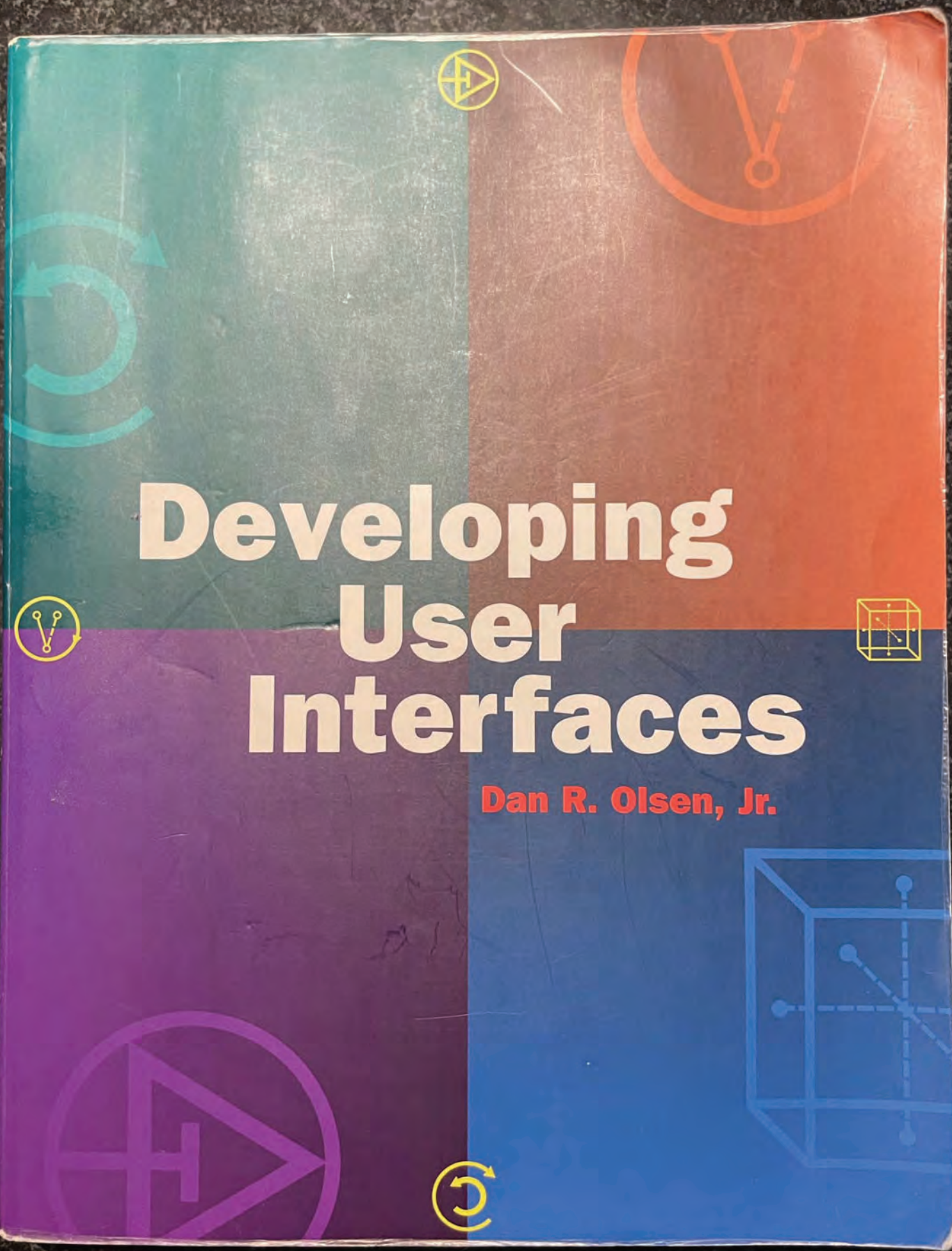
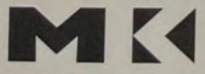


Developing User Interfaces

Dan R. Olsen, Jr.
Carnegie Mellon University/Brigham Young University



MORGAN KAUFMANN PUBLISHERS, INC.
San Francisco, California

Sponsoring Editor	Michael B. Morgan
Production Manager	Yonie Overton
Production Editor	Julie Pabst
Editorial Coordinator	Marilyn Uffner Alan
Copyeditor	Jeff Van Bueren
Text Design	Side by Side Studios
Illustration	Cherie Plumlee
Composition	Nancy Logan
Cover Design	Ross Carron Design
Proofreaders	Erin Milnes, Gary Morris
Indexer	Steve Rath
Printer	Courier Corporation

Morgan Kaufmann Publishers, Inc.
Editorial and Sales Office:
340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205
USA

Telephone	415/392-2665
Facsimile	415/982-2665
Email	*mkp@mkp.com*
WWW	*http://www.mkp.com*
Order toll free	800/745-7323

Printed in the United States of America

02 01 00 99 98 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data
Olsen, Dan R., 1953–
Developing user interfaces/Dan R. Olsen, Jr.
p. cm.
ISBN 1-55860-418-9
1. User interfaces (Computer systems) 2. Computer software—Development. I. Title.
QA76.9.U83043 1998
005.4'28--dc21

97–45231

Contents

3

Basic Computer Graphics

Having completed a functional design of what our user interface is to accomplish and how it is logically structured, we must lay the technical groundwork required for implementing graphical presentations. This chapter covers the basics of 2D computer graphics, involving the basic drawing techniques and hardware. Also included is the necessary 2D geometry required for interaction with primitive forms. We also discuss the issues of text, clipping a drawing to stay within a particular region, and color.

In this chapter, we focus on the basic 2D primitives that are required to present information to the users so they can interactively manipulate it. The field of computer graphics is quite diverse, and only the bare essentials are presented here. In creating realistic images of physical objects, we must consider 3D perspective viewing, realistic reflection of light and shadow, texturing physical objects, and removing hidden surfaces. The whole area of realistic physical models as well as the animation of such models is not considered here. A variety of computer graphics texts already address these issues.

In addition to ignoring high-level 3D modeling and rendering, we also ignore scan conversion. *Scan conversion* is the process of converting a geometric shape such as a line or polygon into a set of pixels that represent the shape on the screen or a printer. We assume that any user interface tool kit will provide methods or procedures that perform such tasks for us.

In this chapter, we assume that we are using a graphics package that can draw a variety of 2D shapes, given the appropriate inputs. Our problem is to understand how to invoke such methods and how to organize them in a way that will efficiently display our user's information. Understanding the various display architectures will help us understand how those architectures affect interactive software.

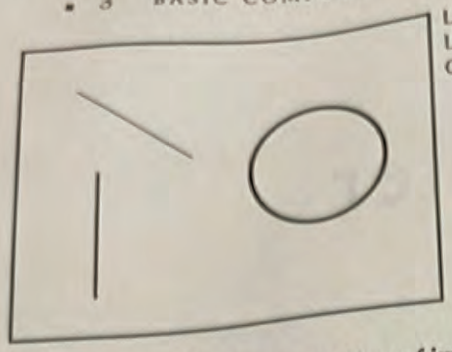


Figure 3-1 Stroke representation of images

3.1 Models for Images

There are three primary ways to represent 2D images—strokes, pixels, and regions. Each model has properties that are helpful for display, modeling, or interaction, and each has its disadvantages.

3.1.1 Stroke Model

The stroke model is the earliest form used for representing geometric objects in a computer. In this model, we describe all images as strokes of some specified color and thickness. For example, Figure 3-1 shows a simple diagram and its accompanying stroke representation.

In the stroke representation, the type of stroke and its geometry are represented. Early vector-refresh displays and direct-view storage tubes represented their images in this fashion. Special control hardware was required to translate the stroke representation into images on the screen. Plotters accept such representations and convert them into paths that the pen should follow to draw the stroke on paper. PostScript printers also accept such a representation, which they then convert to a region model and then to a pixel model for actual printing.

The stroke model is the one that is commonly used for many interactive applications. Most graphics packages in user interface tool kits provide more complex stroked objects, including arcs, ellipses, elliptical arcs, rectangles with rounded corners, and various curved shapes.

3.1.2 Pixel Model

The stroke model, however, is not adequate for representing more realistic or complex images, such as Figure 3-2. Such images require a pixel model, which

Figure 3-2 A pixel image

divides the image into a discrete number of pixels and then stores an intensity or color value for each pixel. In addition, all of today's graphics hardware is pixel based, which means that ultimately all models must be reduced to pixels before printing or display.

There are three key aspects to the resolution of a pixel image. The number of rows of pixels and the number of pixels per row constitute the spatial resolution of the image. For good display screens, this is 1024×768, with some displays going higher. For most laser printers, the resolution varies from 3000×2400 to 6000×4800 in order to create a high-quality printed page. The third aspect is the image depth or number of bits required to represent each pixel.

There are four basic forms of pixel-based images. The first is a simple bitmap, where each pixel consists of one bit that is either on or off. Such a model is suitable for representing the black and white of a printed page. In order to represent levels of gray, several pixels are grouped together in various patterns of on and off, producing an appearance of grayness. This is the approach used in laser printers, which can either put down a spot of ink or not. With very high resolution, this halftoning or dithering process can produce acceptable gray.

