some transparent materials exhibit *double refraction*, in which two refracted light rays are generated. For most applications, however, we can use a single average index of refraction for each material, as listed in Table 1. Using the index of refraction for air (approximately 1.0) surrounding a pane of heavy crown glass (refractive index ≈ 1.61) in Equation 28, with an angle of incidence of 30°, we obtain a refraction angle of about 18° for the light passing through the crown glass.

Refraction occurs whenever a ray moves through the boundary between materials, so in a situation where the ray passes completely through an object, the ray will be refracted twice—one refraction for each boundary transition. Figure 21 illustrates the refraction changes for a ray of light passing through a thin sheet of glass. The overall effect of the refraction is to shift the incident light to a parallel path as it emerges from the material. Because the evaluations for the trigonometric functions in Equation 28 are time-consuming, these refraction effects could be approximated by simply shifting the path of the incident light by an appropriate amount for a given material.

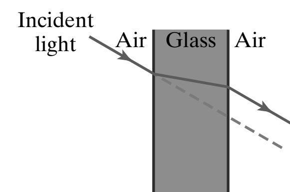


FIGURE 21
Refraction of light through a pane of glass. The emerging refracted ray travels along a path that is parallel to the incident light path (dashed line).

https://hemanthrajhemu.github.io

TABLE 1

Average Index of Refraction for Common Materials

Material	Index of Refraction
Vacuum, air or other gas	1.00
Ordinary crown glass	1.52
Heavy crown glass	1.61
Ordinary flint glass	1.61
Heavy flint glass	1.92
Rock salt	1.55
Quartz	1.54
Water	1.33
Ice	1.31

From Snell's law and the diagram in Figure 20, we can obtain the unit transmission vector **T** in the refraction direction θ_r as follows:

$$\mathbf{T} = \left(\frac{\eta_i}{\eta_r}\cos\theta_i - \cos\theta_r\right)\mathbf{N} - \frac{\eta_i}{\eta_r}\mathbf{L} \tag{29}$$

where **N** is the unit surface normal and **L** is the unit vector in the direction from the surface position to the light source. Transmission vector **T** can be used to locate intersections of the refraction path with objects behind the transparent surface. Including refraction effects in a scene can produce highly realistic displays, but the determination of refraction paths and object intersections requires considerable computation. Most scan-line image-space methods model light transmission with approximations that reduce processing time. Accurate refraction effects are displayed using ray-tracing algorithms.

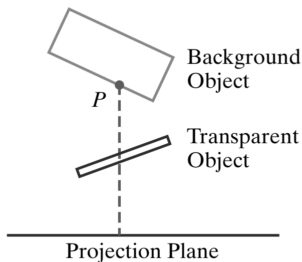


FIGURE 22
The intensity of a background object at point **P** can be combined with the reflected intensity off the surface of a transparent object along a perpendicular projection line (dashed).

Basic Transparency Model

A simpler procedure for modeling transparent objects is to ignore the path shifts due to refraction. In effect, this approach assumes there is no change in the index of refraction from one material to another, so that the angle of refraction is always the same as the angle of incidence. This method speeds up the calculation of intensities and can produce reasonable transparency effects for thin, polygonal surfaces.

We can combine the transmitted intensity I_{trans} through a transparent surface from a background object with the reflected intensity I_{refl} from the surface (Figure 22) using a **transparency coefficient** k_t. We assign parameter k_t a value between 0.0 and 1.0 to specify how much of the background light is to be transmitted. Total surface intensity is then calculated as

$$I = (1 - k_t)I_{\text{refl}} + k_t I_{\text{trans}} \tag{30}$$

The term $(1 - k_t)$ is the **opacity factor.** For example, if the transparency factor is assigned the value 0.3, then 30 percent of the background light is combined with 70 percent of the reflected surface illumination.

https://hemanthrajhemu.github.io

This procedure can be used to combine the lighting effects from any number of transparent and opaque objects, so long as we process the surfaces in a depth-first order (i.e., back to front). For example, looking through an empty drinking glass, we can see opaque objects that are behind its two transparent surfaces. Similarly, when we look through the windshield of an automobile, objects inside the car are visible, as well as objects that may be behind the back window.

For highly transparent objects, we assign k_t a value near 1.0. Nearly opaque objects transmit very little light from background objects, and we can set k_t to a value near 0.0 for these materials. It is also possible to allow k_t to be a function of position over the surface such that different parts of an object can transmit more or less of the light from the background surfaces.

A depth-sorting visibility algorithm can be modified to handle transparency by first sorting surfaces in depth order, then determining whether any visible surface is transparent. If it is, its reflected surface intensity is combined with the surface intensity of objects behind it to obtain the pixel intensity at each projected surface point.

Transparency effects could also be implemented using a modified depth-buffer approach. We can divide the surfaces in a scene into two groups so that all the opaque surfaces are processed first. At this point, the frame buffer contains the intensities of the visible surfaces, and the depth buffer contains their depths. Then, the depth positions of the transparent objects are compared to the values previously stored in the depth buffer. If any transparent surface is visible, its reflected intensity is calculated and combined with the opaque surface intensity previously stored in the frame buffer. This method can be modified to produce more accurate displays by using additional storage for the depth and other parameters of the transparent surfaces. This allows depth values for the transparent surfaces to be compared to each other, as well as to the depth values of the opaque surfaces. Visible transparent surfaces are then rendered by combining their surface intensities with those of the visible and opaque surfaces behind them.

Another approach is the A-buffer method. For each pixel position in the A-buffer, surface patches for all overlapping surfaces are saved and sorted in depth order. Then, intensities for the transparent and opaque surface patches that overlap in depth are combined in the proper visibility order to produce the final averaged intensity for the pixel.

5 Atmospheric Effects

Another factor that is sometimes included in an illumination model is the effect of the atmosphere on an object's color. A hazy atmosphere makes colors fade and objects appear dimmer. Thus, we could specify a function to modify surface colors according to the amount of dust, smoke, or smog that we want to simulate in the atmosphere. The hazy-atmosphere effect is often simulated with an exponential attenuation function such as

$$f_{\text{atmo}}(d) = e^{-\rho d} \tag{31}$$

or

$$f_{\text{atmo}}(d) = e^{-(\rho d)^2} \tag{32}$$

The value assigned to d is the distance of the object from the viewing position. In addition, we use parameter ρ in either of these exponential functions to set a positive density value for the atmosphere. Higher values for ρ produce a denser atmosphere and cause surface colors to be more muted. After the surface color of an object has been computed, we multiply that color by one of the atmosphere

https://hemanthrajhemu.github.io

functions to decrease its intensity by an amount that depends on the value we set for the density of the atmosphere.

Instead of an exponential function, we could simplify the atmospheric attenuation calculations by using the linear depth-cueing function. This decreases the intensity of surface colors for distant objects, but we then have no provision for varying the density of the atmosphere.

Sometimes we might also want to simulate an atmosphere color. For example, the air in a smoky room could be modeled with a slate-gray color, or perhaps a pale blue. The following calculation could then be used to combine the atmosphere color with an object's color:

$$I = f_{\text{atmo}}(d) I_{\text{obj}} + [1 - f_{\text{atmo}}(d)] I_{\text{atmo}} \tag{33}$$

where f_{atmo} is an exponential or linear atmosphere-attenuation function.

6 Shadows

Visibility detection methods can be used to locate regions that are not illuminated by light sources. With the viewing position at the location of a light source, we can determine which surface sections in the scene are not visible. These are the shadow areas. Once we have determined the shadow areas for all light sources, the shadows could be treated as surface patterns and stored in pattern arrays.

Shadow patterns generated by a visible-surface detection method are valid for any selected viewing position, so long as the light-source positions are not changed. Surfaces that are visible from the view position are shaded according to the lighting model, which can be combined with texture patterns. We can display shadow areas with ambient light intensity only, or we could combine the ambient light with specified surface textures.

7 Camera Parameters

The viewing and illumination procedures we have considered so far produce sharp images that are equivalent to photographing a scene with a pinhole camera. When we photograph an actual scene, however, we can adjust the camera so that only selected objects are in focus. Other objects are then more or less out of focus, depending on the depth distribution of the objects in the scene. We can simulate the appearance of out-of-focus positions in a computer-graphics program, by projecting each position to an area covering multiple pixel positions, with the object colors merged into other objects to produce a blurred projection pattern. This procedure is similar to the methods used in antialiasing, and we can incorporate the camera effects into either a scan-line or a ray-tracing algorithm. Computer-generated scenes appear more realistic when focusing effects are included, but the focusing calculations are time-consuming.

8 Displaying Light Intensities

A surface intensity calculated by an illumination model can have any value in the range from 0.0 to 1.0, but a computer-graphics system can display only a limited set of intensities. Therefore, a calculated intensity value must be converted to one of

https://hemanthrajhemu.github.io

the allowable system values. In addition, the allowable number of system intensity levels can be distributed so that they correspond to the way that our eyes perceive intensity differences. When we display scenes on a bilevel system, we could convert calculated intensities into halftone patterns, as discussed in Section 9.

Distributing System Intensity Levels

For any system, the allowable number of intensity levels can be distributed over the range from 0.0 to 1.0 so that this distribution corresponds to our perception of equal intensity intervals between levels. We perceive relative light intensities the same way that we perceive relative sound intensities: on a logarithmic scale. This means that if the ratio of two intensity values is the same as the ratio of two other intensities, we perceive the difference between each pair of intensities to be the same. For example, we perceive the difference between intensities 0.20 and 0.22 to be the same as the difference between 0.80 and 0.88. Therefore, to display $n + 1$ successive intensity levels with equal perceived brightness, the intensity levels on the monitor should be spaced so that the ratio of successive intensities is constant, as follows:

$$\frac{I_1}{I_0} = \frac{I_2}{I_1} = \cdots = \frac{I_n}{I_{n-1}} = r \tag{34}$$

where I represents the intensity of one of the color components of a light. The lowest level that can be displayed is represented as I_0 and the highest is represented as I_n. Any intermediate intensity can then be expressed in terms of I_0 as

$$I_k = r^k I_0 \tag{35}$$

We can calculate the value of r, given the values of I_0 and n for a particular system, by substituting $k = n$ in the previous expression. Because $I_n = 1.0$, we have

$$r = \left(\frac{1.0}{I_0}\right)^{1/n} \tag{36}$$

Thus, the calculation for I_k in Equation 35 can be rewritten as

$$I_k = I_0^{(n-k)/n} \tag{37}$$

For example, if $I_0 = \frac{1}{8}$ for a system with $n = 3$, we have $r = 2$ and the four intensity values are $\frac{1}{8}$, $\frac{1}{4}$, $\frac{1}{2}$, and 1.0.

The lowest intensity value I_0 depends on the characteristics of the monitor and is typically in the range from 0.005 to around 0.025. This residual intensity on a video monitor is due to reflected light from the screen phosphors. Therefore, a "black" region on the screen will always have some intensity value above 0.0. For a grayscale display with 8 bits per pixel ($n = 255$) and $I_0 = 0.01$, the ratio of successive intensities is approximately $r = 1.0182$. The approximate values for the 256 intensities on this system are 0.0100, 0.0102, 0.0104, 0.0106, 0.0107, 0.0109, . . . , 0.9821, and 1.0000.

Similar methods are used with RGB color components. For example, we can express the intensity of the blue component of a color at level k in terms of the lowest attainable blue value as

$$I_{Bk} = r_B^k I_{B0} \tag{38}$$

where

$$r_B = \left(\frac{1.0}{I_{B0}}\right)^{1/n} \tag{39}$$

and n is the number of intensity levels.

https://hemanthrajhemu.github.io

Gamma Correction and Video Lookup Tables

When we display color or monochromatic images on a video monitor, the perceived brightness variations are nonlinear, but illumination models produce a linear variation for intensity values. The RGB color (0.25, 0.25, 0.25) obtained from a lighting model represents one-half the intensity of the color (0.5, 0.5, 0.5). Usually, these calculated intensities are then stored in an image file as integer values ranging from 0 to 255, with one byte for each of the three RGB components. This intensity file is also linear, so a pixel with the value (64, 64, 64) represents half the intensity of a pixel with the value (128, 128, 128). The electron-gun voltages, which control the number of electrons striking the phosphor screen, produce brightness levels as determined by the **monitor response curve** shown in Figure 23. Therefore, the displayed intensity value (64, 64, 64) would not appear to be half as bright as the value (128, 128, 128).

To compensate for monitor nonlinearities, graphics systems use a **video lookup table** that adjusts the linear input intensity values. The monitor response curve is described with the exponential function

$$I = aV^{\gamma} \tag{40}$$

Parameter I is displayed intensity and parameter V is the corresponding electron-gun voltage. Values for parameters a and γ depend on the characteristics of the monitor used in the graphics system. Thus, if we want to display a particular intensity value I, the voltage value to produce this intensity is

$$V = \left(\frac{I}{a}\right)^{1/\gamma} \tag{41}$$

This calculation is referred to as the **gamma correction** of intensity, and gamma values are typically in the range from about 1.7 to 2.3. The National Television System Committee (NTSC) signal standard is $\gamma = 2.2$. Figure 24 shows a gamma-correction curve using the NTSC gamma value with both intensity and voltage normalized on the interval from 0 to 1.0. Equation 41 is used to set up

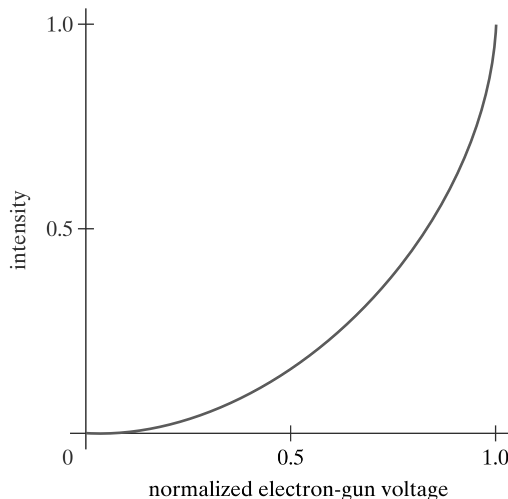


FIGURE 23
A typical monitor response curve, showing the variation in displayed intensity (or "brightness") as a function of the normalized electron-gun voltage.

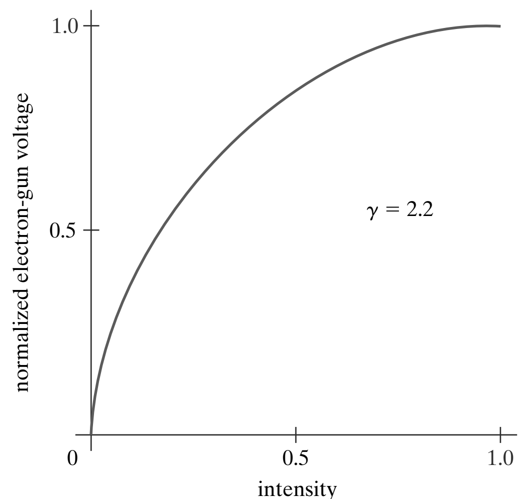


FIGURE 24
A video lookup correction curve for mapping a normalized intensity value to a normalized electron-gun voltage, using gamma correction with $\gamma = 2.2$.

https://hemanthrajhemu.github.io

the video lookup table that converts integer intensity values in an image file to values that control the electron-gun voltages.

We can combine gamma correction with logarithmic intensity mapping to produce a lookup table. If I is an input intensity value from an illumination model, we first locate the nearest intensity I_k from a table of values created with Equation 34 or Equation 37. Alternatively, we could determine the level number for this intensity value with the calculation

$$k = \text{round}\left[\log_r\left(\frac{I}{I_0}\right)\right] \tag{42}$$

then we compute the intensity value at this level using Equation 37. Once we have the intensity value I_k, we can calculate the electron-gun voltage as

$$V_k = \left(\frac{I_k}{a}\right)^{1/\gamma} \tag{43}$$

Values V_k can then be placed in the lookup tables, with values for k stored in the frame-buffer pixel positions. If a particular system has no lookup table, computed values for V_k could be stored directly in the frame buffer. The combined conversion to a logarithmic intensity scale followed by calculation of the V_k using Equation 43 is also sometimes referred to as *gamma correction*.

If the video amplifiers of a monitor are designed to convert the linear intensity values to electron-gun voltages, we cannot combine the two intensity conversion processes. In this case, gamma correction is built into the hardware, and the logarithmic values I_k must be precomputed and stored in the frame buffer (or the color table).

Displaying Continuous-Tone Images

High-quality computer graphics systems generally provide 256 intensity levels for each color component, but acceptable displays can be obtained for many applications with fewer levels. A four-level system provides minimum shading capability for continuous-tone images, while photo-realistic images can be generated on systems that are capable of from 32 to 256 intensity levels per pixel.