In the second method, gray-scale images provide more than 1 bit per pixel. Some systems such as the early NeXT provided 2 bits per pixel. This approach only doubled the space required to store the screen image while providing four levels of gray. This 2-bit image only marginally improved the quality of photographs and other realistic images, but it significantly improved the appearance of interface items such as buttons and type-in boxes. Items that the Macintosh (using bitmaps) had to represent as grainy patterns appeared smooth on the NeXT. Most modern gray-scale systems provide 8 bits (1 byte) per pixel, which can represent 256 levels of gray ranging from 0 (black) to 255 (white). A 1024×1024 gray-scale image requires 1 megabyte of storage.

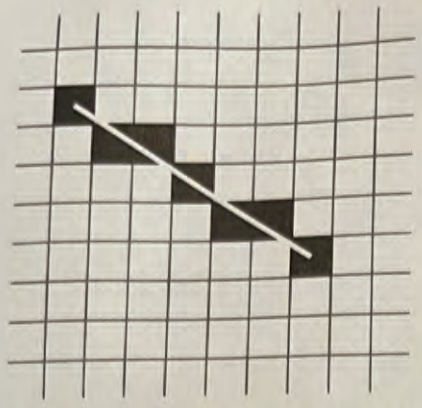


Figure 3-3 Pixel representation of a line

The third, and most flexible, representation of images is the full-color representation. Colors are assembled from the three additive primary colors, red, green, and blue. With 8 bits per primary color (3 bytes), the entire range of colors can be represented. Such an image format requires 3 megabytes for a 1024×1024 image. Space becomes a problem when dealing with highly realistic full-color images. For photograph-quality images that are 3000×2400, approximately 22 MB of storage are required. Because of these requirements, compression techniques are often used.

The fourth representation is the color-mapped image. In this case, only 8 bits are used per pixel. Instead of storing a color, each pixel stores an index into a color table that contains the full 24 bits for each indexed color. This allows many color images to be represented in the same space as a gray-scale image. Many color displays that are being used for normal interaction, rather than photograph-quality applications, are based on the lookup-table image model. In addition to cutting the storage space to a third, the smaller number of bytes also reduces the amount of computation required to manipulate the pixels.

Although the pixel model can represent any image, there are some problems when converting stroke or region objects into pixels. If the spatial resolution of the pixel image is low, as shown in Figure 3-3, aliasing can occur where smooth objects, such as a line, appear jagged.

If the resolution is very high, as with a good laser printer, the jagged edges are so small that the eye does not perceive them. In the case of lower-resolution displays, gray scale can be used to alleviate the problem by filling in some of the jagged places. This process, known as *antialiasing*, is shown in Figure 3-4. The appearance is improved, but significantly more computation is required to produce the correct gray settings. Similar effects are used with each of the primary colors in color images.

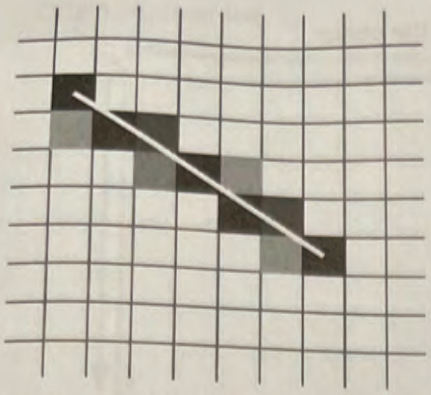


Figure 3-4 Antialiased line

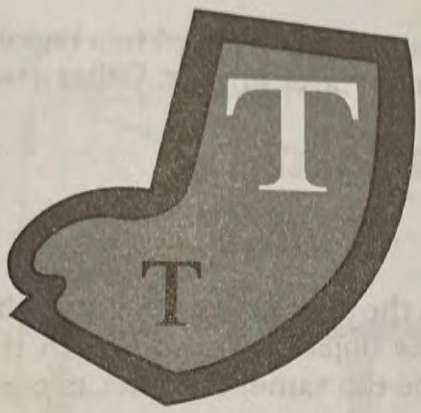


Figure 3-5 Region-based image

3.1.3 Region Model

The third image model is the region model. In this model, stroke objects are used to outline the region to be filled, as shown in Figure 3-5. There are various models by which regions are filled, including constant colors or various blendings to produce shaded effects. A major advantage of region models is that filled shapes can be represented in very little memory and in a way that is independent of the display resolution. This is very advantageous for high-resolution display devices, such as laser printers, where transferring a full page to the printer would require at least 2 MB in pixel form (thus slowing communications) and would require that the computer software be aware of the printer's resolution.

Even text is represented as regions and then converted to bitmaps inside of the printer; this method allows good clear text of any size to be generated. In

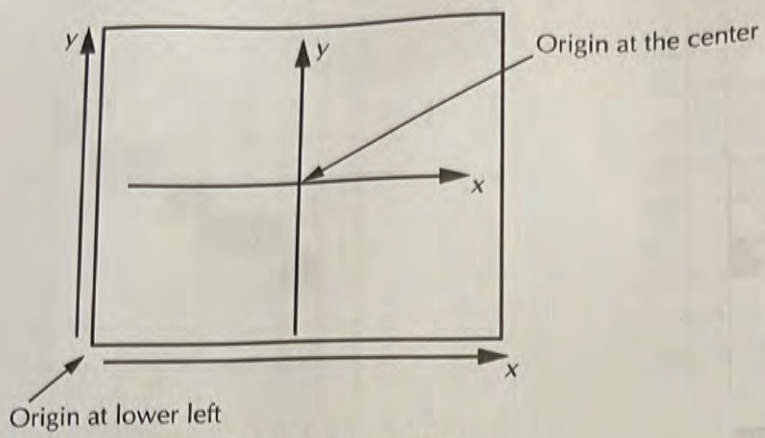


Figure 3-6 Normal Cartesian coordinates

fact, even a line that is 1 pixel wide on a screen must be converted to a region of pixels when rendered on a 600-dpi (dots per inch) laser printer. Otherwise the line would be too thin in print.

3.2 Coordinate Systems

An important consideration in drawing objects is the coordinate system of the drawing and the coordinate system in which the objects are defined in the application. These coordinate systems may not be the same; this merits careful discussion

3.2.1 Device Coordinates

Device coordinates are the coordinates of the actual display device. In most geometry texts, the origin of the coordinate system is either in the lower left or the center with positive *x* going to the right and positive *y* going up, as shown in Figure 3-6.

This coordinate system is rarely used for frame buffers, graphics displays, or laser printers, because most display devices work from the upper left and downward. For this reason, most devices use the coordinates shown in Figure 3-7. In this model, device coordinates are always positive integers, except in cases such as laser printers where additional processing of the image is done by the printer's processor before actual display.

One significant modification to the use of device coordinates in displaying graphics is the window. In most modern interface tool kits, the programmer is presented with a window as the abstraction for the display. This window is presented to the programmer as a virtual display on which the programmer is

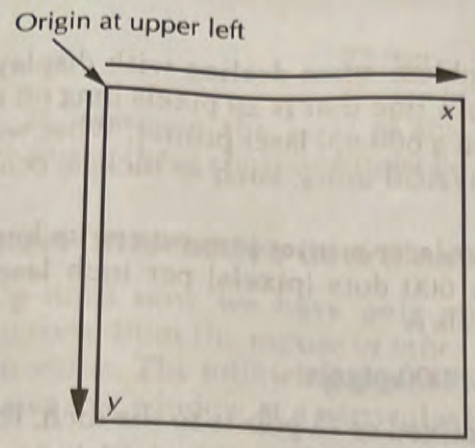


Figure 3-7 Device coordinates

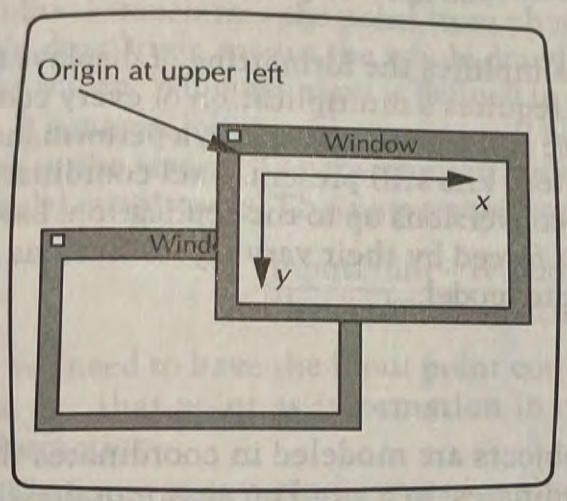


Figure 3-8 Window coordinates

free to draw. Most such systems also provide a frame around this window that allows the user to manipulate the window's location and size. The program's coordinates are placed inside of the window as if the frame did not exist, as shown in Figure 3-8. The programmer then treats the window as a display in and of itself and generally ignores the larger space of device coordinates.

Window coordinates and sizes, like display coordinates and sizes, are almost always expressed in pixels. Because some windows can receive mouse events that are outside of their boundaries, mouse events in some systems are reported in display coordinates rather than window coordinates. It is important to know how a particular system handles mouse coordinates so that the appropriate conversions are done to make the input coordinates and the drawing coordinates consistent.

3.2.2 Physical Coordinates

Pixel-based device coordinates can be a problem when dealing with display devices of varying resolutions. For example, a line that is 20 pixels long on a 1024×1024 screen appears very differently on a 600-dpi laser printer. What we need is to specify display coordinates in physical units, such as inches, centimeters, or printer points.

Let us suppose that we want to access our laser printer in terms of inches. Suppose also that we know that this is a 600 dots (pixels) per inch laser printer. The conversion from 5 inches to pixels is

$$5 \text{ inches} \cdot 600 \text{ dpi} = 3000 \text{ pixels}$$

Printer points for defining font sizes are defined as 72 points to the inch. For a 400-dpi printer, the height of 12-point text would be

$$\frac{12 \text{ points} \cdot 1 \text{ inch}}{72 \text{ points}} \cdot 400 \text{ dpi} = 67 \text{ pixels}$$

Defining devices in physical units simplifies the formatting of displays to be used on a variety of media but also requires a multiplication of every coordinate by some constant. Because these conversions can cause a performance problem on low-end machines, some tool kits still present pixel coordinates as the model for devices and leave any conversions up to the application. Laser printers, on the other hand, have been forced by their varying resolutions to present physical units as their coordinate model.

3.2.3 Model Coordinates

In many situations, the information objects are modeled in coordinates that are very different from the display coordinates. In a word processor or drawing package, the coordinates of the model are in physical units for printing because the application is modeling exactly what is going onto the page.

For an architectural drawing, however, the model units are feet or meters when designing buildings. Thus a scaling transformation is required that transforms feet, for example, into inches, which can then be used as physical display units. A possible scaling might be 10 feet to the inch. This would make the total transformation (for an example length of 22 feet) from model coordinates to a 100-pixel-per-inch display

$$22 \text{ feet} \cdot \frac{1 \text{ inch}}{10 \text{ feet}} \cdot \frac{100 \text{ pixels}}{1 \text{ inch}} = 220 \text{ pixels}$$

The constants in this formula can be combined into a simpler formula that can be used on all parts of the drawing of the building:

$$22 \text{ feet} \cdot \frac{10 \text{ pixels}}{\text{foot}} = 220 \text{ pixels}$$

If, however, the scale of the drawing changes or the display device is changed, these constants must be recalculated.

3.2.4 Interactive Coordinates

Up until now we have only considered output. When an input point is received from the mouse or other device, the mapping must work in the other direction. The following general formula maps a point in some model coordinates to a window at a particular location on the screen:

$$\text{ModelPoint} \cdot \text{DrawScale} \cdot \text{PhysicalToPixel} + \text{WindowOrigin} = \text{OutputPoint}$$

The DrawScale transforms this point into physical display units, PhysicalToPixel transforms the point from physical display units into pixels, and the WindowOrigin moves the whole drawing to where the window is located on the screen. WindowOrigin is defined in pixels.

If a mouse input is received, it will be defined in pixels relative to the upper left of the screen. The transformation must be reversed to produce a point in model coordinates. This new transformation is

$$\frac{(\text{InputPoint} - \text{WindowOrigin})}{\text{PhysicalToPixel} \cdot \text{DrawScale}} = \text{ModelPoint}$$

We need to have the input point converted to model coordinates so that we can use that point as information in changing our model of the application information.

There are many more issues involved in defining the geometric mappings between application models and actual displays. The simple analysis we have done here is only the beginning. In Chapter 10, we will discuss a more complete system based on homogeneous coordinates and matrix algebra.

3.3 Human Visual Properties

In order to interactively present information to users, we need to understand a little about the human visual system. This understanding is essential in designing presentations and interactions that are understandable and pleasing to the eye.

3.3.1 Update Rates

Early moviemakers were limited by their technology in the number of frames of film that could be presented to a viewer in a second. Consequently, early silent films appear very jerky. We perceive the movement but it does not look real. Experimentation has shown that when images are presented to viewers at more than 20–30 frames per second (fps), the images fuse together and appear to be moving continuously. This is because the frame rate has exceeded the rate at which the visual system samples the inputs to the retina. NTSC video is 29.997 fps, film is 24 fps, and PAL video is 25 fps.

On a 30-MIPS (millions of instructions per second) computer—the available rate when this book was begun—there are 1 million instructions available to update the display frame in 1/30 of a second. Although 1 million instructions seems like a lot, remember that a 640×480 display (television resolution) has 307,200 pixels, leaving 3.25 instructions per pixel if every pixel is to be manipulated between each frame. On a 300-MIPS computer—the rate in common use by the time many of you read this book—there are still only 32.5 instructions per pixel per frame. This is a serious problem for video applications.

For normal interactive use, however, we do not change every pixel in every frame; therefore the magic number of 30 fps for smooth motion is not required for most interactive uses. If the user wants to drag an object across the screen, experiments have shown that 5 updates per second to the object being dragged are sufficient to maintain the "interactive feel." Obviously, improving this update rate up to 30 updates per second will make the movement appear more smooth and natural. For dragging tasks, 5 updates per second is the lower bound for acceptable interaction. This means that our display update process, for dragging purposes, must complete in less than 1/5th of a second to be acceptable.

For interactions that are not continuous, such as displaying a new student record or finding a word in a document, delays of 1–2 seconds are acceptable. In these situations, we are not trying to create a smooth movement but are visually moving to a new context. Such context movements on the part of users are much slower because of the scanning and understanding that must occur. For important or time-consuming tasks, users will tolerate delays of much longer than 2 seconds, but beyond 10–15 seconds the feel of the interface is no longer interactive.

3.4 Graphics Hardware

A brief introduction to basic graphics devices is in order, since these are the devices for which our images will be generated.

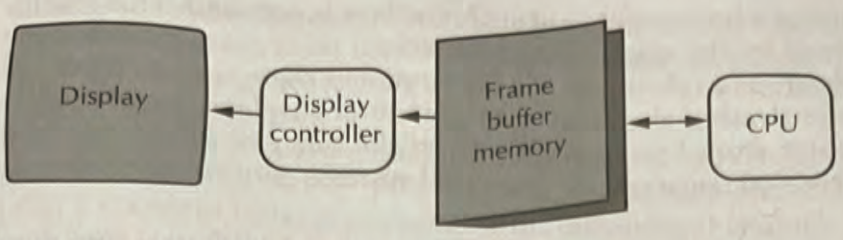


Figure 3-9 Frame buffer architecture

3.4.1 Frame Buffer Architecture

The frame buffer, shown in Figure 3-9, is the dominant architecture for display devices. The graphics package, which is part of the interactive tool kit running in the central processing unit (CPU), sets pixel values into the frame buffer memory. The frame buffer is the repository of the image that is being displayed on the screen. In many cases, the CPU can read the image out of the frame buffer as well as modify it. Various display technologies use differing techniques to convert the contents of the frame buffer into a visible image.

3.4.2 Cathode Ray Tube

The most popular display device is the cathode ray tube (CRT), which is the basis for standard televisions. For such a device, the display controller scans the frame buffer memory from top to bottom, left to right to retrieve pixel values. Simultaneously with this scan of the frame buffer, an electron beam is scanning across the face of the display in a similar pattern. The display controller modifies the intensity of this beam based on the pixel values found in the frame buffer. This produces an image on the phosphor of the screen. In most color displays, there are three beams, one for each of the primary colors, red, green, and blue.

The phosphor image decays based on the persistence of the phosphor. If a long-persistence phosphor is used, the image will not decay by the time the display is refreshed again. Too long a persistence means that fast-moving objects have ghosts of the object trailing behind. If a low-persistence phosphor is used, then the image will start to decay before the display controller can redraw it, causing the display to appear to flicker. If the display is refreshed 30 times per second, there is a conflict between ghosting from high-persistence phosphors and the flicker of low-persistence phosphors. On good displays, this is resolved by refreshing the screen 60 times per second (60 Hz) so that the user cannot perceive the flicker. On higher-quality displays, this may go up to 75 Hz.

The resolution of a huge number of such displays is 640×480. This resolution is determined by the resolution of American television. By using the same CRT hardware as a television, the development costs can be amortized over all televisions rather than just over computer displays. Good-quality CRTs for computer work have resolutions on the order of 1024×768, with some in the 1200×1000 range and experimental systems going much higher.

3.4.3 Liquid Crystal Display

Most portable computers use liquid crystal displays (LCD) because they are flat and have low power consumption. By placing a charge on the liquid crystals, the polarization of light passing through those crystals is changed. When used in conjunction with polarizing filters, such crystals can be made transparent or opaque, which produces the image.

Low-cost LCDs use a passive-matrix technique where the circuits for controlling the crystals are located at the periphery of the screen and the settings of the various pixels must be done sequentially through the screen. This results in a constant image without flicker, but the time required to change a pixel's color is much slower than 30 fps. Depending on how slow this time is, fast-moving objects, such as cursors, may "submarine" or disappear while moving quickly. This is because the cursor has moved from its location before the display was fully changed.