Figure 25 shows a continuous-tone photograph displayed with various intensity levels. When a small number of intensity levels are used to reproduce a continuous-tone image, the borders between the different intensity regions (called *contours*) are clearly visible. In the 2-level reproduction, the facial features in the photograph are just barely identifiable. Using 4 intensity levels, we begin to identify the original shading patterns, but the contouring effects are glaring. With 8 intensity levels, contouring effects are still obvious, but we begin to have a better indication of the original shading. At 16 or more intensity levels, contouring effects diminish and the reproductions are very close to the original. Reproductions of continuous-tone images using more than 32 intensity levels show only very subtle differences from the original.

9 Halftone Patterns and Dithering Techniques

With a system that has very few available intensity levels, we can create an apparent increase in the number of available intensities by incorporating multiple pixel positions into the display of each intensity value for a scene. When we view a small region consisting of several pixel positions, our eyes tend to integrate

https://hemanthrajhemu.github.io

(a) (b) (c) (d)

FIGURE 25
A continuous-tone photograph (a) printed with two intensity levels (b), four intensity levels (c), and eight intensity levels (d).

FIGURE 26
An enlarged section of a photograph reproduced with a halftoning method, showing how tones are represented with "dots" of varying sizes.

or average the fine detail into an overall intensity. Bilevel monitors and printers, in particular, can take advantage of this visual effect to produce pictures that appear to be displayed with multiple intensity values.

Continuous-tone photographs are reproduced for publication in newspapers, magazines, and books with a printing process called **halftoning,** and the reproduced pictures are called **halftones.** For a black-and-white photograph, each constant intensity area is reproduced as a set of small black circles on a white background. The diameter of each circle is proportional to the darkness required for that intensity region. Darker regions are printed with larger circles, and lighter regions are printed with smaller circles (more white space). Figure 26 shows an enlarged section of a grayscale halftone reproduction. Color halftones are printed using small circular dots of various sizes and colors. Book and magazine halftones are printed on high-quality paper using approximately 60 to 80 circles of varying diameter per centimeter. Newspapers use lower-quality paper and lower resolution (about 25 to 30 dots per centimeter).

https://hemanthrajhemu.github.io

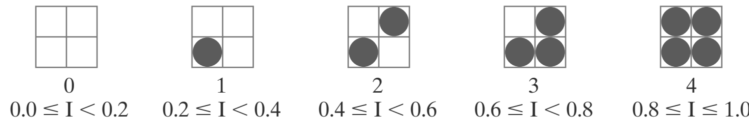


FIGURE 27
A set of 2×2 pixel grid patterns that can be used to display five intensity levels on a bilevel system, showing the "on" pixels as green circles. The intensity values that are mapped to each of the grid patterns are listed below the pixel arrays.

Halftone Approximations

In computer graphics, halftone reproductions are simulated using rectangular pixel regions that are called **halftone approximation patterns,** or just **pixel patterns.** The number of intensity levels that we can display with this method depends on how many pixels we include in the rectangular grids and how many levels a system can display. With $n \times n$ pixels for each grid on a bilevel system, we can represent $n^2 + 1$ intensity levels. Figure 27 shows one way to set up pixel patterns to represent five intensity levels that could be used with a bilevel system. In pattern 0, all pixels are turned off; in pattern 1, one pixel is turned on; and in pattern 4, all four pixels are turned on. An intensity value I in a scene is mapped to a particular pattern according to the range listed below each grid shown in the figure. Pattern 0 is used for $0.0 \leq I < 0.2$, pattern 1 for $0.2 \leq I < 0.4$, and pattern 4 is used for $0.8 \leq I \leq 1.0$.

With 3×3 pixel grids on a bilevel system, we can display 10 intensity levels. One way to set up the 10 pixel patterns for these levels is shown in Figure 28. Pixel positions are chosen at each level so that the patterns approximate the increasing circle sizes used in halftone reproductions. That is, the "on" pixel positions are near the center of the grid for lower intensity levels and expand outward as the intensity level increases.

For any pixel-grid size, we can represent the pixel patterns for the various possible intensities with a mask (matrix) of pixel position numbers. For example, the following mask can be used to generate the nine 3×3 grid patterns for intensity levels above 0 shown in Figure 28:

$$\begin{bmatrix} 8 & 3 & 7 \\ 5 & 1 & 2 \\ 4 & 9 & 6 \end{bmatrix} \tag{44}$$

To display a particular intensity with level number k, we turn on each pixel whose position number is less than or equal to k.

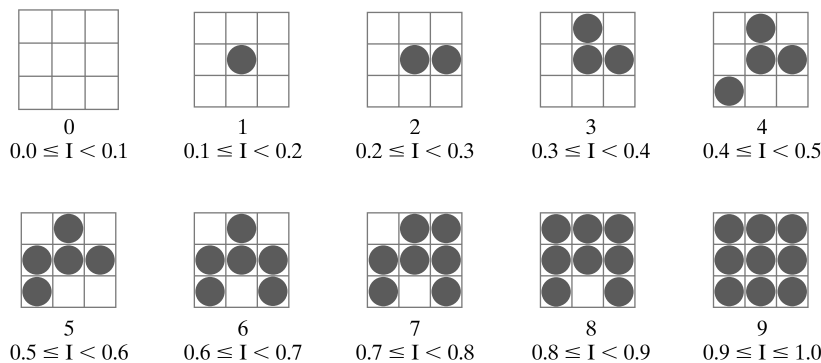


FIGURE 28
A set of 3×3 pixel grid patterns that can be used to display 10 intensities on a bilevel system, showing the "on" pixels as green circles. The intensity values that are mapped to each of the grid patterns are listed below the pixel arrays.

https://hemanthrajhemu.github.io

Although the use of $n \times n$ pixel patterns increases the number of intensities that can be represented, the resolution of the display area is reduced by a factor of $1/n$ in the x and y directions. Using 2×2 grid patterns on a 512×512 screen area, for instance, reduces the resolution to 256×256 intensity positions; and with 3×3 patterns, we reduce the resolution of the 512×512 area to 128×128.

Another problem with pixel grids is that subgrid patterns become apparent as the grid size increases. The grid size that can be used without distorting the intensity variations depends on the size of a displayed pixel. Therefore, for systems with lower resolution (fewer pixels per centimeter), we must be satisfied with fewer intensity levels. On the other hand, high-quality displays require at least 64 intensity levels. This means that we need 8×8 pixel grids. And to achieve a resolution equivalent to that of halftones in books and magazines, we must display 60 dots per centimeter. Thus, we need to be able to display $60 \times 8 = 480$ dots per centimeter. Some devices, such as high-quality film recorders, can display this resolution.

Pixel-grid patterns for halftone approximations must also be constructed to minimize contouring and other visual effects not present in the original scene. We can minimize contouring by evolving each successive grid pattern from the previous pattern. That is, we form the pattern at level k by adding an "on" position to the grid pattern used for level $k - 1$. Thus, if a pixel position is on for one grid level, it is on for all higher levels (Figs. 27 and 28). We can minimize the introduction of other visual effects by avoiding symmetrical patterns. With a 3×3 pixel grid, for instance, the third intensity level above zero would be better represented by the pattern in Figure 29(a) than by any of the symmetrical arrangements in Figure 29(b). The symmetrical patterns in this figure would produce either vertical, horizontal, or diagonal streaks in any large area shaded with intensity level 3. For hardcopy output on devices such as film recorders and some printers, isolated pixels are not effectively reproduced. Therefore a grid pattern with a single "on" pixel or with isolated "on" pixels, as in Figure 30, should be avoided.

Halftone-approximation methods can be applied also to increase the number of intensity options on systems that are capable of displaying more than two intensities per pixel. For example, on a grayscale system that can display four intensity values per pixel, we can use 2×2 pixel grids to represent 13 different intensity levels. Figure 31 illustrates one way to set up the 13 pixel-grid patterns, where each pixel can be set to intensity level 0, 1, 2, or 3.

Similarly, we can use pixel-grid patterns to increase the number of intensities that can be represented on a color system. A three-bit-per-pixel RGB system, for example, uses one bit per pixel for each color gun. Thus, a pixel is displayed with three phosphor dots, so that the pixel can be assigned any one of eight different

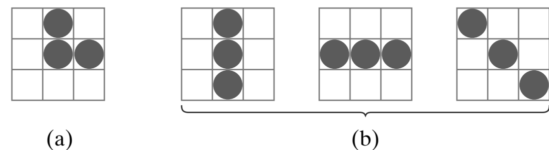


FIGURE 29
For a 3×3 pixel grid, the pattern in (a) is better than any of the symmetrical patterns in (b) for representing the third intensity level above 0.

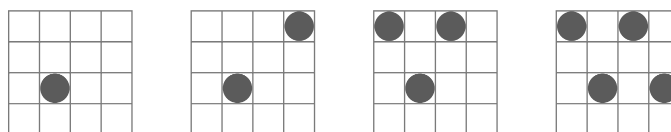


FIGURE 30
Halftone grid patterns with isolated pixels that cannot be reproduced effectively on some hardcopy devices.

https://hemanthrajhemu.github.io

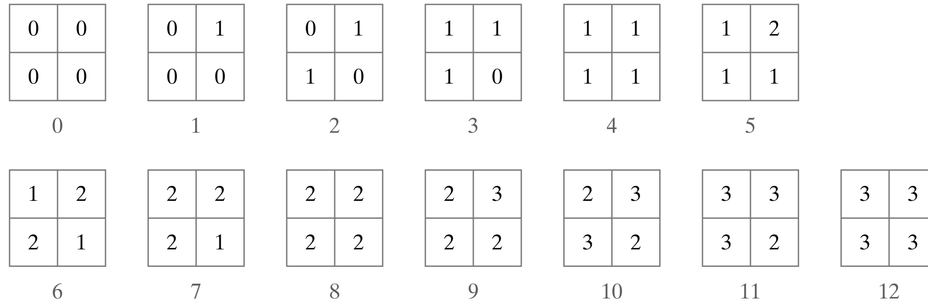


FIGURE 31
Intensity representations 0 through 12 obtained with halftone- approximation patterns using 2 × 2 pixel grids on a four-level system, with pixel-intensity levels labeled 0 through 3.

colors (including black and white). But with 2 × 2 pixel-grid patterns, we have 12 phosphor dots that we can use to represent a color, as shown in Figure 32. The red electron gun can activate any combination of the four red dots in the grid pattern, and this provides five possible settings for the red color of the pattern. The same is true for the green and blue guns, which gives us a total of 125 different color combinations that can be represented with our 2 × 2 grid patterns.

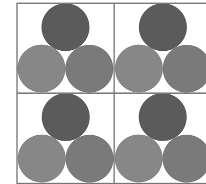


FIGURE 32
A 2 × 2 pixel-grid pattern for displaying RGB colors.

Dithering Techniques

The term **dithering** is used in various contexts. Primarily, it refers to techniques for approximating halftones without reducing resolution, as pixel-grid patterns do. However, dithering is sometimes used also as a synonym for any halftone-approximation scheme, and sometimes it is used as another term for color halftone approximations.

Random values added to pixel intensities to break up contours are often referred to as **dither noise.** Various algorithms have been used to generate the random distributions. The effect is to add noise over an entire picture, which tends to soften intensity boundaries.

A method called **ordered dither** generates intensity variations with a one-to-one mapping of points in a scene to pixel positions using a **dither matrix** $\mathbf{D}_n$ to select an intensity level. Matrix $\mathbf{D}_n$ contains n x n elements that are assigned distinct positive integer values in the range from 0 to $n^2 - 1$. For example, we can generate four intensity levels with

$$\mathbf{D}_2 = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \tag{45}$$

and we can generate nine intensity levels with

$$\mathbf{D}_3 = \begin{bmatrix} 7 & 2 & 6 \\ 4 & 0 & 1 \\ 3 & 8 & 5 \end{bmatrix} \tag{46}$$

The matrix elements for $\mathbf{D}_2$ and $\mathbf{D}_3$ are in the same order as the pixel mask for setting up 2 × 2 and 3 × 3 pixel grids, respectively. With a bilevel system, we determine the display intensity values by comparing input intensities to the matrix elements. Each input intensity is first scaled to the range $0 \leq I \leq n^2$. If the intensity I is to be applied to screen position (x, y), we calculate the reference position (row and column) in the dither matrix as

$$j = (x \mod n) + 1, \qquad k = (y \mod n) + 1 \tag{47}$$

If $I > \mathbf{D}_n(j, k)$, we turn on the pixel at position (x, y). Otherwise, the pixel is off. For RGB color applications, this procedure is implemented for the intensity of each of the individual color components (red, green, and blue).

https://hemanthrajhemu.github.io

Elements of the dither matrix are assigned in accordance with the guidelines discussed for pixel grids. That is, we want to minimize artificial visual effects, such as contouring. Order dither produces constant intensity areas identical to those generated with pixel-grid patterns when the values of the matrix elements correspond to those in the halftone-approximation grid mask. Variations from the pixel-grid displays occur at the boundary of two different intensity areas.

Typically, the number of intensity levels is taken to be a multiple of 2. Higher-order dither matrices, $n \geq 4$, are then obtained from lower-order matrices using the recurrence relation

$$\mathbf{D}_n = \begin{bmatrix} 4\mathbf{D}_{n/2} + \mathbf{D}_2(1,1)\,\mathbf{U}_{n/2} & 4\mathbf{D}_{n/2} + \mathbf{D}_2(1,2)\,U_{n/2} \\ 4\mathbf{D}_{n/2} + \mathbf{D}_2(2,1)\,U_{n/2} & 4\mathbf{D}_{n/2} + \mathbf{D}_2(2,2)\,U_{n/2} \end{bmatrix} \tag{48}$$

Parameter $\mathbf{U}_{n/2}$ represents the "unity" matrix (all elements are 1). For example, if $\mathbf{D}_2$ is specified as in Equation 45, then recurrence relation 48 yields

$$\mathbf{D}_4 = \begin{bmatrix} 15 & 7 & 13 & 5 \\ 3 & 11 & 1 & 9 \\ 12 & 4 & 10 & 6 \\ 0 & 8 & 2 & 10 \end{bmatrix} \tag{49}$$

Another method for mapping a picture with $m \times n$ points to a display area with $m \times n$ pixels is **error diffusion.** Here, the error between an input intensity value and the selected intensity level at a given pixel position is dispersed, or diffused, to pixel positions to the right and below the current pixel position. Starting with a matrix **M** of intensity values obtained by scanning a photograph, we want to construct an array **I** of pixel intensity values for an area of the screen. We do this by first scanning across the rows of **M**, from left to right, starting with the top row, and determining the nearest available pixel-intensity level for each element of **M**. Then the error between the value stored in matrix **M** and the displayed intensity level at each pixel position is distributed to neighboring elements using the following simplified algorithm:

```
for (j = 0; j < m; j++)
   for (k = 0; k < n; k++) {
      /*  Determine the available system intensity value
       *  that is closest to the value of M [j][k] and
       *  assign this value to I [j][k].
       */
      error = M [j][k] - I [j][k];
      I [j][k+1]   = M [j][k+1]   + alpha * error;
      I [j+1][k-1] = M [j+1][k-1] + beta  * error;
      I [j+1][k]   = M [j+1][k]   + gamma * error;
      I [j+1][k+1] = M [j+1][k+1] + delta * error;
   }
```

Once the elements of matrix **I** have been assigned intensity-level values, we then map the matrix to an area of a display device such as a printer or video monitor. Of course, we cannot disperse the error past the last matrix column ($k = n$) or below the last matrix row ($j = m$), and for a bilevel system, the system intensity values are just 0 and 1. Parameters for distributing the error can be chosen to satisfy the following relationship:

$$\alpha + \beta + \gamma + \delta \leq 1 \tag{50}$$

https://hemanthrajhemu.github.io

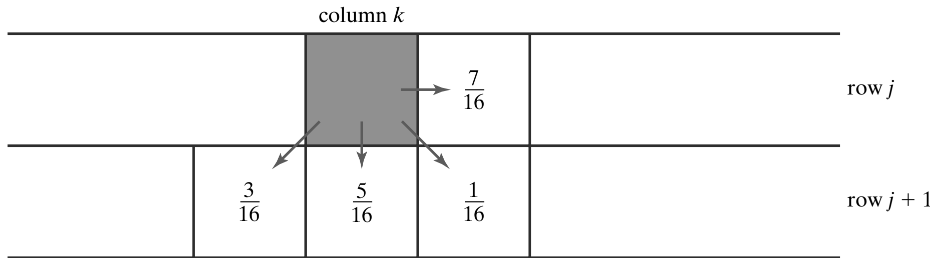


FIGURE 33 Fraction of intensity error that can be distributed to neighboring pixel positions using an error-diffusion scheme.