More expensive LCDs use an active-matrix technology that places the control circuitry for each pixel next to the crystals that are being controlled. This yields higher-contrast images, much faster times to change pixels, and eliminates the submarining behavior of cursors.

All LCDs use a frame buffer architecture similar to that of the CRT. The difference is that the display controller is controlling the opacity of crystals rather than the intensity of an electron beam. Similar technology is used by display devices that have not yet gained a wide market, such as plasma panels and others.

3.4.4 Hardcopy Devices

Hardcopy devices are only indirectly involved with the user interface. However, many of the same software techniques—and in a good software design the same code—are used to generate both hardcopy and screen output. The major difference between hardcopy devices and the screen is that the CPU does not have direct access to the memory in the frame buffer of a hardcopy device. Communication bandwidth is a large factor in drawing on a hardcopy device.

One of the oldest technologies for hardcopy graphical output is the pen plotter. This device is based on the stroke image model, except that the coor-

dinates of stroke objects are sent to the plotter in physical coordinates. The plotter then moves a pen across the paper to draw the stroke. This process has all of the imaging limitations of the stroke model, including the inability to present realistic images.

Modern laser printers are based on the frame buffer model. The primary difficulty is that in a 600-dpi printer, the frame buffer requires 3.6 megabytes to print a standard letter-size page. Communicating this much information for each page is prohibitive. For these reasons, laser printers generally use a region model. The most popular of these is PostScript.[1] The PostScript language is based on FORTH,[2] which allows the printer to be programmed by the print driver software. The primitive elements are regions bounded by splines and straight lines, and the printer itself fills the frame buffer from these region definitions. This method allows for very compact communication of very precise and sophisticated images.

In the case of a photograph, however, the region image model is not sufficient. Most laser printers can also accept pixel-based images that are then rendered into the printer's frame buffer. Such images are usually much lower in resolution than the printer's resolution, which somewhat mitigates the communication-bandwidth problem.

From the frame buffer a laser writes an electrostatic charge on a drum or belt. Toner (powdered ink) is attracted to the charges and sticks to the surface wherever a charge has been written. The drum or belt is then rolled across a piece of paper to which the toner sticks. The toner is heat fused to the surface of the paper so that it will not rub off. In addition to the standard laser printer technology, there are ink jet printers, which use frame buffers of lower resolution, and various high-quality color printing technologies such as dye sublimation and wax transfer.

3.5 Abstract Canvas Class

In building our software architecture for the presentation of user information, we need an abstraction for a drawing surface. We will call this a Canvas. A Canvas is an abstract class that defines the methods we will use for drawing. Many user interface tool kits or graphics packages provide such a class. If they do not, it is essential that the application programmer design one.

The Canvas class defines a uniform model for drawing on a 2D surface that can be used everywhere in the application. A Canvas has a width and height and defines its physical units. From this abstract class, we can define subclasses for a window on the screen, an image in memory, various hardcopy devices, or a file in which a picture is to be saved for drawing later. Each of these subclasses is implemented differently because of the varying destinations for the output image, but through the interface defined by the abstract class they can all be treated the same.

There are some variations among the subclasses to Canvas that must be considered. The first is the actual pixel resolution of the Canvas. A screen window that is 8 inches high cannot represent the same information as a laser-printed page of the same size. In some cases, the application program may need to be aware of the resolution limitations of the screen window and may need to adapt the kind of detail it attempts to display. Another major difference is in the capacity of the pixels. It is important to know if we are drawing on a Canvas that only supports black and white or on one that supports gray scale or full color, because it makes a significant difference in the way the presentation is defined. A third difference is that some Canvases, such as a screen window, can dynamically change size while others, such as a printer page, cannot.

All modern user interface packages provide a uniform model for drawing on a 2D surface. This is independent of what or where this surface might be. In X, there is a standard interface for drawing to windows, drawing to images in memory, and saving to files. There is no such interface to hardcopy devices. In NeWS and on the NeXT, PostScript is the model for drawing on any drawable surface, including windows, memory images, files, or printers. The View class in NeXTSTEP performs the Canvas role of providing this abstraction. On the Macintosh, the basic Quickdraw package provides GrafPorts as the abstraction for drawing. In MacApp, the GrafPort features are more carefully abstracted into the View class. View has, for example, a subclass called TStdPrintHandler that will draw on the printer using the View abstraction for drawing. Microsoft Windows provides the Graphical Device Interface (GDI), which is not object oriented but provides a uniform interface for drawing on various media including the screen. The GDI associates a particular graphical device driver with a "device context." All drawing occurs through a device context that then translates the drawing commands into appropriate output. In Visual C++, the device context concept from MS Windows is encapsulated into the abstract CDC class, which can handle printing, windows, files, and images.

3.5.1 Methods and Properties

Our drawing functionality can be defined in terms of the methods and properties that our Canvas class provides. The methods can be grouped into drawing of lines and shapes, drawing of text, clipping our drawings so that they do not go outside of specified portions of the Canvas, and controlling color and texture. Properties here are like fields of the class except that they are accessed by means of methods rather than directly. The reason for this form of access is that when they are set, most subclasses such as a printer require that additional processing be performed. Such properties include the coordinate system, the physical drawing units, the current drawing color, or the line width.

Figure 3-10 Properties of a shape

All of these properties are accessed by methods on the Canvas class. The actual implementation of these methods is different for each kind of output surface that we are trying to draw onto.

3.6 Drawing

A fundamental part of graphical output is the drawing of geometric shapes. This section discusses how we model such shapes and also covers some of the geometry that will be needed as we interact with these shapes. Text is such a special case that we treat it separately.

All graphical shapes have a few properties in common. These include basic geometry of the shape, line or border width, fill or line color, and a pattern or texture. For example, the rectangle in Figure 3-10 has its geometry defined by its two corner points. It has a width and color for its border as well as a color for filling in the rectangle.

We could define our rectangle facility with a single method of the following form:

```
void Canvas::Rectangle(X1, Y1, X2, Y2, LineWidth, LineColor, FillColor)
```

Such methods, however, are painful to use because of the number of parameters required. In most cases, the LineWidth, LineColor, and FillColor will be the same for a large number of rectangles and other shapes.

For this reason, most graphics packages limit their drawing methods to specify only geometric information. The other properties are handled by current settings on the Canvas object. For example, we might provide Canvas with the following methods:

```
void Canvas::Rectangle(X1, Y1, X2, Y2)
void Canvas::SetLineWidth(LW)
long Canvas::GetLineWidth()
void Canvas::SetLineColor(LC)
```

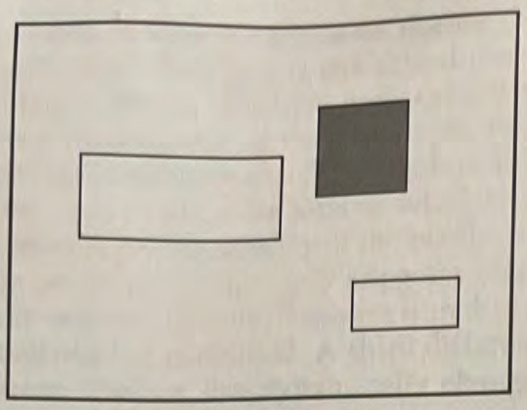


Figure 3-11 Rectangles and their properties (see text)

Color Canvas::GetLineColor()
void Canvas::SetFillColor(FC)
Color Canvas::GetFillColor()

Note that each property has a Set and a Get method. When a rectangle—or any other shape—is drawn, the settings of the current properties are used to supply the remaining information. In most systems, these settings are stored in the Canvas objects; thus they may be different for different windows or printers. To draw several rectangles, we might do the following:

```
Canvas Cnv;

Cnv.SetLineWidth(1);
Cnv.SetLineColor(Black);
Cnv.SetFillColor(White);
Cnv.Rectangle(50, 50, 150, 100);
Cnv.Rectangle(200, 120, 250, 140);
Cnv.SetFillColor(Gray);
Cnv.Rectangle(180, 20, 220, 40);
```

This produces the picture shown in Figure 3-11.

Current property settings are used widely in graphics packages. This method does have some drawbacks, however. If you're not careful, you may forget to set the properties before drawing the objects. In some cases this might work, because the properties are already correct. Sometime later, when the program is changed, the properties will be different and suddenly your drawing code does not work correctly.

Most drawing in graphics systems is performed in terms of geometric primitives and text. The geometric primitives can be divided into paths and closed

Figure 3-12 Control points of a line

shapes. We only discuss the basic geometries of these shapes in terms of how they are specified by the programmer. You must consult a graphics text for the details of how to convert these shapes into pixels on a screen or page.

3.6.1 Paths

The first set of geometric objects we discuss are paths, or 1D objects that are drawn in a 2D space. The simplest description of these objects is that they have no inside or outside; they are infinitely thin. Most of our geometric questions involve paths. When paths define the border of a filled shape, the geometry of the shape is determined by the geometry of the path, which is its border. The paths that we discuss are lines, circles, arcs, ellipses, splines, and complex piecewise paths.