34	48	40	32	29	15	23	31
42	58	56	53	21	5	7	10
50	62	61	45	13	1	2	18
38	46	54	37	25	17	9	26
28	14	22	30	35	49	41	33
20	4	6	11	43	59	57	52
12	0	3	19	51	63	60	44
24	16	8	27	39	47	55	36

FIGURE 34 One possible distribution scheme for dividing the intensity array into 64 dot-diffusion classes numbered from 0 through 63.

One choice for the error-diffusion parameters that produces fairly good results is $(\alpha, \beta, \gamma, \delta) = (\frac{7}{16}, \frac{3}{16}, \frac{5}{16}, \frac{1}{16})$. Figure 33 illustrates the error distribution using these parameter values. Error diffusion sometimes produces "ghosts" in a picture by repeating, or echoing, certain parts of the picture, particularly with facial features such as hairlines and nose outlines. Ghosting can often be reduced in these cases by choosing values for the error-diffusion parameters that sum to a value less than 1 and by rescaling the matrix values after the dispersion of errors. One way to rescale is to multiply all matrix elements by 0.8 and then add 0.1. Another method for improving picture quality is to alternate the scanning of matrix rows from right to left and left to right.

A variation on the error-diffusion method is **dot diffusion.** In this method, the $m \times n$ array of intensity values is divided into 64 classes numbered from 0 to 63, as shown in Figure 34. The error between a matrix value and the displayed intensity is then distributed only to those neighboring matrix elements that have a larger class number. Distribution of the 64 class numbers is based on minimizing the number of elements that are completely surrounded by elements with a lower class number, because this would tend to direct all errors of the surrounding elements to that one position.

10 Polygon Rendering Methods

Intensity calculations from an illumination model can be applied to surface rendering in various ways. We could use an illumination model to determine the surface intensity at every projected pixel position, or we could apply the illumination model to a few selected points and approximate the intensity at the other surface positions. Graphics packages typically perform surface rendering using scan-line algorithms that reduce processing time by dealing only with polygon surfaces and by calculating surface intensity only at the vertices. The vertex intensities are then

https://hemanthrajhemu.github.io

interpolated to the other positions on the polygon surface. Other, more accurate polygon scan-line rendering methods have been developed, and ray-tracing algorithms calculate the intensity at each projected surface point for curved or planar surfaces. In this section, we consider the scan-line surface-rendering schemes that are applied to polygons.

Constant-Intensity Surface Rendering

The simplest method for rendering a polygon surface is to assign the same color to all projected surface positions. In this case, we use the illumination model to determine the intensity for the three RGB color components at a single surface position, such as a vertex or the polygon centroid. This approach, called **constant-intensity surface rendering** or **flat surface rendering,** provides a fast and simple method for displaying polygon facets on an object, which can be useful for quickly generating the general appearance of a curved surface, as in Color Plate 17(b). Flat rendering is also useful in design or other applications where we might want quickly to identify the individual polygonal facets used to model a curved surface.

In general, flat surface rendering of a polygon provides an accurate display of the surface if all of the following assumptions are valid:

- The polygon is one face of a polyhedron and not a section of a curved-surface approximation mesh.
- All light sources illuminating the polygon are sufficiently far from the surface that $\mathbf{N} \cdot \mathbf{L}$ and the attenuation function are constant over the area of the polygon.
- The viewing position is sufficiently far from the polygon that $\mathbf{V} \cdot \mathbf{R}$ is constant over the area of the polygon.

Even if some of these conditions are not true, we can still reasonably approximate surface lighting effects using constant-intensity surface rendering if the polygon facets of an object are small.

Gouraud Surface Rendering

The **Gouraud surface rendering** scheme, devised by Henri Gouraud and also referred to as **intensity-interpolation surface rendering,** linearly interpolates vertex intensity values across the polygon faces of an illuminated object. Developed for rendering a curved surface that is approximated with a polygon mesh, the Gouraud method smoothly transitions the intensity values for each polygon facet into the values for adjacent polygons along the common edges. This interpolation of intensities across the polygon area eliminates the intensity discontinuities that can occur in flat surface rendering.

Each polygon section of a tessellated curved surface is processed by the Gouraud surface-rendering method using the following procedures:

1. Determine the average unit normal vector at each vertex of the polygon.
2. Apply an illumination model at each polygon vertex to obtain the light intensity at that position.
3. Interpolate the vertex intensities linearly over the projected area of the polygon.

https://hemanthrajhemu.github.io

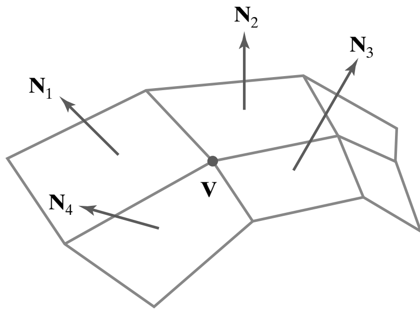


FIGURE 35
The normal vector at vertex **V** is calculated as the average of the surface normals for each polygon sharing that vertex.

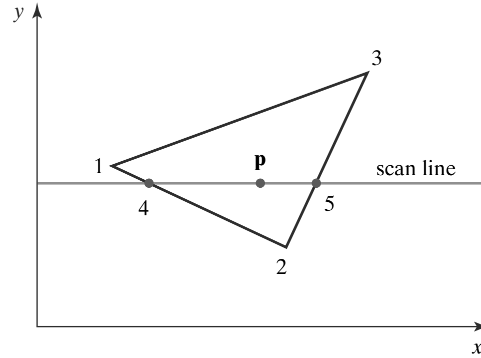


FIGURE 36
For Gouraud surface rendering, the intensity at point 4 is linearly interpolated from the intensities at vertices 1 and 2. The intensity at point 5 is linearly interpolated from intensities at vertices 2 and 3. An interior point **p** is then assigned an intensity value that is linearly interpolated from intensities at positions 4 and 5.

At each polygon vertex, we obtain a normal vector by averaging the normal vectors of all polygons in the surface mesh that share that vertex, as illustrated in Figure 35. Thus, for any vertex position **V**, we obtain the unit vertex normal with the calculation

$$\mathbf{N}_V = \frac{\sum_{k=1}^{n} \mathbf{N}_k}{\left|\sum_{k=1}^{n} \mathbf{N}_k\right|} \tag{51}$$

Once we have obtained the normal vector at a vertex, we invoke the illumination model to obtain the surface intensity at that point.

After all vertex intensities have been computed for a polygonal facet, we can interpolate the vertex values to obtain the intensities at positions along scan lines that intersect the projected area of the polygon, as demonstrated in Figure 36. For each scan line, the intensity at the intersection of the scan line with a polygon edge is linearly interpolated from the intensities at the endpoints of that edge. For the example in Figure 36, the polygon edge with endpoint vertices at positions 1 and 2 is intersected by the scan line at point 4. A fast method for obtaining the intensity at point 4 is to interpolate between the values at vertices 1 and 2 using only the vertical displacement of the scan line, as follows:

$$I_4 = \frac{y_4 - y_2}{y_1 - y_2}\, I_1 + \frac{y_1 - y_4}{y_1 - y_2}\, I_2 \tag{52}$$

In this expression, the symbol I represents the intensity for one of the RGB color components. Similarly, the intensity at the right intersection of this scan line (point 5) is interpolated from intensity values at vertices 2 and 3. From these two boundary intensities, we linearly interpolate to obtain the pixel intensities for positions across the scan line. The intensity for one of the RGB color components at point **p** in Figure 36, for instance, is calculated from the intensities at points 4 and 5 as

$$I_p = \frac{x_5 - x_p}{x_5 - x_4}\, I_4 + \frac{x_p - x_4}{x_5 - x_4}\, I_5 \tag{53}$$

In the implementation of Gouraud rendering, we can perform the intensity calculations represented by Equations 52 and 53 efficiently by using

https://hemanthrajhemu.github.io

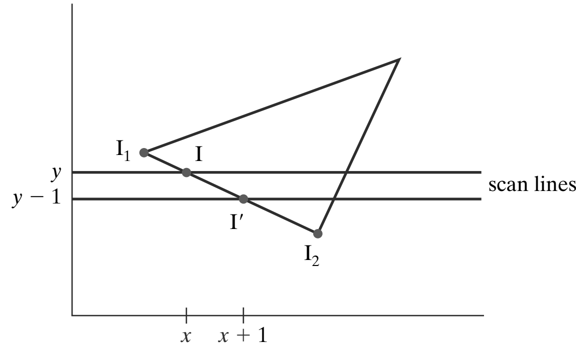


FIGURE 37
Incremental interpolation of intensity values along a polygon edge for successive scan lines.

incremental methods. Starting from a scan line that intersects one of the polygon vertices, we can incrementally obtain intensity values for other scan lines that intersect an edge that is connected to that vertex. Assuming that the polygon facets are convex, each scan line crossing the polygon has two edge intersections, such as points 4 and 5 in Figure 36. Once we have obtained the intensities at the two edge intersections for a scan line, we apply the incremental procedures to obtain pixel intensities across the scan line.

As an example of the incremental calculation of intensities, we consider scan lines y and $y-1$ in Figure 37, which intersect the left edge of a polygon. If scan line y is the next scan line below the vertex at y_1 with intensity I_1, that is $y = y_1 - 1$, then we can compute the intensity I on scan line y from Equation 52 as

$$I = I_1 + \frac{I_2 - I_1}{y_1 - y_2} \tag{54}$$

Continuing down the polygon edge, the intensity along this edge for the next scan line, $y - 1$, is

$$I' = I + \frac{I_2 - I_1}{y_1 - y_2} \tag{55}$$

Thus, each successive intensity value down the edge is computed simply by adding the constant term $(I_2 - I_1)/(y_1 - y_2)$ to the previous intensity value. Similar incremental calculations are used to obtain intensities at successive horizontal pixel positions along each scan line.

Gouraud surface rendering can be combined with a hidden-surface algorithm to fill in the visible polygons along each scan line. An example of a three-dimensional object rendered with the Gouraud method appears in Color Plate 17(c).

This intensity-interpolation method eliminates the discontinuities associated with flat rendering, but it has some other deficiencies. Highlights on the surface are sometimes displayed with anomalous shapes, and the linear intensity interpolation can cause bright or dark intensity streaks, called **Mach bands,** to appear on the surface. These effects can be reduced by dividing the surface into a greater number of polygon faces or by using more precise intensity calculations.

Phong Surface Rendering

A more accurate interpolation method for rendering a polygon mesh was subsequently developed by Phong Bui Tuong. This approach, called **Phong surface rendering** or **normal-vector interpolation rendering,** interpolates normal vectors instead of intensity values. The result is a more accurate calculation of intensity values, a more realistic display of surface highlights, and a great reduction in the

https://hemanthrajhemu.github.io

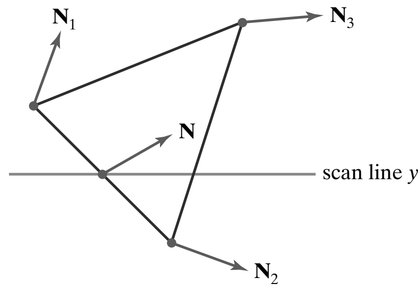


FIGURE 38
Interpolation of surface normals along a polygon edge.

Mach-band effect. However, the Phong method requires more computation than the Gouraud method.

Each polygon section of a tessellated curved surface is processed by the Phong surface-rendering method using the following procedures:

1. Determine the average unit normal vector at each vertex of the polygon.
2. Interpolate the vertex normals linearly over the projected area of the polygon.
3. Apply an illumination model at positions along scan lines to calculate pixel intensities using the interpolated normal vectors.

Interpolation procedures for normal vectors in the Phong method are the same as those for the intensity values in the Gouraud method. The normal vector **N** in Figure 38 is interpolated vertically from the normal vectors at vertices 1 and 2 as

$$\mathbf{N} = \frac{y - y_2}{y_1 - y_2}\mathbf{N}_1 + \frac{y_1 - y}{y_1 - y_2}\mathbf{N}_2 \tag{56}$$

This result must be re-normalized before we perform our shading calculations. We apply the same incremental methods for obtaining normal vectors on successive scan lines and at successive pixel positions along scan lines. The difference between the two surface-rendering approaches is that we must now apply the illumination model at every projected pixel position along the scan lines to obtain the surface intensity values.

Fast Phong Surface Rendering

We can reduce processing time in the Phong-rendering method by approximating some of the illumination-model calculations. **Fast Phong surface rendering** performs the intensity calculations using a truncated Taylor-series expansion and limiting the polygon facets to triangular surface patches.

Because the Phong method interpolates normal vectors from the vertex normals, we can write the expression for calculating the surface normal **N** at position (x, y) in a triangular patch as

$$\mathbf{N} = \mathbf{A}x + \mathbf{B}y + \mathbf{C} \tag{57}$$

where vectors **A**, **B**, and **C** are determined from the three vertex equations:

$$\mathbf{N}_k = \mathbf{A}x_k + \mathbf{B}y_k + \mathbf{C}, \qquad \text{for } k = 1, 2, 3 \tag{58}$$

with (x_k, y_k) denoting a projected triangle vertex position on the pixel plane.

https://hemanthrajhemu.github.io

Omitting the reflectivity and attentuation parameters, we can write the calculation for light-source diffuse reflection from a surface point (x, y) as

$$I_{\text{diff}}(x, y) = \frac{\mathbf{L} \cdot \mathbf{N}}{|\mathbf{L}||\mathbf{N}|}$$

$$= \frac{\mathbf{L} \cdot (\mathbf{A}x + \mathbf{B}y + \mathbf{C})}{|\mathbf{L}||\mathbf{A}x + \mathbf{B}y + \mathbf{C}|}$$

$$= \frac{(\mathbf{L} \cdot \mathbf{A})x + (\mathbf{L} \cdot \mathbf{B})y + \mathbf{L} \cdot \mathbf{C}}{|\mathbf{L}||\mathbf{A}x + \mathbf{B}y + \mathbf{C}|} \tag{59}$$

This expression can be written in the form

$$I_{\text{diff}}(x, y) = \frac{ax + by + c}{[dx^2 + exy + fy^2 + gx + hy + i]^{1/2}} \tag{60}$$

where parameters such as a, b, c, and d are used to represent the various dot products. For example,

$$a = \frac{\mathbf{L} \cdot \mathbf{A}}{|\mathbf{L}|} \tag{61}$$

Finally, we can express the denominator in Equation 60 as a Taylor series expansion and retain terms up to the second degree in x and y. This yields

$$I_{\text{diff}}(x, y) = T_5\, x^2 + T_4\, xy + T_3\, y^2 + T_2\, x + T_1\, y + T_0 \tag{62}$$

where each T_k is a function of the various parameters in Equation 60, such as a, b, and c.

Using forward differences, we then evaluate Equation 62 with only two additions for each pixel position (x, y) once the initial forward-difference parameters have been evaluated. Although the simplifications in the fast-Phong approach reduce the Phong surface-rendering calculations, it still takes approximately twice as long to render a surface with the fast-Phong method as it does with Gouraud surface rendering. And the basic Phong method, using forward-difference calculations, takes about 6 to 7 times longer than Gouraud rendering.

Fast-Phong rendering for diffuse reflection can be extended to include specular reflections, using similar approximations for evaluating the specular terms such as $(\mathbf{N} \cdot \mathbf{H})^{n_s}$. In addition, we can generalize the algorithm to include a finite viewing position and polygons other than triangles.

11 OpenGL Illumination and Surface-Rendering Functions

A variety of routines are available in OpenGL for setting up point light sources, selecting surface-reflection coefficients, and choosing values for other parameters in the basic illumination model. In addition, we can simulate transparency, and objects can be displayed using either flat surface rendering or Gouraud surface rendering.

OpenGL Point Light-Source Function

Multiple point light sources can be included in an OpenGL scene description, and various properties, such as position, type, color, attenuation, and spotlight effects,

https://hemanthrajhemu.github.io

are associated with each light source. We set a property value for a light source with the function

```
glLight* (lightName, lightProperty, propertyValue);
```

A suffix code of `i` or `f` is appended to the function name, depending on the data type of the property value. For vector data, the suffix code `v` is also appended and parameter `propertyValue` is then a pointer to an array. Each light source is referenced with an identifier, and parameter `lightName` is assigned one of the OpenGL symbolic identifiers `GL_LIGHT0`, `GL_LIGHT1`, `GL_LIGHT2`, . . . , `GL_LIGHT7`, although some implementations of OpenGL may allow more than 8 light sources. Similarly, parameter `lightProperty` must be assigned one of the 10 OpenGL symbolic property constants. After all properties have been assigned to a light source, we turn on that light with the command

```
glEnable (lightName);
```

However, we also need to activate the OpenGL lighting routines, and we do that with the command

```
glEnable (GL_LIGHTING);
```

Object surfaces are then rendered using lighting calculations that include contributions from each light source that has been enabled.