Lines

Lines are the simplest of all paths and provide the easiest geometric solutions. Figure 3-12 shows a line and its control points. Control points define the line's geometry and we use them to compute the coefficients for our line equations.

Circles

The next most interesting geometric shape is the circle. The simplest model for a circle is defined by its center and its radius. The radius is not a control point but it does provide the simplest geometry. Circles can also be defined by their center and some point on the circumference of the circle, as shown in Figure 3-13.

Arcs

Arcs are fragments of circles and as such use the same equations as circles. The question is how to define the restricted part of the circle that forms the arc. This is most easily done by using the parametric equations for a circle and restricting the parameter values; this geometry is discussed in Chapter 9.

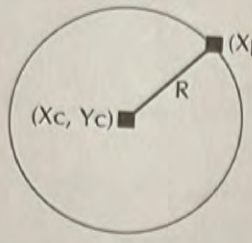


Figure 3-13 Geometry of a circle

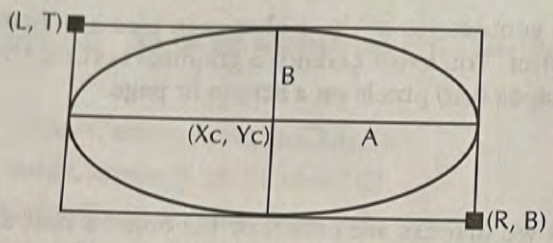


Figure 3-14 An ellipse

Ellipses and Elliptical Arcs

There are two major classes of elliptical shapes to consider. The simplest class is the set of ellipses whose major and minor axes are parallel to the *x* and *y* axes. These are computationally quite tractable and can be understood by expanding on the equations for circles. A much more difficult class is the set of ellipses that can have any orientation for their major and minor axes; this class of shapes is more computationally expensive. There are applications where the general class of ellipses is required, but we can go a long way without them. For our discussion, we restrict ourselves to the ellipses aligned with the *x* and *y* axes, as shown in Figure 3-14.

Splines

Curved shapes are frequently called splines because of the original mechanical tools used for drawing such curves by hand. Most curves are defined as parametric cubic equations. Their geometry and definitions are discussed in detail in Chapter 9.

Piecewise Path Objects

The path objects defined so far are generally not sufficient for all of our drawing needs. Paths such as those shown in Figure 3-15 are constructed by piecing together a variety of simpler path objects to construct the desired figure. A major issue with such drawings is the continuity between pieces, that

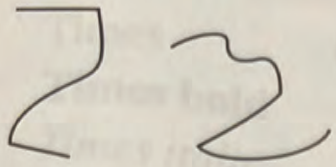


Figure 3-15 Piecewise paths

Figure 3-16 Closed Shapes

is, whether the pieces connect smoothly or whether there is a sharp corner. Issues of how to ensure continuity are discussed in Chapter 9.

3.6.2 Closed Shapes

The graphical objects described above only draw lines. Frequently we want closed shapes that can be filled, as shown in Figure 3-16. They are generally defined by a border, which is a path object. The filled shape is defined as all points lying inside of the closed shape. The definition of the inside and the outside of a closed shape will be deferred until we have a better definition of the geometry of these shapes.

3.7 Text

The drawing of text is one of the most common needs in graphical user interfaces, not only for text editors but also for a variety of labeling and information presentation needs. Unfortunately, this can be one of the more complex parts of an application, due to the wide variety of textual representations that have evolved over time and the various mechanisms used for representing them.

Courier (fixed-space font)
Avant Garde (sans serif)
Helvetica (sans serif)
Times Roman (serif)

Figure 3-17 Font families

3.7.1 Font Selection

The first task in drawing graphical text is to select the font in which the text is to be drawn. There are a variety of ways in which fonts are specified, but in general there are three primary arguments: the font family, the style, and the size. The font family defines the general shape of the characters in the font, as shown in Figure 3-17.

The Courier font family is a monospaced or fixed-space font, meaning that every character in the font has the same width. Fixed-space fonts work like typewriters and character-based terminals. Alignment and spacing of characters is very easy, but the resulting text is not as pleasant to read nor is it very space efficient, since the character "i" gets the same space as "G." The remaining fonts shown in Figure 3-17 are proportionally spaced fonts. Each character has a different width depending upon its needs.

The Times Roman font is an example of a serifed font. Each vertical stroke that reaches the baseline has a little foot, or serif, on the bottom. The Avant Garde and Helvetica fonts do not have such serifs. Compare the "i" in Times Roman with the "i" in Helvetica. When characters from a serifed font are strung together, the serifs tend to form a line, which visually defines the line of text. This facilitates eye tracking across the line while reading. In general, long lines of serifed text can be read more quickly than sans serif text because the eye has less difficulty in horizontal tracking. The advantage with serif type for reading on paper does not necessarily hold true for reading on screens due to resolution problems. With lower resolutions, the serifs may make letters harder to discriminate.

Font Style

Within a font family, there are frequently a variety of styles of font face. These styles are variations on the basic character shapes, as shown in Figure 3-18.

On some systems, various styles such as Times Roman bold are treated as completely separate fonts. On other systems, they are separate fonts but are

Times
Times bold
Times italic
Times bold italic

Helvetica
Helvetica bold
Helvetica italic
Helvetica bold italic

Figure 3-18 Font styles

9 point
10 point
12 point
14 point
18 point
24 point
36 point

Figure 3-19 Font sizes

grouped together within families. On the Macintosh, there may only be a Times Roman font, and the font system automatically distorts the base font to create bold, italic, and outline versions of the font.

Font Size

The third primary control on font selection is the font size or vertical height of characters. This is usually expressed in *points*, where 1 point = 1/72 of an inch, as shown in Figure 3-19. A point is a unit of measure that has carried over from printing. Unfortunately, in many systems the point size of a font is only loosely related to the actual vertical size of the characters as displayed on the screen. The problem lies in the low resolution of many graphics displays. Some more complex fonts are slightly larger than their point size would indicate because that is how many pixels are required to clearly display characters in the font. On most laser printers, however, the point size of a font accurately reflects its height.

The set of things that can be specified about a font is not restricted to family, style, and size even though these are the most common specifications. Other controls vary widely with the particular windowing system being used.

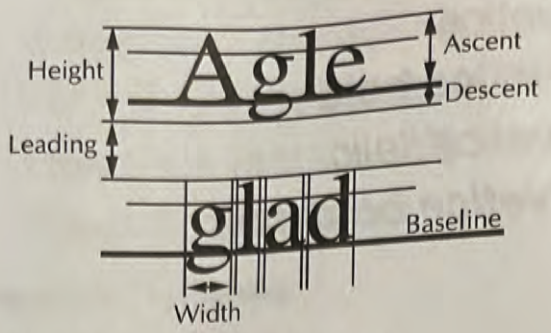


Figure 3-20 Font size information

Font selection is a major problem in developing applications that port from one system to another. In fact, porting a drawing or document between machines that are running the same windowing system can cause font problems because of the differences in the fonts actually installed on the machine.

3.7.2 Font Information

Having selected a font family, style, and size, there is a variety of information we need to know about the font in order to display text at the appropriate positions on the screen. Consider the characters in Figure 3-20.

As with the font family, style, and size, there are a variety of ways in which font geometry is represented. Figure 3-20 shows the general characteristics, although these vary among graphics packages. The most useful piece of information is the height since that indicates how much vertical space should be allocated to accommodate a line of characters in this font. The leading is the space between multiple lines of text. In some systems, the leading is incorporated into the height of the font. In some, it is an additional parameter on the font. In others, the leading has to be handled separately by programmers using the font.

In most systems, the vertical position of text is indicated by the y coordinate of the baseline. Note that in the case of the character "g," this is not the lowest extent of the line of text. The distance between the baseline and the lowest extent of any character in the font is the descent. The distance between the baseline and the highest extent of the font is the ascent. These two measures, along with the height, define all of the information normally needed to position text vertically.

Horizontal positioning of text is a little more problematic because the width of each character may vary. All windowing systems provide calls that return the width of an individual character or a string of characters, given a particular font selection. In some fonts, the space between characters is actually included in the width of each character. In others, it is specified sepa-

rately. The vertical measurements are uniform for an entire font. The horizontal measures vary from character to character. The only exceptions to this are fixed-spaced fonts such as Courier.

3.7.3 Drawing Text

In most systems, drawing text is relatively simple. A current font is selected in which all following text will be drawn. For each text string to be drawn, a reference point and the string to be drawn are specified, for example:

```
Canvas Cnv;

Cnv.SetFont("Times", Bold, 10);
    Times Roman; bold; 10 point
Cnv.Text(10, 20,"This is the text");
```

In most cases, the reference point (10, 20) specifies the baseline and the leftmost position of the first character. There are other alternatives, however. We could specify that the *y* coordinate of the reference point defines the bottom or the top of all characters rather than the baseline. We could also specify that the *x* coordinate defines the center or rightmost position of the string. Some systems provide only reference points at the leftmost baseline, since all of the others are easily calculated by programmers using the system. MS Windows, on the other hand, provides a SetTextAlign routine that sets the current alignment for subsequent reference points. This allows for all of the possible variants.

Suppose we wanted to output multiple lines of text that are appropriately spaced depending on the font being used. It is never wise to hard-code font heights and widths because sooner or later some user will want a different font and all that code will need to be rewritten in order to accommodate the new flexibility. To solve this problem, we can use the font height information, as follows:

```
Canvas Cnv;
long Height;

Cnv.SetFont("Times", Plain, 12);
Height = Cnv.GetFontHeight();
Cnv.Text(10, 20, "This is the first line");
Cnv.Text(10, 20 + Height, "This is the second line");
Cnv.Text(10, 20 + 2*Height, "This is the third line");
```

For an arbitrary number of lines, this is done in a loop incrementing the y position by Height each time.

A slightly more complex problem is to output "Some **bold** text" on a single line. This string must be output as three separate strings with different fonts, and they must be correctly positioned relative to each other. This can be done as follows:

```
Canvas Cnv;
long Width;

Cnv.SetFont("Times", Plain, 12);
Cnv.Text(10, 20, "Some");
Width = Cnv.StringWidth("Some");
Cnv.SetFont("Times", Bold, 12);
Cnv.Text(10 + Width, 20, "bold");
Width = Width + Cnv.StringWidth("bold");
Cnv.SetFont("Times", Plain, 12);
Cnv.Text(10 + Width, 20, "text");
```

These two examples illustrate what is necessary when outputting strings in multiple fonts. If we were faced with multiple lines and varying fonts per line, we would need more complex calculations of line height and width.

3.7.4 Outline vs. Bitmapped Fonts

In order to draw a textual character into a frame buffer or onto some other pixel-based display, we must know the set of pixels that make up the character. In many windowing systems, fonts of characters are simply defined as the bitmaps, or set of pixels, that make up each character. This technique is very efficient but it has several problems. When the size of the font is very large, the amount of space required to store all of the bitmaps becomes a problem, especially when each font size requires a separate set of bitmaps. This is even a problem with small font sizes when drawing onto high-resolution printers. At 300 dpi, a 10-point font can take over 20 KB of space. A 72-point font at 300 dpi would take 1.4 MB. Using very many sizes of a particular font can easily consume a large amount of space.

To resolve this problem, many systems now represent fonts by storing characters as closed shapes. This means that only the outline needs to be stored, as shown in Figure 3-21. The outline of the letter A is stored as a piecewise path of lines and splines, or on some systems as lines and elliptical arcs. Such character definition can be readily scaled to any size and then converted to bitmaps as needed rather than storing all of the font sizes that might ever be required. This also allows advanced drawing packages to treat characters as geometric shapes that can be manipulated like any other graphical object.

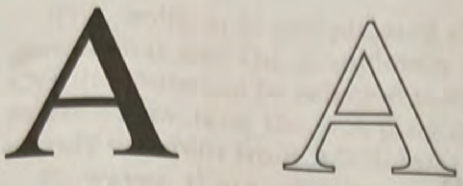


Figure 3-21 Outline fonts

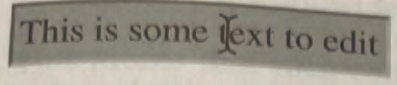


Figure 3-22 Editing a line of text

3.7.5 Character Selection

Since text is a major part of many user interfaces, editing text requires that the user interface software be able to select a given character in a text string, given a specific mouse position. Take, for example, the text type-in box shown in Figure 3-22. The problem is that the sizes of each of the characters are different. All systems that support proportionally spaced text provide a mechanism for obtaining the width of each character for a given selected font. On most systems, there is a single call that returns an array of character widths. Indexing this array with the ASCII value of a character yields the desired information. The algorithm for selecting the character, then, is as follows:

```
Canvas Cnv;
int StringLeft, =
```
The leftmost location where the string was drawn.
```
int MouseX, =
```
The X coordinate of the mouse location.
```
char * Str, =
```
A pointer to the string being edited.
```
int CharWidths[256];
int I, X, StrLen;

Cnv.SetFont("Times", Plain, 12);
Cnv.GetCharWidths(CharWidths);
StrLen = Length(Str);
I = 0;
X = StringLeft;
while(I < StrLen && X < MouseX)
```

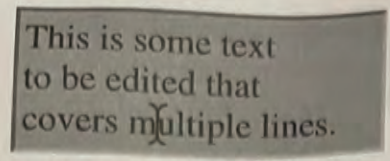


Figure 3-23 Selecting text from among multiple lines

```
{
    X = X + CharWidths[ Str[I] ];
    I = I + 1;
}
```

Selected character is at Str[I–1].

Sometimes, as in Figure 3-23, there is a need to select a text point from within multiple lines of text. In this case, we must take into account the text height in determining which line of text is being selected. The general algorithm for this is as follows:

1. Compute the line of text being selected using the *y* position of the mouse and the text height plus leading.
2. Starting at the beginning of the text string, run through the string, counting new line characters, until the beginning of the appropriate line is found.
3. Apply the single-line selection algorithm described above, starting at the beginning of the line of text.

3.7.6 Complex Strings

Our discussion so far has covered the drawing of simple strings of ASCII text. Some graphics packages support the creation of compound strings that contain additional formatting information. The simplest of these include style information such as, bold, italic, or change of font. These issues complicate selection and computation of text height.

The most complicated issues in text output arise from internationalization. The ASCII standard for representing text is an American standard and does not support the special needs of even closely related languages such as French or German. These languages, although they use the Latin alphabet, require special accents (umlaut, circumflex, etc.) over letters that are necessary to the pronunciation of those languages. Most of the Latin alphabet issues are met by extending ASCII beyond the 128 characters to use the full 256 possibilities in an 8-bit byte.

The problem is complicated slightly by languages such as Russian or Bulgarian that use the completely different Cyrillic alphabet. In such a case, Cyrillic fonts can be selected that map the 256 values in a byte into the appropriate characters; the mapping of 8-bit integers into character faces is completely different from ASCII but the software techniques are identical.

However, there are languages—such as Hebrew and Arabic—that are written right to left instead of left to right. This means that the text reference point for drawing and the algorithms for text selection are all reversed. Because of the worldwide use of English in technology, users of such languages sometimes want to embed English words in the middle of sentences. This means that such compound strings contain fragments of both left-to-right and right-to-left text.

Oriental languages, such as Chinese or Japanese, use ideographic representations that have a large number of characters, each of which represents a single word or a word fragment. Except for the fact that such languages are frequently written top to bottom instead of left to right, many of the algorithms are the same. The biggest problem is that there are far more than 256 characters. To handle such strings, 2-byte character encodings and a much larger font are required. Such languages—with thousands of characters—also have keyboard entry problems in addition to the graphics problem of drawing text strings.

In order to be truly international, an interactive application must support the mixing of a variety of languages in a single drawing, document, or database. This requires more general character representations such as UNICODE, which attempts to define a 2-byte encoding for all of the world's major written languages.[3]

3.8 Clipping

The clipping problem arises when we want to limit drawing to a particular area of the screen. There are a variety of cases in which this might occur. The simplest is clipping to a rectangle, as shown in Figure 3-24. This is a commonly used form in computer graphics for clipping displayed objects to the bounds of the window in which they are to be displayed.

The rectangular-window clipping model is not sufficient, however, for most modern windowing systems. As Figure 3-25 shows, at least rectilinear clipping regions are required. If any drawing is to be done on the window in the back, all of its output must be clipped to the rectilinear visible region that remains when other windows are laid over the top. And as shown in Figure 3-26, such rectilinear regions must allow for holes if they are to accommodate all of the cases.