Specifying an OpenGL Light-Source Position and Type

The OpenGL symbolic property constant for designating a light-source position is `GL_POSITION`. Actually, this symbolic constant is used to set two light-source properties at the same time: the light-source position and the *light-source type*. Two general classifications of light sources are available in OpenGL to illuminate a scene. A point light source can be classified as near the objects to be illuminated (a local source), or it can be treated as if it were infinitely far from the scene. And this classification is independent of the position that we assign to a light source. For a nearby light source, the emitted light radiates in all directions, and the position of the light source is included in the lighting calculations. However, the emitted light from a distant source is allowed to emanate in one direction only, and this direction is applied to all surfaces in the scene, independently of the assigned light-source position. The direction for the emitted rays from a light that is classified as a distant source is calculated as the direction from the assigned position of the light source to the coordinate origin.

A four-element floating-point vector is used to designate both the type of light and the coordinate values for the light position. The first three elements of this vector give the world-coordinate position, and the fourth element is used to designate the light-source type. If we assign the value 0.0 to the fourth element of the position vector, the light is considered to be a very distant source (referred to in OpenGL as a “directional” light), and the light-source position is then used only to determine the light direction. Otherwise, the light is taken to be a local point source (referred to in OpenGL as a “positional” light), and the light position is used by the lighting routines to determine the light direction to each object in the scene. In the following code example, light 1 is designated as a local source

https://hemanthrajhemu.github.io

at location (2.0, 0.0, 3.0), and light 2 is a distant source with light emission in the negative y direction:

```
GLfloat light1PosType [ ] = {2.0, 0.0, 3.0, 1.0};
GLfloat light2PosType [ ] = {0.0, 1.0, 0.0, 0.0};

glLightfv (GL_LIGHT1, GL_POSITION, light1PosType);
glEnable (GL_LIGHT1);

glLightfv (GL_LIGHT2, GL_POSITION, light2PosType);
glEnable (GL_LIGHT2);
```

If we do not specify a position and type for a light source, the default values are (0.0, 0.0, 1.0, 0.0), which indicates a distant source with light rays traveling in the negative z direction.

The position of a light source is included in the scene description, and it is transformed to viewing coordinates along with the object positions by the OpenGL geometric-transformation and viewing-transformation matrices. Therefore, if we want to keep the light source at a fixed position relative to the objects in a scene, we set its position after the specification of the geometric and viewing transformations in the program. However, if we want the light source to move as the viewpoint moves, we set its position before the specification of the viewing transformation. Also, we can apply a translation or rotation to a light source to move it around in a stationary scene.

Specifying OpenGL Light-Source Colors

Unlike an actual light source, an OpenGL light has three different color properties. In this empirical scheme, the three light-source colors provide options for varying the lighting effects in a scene. We set these colors using the symbolic color-property constants GL_AMBIENT, GL_DIFFUSE, and GL_SPECULAR. Each of these colors is assigned by specifying a four-element floating-point set of values representing the red, green, blue, and alpha (RGBA) components of the color, specified in that order. The alpha component controls color-blending, and is used only if the OpenGL color-blending routines are activated. As we might guess from the names of the symbolic color-property constants, one of the light-source colors contributes to the background (ambient) light in a scene, another color is used in diffuse-lighting calculations, and the third color is used to compute specular-lighting effects for a surface. Realistically, a light source has just one color, but we can use the three OpenGL light-source colors to create various lighting effects. In the following code example, we set the ambient color for a local light source, labeled GL_LIGHT3, to black, and we set the diffuse and specular colors to white:

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

glLightfv (GL_LIGHT3, GL_AMBIENT, blackColor);
glLightfv (GL_LIGHT3, GL_DIFFUSE, whiteColor);
glLightfv (GL_LIGHT3, GL_SPECULAR, whiteColor);
```

The default colors for light source 0 are black for the ambient color and white for the diffuse and specular colors. All the other light sources have a default color of black for each of the ambient, diffuse, and specular color properties.

https://hemanthrajhemu.github.io

Specifying Radial-Intensity Attenuation Coefficients for an OpenGL Light Source

We can apply radial-intensity attenuation to the light emitted from an OpenGL local light source, and the OpenGL lighting routines calculate this attenuation using Equation 2, with d_l as the distance from a light-source position to an object position. The three OpenGL property constants for radial intensity attenuation are GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, and GL_QUADRATIC_ATTENUATION, which correspond to the coefficients a_0, a_1, and a_2 in Equation 2. Either a positive integer value or a positive floating-point value can be used to set each attenuation coefficient. For example, we could assign the radial-attenuation coefficient values as

```
glLightf (GL_LIGHT6, GL_CONSTANT_ATTENUATION, 1.5);
glLightf (GL_LIGHT6, GL_LINEAR_ATTENUATION, 0.75);
glLightf (GL_LIGHT6, GL_QUADRATIC_ATTENUATION, 0.4);
```

Once the values for the attenuation coefficients have been set, the radial attenuation function is applied to all three colors (ambient, diffuse, and specular) of the light source. Default values for the attenuation coefficients are $a_0 = 1.0$, $a_1 = 0.0$, and $a_2 = 0.0$. Thus, the default is no radial attenuation: $f_{l,\text{radatten}} = 1.0$. Although radial attenuation can produce more realistic displays, the calculations are time-consuming.

OpenGL Directional Light Sources (Spotlights)

For local light sources (those not considered to be at infinity), we can also specify a directional, or spotlight, effect. This limits the light that is emitted from a source to a cone-shaped region of space. We define the conical region with a direction vector along the axis of the cone and an angular spread θ_l from the cone axis, as shown in Figure 39. In addition, we can specify an angular-attenuation exponent a_l for the light source that determines how much the light intensity decreases as we move from the center of the cone toward the cone surface. Along any direction within the light cone, the angular attenuation factor is $\cos^{a_l} \alpha$ (Eq. 5), where $\cos \alpha$ is calculated as the dot product of the cone axis vector and the vector from the light source to an object position. We compute the value for each of the ambient, diffuse, and specular light colors at angle α by multiplying the intensity components by this angular attenuation factor. If $\alpha > \theta_l$, the object is outside the light-source cone, and the object is not illuminated by this light source. For light rays within the cone, we can also attenuate the intensity values radially.

There are three OpenGL property constants for directional effects: GL_SPOT_DIRECTION, GL_SPOT_CUTOFF, and GL_SPOT_EXPONENT. We specify the light direction as either an integer or a floating-point world-coordinate vector. The cone angle θ_l is given as an integer or floating-point value in degrees, and this angle can be either 180° or any value in the range from 0° to 90°. When the cone angle is set to 180°, the light source emits rays in all directions (360°). We set the exponent value for intensity attenuation either as an integer or floating-point number in the range from 0 to 128. The following statements set the directional effects for light source 3 so that the cone axis is in the positive x direction, the cone angle θ_l is 30°, and the attenuation exponent is 2.5:

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

glLightfv (GL_LIGHT3, GL_SPOT_DIRECTION, dirVector);
glLightf (GL_LIGHT3, GL_SPOT_CUTOFF, 30.0);
glLightf (GL_LIGHT3, GL_SPOT_EXPONENT, 2.5);
```


https://hemanthrajhemu.github.io

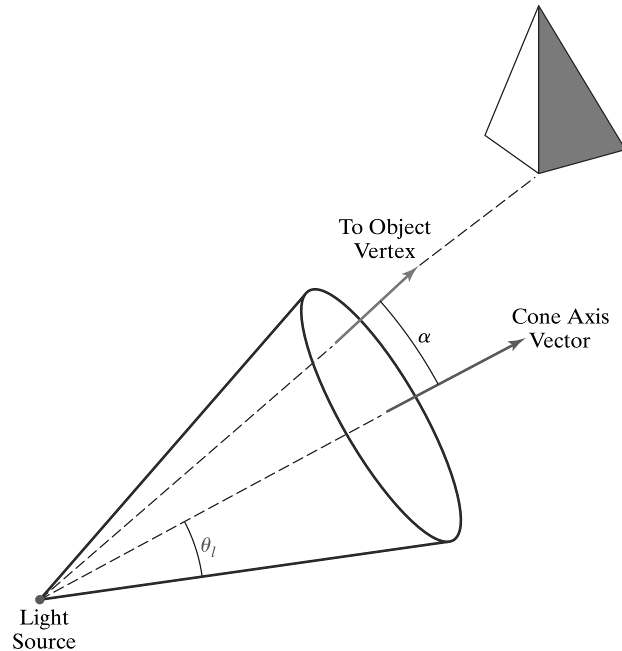


FIGURE 39
A circular cone of light emitted from an OpenGL light source. The angular extent of the light cone, measured from the cone axis, is θ_l, and the angle from the axis to an object direction vector is labeled as α.

If we do not specify a direction for a light source, the default direction is parallel to the negative z axis; that is, (0.0, 0.0, −1.0). Also, the default cone angle is 180° and the default attenuation exponent is 0. Thus, the default is a point light source that radiates in all directions, with no angular attenuation.

OpenGL Global Lighting Parameters

Several OpenGL lighting parameters can be specified at the global level. These values are used to control the way that some lighting calculations are performed, and a global parameter value is set with the following function:

```
glLightModel* (paramName, paramValue);
```

We append a suffix code of `i` or `f`, depending on the data type of the parameter value. In addition, for vector data, we append the suffix code `v`. Parameter `paramName` is assigned an OpenGL symbolic constant that identifies the global property to be set, and parameter `paramValue` is assigned a single value or set of values. Using the `glLightModel` function, we can set a global ambient-light level, we can specify how specular highlights are to be calculated, and we can choose to apply the illumination model to the back faces of polygon surfaces.

In addition to the ambient color for individual light sources, we can set an independent value for the OpenGL background lighting as a global value. This provides just one more option in the empirical lighting calculations. To set this option, we use the symbolic constant `GL_LIGHT_MODEL_AMBIENT`. The following statement, for example, sets the general background lighting for a scene to a low-intensity (dark) blue color, with an alpha value of 1.0:

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

glLightModelfv (GL_LIGHT_MODEL_AMBIENT, globalAmbient);
```


https://hemanthrajhemu.github.io

If we do not set a global ambient-light level, the default is the low-intensity white (dark gray) color (0.2, 0.2, 0.2, 1.0).

Specular-reflection calculations require the determination of several vectors, including the vector **V** from a surface position to the viewing position. To speed up specular calculations, the OpenGL lighting routines can use a constant direction for vector **V**, regardless of the surface position relative to the view point. This constant unit vector is in the positive z direction, (0.0, 0.0, 1.0), and this is the default value for **V**. However, if we want to turn off this default and use the actual viewing position (which is the viewing-coordinate origin) to calculate **V**, we issue the following command:

```
glLightModeli (GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

Although the specular calculations take more time when we use the actual viewing position to calculate **V**, we do obtain more realistic displays. We turn off the surface calculations for vector **V** when we use the default value `GL_FALSE` (or 0, or 0.0) for the local-viewer parameter.

When surface textures are added to the OpenGL lighting calculations, surface highlights can be dulled and the texture patterns may be distorted by the specular terms. Therefore, as an option, texture patterns can be applied only to the nonspecular terms that contribute to a surface color. These nonspecular terms include ambient effects, surface emissions, and diffuse reflections. Using this option, the OpenGL lighting routines generate two colors for each surface lighting calculation: a specular color and the nonspecular color contributions. Texture patterns are combined only with the nonspecular color, and then the two colors are combined. We select this two-color option with

```
glLightModeli (GL_LIGHT_MODEL_COLOR_CONTROL,
                    GL_SEPARATE_SPECULAR_COLOR);
```

We need not separate the color terms if we are not using texture patterns, and the lighting calculations are performed more efficiently if this option is not invoked. The default value for this property is `GL_SINGLE_COLOR`, which does not separate the specular color from the other surface-color components.

In some applications, we may want to display back-facing surfaces of an object. An example is the inside, cutaway view of a solid, in which some back-facing surfaces, in addition to the front-facing surfaces, are to be displayed. However, by default, the lighting calculations use the assigned material properties only for the front faces. To apply the lighting calculations to both the front and back faces, using the corresponding front-face and back-face material properties, we issue the command

```
glLightModeli (GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

The surface normal vectors for the back faces are then reversed, and the lighting calculations are applied using the material properties that have been assigned to the back faces. To turn off the two-sided lighting calculations, we use the value `GL_FALSE` (or 0, or 0.0) in the `glLightModel` function, which is the default.

OpenGL Surface-Property Function

Reflection coefficients and other optical properties for surfaces are set using the function

```
glMaterial* (surfFace, surfProperty, propertyValue);
```


https://hemanthrajhemu.github.io

A suffix code of i or f is appended to the function, depending on the data type for the property value, and we also append the code v when we supply vector-valued properties. Parameter surfFace is assigned one of the symbolic constants GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK; parameter surfProperty is a symbolic constant identifying a surface parameter such as I_{surf}, k_a, k_d, k_s, or n_s; and parameter propertyValue is set to the corresponding value. All properties except the specular-reflection exponent n_s are specified as vector values. We use a sequence of glMaterial functions to set all the illumination properties for an object before we issue the commands that describe the object geometry.

An RGBA value for the surface emission color, I_{surf}, is selected using the OpenGL symbolic surface-property constant GL_EMISSION. For example, the following statement sets the emission color for front surfaces to a light gray (off-white):

```
surfEmissionColor [ ] = {0.8, 0.8, 0.8, 1.0};

glMaterialfv (GL_FRONT, GL_EMISSION, surfEmissionColor);
```

The default emission color for a surface is black (0.0, 0.0, 0.0, 1.0). Although an emission color can be assigned to a surface, this emission does not illuminate other objects in the scene. To do that, we must define the surface as a light source using the methods discussed in Section 3.

We use the OpenGL symbolic property names GL_AMBIENT, GL_DIFFUSE, and GL_SPECULAR to set values for the surface reflection coefficients. Realistically, the ambient and diffuse coefficients should be assigned the same vector values, and we can do that using the symbolic constant GL_AMBIENT_AND_DIFFUSE. The default values for the ambient coefficient are (0.2, 0.2, 0.2, 1.0), the default values for the diffuse coefficient are (0.8, 0.8, 0.8, 1.0), and the default values for the specular coefficient are (1.0, 1.0, 1.0, 1.0). To set the specular-reflection exponent, we use the constant GL_SHININESS. We can assign any value in the range from 0 to 128 to this property, and the default value is 0. For example, the following statements set the values for the three reflection coefficients and the specular exponent. The diffuse and ambient coefficients are set so that the surface is displayed as a light-blue color when it is illuminated with white light; specular reflection is the color of the incident light; and the specular exponent is assigned a value of 25.0.