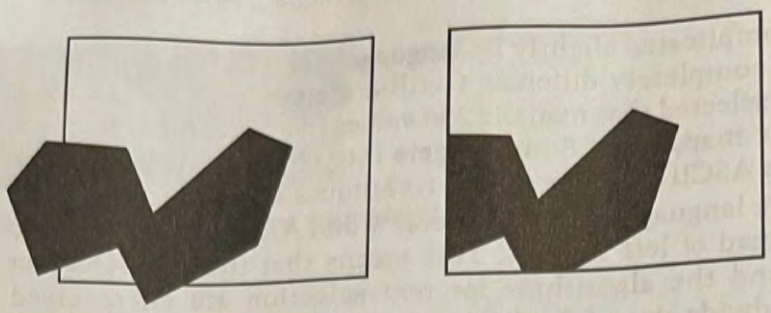


Figure 3-24 Rectangular clipping

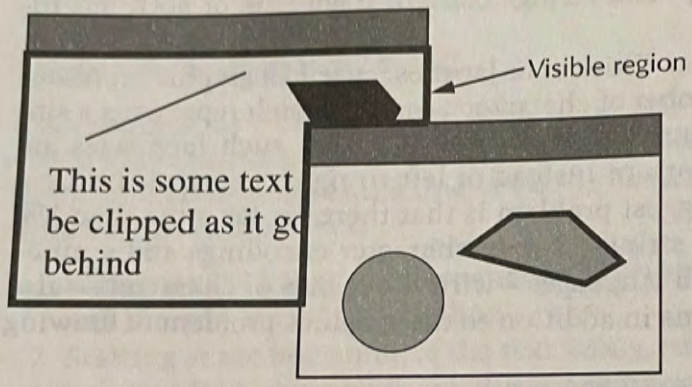


Figure 3-25 Rectilinear clipping to visible regions

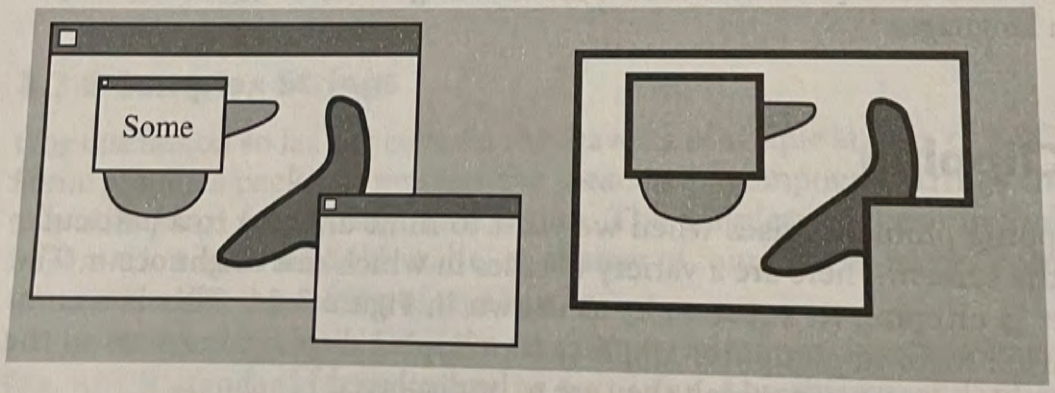


Figure 3-26 Rectilinear region with holes

3.8.1 Regions

A variety of types of regions can be used as the basis for clipping objects to the exact visible area. The critical consideration when choosing a particular type of region as the basis for clipping is the complexity involved in computing the

intersections of the graphical objects that are being clipped and the boundaries of the region.

Types of Regions

As noted above, the simplest clipping region basis is rectangles. The most common region definition in current windowing systems is rectilinear. Rectilinear regions are closed shapes with their edges defined entirely by vertical and horizontal lines. Intersections with vertical and horizontal lines are just about the simplest geometry to compute; that, coupled with the general usefulness of such regions, is the reason they are used so frequently.

Some windowing systems support clipping to arbitrary polygons, which is a generalization of the rectilinear region in that the lines are not restricted to being vertical or horizontal. This form is not used very often because it only provides a limited increase in the kinds of regions that can be defined but it substantially increases the computation cost of the clipping.

PostScript-based systems, such as NeWS or NeXTSTEP, can clip to any complex shape bounded with a combination of lines and curves. This is a highly flexible and powerful model but it requires significantly more computation to do the clipping.

Some systems support clipping regions that are defined as pixel masks. A set of pixels with a 1 value is defined for the entire area within a bounding region. If a given pixel is 1, then objects drawn in the region will show at that pixel. If the pixel is 0, then drawn objects will not show at that pixel. This model is very fast when handled at the lowest levels of the system where objects are actually scan-converted to pixels. Also, the model is very powerful in that any shaped region can be defined. It is not very space efficient, however, and it has the problem of being very resolution dependent. One alternative is to define the clipping region as complex curved or polygonal shapes and then to scan-convert those shapes to a pixel map at the particular desired screen resolution.

One mechanism for reducing the space requirements of pixel-based regions is to run-encode them. For a given horizontal line of pixels, the 1s and 0s are grouped together. Instead of storing each 1 or 0 for each pixel, we can store the number of 1s and 0s in a row. Consider the shape in Figure 3-27. For any given horizontal line, there is a string of 0s for the empty space to the left, a row of 1s for the space inside the shape, and a row of 0s for the empty space to the right. By storing just three numbers, we represent all of the shape information on any given row.

Set Operations on Regions

When working with a clipping region, it is helpful to think of such a region as a set of points on the 2D plane that lie inside of the region. We can then work with the region in terms of set operations. Throughout the rest of the

Figure 3-27 An arbitrary clipping region

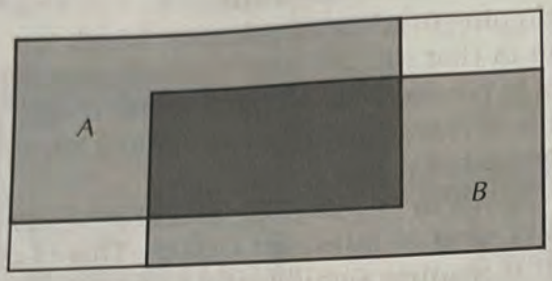


Figure 3-28 Set operations on rectangles

book, we ignore the particular style of clipping region that is supported by a given graphics package and simply treat them as abstract objects defined in terms of sets. The set operations that we are interested in are union, intersection, and difference.

Closure A most important property when considering types of regions is closure under the set operations. *Closure* means that given some set, the result of any combination of operations on that set will yield a member of the same set. Take, for example, the rectangular regions A and B in Figure 3-28.

Note that the union $A \cup B$ is not a rectangle and that the difference $A - B$ is also not a rectangle. The intersection $A \cap B$ is a rectangle. This example illustrates that rectangular regions are closed under intersection but are not closed under union or difference. This lack of closure can be a problem in terms of the usability of a particular class of regions.

We can overcome the closure problems with rectangular regions by representing regions as a list of nonoverlapping rectangles rather than as a single rectangle. Lists of rectangles are closed under union and difference. In Figure 3-28, we can represent $A \cup B$ by dividing B into three small rectangles, one of which is $A \cap B$ and the other two of which make up the remainder of B. The result $A \cup B$ is A plus the two smaller rectangles from B. Similarly $A - B$ can be represented by slicing A into three smaller rectangles and retaining two of them. Any rectilinear region can be represented by a list of rectangles.

Abstract Class for a Region

For the purpose of this text, we assume that all regions are rectilinear and are composed of lists of rectangles. All subsequent uses of regions, however, are based on the abstract Region class that has the following methods:

Region Region::Union(Region)

Returns a region that is the union of the target region with the argument.

Region Region::Intersect(Region)

Returns a region that is the intersection of the target region with the argument.

Region Region::Difference(Region)

Returns all parts of the target region that are not in the argument.

int Region::IsEmpty()

Returns true if the region is empty and false otherwise.

Rectangle Region::Bounds()

Returns the smallest rectangle that completely encloses the region.

int Region::IsInside(Point)

Returns true if the point is inside of the region and false otherwise.

Region MakeRegion(basic primitive shape)

Constructs a region from a basic primitive object.

In addition to the set operations, there is also a method for determining the rectangular bounds for a region, used for quick tests on the region before considering all of the complexity of the actual region definition. The IsInside method is used with mouse selection to determine if the region has been selected.

By using these methods on the abstract Region class, we can take care of most of our user interface needs. We use the difference operator extensively to handle problems with overlapping windows.

In order to use regions, we need to augment our abstract Canvas class with several methods, as follows:

Region Canvas::BoundingRegion()

Returns the regions that define the outside of the canvas.

Region Canvas::VisibleRegion()

Returns the region of the canvas actually visible through all other canvases that might be overlapping.

void Canvas::SetClipRegion(Region)

Sets the clipping region to the intersection of the argument and the visible region. All future drawing (until this is changed) will be clipped to this region.

Region Canvas::GetClipRegion()

Returns the clipping region.

The BoundingRegion method returns a region because windows may not be rectangular in their definitions. On some systems such as X, nonrectangular windows can be defined. The visible region for a window canvas is defined as its bounding region minus the bounding regions of all windows that lie in front. As will be discussed later, this visible region is very important for drawing information efficiently. A clipping region is used to restrict drawing to some smaller area inside of the canvas. Clipping regions will be used extensively when updating the screen in response to changes in displayed information.

3.9 Color

To understand the use of color, it is important to understand how the human eye perceives color. The retina of the eye is covered with two types of light sensors, rods and cones. The rods are sensitive to a broad spectrum of light. Because the rods are sensitive to a broad spectrum, they cannot discriminate between colors; they primarily sense light intensity or shades of gray. In contrast, there are three types of cones. Because of variations in pigment in the cones, each type is sensitive to a different band of the light spectrum. In humans, there are cones for red, green, and blue.

The eye does not directly measure all of the wavelengths of visible light. For example, yellow light, which falls in the spectrum between red and green, excites both the red cones and the green cones and thus gives a visual sensation of yellow light. The same visual sensation of yellow can be produced by simultaneously providing some light in the red band and some in the green band. The human eye is incapable of differentiating between light in the yellow wavelengths and light that consists of both red and green wavelengths. The fact is exploited by color modeling systems.

3.9.1 Models for Representing Color

There are various ways that color can be represented. Each of these models has different properties and provides a different set of controls for the user to specify color. In the end, however, they must all be converted to red, green, and blue (RGB) colors for actual display.