```
 diffuseCoeff [ ] = {0.2, 0.4, 0.9, 1.0};
 specularCoeff [ ] = {1.0, 1.0, 1.0, 1.0};

glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
                 diffuseCoeff);
glMaterialfv (GL_FRONT_AND_BACK, GL_SPECULAR, specularCoeff);
glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 25.0);
```

Components for the reflection coefficients can also be set using color-table values, and the OpenGL symbolic constant GL_COLOR_INDEXES is provided for this purpose. We assign the color-table indices as a three-element integer or floating-point array, and the default is (0, 1, 1).

OpenGL Illumination Model

Surface lighting effects are calculated by OpenGL using the basic illumination model 19, with some variations in the way that the parameters are

https://hemanthrajhemu.github.io

specified. The ambient light level is the sum of the light-source ambient components and the global ambient setting. Diffuse-reflection calculations use the diffuse-intensity component of the light sources, and specular-reflection calculations use the specular-intensity component of each light source.

Also, the unit vector `V`, specifying the direction from a surface position to a viewing position, can be set to the constant value (0.0, 0.0, 0.0) if the local-viewer option is not used. For a light source positioned at “infinity,” the unit light-direction vector `L` is in the opposite direction to the assigned direction for the light rays from that source.

OpenGL Atmospheric Effects

After the OpenGL illumination model has been applied to obtain surface colors, we can assign a color to the atmosphere in a scene and combine the surface colors with the atmosphere color. Also, we can use an atmosphere intensity-attenuation function to simulate viewing the scene through a hazy or smoky atmosphere. The various atmosphere parameters are set using the `glFog` function:

```
glEnable (GL_FOG);

glFog* (atmoParameter, paramValue);
```

A suffix code of `i` or `f` is appended to indicate data-value type, and the suffix code `v` is used with vector data.

To set an atmosphere color, we assign the OpenGL symbolic constant `GL_FOG_COLOR` to parameter `atmoParameter`. For example, we can designate the atmosphere as having a bluish-gray color with

```
GLfloat atmoColor [4] = {0.8, 0.8, 1.0, 1.0};

glFogfv (GL_FOG_COLOR, atmoColor);
```

The default value for the atmosphere color is black (0.0, 0.0, 0.0, 0.0).

We can next choose the atmosphere-attenuation function that is to be used to combine the object color with the atmosphere color. This is accomplished using the symbolic constant `GL_FOG_MODE`:

```
glFogi (GL_FOG_MODE, atmoAttenFunc);
```

If parameter `atmoAttenFunc` is assigned the value `GL_EXP`, Equation 31 is used as the atmosphere-attenuation function. With the value `GL_EXP2`, we select Equation 32 as the atmosphere-attenuation function. For either of the exponential functions, we select an atmosphere density value with

```
glFog (GL_FOG_DENSITY, atmoDensity);
```

A third option for atmospheric attenuation is the linear depth-cueing function. In this case, parameter `atmoAttenFunc` is assigned the value `GL_LINEAR`. The default value for parameter `atmoAttenFunc` is `GL_EXP`.

Once an atmosphere-attenuation function has been selected, this function is used to calculate a blended atmosphere-surface color for the object. Equation 33 is used by the OpenGL atmosphere routines to calculate this blended color.

https://hemanthrajhemu.github.io

OpenGL Transparency Functions

Some simulated transparency effects are possible in OpenGL using color-blending routines. However, the implementation of transparency in an OpenGL program, in general, is not straightforward. We can combine object colors for a simple scene containing a few opaque and transparent surfaces by using the alpha blending value to specify the degree of transparency and by processing surfaces in a depth-first order. The OpenGL color-blending operations ignore refraction effects, however, and dealing with transparent surfaces in complex scenes with a variety of lighting conditions or animations can be formidable. Also, OpenGL provides no direct provisions for simulating the surface appearance of a translucent object (such as a grainy sheet of plastic or a pane of frosted glass), which diffusely scatters the light transmissions through the semitransparent material. Thus, to display translucent surfaces or the lighting effects resulting from refraction, we would need to write our own routines. To simulate lighting effects through a translucent object, we could use a combination of values for surface texture and material properties. For refraction effects, we could shift the pixel positions for surfaces behind a transparent object using Equation 29 to calculate the amount of offset needed.

We designate objects in a scene as transparent using the alpha parameter in the OpenGL RGBA surface-color commands such as `glMaterial` and `glColor`. A surface alpha parameter can be set to the value of the transparency coefficient (Eq. 30) for that object. For example, if we specify the color for a transparent surface with the function

```
glColor4f (R, G, B, A);
```

then we set the alpha parameter to the value $A = k_t$. A completely transparent surface is assigned the alpha value $A = 1.0$, and an opaque surface has the alpha value $A = 0.0$.

Once we have assigned the transparency values, we activate the color-blending features of OpenGL and process the surfaces, starting with the most distant objects and proceeding in order to the objects closest to the viewing position. With color blending activated, each surface color is combined with any overlapping surfaces that are already in the frame buffer, using the assigned surface alpha values.

We set the color-blending factors so that all color components of the current surface (the "source" object) are multiplied by $(1 - A) = (1 - k_t)$, and all color components of the corresponding frame-buffer positions (the "destination") are multiplied by the factor $A = k_t$:

```
glEnable (GL_BLEND);

glBlendFunc (GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);
```

The two colors are then blended using Equation 30 with the alpha parameter set to k_t, where the frame-buffer colors are those for a surface that is behind the transparent object being processed. For instance, if $A = 0.3$, then the new frame-buffer color is the sum of 30 percent of the current frame-buffer color and 70 percent of the object reflection color, for each surface position. (Alternatively, we could use the alpha color parameter as an opacity factor, instead of a transparency factor. If we set A to an opacity value, though, we also must interchange the two arguments in the function `glBlendFunc`.)

Visibility testing can be accomplished using the OpenGL depth-buffer functions. As each visible opaque surface is processed, both the surface colors and the

https://hemanthrajhemu.github.io

surface depth values are stored. However, when we process a visible transparent surface, we want to save only its colors because the surface does not obscure background surfaces. Therefore, as we process each transparent surface, we put the depth buffer into a read-only status using the `glDepthMask` function.

If we process all objects in depth order, the depth-buffer write mode is turned off and then back on again as we process each transparent surface. Alternatively, we could separate the two object classes, as in the following code outline:

```
glEnable (GL_DEPTH_TEST);
/*  Process all opaque surfaces.  */

glEnable (GL_BLEND);
glDepthMask (GL_FALSE);
glBlendFunc (GL_ONE_MINUS_SRC_ALPHA, GL_SRC_ALPHA);
/*  Process all transparent surfaces.  */

glDepthMask (GL_TRUE);
glDisable (GL_BLEND);

glutSwapBuffers ( );
```

If the transparent objects are not processed in a strictly back-to-front order, this approach will not accumulate surface colors accurately for all cases. But for simple scenes, this is a fast and effective method for generating an approximate representation for the transparency effects.

OpenGL Surface-Rendering Functions

Surfaces can be displayed with OpenGL routines using either constant-intensity surface rendering or Gouraud surface rendering. No OpenGL routines are provided for applying Phong surface rendering, ray tracing, or radiosity methods. A rendering method is selected with

```
glShadeModel (surfRenderingMethod);
```

We select constant-intensity surface rendering by assigning the symbolic value `GL_FLAT` to parameter `surfRenderingMethod`. For Gouraud shading (the default), we use the symbolic constant `GL_SMOOTH`.

When the `glShadeModel` function is applied to a tessellated curved surface, such as a sphere that is approximated with a polygon mesh, the OpenGL rendering routines use the surface-normal vectors at the polygon vertices to calculate the polygon color. The Cartesian components of a surface-normal vector in OpenGL are specified with the command

```
glNormal3* (Nx, Ny, Nz);
```

Suffix codes for this function are `b` (byte), `s` (short), `i` (integer), `f` (float), and `d` (double). In addition, we append the suffix code `v` when the vector components are designated with an array. Byte, short, and integer values are converted to floating-point values in the range from −1.0 to 1.0. The `glNormal` function sets the components for the surface-normal vector as state values that apply to all subsequent `glVertex` commands, and the default normal vector is in the positive z direction: (0.0, 0.0, 1.0).

https://hemanthrajhemu.github.io

For flat surface rendering, we need only one surface normal for each polygon. Thus, we can set each polygon normal as, for example,

```
glNormal3fv (normalVector);
glBegin (GL_TRIANGLES);
    glVertex3fv (vertex1);
    glVertex3fv (vertex2);
    glVertex3fv (vertex3);
glEnd ( );
```

If we want to apply the Gouraud surface-rendering procedure to the above triangle, we need to designate a normal vector for each vertex as follows:

```
glBegin (GL_TRIANGLES);
    glNormal3fv (normalVector1);
    glVertex3fv (vertex1);
    glNormal3fv (normalVector2);
    glVertex3fv (vertex2);
    glNormal3fv (normalVector3);
    glVertex3fv (vertex3);
glEnd ( );
```

Although normal vectors need not be specified as unit vectors, we can reduce computations if do state all surface normals as unit vectors. Any non-unit surface normal is converted to a unit normal automatically if we have issued the command

```
glEnable (GL_NORMALIZE);
```

This command also renormalizes surface vectors if they have been modified by geometric transformations such as scaling or shear.

Another available option is the designation of a list of normal vectors that are to be combined or associated with a vertex array. The statements for creating an array of normal vectors are

```
glEnableClientState (GL_NORMAL_ARRAY);

glNormalPointer (dataType, offset, normalArray);
```

Parameter `dataType` is assigned the constant value `GL_BYTE`, `GL_SHORT`, `GL_INT`, `GL_FLOAT` (the default value), or `GL_DOUBLE`. The number of bytes between successive normal vectors in the array `normalArray` is given by parameter `offset`, which has a default value of 0.

OpenGL Halftoning Operations

A variety of colors and gray-scale effects are possible on some systems using OpenGL halftone routines. The halftone-approximation patterns and operations are hardware dependent, and they typically have no effect on systems with full-color graphics capabilities. However, when a system has only a small number of bits per pixel, RGBA color settings can be approximated with halftone patterns. We activate the halftone routines with

```
glEnable (GL_DITHER);
```

which is the default, and the halftoning routines are deactivated with the function

```
glDisable (GL_DITHER);
```


https://hemanthrajhemu.github.io

12 Summary

In general, an object is illuminated with radiant energy from light emitters and from the reflective surfaces in a scene. Light sources can be modeled as point objects or they can have an extended size. In addition, light sources can be directional, and they can be treated as infinitely distant sources or as local light sources. Radial attenuation is typically applied to transmitted light using an inverse quadratic function of distance, and spotlights can be angularly attenuated as well. Reflecting surfaces in a scene are opaque, completely transparent, or partially transparent; and lighting effects are described in terms of diffuse and specular components for both reflections and refractions.

Light intensity at a surface position is calculated using an illumination model, and the basic illumination model in most graphics packages uses simplified approximations of physical laws. These lighting calculations provide a light-intensity value for each RGB component of the reflected light from a surface position, and for the transmitted light through a transparent object. The basic illumination model typically accommodates multiple light sources as point emitters, but they can be distant sources, local sources, or spotlights. Ambient light for a scene is described with a fixed intensity for each RGB color component and for all surfaces. Diffuse-intensity reflections from a surface are taken to be proportional to the cosine of the angular distance from the direction of the surface normal. Specular-intensity reflections are computed using the Phong model. In addition, transparency effects are usually approximated using a simple transparency coefficient for a material, although accurate refraction effects can be modeled using Snell's law. Shadow effects from the individual light sources can be added by identifying the regions in a scene that are not visible from the light source. Also, the calculations necessary for obtaining light reflections and transmission effects for translucent materials are not usually part of a basic illumination model, but we can model them using methods that disperse the diffuse light components.

Intensity values calculated with an illumination model are mapped to the intensity levels available on the display system in use. A logarithmic intensity scale is used by systems to provide a set of intensity levels that increase with equal perceived brightness differentials. Gamma correction is applied to intensity values to correct for the nonlinearity of display devices. With bilevel monitors, we can use halftone patterns and dithering techniques to simulate a range of intensity values. Halftone approximations can also be used to increase the number of intensity options on systems that are capable of displaying more than two intensities per pixel. Ordered-dither, error-diffusion, and dot-diffusion methods are used to simulate a range of intensities when the number of points to be plotted in a scene is equal to the number of pixels on the display device.

Surface rendering in graphics packages is accomplished by applying the calculations from the basic illumination model to scan-line procedures that extrapolate the intensity values from a few surface points to all projected pixel positions of a surface. With constant-intensity surface rendering, also called *flat rendering*, we use one calculated color to display all points of a surface. Flat surface rendering is accurate for polyhedrons or curved-surface polygon meshes when the viewing and light-source positions are far from the objects in a scene. Gouraud surface rendering approximates light reflections from tessellated curved surfaces by calculating intensity values at polygon vertices and linearly interpolating these intensity values across the polygon facets. A more accurate, but slower, surface-rendering procedure is Phong surface rendering, which interpolates the average normal vectors for polygon vertices over the polygon facets. Then, the basic

https://hemanthrajhemu.github.io

illumination model is employed to compute surface intensities at each projected surface position, using the interpolated values for the surface normal vectors. Fast Phong surface rendering uses Taylor series approximations to reduce processing time for the intensity calculations.

The core library of OpenGL contains an extensive set of functions for setting up point light sources, specifying the various parameters in the basic illumination model, selecting a surface-rendering method, activating halftone-approximation routines, and for applying texture array patterns to objects. Table 2 provides a summary of these OpenGL illumination and surface-rendering functions.

TABLE 2

Summary of OpenGL Illumination and Surface-Rendering Functions

Function	Description
`glLight`	Specifies a light-source property value.
`glEnable (lightName)`	Activates a light source.
`glLightModel`	Specifies global-lighting parameter values.
`glMaterial`	Specifies a value for an optical surface parameter.
`glFog`	Specifies a value for an atmosphere parameter; activates atmospheric effects with the `glEnable` function.
`glColor4f (R, G, B, A)`	Specifies an alpha value for a surface to simulate transparency. In the function `glBlendFunc`, sets the source blending factor to `GL_SRC_ALPHA` and the destination blending factor to `GL_ONE_MINUS_SRC_ALPHA`.
`glShadeModel`	Specifies either Gouraud surface rendering or single-color surface rendering.
`glNormal3`	Specifies a surface-normal vector.
`glEnable (GL_NORMALIZE)`	Specifies that surface normals are to be converted to unit vectors.
`glEnableClientState (GL_NORMAL_ARRAY)`	Activates processing routines for an array of surface-normal vectors.
`glNormalPointer`	Creates a list of surface-normal vectors that are to be used with a vertex array.
`glEnable (GL_DITHER)`	Activates operations for applying surface rendering as halftone approximation patterns.

https://hemanthrajhemu.github.io

REFERENCES

Basic illumination models and surface-rendering techniques are discussed in Gouraud (1971) and Phong (1975), Freeman (1980), Bishop and Wiemer (1986), Birn (2000), Akenine-Möller and Haines (2002), and Olano, et al. (2002). Implementation algorithms for illumination models and rendering methods are presented in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), Paeth (1995), and Sakaguchi, et al. (2001). Halftoning methods are given in Velho and Gomes (1991). For further information on ordered dither, error diffusion, and dot diffusion see Knuth (1987).

Additional programming examples using OpenGL illumination and rendering functions are given in Woo, et al. (1999). Programming examples for the OpenGL lighting and rendering functions are also available at Nate Robins's tutorial website: http://www.xmission.com/~nate/opengl.html. Finally, a complete listing of OpenGL illumination and rendering functions is provided in Shreiner (2000).

EXERCISES

1. Write a routine to implement Equation 12 for diffuse reflection using a single point light source and constant surface rendering for the faces of a tetrahedron. The object description is to be given in polygon tables, including surface normal vectors for each of the polygon faces. Additional input parameters include the ambient intensity, light-source intensity, and surface reflection coefficients. All coordinate information can be specified directly in the viewing reference frame.
2. Modify the routine in Exercise 1 to render the polygon facets of a tessellated spherical surface.
3. Modify the routine in Exercise 2 to display the spherical surface using Gouraud surface rendering.
4. Modify the routine in Exercise 3 to display the spherical surface using Phong surface rendering.
5. Use the routines developed in the previous exercises to write a program that displays an input set of objects given as polygon meshes using the same parameters as described in Exercise 1. The program should allow the user to switch between constant surface rendering, Gouraud shading, and Phong shading via keyboard input. Run the program with a sample set of objects and the light source in various positions and examine the visual differences between the three different rendering schemes.
6. Write a routine to implement Equation 17 for diffuse and specular reflections using a single point light source and Gouraud surface rendering for the polygon facets of a tessellated spherical surface. The object description is to be given in polygon tables, including surface normal vectors for each of the polygon faces. Additional input includes values for the ambient intensity, light-source intensity, surface reflection coefficients, and specular-reflection parameter. All coordinate information can be specified directly in the viewing reference frame.
7. Modify the routine in preceding exercise to display the polygon facets using Phong surface rendering.
8. Modify the routine in the preceding exercise to include a linear intensity attenuation function.
9. Modify the routine in the preceding exercise to include two light sources in the scene.
10. Modify the routine in the preceding exercise so that the spherical surface is viewed through a pane of glass.
11. Discuss the differences you might expect to see in the appearance of specular reflections modeled with $(\mathbf{N} \cdot \mathbf{H})^{n_s}$ compared to specular reflections modeled with $(\mathbf{V} \cdot \mathbf{R})^{n_s}$.
12. Verify that $2\alpha = \phi$ in Figure 18 when all vectors are coplanar, but that, in general, $2\alpha \neq \phi$.
13. Discuss how the different visible-surface detection methods can be combined with an intensity model for displaying a set of polyhedrons with opaque surfaces.
14. Discuss how the various visible-surface detection methods can be modified to process transparent objects. Are there any visible-surface detection methods that cannot handle transparent surfaces?
15. Set up an algorithm, based on one of the visible-surface detection methods, that will identify shadow areas in a scene illuminated by a distant point source.
16. How many intensity levels can be displayed with halftone approximations using $n \times n$ pixel grids, where each pixel can be displayed with m different intensities?
17. How many different color combinations can be generated using halftone approximations on a four-level RGB system with a 3 x 3 pixel grid?
18. How many different color combinations can be generated using halftone approximations on a two-level RGB system with a 4 x 4 pixel grid?
19. Write a routine to display a given set of surface-intensity variations using halftone approximations with 4 x 4 pixel grids and two intensity levels (0 and 1) per pixel.
20. Write a routine to generate ordered-dither matrices using the recurrence relation in Equation 48.

https://hemanthrajhemu.github.io

21 Write a procedure to display a given array of intensity values using the ordered-dither method.

22 Write a procedure to implement the error-diffusion algorithm for a given $m \times n$ array of intensity values.

23 Write an OpenGL program to display a scene containing a sphere, a cube, and a tetrahedron illuminated by two light sources: one is to be a local green source and the other a distant white-light source. Set surface parameters for both diffuse and specular reflections with Gouraud surface rendering, and apply a linear intensity-attenuation function.

24 Modify the program in the preceding exercise so that the single local green source is replaced with two spotlights: one green and one blue.

25 Modify the program in the preceding exercise so that a smoky atmosphere is added to the scene.

26 Modify the program in the preceding exercise so that the scene is viewed through a semitransparent pane of glass.

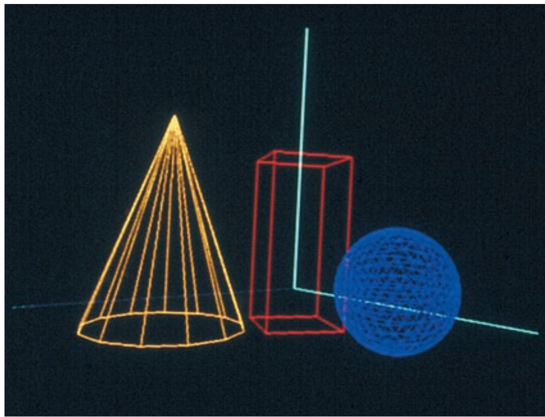
IN MORE DEPTH

1 Using the techniques presented in this chapter, choose a lighting scheme appropriate to your application and write out its specification in detail. Decide what types of light sources are most appropriate (point sources, directional sources, ambient sources, etc.) and what their colors, positions, and orientations should be, if applicable. In addition, include appropriate lighting effects, such as intensity attenuation, shadows, or atmospheric effects where appropriate. Next, choose appropriate surface properties for the objects in your scene based on the material that they represent and include these properties in your specification. If transparency is an important aspect in your scene, be sure to specify the transparency properties of the objects in your scene as well.

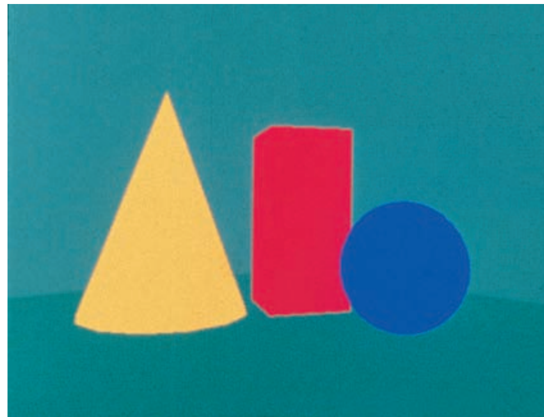
2 Implement the specification that you developed in the previous exercise using the OpenGL illumination and surface-rendering functions. Create and position/orient light sources accordingly within the scene and turn on any atmospheric effects if necessary. If you are using attenuation functions, then experiment with the different models and parameters that define their visual appearance. Next, set the material properties of the surfaces in your scene, including diffuse and specular reflection parameters, and turn on appropriate color-blending routines for transparent surfaces if applicable. Finally, render the scene using Gouraud shading and experiment with modifying each of the parameters discussed here to produce the most visually appealing result.

https://hemanthrajhemu.github.io

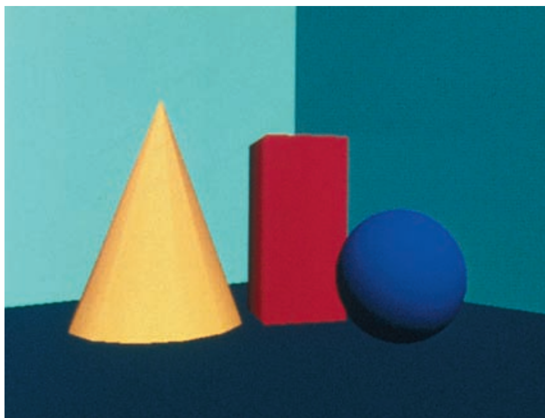
Illumination Models and Surface-Rendering Methods Color Plates



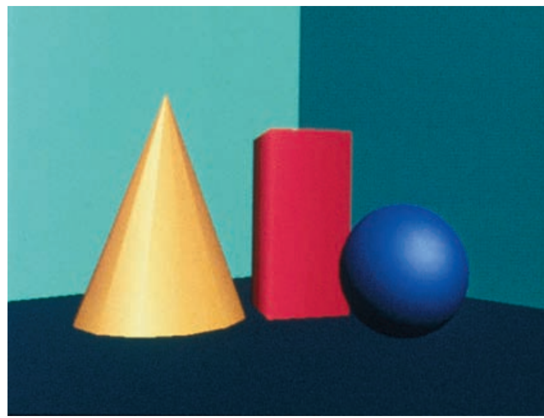
(a)



(b)



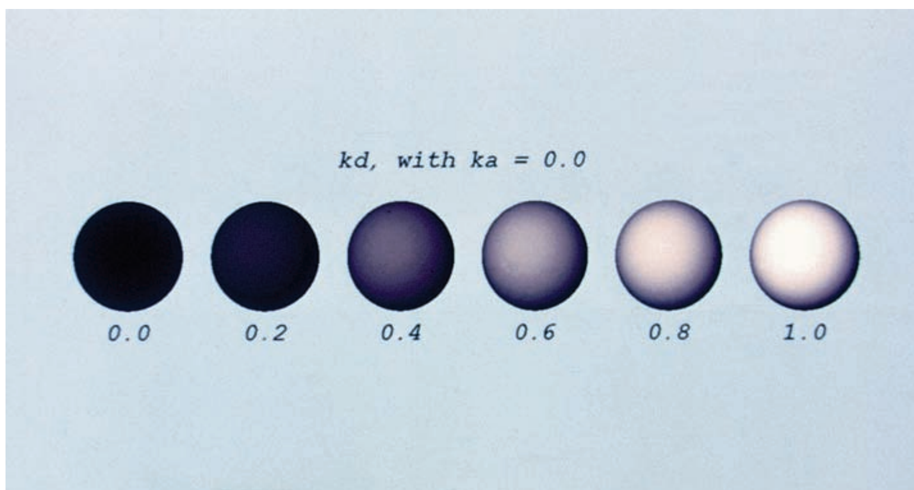
(c)



(d)

Color Plate 12

A wire-frame scene (a) is displayed in (b) using ambient lighting only, with a different color for each object. Diffuse reflections resulting from illumination with ambient light and a single point source are illustrated in (c). For this display, $k_s = 0$ for all surfaces. In (d), both diffuse and specular reflections are shown for the illumination from a point source and the background lighting.



Color Plate 13

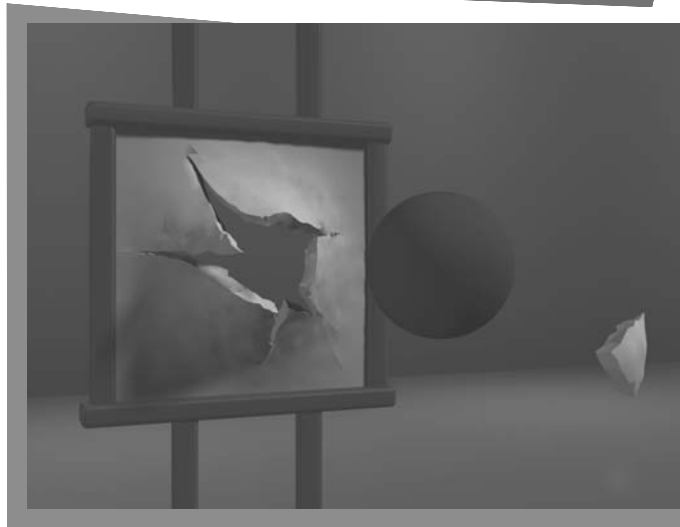
Diffuse reflections from a spherical surface illuminated by a point source emitting white light, with values of the diffuse reflectivity coefficient in the interval $0 \leq k_d \leq 1$.

From *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.
Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

https://hemanthrajhemu.github.io

Color Models and Color Applications

1 Properties of Light
2 Color Models
3 Standard Primaries and the Chromaticity Diagram
4 The RGB Color Model
5 The YIQ and Related Color Models
6 The CMY and CMYK Color Models
7 The HSV Color Model
8 The HLS Color Model
9 Color Selection and Applications
10 Summary



Our discussions of color up to this point have concentrated on methods involving red, green, and blue (RGB) components, which we use for generating displays on video monitors. Several other color descriptions are useful as well in computer-graphics applications. Some methods are used to describe color output on printers and plotters, some are used for transmitting and storing color information, and others are used to provide a more intuitive color-parameter interface to a program.

From Chapter 19 of *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.

Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

https://hemanthrajhemu.github.io

1 Properties of Light

Light exhibits many different characteristics, and we describe the properties of light in different ways in different contexts. Physically, we can characterize light as radiant energy, but we also need other concepts to describe our perception 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.

Each frequency value within the visible region of the electromagnetic spectrum corresponds to a distinct **spectral color.** At the low-frequency end (approximately 3.8×10^{14} hertz) are the red colors, and at the high-frequency end (approximately 7.9×10^{14} 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

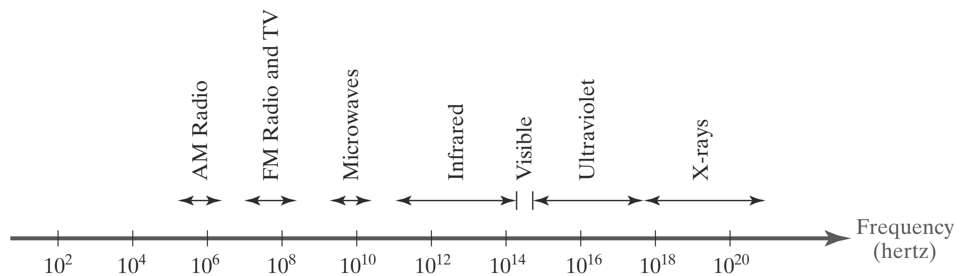


FIGURE 1
Electromagnetic spectrum.

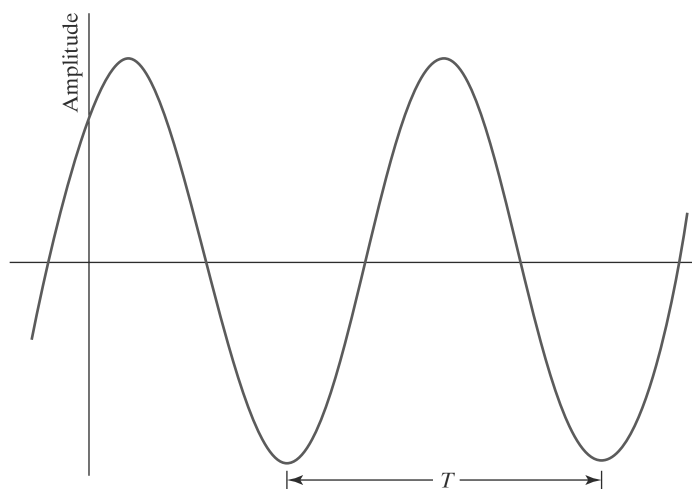


FIGURE 2
Time variations for the amplitude of the electric field for one frequency component of a plane-polarized electromagnetic wave. The time between two consecutive amplitude peaks or two consecutive amplitude minimums is called the period of the wave.

https://hemanthrajhemu.github.io

illustrates the time-varying oscillations for the magnitude of the electric field within one plane. 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):

$$c = \lambda f \tag{1}$$

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 $c = 3 \times 10^{10}$ 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 (1 nm $= 10^{-7}$ cm). An equivalent term for nanometer is *millimicron*. 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. 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. 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

https://hemanthrajhemu.github.io

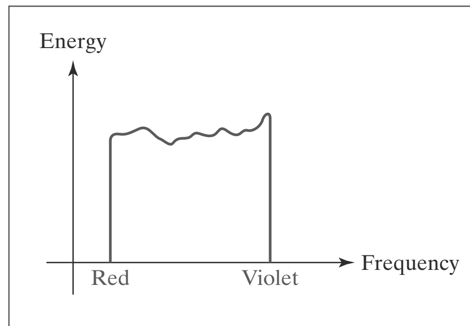


FIGURE 3
Energy distribution for a white light source.

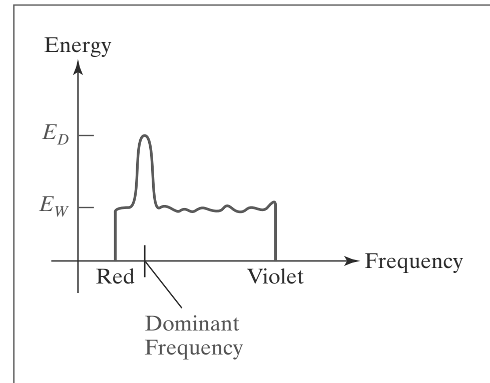


FIGURE 4
Energy distribution for a light source with a dominant frequency near the red end of the frequency range.

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 $E_W = E_D$.

2 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. 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.

https://hemanthrajhemu.github.io

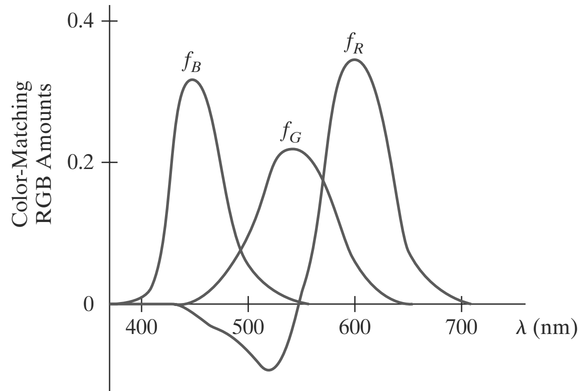


FIGURE 5
Three color-matching functions for displaying spectral frequencies within the approximate range from 400 nm to 700 nm.

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.

3 Standard Primaries and the Chromaticity Diagram

Because no finite set of light sources can be combined to display all possible colors, three standard primaries were defined in 1931 by the International Commission on Illumination, referred to as the CIE (Commission Internationale de l'Éclairage). The three standard primaries are imaginary colors. They are defined mathematically with positive color-matching functions (Figure 6) that

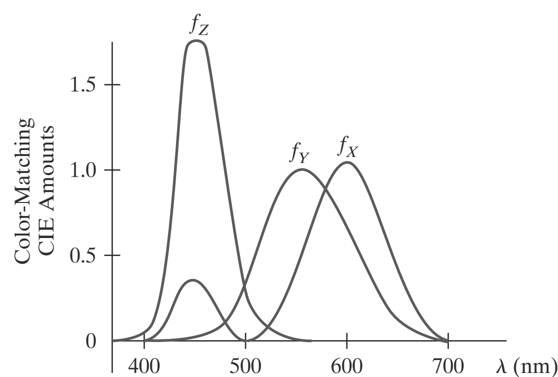


FIGURE 6
The three color-matching functions for the CIE primaries.

https://hemanthrajhemu.github.io

specify the amount of each primary needed to describe any spectral color. This provides an international standard definition for all colors, and the CIE primaries eliminate negative-value color-matching and other problems associated with selecting a set of real primaries.

The XYZ Color Model