The RGB Color Model

The mixing of different wavelengths of light to produce the visual sensation of some particular intermediate wavelength forms the basis for all color displays. We can produce any of the human visual sensations of color by using varying intensities of the primary colors red, green, and blue. It has also been shown that the human retina can only distinguish about 64 levels of intensity. This means that we can represent all of the colors that can be sensed by the retina using 6 bits for each primary color, or a total of 18 bits.

The 64 levels of intensity, however, only account for retinal sensitivity, not the sensitivity of the total ocular system. The pupil, by varying the amount of light that enters the eye and strikes the retina, allows the eye to cover a much larger range of light intensity and allows us to function both in bright sunlight as well as in a dark basement. For most reading and other uses of a computer screen, the pupil only makes minor adjustments. This means that 256 levels of intensity for each primary color are more than adequate for human visual needs. We thus find that almost all computer displays represent colors with 24 bits (3 bytes) per pixel. There are frame buffers with a greater range than 8 bits per primary color but the additional bits are for image processing or composition functions, rather than for visual presentation of information.

Most windowing systems allow the programmer to set the current color of the canvas by specifying the RGB values of the desired color. This color is then used in subsequent drawing operations. Because a color can be represented in 24 bits, it is frequently represented by a single long word with special functions that assemble and extract the RGB components of the color. We can augment our canvas with the following methods:

```
long MakeRGB(int Red, int Green, int Blue);
int GetRed(long RGB);
int GetGreen(long RGB);
int GetBlue(long RGB);

void Canvas::SetColor(long RGB);
void Canvas::SetRGB(int Red, int Green, int Blue);
long Canvas::GetRGB();
```

For our abstract Canvas class, we have chosen to represent each of the colors red, green, and blue with an integer number between 0 and 255. In other systems that wish to remain more independent of the color resolution, red, green, and blue are each represented by a floating-point number between 0.0 and 1.0.

The HSV Color Model

Although the RGB model accurately represents what display devices must use to produce color sensations in the human eye, it is not a very good model for users to express a particular color. For example, if users are asked for the RGB values that represent the color of a manila envelope, they are very hard pressed to do so. For this reason, many user interfaces use the hue, saturation, and value (HSV) system for specifying colors.

Hue represents the primary wavelength of light. For most people, this is what they think of when they think of color. The hue might be red, yellow, orange, green, cyan, blue, or purple. The *value* is the intensity or brightness of the light. Thus we have light red or dark red. A yellow or orange hue with very low value would produce brown.

Saturation is a little more difficult to explain, it is a measure of how pure the primary light wavelength is. If there is only red light present, then the color is highly saturated. On the other hand, there may be the same amount of red light but also a lot of blue and green. The hue or primary wavelength is still red and the value or intensity is unchanged, but the color is pink rather than red. With pink, which has low saturation, there is a lot of white mixed with the red color. High saturation means a very pure color while low saturation means that a lot of white or gray has been mixed in.

The HSV model is directly linked to the RGB model, which is required for actually drawing on the display. The code for converting between HSV and RGB is somewhat complicated but not excessively.[4]

The CMY Color Model

For people trained in the visual arts, such as painting and printing, color is not defined in terms of mixtures of light but rather mixtures of pigment. A pigment gets its color from the light that it absorbs (does not reflect). For example, a green piece of plastic appears green not because it is generating green light but because it has absorbed the red and the blue light and is only reflecting green.

For mixing pigments, the cyan, magenta, and yellow (CMY) system is used. These are called the subtractive primary colors; red, green, and blue are the additive primaries. Each of these colors corresponds to the absence of one of the additive primaries. Cyan occurs when there is no red, magenta indicates no green, and yellow is what is left when blue light is removed. Mixing all of the subtractive primaries produces black rather than white because when all of the subtractive primaries are mixed, all wavelengths are absorbed.

In the printing business, it is also helpful to specify a black component in addition to cyan, magenta, and yellow. This need for black is a property of printing and ink rather than the mathematical mixing of colors. Images with greater contrast and clarity can be achieved through the printing process by actually using black ink rather than by mixing the CMY pigments to produce

black. (This is the basis for four-color printing; the system acronym is CMYK, with *K* standing for black.)

3.9.2 Human Color Sensitivity

The human retina has approximately 7,000,000 cones and between 75,000,000 and 150,000,000 rods. These numbers mean that humans are 10 times more sensitive to variations in intensity than they are to variations in hue or the color wavelength of light. This also means that our ability to discriminate fine detail between two areas that have different hues but similar intensity is much worse than our ability to distinguish details that vary in intensity rather than hue.

The makeup of the human visual system has a strong impact on the use of color in a user interface. If the user is presented bright green text on a bright red background, it will be much harder to read than light green text on a dark red background. When presenting bright green on bright red, the rods cannot distinguish between the text and the background; only the cones will be able to distinguish between the two. This means that 90% of human visual capacity is useless in reading the text. In the case of dark on light, however, the rods can distinguish between text and background and can thus contribute to resolving the shape of each character.

The ability of the retina to distinguish differences of intensity is not uniform across the visible spectrum. The eye is much less capable of distinguishing intensity at the fringes of the spectrum (deep red and violet) than it is in the center where the yellows and greens are found. Reds and blues are less easy to resolve than other colors. This means that putting blue text on a dark red background will cause problems because we are forcing the eye to distinguish detail in the areas of the spectrum where it is least capable of resolving differences.

In terms of usability for all people, we must remember that many segments of the population are colorblind. In 1% of the male population, one of the primary colors cannot be sensed. A protanope (red blind) individual is not able to pick up the red primary; for such a person, bright cyan (green and blue) cannot be distinguished from white. A deuteranope cannot detect the green primary, and the very rare tritanope cannot sense the blue primary. In most situations, this is not a problem because most of our visual processing depends on the rods, which are sensitive across the entire visible spectrum. What this means, however, is that we must not present users with visual tasks that require discrimination based solely upon hue. There must also be a contrast in value or intensity.

In addition, the rods and cones are not distributed uniformly across the retina; there is a central area that has a much higher density. This high density of sensors allows much sharper resolution of images. It is this part of the eye

that we use in reading. This makeup of the eye accounts for the very narrow focus of visual attention discussed earlier. We accommodate such a narrow focus by moving our eyes.

3.10 Summary

This chapter has covered the basics of drawing the images that are a central part of graphical interaction. Central to the process of drawing is the question of the coordinate system used to specify the geometry. Device coordinates are those actually supported by the device or system on which images are to be drawn. Physical coordinates provide resolution independence by specifying the physical size of the images that are drawn. Model coordinates allow an object to be specified in the size it is actually being represented rather than the size in which it is rendered. The transformations between these coordinate systems are discussed in more detail in Chapter 10.

Only a brief introduction has been given for output devices. The most common of these devices are the cathode ray tube (CRT), the liquid crystal display (LCD), and various hardcopy devices. For purposes of interactive systems, we encapsulate all of these in the abstract Canvas class; using this class, we can draw on any device regardless of its implementation. The actual details of image creation are handled in the implementations of the various subclasses of Canvas. Most of the drawing on a Canvas is done in terms of geometric primitives such as lines, circles, ellipses, arcs, and splines, or combinations of these primitives. Closed shapes are also formed from such geometries. Such stroke models for graphics objects are easy to specify and manipulate, but ultimately they must be converted to pixels for printing or display. It is also frequently the case that rectangular arrays of pixel values can be drawn on a canvas. Such direct use of the image model supports painting and image manipulations.

Of special interest is the drawing of text, which is complicated by a variety of factors related to various fonts. Outline fonts have clear geometric definitions while resolution-dependent, pixel-based fonts provide speed at some cost of space. Fonts with variable sizes and variable spacing pose additional algorithmic problems when they are being drawn and selected. International fonts, which will become increasingly important, also complicate the problem.

The concept of clipping to restrict drawing to a defined region has been introduced. This is the basis for most windowing and for a wide variety of other interactive techniques. Clipping has also been encapsulated in the abstract Region class, which provides set operations on regions of the drawable space. The geometry of regions is discussed in Chapter 9.

The specification of color is also important. We deceive the human eye in terms of color by only presenting mixtures of red, green, and blue (RGB). This physically based model for color, however, is not very intuitive. Instead, the hue, saturation, and value (HSV) model is a better way to express intended colors. For those trained in the visual arts, the cyan, magenta, yellow, and black (CMYK) model mirrors the behavior of pigments rather than light.

Throughout this chapter, human issues have also been addressed. When drawing images on the screen, the rate at which those images change is important. If we draw changes faster than 30 times per second, the changes fuse into continuous motion because this rate is faster than the sampling rate of the human eye. If we move something across the screen, a redraw rate as slow as 5 times per second preserves the "feel" of movement even if the movement no longer appears smooth to the eye. When drawing text, the notion of serifs, which aid the eye in tracking across lines of text while reading, was discussed. Finally, the issues of color perception were addressed, in particular the fact that humans differentiate light and dark far better than variations in color, and the fact that resolving detail in the blue and red areas of the spectrum is much harder than at the center of the spectrum in the green and yellow bands.