The set of CIE primaries is generally referred to as the XYZ color model, where parameters X, Y, and Z represent the amount of each CIE primary needed to produce a selected color. Thus, a color is described with the XYZ model in the same way that we described a color using the RGB model.

In the three-dimensional XYZ color space, we represent any color $C(\lambda)$ as

$$C(\lambda) = (X,\ Y,\ Z) \tag{2}$$

where X, Y, and Z are calculated from the color-matching functions (Figure 6):

$$\begin{aligned} X &= k \int_{\text{visible}\,\lambda} f_X(\lambda)\, I(\lambda)\, d\lambda \\ Y &= k \int_{\text{visible}\,\lambda} f_Y(\lambda)\, I(\lambda)\, d\lambda \\ Z &= k \int_{\text{visible}\,\lambda} f_Z(\lambda)\, I(\lambda)\, d\lambda \end{aligned} \tag{3}$$

Parameter k in these calculations has the value 683 lumens/watt, where lumen is a unit of measure for light radiation per unit solid angle from a "standard" point light source (once called a *candle*). The function $I(\lambda)$ represents the spectral radiance, which is the selected light intensity in a particular direction, and the color-matching function f_Y is chosen so that parameter Y is the luminance for that color. Luminance values are normally adjusted to the range from 0 to 100.0, where 100.0 represents the luminance of white light.

Any color can be represented in the XYZ color space as an additive combination of the primaries using unit vectors $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$. Thus, we can write Equation 2 as

$$C(\lambda) = X\mathbf{X} + Y\mathbf{X} + Z\mathbf{X} \tag{4}$$

Normalized XYZ Values

In discussing color properties, it is convenient to normalize the amounts in Equation 3 against the sum $X + Y + Z$, which represents the total light energy. Normalized amounts are thus calculated as

$$x = \frac{X}{X+Y+Z}, \qquad y = \frac{Y}{X+Y+Z}, \qquad z = \frac{Z}{X+Y+Z} \tag{5}$$

Because $x+y+z = 1$, any color can be represented with just the x and y amounts. Also, we have normalized against total energy, so parameters x and y depend only on hue and purity and are called the **chromaticity values.** However, the x and y values alone do not allow us to describe all properties of the color completely, and we cannot obtain the amounts X, Y, and Z. Therefore, a complete description of a color is typically given with three values: x, y, and the luminance Y. The remaining CIE amounts are then calculated as

$$X = \frac{x}{y}Y, \qquad Z = \frac{z}{y}Y \tag{6}$$

where $z = 1 - x - y$. Using chromaticity coordinates (x, y), we can represent all colors on a two-dimensional diagram.

https://hemanthrajhemu.github.io

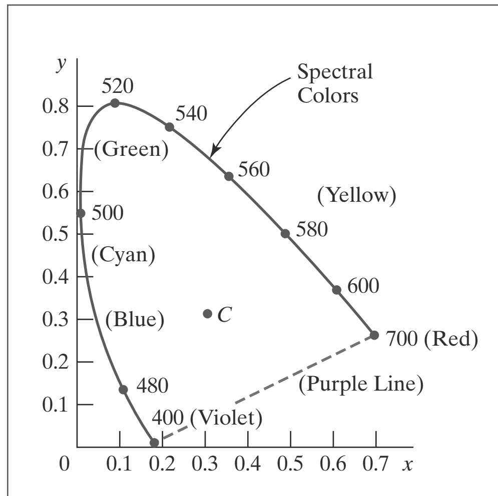


FIGURE 7
CIE chromaticity diagram for the spectral colors from 400 nm to 700 nm.

The CIE Chromaticity Diagram

When we plot the normalized amounts x and y for colors in the visible spectrum, we obtain the tongue-shaped curve shown in Figure 7. This curve is called the **CIE chromaticity diagram.** Points along the curve are the spectral colors (pure colors). The line joining the red and violet spectral points, referred to as the *purple line,* is not part of the spectrum. Interior points represent all possible visible color combinations. Point C in the diagram corresponds to the white-light position. Actually, this point is plotted for a white light source known as **illuminant C,** which is used as a standard approximation for average daylight.

Luminance values are not available in the chromaticity diagram because of normalization. Colors with different luminance but with the same chromaticity map to the same point. The chromaticity diagram is useful for:

- Comparing color gamuts for different sets of primaries.
- Identifying complementary colors.
- Determining purity and dominant wavelength for a given color.

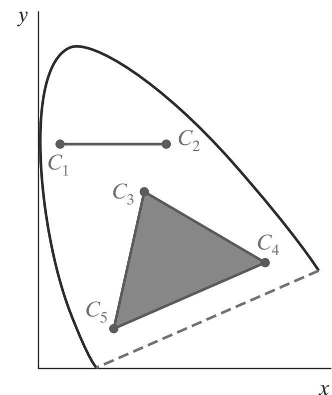


FIGURE 8
Color gamuts defined on the chromaticity diagram for a two-color and a three-color system of primaries.

Color Gamuts

We identify color gamuts on the chromaticity diagram as straight-line segments or polygon regions. All colors along the straight line joining points C_1 and C_2 in Figure 8 can be obtained by mixing appropriate amounts of the colors C_1 and C_2. If a greater proportion of C_1 is used, the resultant color is closer to C_1 than to C_2. The color gamut for three points, such C_3, C_4, and C_5 in Figure 8, is a triangle with vertices at the three color positions. These three primaries can generate only the colors inside or on the bounding edges of the triangle. Thus, the chromaticity diagram helps us to understand why no set of three primaries can be additively combined to generate all colors, because no triangle within the diagram can encompass all colors. Color gamuts for video monitors and hard-copy devices are compared conveniently on the chromaticity diagram.

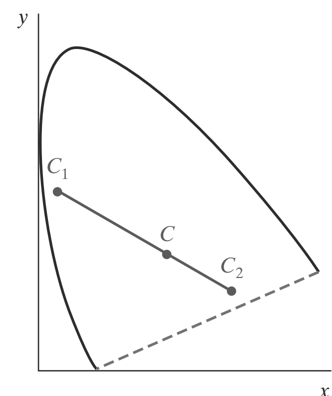


FIGURE 9
Representing complementary colors on the chromaticity diagram.

Complementary Colors

Because the color gamut for two points is a straight line, complementary colors must be represented on the chromaticity diagram as two points on opposite sides of C and collinear with C, as in Figure 9. The distances of the two colors C_1 and C_2 to C determine the amounts of each needed to produce white light.

https://hemanthrajhemu.github.io

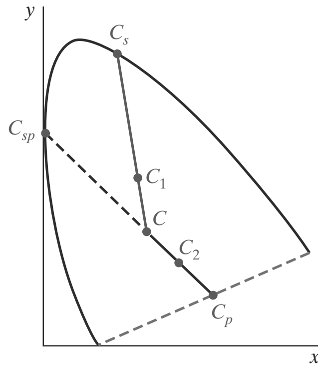


FIGURE 10
Determining dominant wavelength and purity using the chromaticity diagram.

Dominant Wavelength

To determine the dominant wavelength of a color, we draw a straight line from C through that color point to a spectral color on the chromaticity curve. The spectral color C_s in Figure 10 is the dominant wavelength for color C_1 in this diagram. Thus, color C_1 can be represented as a combination of white light C and the spectral color C_s. This method for determining dominant wavelength will not work for color points that are between C and the purple line. Drawing a line from C through point C_2 in Figure 10 takes us to point C_p on the purple line, which is not in the visible spectrum. In this case, we take the compliment of C_p on the spectral curve, which is the point C_{sp}, as the dominant wavelength. Colors such as C_2 in this diagram have spectral distributions with subtractive dominant wavelengths. We can describe such colors by subtracting the spectral dominant wavelength from white light.

Purity

For a color point such as C_1 in Figure 10, we determine the purity as the relative distance of C_1 from C along the straight line joining C to C_s. If d_{c1} denotes the distance from C to C_1 and d_{cs} is the distance from C to C_s, we can represent purity as the ratio d_{c1}/d_{cs}. Color C_1 in this figure is about 25 percent pure, because it is situated at about one-fourth the total distance from C to C_s. At position C_s, the color point would be 100 percent pure.

4 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. The origin represents black and the diagonally opposite

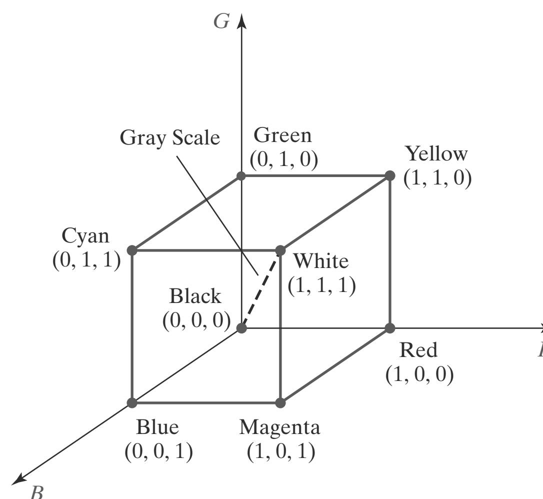


FIGURE 11
The RGB color model. Any color within the unit cube can be described as an additive combination of the three primary colors.

https://hemanthrajhemu.github.io

TABLE 1

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

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

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**:

$$C(\lambda) = (R,\ G,\ B) = R\,\mathbf{R} + G\,\mathbf{G} + B\,\mathbf{B} \tag{7}$$

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.

Chromaticity coordinates for the National Television System Committee (NTSC) standard RGB phosphors are listed in Table 1. Also listed are the RGB chromaticity coordinates within the CIE color model and the approximate values used for phosphors in color monitors. Figure 12 shows the approximate color gamut for the NTSC standard RGB primaries.

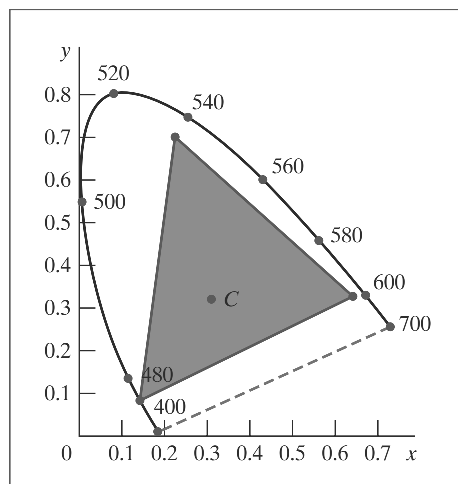


FIGURE 12
The RGB color gamut for NTSC chromaticity coordinates. Illuminant C is at position (0.310, 0.316), with a luminance value of $Y = 100.0$.

https://hemanthrajhemu.github.io

5 The YIQ and Related Color Models

Although an RGB graphics monitor requires separate signals for the red, green, and blue components of an image, a television monitor uses a composite signal. NTSC color encoding for forming the composite video signal is called the *YIQ color model*.

The YIQ Parameters

In the YIQ color model, parameter Y is the same as the Y component in the CIE XYZ color space. Luminance (brightness) information is conveyed by the Y parameter, while chromaticity information (hue and purity) is incorporated into the I and Q parameters. A combination of red, green, and blue is chosen for the Y parameter to yield the standard luminosity curve. Because Y contains the luminance information, black-and-white television monitors use only the Y signal. Parameter I contains orange-cyan color information that provides the flesh-tone shading, and parameter Q carries green-magenta color information.

The NTSC composite color signal is designed to provide information in a form that can be received by black-and-white television monitors, which obtain grayscale information for a picture within a 6-MHz bandwidth. Thus, the YIQ information is also encoded within a 6-MHz bandwidth, but the luminance and chromaticity values are encoded on separate analog signals. In this way, the luminance signal is unchanged for black-and-white monitors, and the color information is simply added within the same bandwidth. Luminance information, the Y value, is conveyed as an amplitude modulation on a carrier signal with a bandwidth of about 4.2 MHz. Chromaticity information, the I and Q values, is combined on a second carrier signal that has a bandwidth of about 1.8 MHz. The parameter names I and Q refer to the modulation methods used to encode the color information on this carrier. An amplitude-modulation encoding (the "in-phase" signal) transmits the I value, using about 1.3 MHz of the bandwidth. And a phase-modulation encoding (the "quadrature" signal), using about 0.5 MHz, carries the Q value.

Luminance values are encoded at a higher precision in the NTSC signal (4.2 MHz bandwidth) than the chromaticity values (1.8 MHz bandwidth), because we can detect small brightness changes more easily compared to small color changes. However, the lower precision for the chromaticity encoding does result in some degradation of the color quality for an NTSC picture.

We can calculate the luminance value for an RGB color. One method for producing chromaticity values is to subtract the luminance from the red and blue components of the color. Thus,

$$\begin{aligned} Y &= 0.299\,R + 0.587\,G + 0.114\,B \\ I &= R - Y \\ Q &= B - Y \end{aligned} \tag{8}$$

Transformations Between RGB and YIQ Color Spaces

An RGB color is converted to a set of YIQ values using an NTSC encoder that implements the calculations in Equation 9 and modulates the carrier signals. The conversion from RGB space to YIQ space is accomplished using the following transformation matrix:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.701 & -0.587 & -0.114 \\ -0.299 & -0.587 & 0.886 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{9}$$

https://hemanthrajhemu.github.io

Conversely, an NTSC video signal is converted to RGB color values using an NTSC decoder, which first separates the video signal into the YIQ components, and then converts the YIQ values to RGB values. The conversion from YIQ space to RGB space is accomplished with the inverse of transformation 9:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 1.000 & 0.000 \\ 1.000 & -0.509 & -0.194 \\ 1.000 & 0.000 & 1.000 \end{bmatrix} \cdot \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \tag{10}$$

The YUV and YC_rC_b Systems

Because of the lower bandwidth assigned to the chromaticity information in the NTSC composite analog video signal, the color quality of an NTSC picture is somewhat impaired. Therefore, variations of the YIQ encoding have been developed to improve the color quality of video transmissions. One such encoding is the YUV set of color parameters, which provides the composite color information for video transmissions by systems such as Phase Alternation Line (PAL) Broadcasting, used in most of Europe, as well as Africa, Australia, and Eurasia. Another variation of YIQ is the digital encoding called YC_rC_b. This color representation is used for digital video transformations, and it is incorporated into various graphics file formats, such as the JPEG system.

6 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.

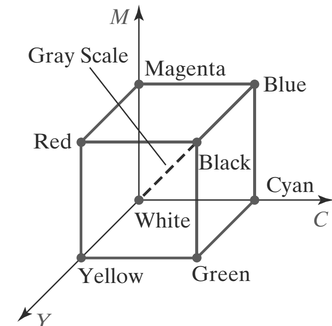


FIGURE 13 The CMY color model. Positions within the unit cube are described by subtracting the specified amounts of the primary colors from white.

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

https://hemanthrajhemu.github.io

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:

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{11}$$

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

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix} \tag{12}$$

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 in Equation 11. 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.

7 The HSV Color Model

Interfaces for selecting colors often use a color model based on intuitive concepts, rather than a set of primary colors. We can give a color specification in an intuitive model by selecting a spectral color and the amounts of white and black that are to be added to that color to obtain different shades, tints, and tones (Section 2).

The HSV Parameters

Color parameters in this model are called *hue* (H), *saturation* (S), and *value* (V). We derive this three-dimensional color space by relating the HSV parameters to the directions in the RGB cube. If we imagine viewing the cube along the diagonal from the white vertex to the origin (black), we see an outline of the cube that has the hexagon shape shown in Figure 14. The boundary of the hexagon represents the various hues, and it is used as the top of the HSV hexcone (Figure 15). In HSV space, saturation S is measured along a horizontal axis, and the value parameter V is measured along a vertical axis through the center of the hexcone.

Hue is represented as an angle about the vertical axis, ranging from 0° at red through 360°. Vertices of the hexagon are separated by 60° intervals. Yellow is at 60°, green at 120°, and cyan (opposite the red point) is at $H = 180^\circ$. Complementary colors are 180° apart.

Saturation parameter S is used to designate the purity of a color. A pure color (spectral color) has the value $S = 1.0$, and decreasing S values tend toward the grayscale line ($S = 0$) at the center of the hexcone.

https://hemanthrajhemu.github.io

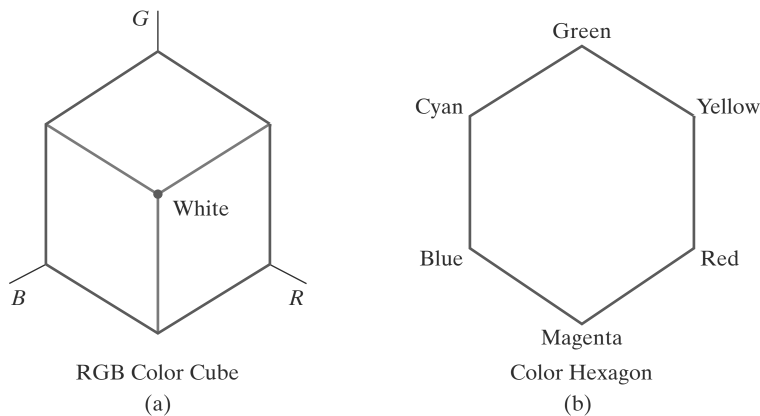


FIGURE 14
When the RGB color cube (a) is viewed along the diagonal from white to black, the color-cube outline is a hexagon (b).

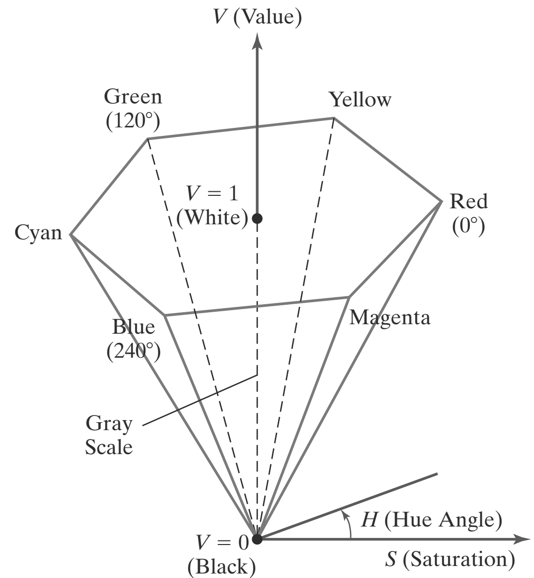


FIGURE 15
The HSV hexcone.

Value V varies from 0 at the apex of the hexcone to 1.0 at the top plane. The apex of the hexcone is the black point. At the top plane, colors have their maximum intensity. When $V = 1.0$ and $S = 1.0$, we have the pure hues. Parameter values for the white point are $V = 1.0$ and $S = 0$.

For most users, this is a more convenient model for selecting colors. Starting with a selection for a pure hue, which specifies the hue angle H and sets $V = S = 1.0$, we describe the color we want in terms of adding either white or black to the pure hue. Adding black decreases the setting for V while S is held constant. To get a dark blue, for instance, V could be set to 0.4 with $S = 1.0$ and $H = 240°$. Similarly, when white is to be added to the selected hue, parameter S is decreased while keeping V constant. A light blue could be designated with $S = 0.3$ while $V = 1.0$ and $H = 240°$. By adding some black and some white, we decrease both V and S. An interface for this model typically presents the HSV parameter choices in a color palette containing sliders and a color wheel.

Selecting Shades, Tints, and Tones

Color regions for selecting shades, tints, and tones are represented in the cross-sectional plane of the HSV hexcone shown in Figure 16. Adding black to a spectral color decreases V along the side of the hexcone toward the black point. Thus, various shades are represented with the values $S = 1.0$ and $0.0 \leq V \leq 1.0$. Adding white to spectral colors produces the tints across the top plane of the hexcone, where parameter values are $V = 1.0$ and $0 \leq S \leq 1.0$. Various tones are obtained by adding both black and white to spectral colors, which generates color points within the triangular cross-sectional area of the hexcone.

The human eye can distinguish about 128 different hues and about 130 different tints (saturation levels). For each of these, a number of shades (value settings) can be detected, depending on the hue selected. About 23 shades are discernible with yellow colors, and about 16 different shades can be seen at the blue end of the spectrum. This means that we can distinguish about $128 \times 130 \times 23 = 382{,}720$ different colors. For most graphics applications, 128 hues, 8 saturation levels, and

https://hemanthrajhemu.github.io

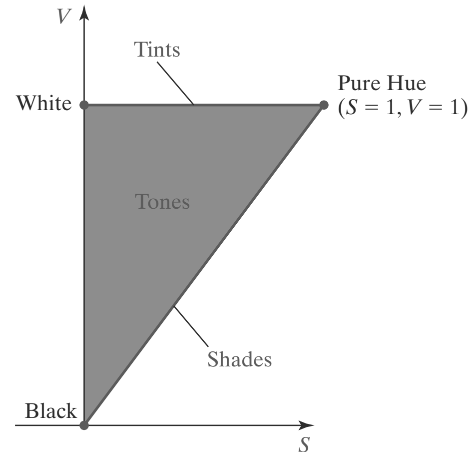


FIGURE 16
Cross section of the HSV hexcone, showing regions for shades, tints, and tones.

16 value settings are sufficient. With this range of parameters in the HSV color model, 16,384 colors are available to a user. These color values can be stored in 14 bits per pixel, or we could use color-lookup tables and fewer bits per pixel.

Transformations Between HSV and RGB Color Spaces

To determine the operations required for the transformations between the HSV and RGB spaces, we first consider how the HSV hexcone can be constructed from the RGB cube. The diagonal of the RGB cube from black (the origin) to white corresponds to the V axis of the hexcone. Also, each subcube of the RGB cube corresponds to a hexagonal cross-sectional area of the hexcone. At any cross section, all sides of the hexagon and all radial lines from the V axis to any vertex have the value V. Thus, for any set of RGB values, V is equal to the value of the maximum RGB component. The HSV point corresponding to this set of RGB values lies on the hexagonal cross section at value V. Parameter S is then determined as the relative distance of this point from the V axis. Parameter H is determined by calculating the relative position of the point within each sextant of the hexagon. An algorithm for mapping any set of RGB values into the corresponding HSV values is given in the following procedure:

```
class rgbSpace {public: float r, g, b;};
class hsvSpace {public: float h, s, v;};

const float noHue = -1.0;
inline float min(float a, float b) {return (a < b)? a : b;}
inline float max(float a, float b) {return (a > b)? a : b;}

void rgbTOhsv (rgbSpace& rgb, hsvSpace& hsv)
{
   /* RGB and HSV values are in the range from 0 to 1.0 */
   float minRGB = min (r, min (g, b)), maxRGB = max (r, max (g, b));
   float deltaRGB = maxRGB - minRGB;

   v = maxRGB;
   if (maxRGB != 0.0)
      s = deltaRGB / maxRGB;
   else
      s = 0.0;
```


https://hemanthrajhemu.github.io

```
    if (s <= 0.0)
       h = noHue;
    else {
       if (r == maxRGB)
          h = (g - b) / deltaRGB;
       else
          if (g == maxRGB)
             h = 2.0 + (b - r) / deltaRGB;
          else
             if (b == maxRGB)
                h = 4.0 + (r - g) / deltaRGB;
       h *= 60.0;
       if (h < 0.0)
          h += 360.0;
       h /= 360.0;
    }
}
```

We obtain the transformation from HSV space to RGB space by determining the inverse of the operations in the preceding procedure. These inverse operations are carried out for each sextant of the hexcone, and the resulting transformation equations are summarized in the following algorithm:

```
class rgbSpace {public: float r, g, b;};
class hsvSpace {public: float h, s, v;};

void hsvT0rgb (hsvSpace& hsv, rgbSpace& rgb)
{
   /*  HSV and RGB values are in the range from 0 to 1.0  */
   int k
   float aa, bb, cc, f;

   if ( s <= 0.0)
      r = g = b = v;       //  Have gray scale if s = 0.
   else {
      if (h == 1.0)
         h = 0.0;
      h *= 6.0;
      k = floor (h);
      f = h - k;
      aa = v * (1.0 - s);
      bb = v * (1.0 - (s * f));
      cc = v * (1.0 - (s * (1.0 - f)));
      switch (k)
      {
         case 0:  r = v;   g = cc;  b = aa; break;
         case 1:  r = bb;  g = v;   b = aa; break;
         case 2:  r = aa;  g = v;   b = cc; break;
         case 3:  r = aa;  g = bb;  b = v; break;
         case 4:  r = cc;  g = aa;  b = v; break;
         case 5:  r = v;   g = aa;  b = bb; break;
      }
   }
}
```


https://hemanthrajhemu.github.io

8 The HLS Color Model

Another model based on intuitive color parameters is the *HLS system* used by the Tektronix Corporation. This color space has the double-cone representation shown in Figure 17. The three parameters in this color model are called hue (H), lightness (L), and saturation (S).

Hue has the same meaning as in the HSV model. It specifies an angle about the vertical axis that locates a hue (spectral color). In this model, $H = 0^\circ$ corresponds to blue. The remaining colors are specified around the perimeter of the cone in the same order as in the HSV model. Magenta is located at $H = 60^\circ$, red is at $H = 120^\circ$, and cyan is at $H = 300^\circ$. Again, complementary colors are 180° apart on the double cone.

The vertical axis in this model is called lightness, L. At $L = 0$, we have black, and at $L = 1.0$, we have white. Grayscale values are along the L axis, and the pure colors lie on the $L = 0.5$ plane.

Saturation parameter S again specifies the purity of a color. This parameter varies from 0 to 1.0, and pure colors are those for which $S = 1.0$ and $L = 0.5$. As S decreases, more white is added to a color. The grayscale line is at $S = 0$.

To specify a color, we begin by selecting hue angle H. Then a particular shade, tint, or tone for that hue is obtained by adjusting parameters L and S. We obtain a lighter color by increasing L, and we obtain a darker color by decreasing L. When S is decreased, the spatial color point moves toward the grayscale line.

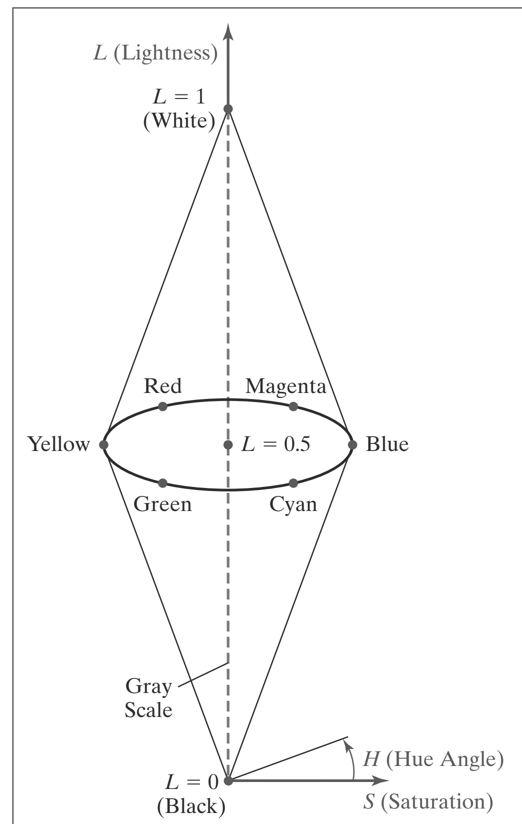


FIGURE 17
The HLS double cone.

https://hemanthrajhemu.github.io

9 Color Selection and Applications

A graphics package can provide color capabilities in a way that aids us in making color selections. For example, an interface can contain sliders and color wheels instead of requiring that all color specifications be provided as numerical values for the RGB components. In addition, some aids can be provided for choosing harmonious color combinations and for basic color selection guidelines.

One method for obtaining a set of coordinating colors is to generate the color combinations from a small subspace of a color model. If colors are selected at regular intervals along any straight line within the RGB or CMY cube, for example, we can expect to obtain a set of well-matched colors. Randomly selected hues can be expected to produce harsh and clashing color combinations. Another consideration in color displays is the fact that we perceive colors at different depths. This occurs because our eyes focus on colors according to their frequency. Blues, in particular, tend to recede. Displaying a blue pattern next to a red pattern can cause eye fatigue, because we continually need to refocus when our attention is switched from one area to the other. This problem can be reduced by separating these colors or by using colors from one-half or less of the color hexagon in the HSV model. With this technique, a display contains either blues and greens or reds and yellows.

As a general rule, the use of a smaller number of colors produces a better-looking display than one with a large number of colors. Also, tints and shades tend to blend better than the pure hues. For a background, gray or the complement of one of the foreground colors is usually best.

10 Summary

Light can be described as electromagnetic radiation with a certain energy distribution propagating through space, and the color components of light correspond to frequencies within a narrow band of the electromagnetic spectrum. However, light exhibits other properties, and we characterize the different aspects of light using a variety of parameters. With the light theories for wave-particle duality, we can explain the physical features of visible radiation. And we quantify our perceptions of a light source using terms such as dominant frequency (hue), luminance (brightness), and purity (saturation). Hue and purity are referred to collectively as the chromaticity properties of a color.

We also use color models to explain the effects of combining light sources. One method for defining a color model is to specify a set of two or more primary colors that are combined to produce various other colors. However, no finite set of primary colors is capable of producing all colors or describing all features of color. The set of colors that can be generated by a set of primaries is called a *color gamut*. Two colors that combine to produce white light are called complementary colors.

In 1931, the International Commission on Illumination (CIE) adopted a set of three hypothetical color-matching functions as a standard. This set of colors is referred to as the *XYZ model*, where X, Y, and Z represent the amounts of each color needed to match any color in the electromagnetic spectrum. The color-matching functions are structured so that all functions are positive and the Y amount for any color represents the luminance value. Normalized X and Y values, called x and y, are used to plot positions for all spectral colors on the CIE chromaticity diagram. We can use the chromaticity diagram to compare color gamuts for different

https://hemanthrajhemu.github.io

color models, to identify complementary colors, and to determinant dominant frequency and purity for a specified color.

Other color models based on a set of three primaries are the RGB, YIQ, and CMY models. We use the RGB model to describe colors that are displayed on a video monitor. The YIQ model is used to describe the composite video signal for television broadcasting. And the CMY model is used to describe color on hard-copy devices.

User interfaces often provide intuitive color models, such as the HSV and HLS models, for selecting color values. With these models, we specify a color as a mixture of a selected hue and certain amounts of white and black. Adding black produces color shades, adding white produces tints, and adding both black and white produces tones.

Color selection is an important factor in the design of effective displays. To avoid clashing color combinations, we can choose adjacent colors in a display that do not differ greatly in dominant frequency. Also, we can select color combinations from a small subspace of a color model. As a general rule, a small number of color combinations formed with tints and shades, rather than pure hues, results in a more harmonious color display.

REFERENCES

A comprehensive discussion of the science of color is given in Wyszecki and Stiles (1982). Color models and color display techniques are treated in Smith (1978), Heckbert (1982), Durrett (1987), Schwartz, Cowan, and Beatty (1987), Hall (1989), and Travis (1991).

Algorithms for various color applications are presented in Glassner (1990), Arvo (1991), Kirk (1992), Heckbert (1994), and Paeth (1995). For additional information on the human visual system and our perception of light and color, see Glassner (1995).

EXERCISES

1. Derive the expressions for converting RGB color parameters to HSV values.
2. Derive the expressions for converting HSV color values to RGB values.
3. Design an interactive procedure that allows selection of HSV color parameters from a displayed menu; then, the HSV values are to be converted to RGB values for storage in a frame buffer.
4. Write a program to select colors using a set of three sliders to select values for the HSV color parameters.
5. Modify the program in the preceding exercise to display the numeric values for the RGB components of a selected color.
6. Modify the program in the preceding exercise to display the RGB color components and the combined color in small display windows.
7. Derive expressions for converting RGB color values to HLS color parameters.
8. Derive expressions for converting HLS color values to RGB values.
9. Design an interactive procedure that allows selection of HLS color parameters from a displayed menu; then, the HLS values are to be converted to RGB values for storage in a frame buffer.
10. Write a program that will produce a set of colors that are linearly interpolated between any two specified positions in RGB space.
11. Write an interactive routine for selecting color values from within a specified subspace of RGB space.
12. Write a program that will produce a set of colors that are linearly interpolated between any two specified positions in HSV space.
13. Write an interactive routine for selecting color values from within a specified subspace of HSV space.
14. Write a program that will produce a set of colors that are linearly interpolated between any two specified positions in HLS space.
15. Write an interactive routine for selecting color values from within a specified subspace of HLS space.
16. Write a program to display two adjacent RGB color rectangles. Fill one rectangle with a set of randomly selected RGB color points, and fill

https://hemanthrajhemu.github.io

the other rectangle with a set of color points that are selected from a small RGB subspace. Experiment with different random selections and different subspaces to compare the two color patterns.

17 Display the two color rectangles in the preceding exercise using color selections from either the HSV or the HLS color space.

18 Write a program that will produce a randomly selected color from within a color gamut specified by three positions in RGB space.

19 Write a program that will produce a randomly selected color from within a color gamut specified by three positions in HSV space.

20 Write a program that will produce a randomly selected color from within a color gamut specified by three positions in HLS space.

IN MORE DEPTH

1 Write a routine that takes in a pixel position of a scene in your application and a color space identifier and returns a vector representing the color value of that pixel in the selected color space. The routine should produce correct output for the RGB, CMY, HSV, and HLS color spaces.

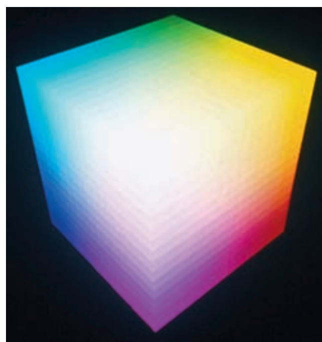
2 Use the routine developed in the previous exercise to write another routine that outputs to a file a bitmap of a scene in your application in a specified color space (either RGB, CMY, HSV, or HLS). That is, the routine should take in a color space identifier, call the routine in the previous exercise on each pixel to obtain the color of that pixel in the specified color space, and write the color value vector of each pixel out to a file. Each color value vector should appear on a separate line, and pixels should be processed in row-major order.

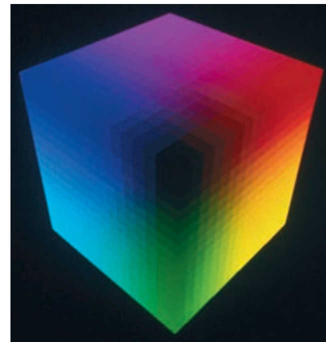
https://hemanthrajhemu.github.io

This page intentionally left blank

https://hemanthrajhemu.github.io

Color Models and Color Applications Color Plates


(a)


(b)

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.

From *Computer Graphics with OpenGL®*, Fourth Edition, Donald Hearn, M. Pauline Baker, Warren R. Carithers.
Copyright © 2011 by Pearson Education, Inc. Published by Pearson Prentice Hall. All rights reserved.

https://hemanthrajhemu.github.io

This page intentionally left blank

https://hemanthrajhemu.github.io