CHAPTER 1

Introduction to QuickDraw

Contents

Introduction to QuickDraw

This chapter introduces you to the terms, concepts, and capabilities of **QuickDraw,** a collection of system software routines that your application can use to perform most image-manipulation operations on Macintosh computers. This chapter also introduces you to the Printing Manager, which your application can use to print the images you create with QuickDraw.

Imaging entails the construction and display of graphical information. Such graphical information can consist of shapes, pictures, and text, and can be displayed on such output devices as screens and printers. You should read this chapter if you are new to Macintosh programming. The topics introduced in this chapter are explained in detail in the rest of the chapters of this book. However, for information on QuickDraw's text-handling facilities, you should instead see *Inside Macintosh: Text*.

Macintosh system software not only provides enormous imaging flexibility, it also handles much of the programming overhead that such flexibility requires. For example, Color QuickDraw automatically handles multiple screens of differing sizes and capabilities, so that most applications don't need to determine where the user has placed a window or what equipment the user has set up.

This rest of this chapter introduces

- **basic QuickDraw,** the imaging engine for all Macintosh computers
- **Color QuickDraw,** the color imaging system with which your application can display hundreds, thousands, even millions of colors on grayscale and color screens
- the **Printing Manager,** the collection of system software routines that allows your application to communicate with printer drivers to print on any variety of printer
- other imaging managers associated with QuickDraw

QuickDraw is the part of the Macintosh Toolbox that performs graphics operations on the user's screen. All Macintosh applications use QuickDraw indirectly whenever they call other Toolbox managers to create and manage the basic user interface elements (such as windows, controls, and menus, as described in *Inside Macintosh: Macintosh Toolbox Essentials*).

As the Macintosh has evolved toward greater graphics capabilities, QuickDraw has grown along with it. Each new generation of QuickDraw has maintained compatibility with those that preceded it, while adding new capabilities and expanding the range of possible display devices. This evolutionary approach has helped to ensure that existing applications, written for earlier Macintosh models, continue to work as more powerful computers are developed.

The development of QuickDraw has progressed along these three main evolutionary stages:

- Basic QuickDraw, which was designed for the earliest Macintosh models with their built-in black-and-white screens. System 7 added new capabilities to basic QuickDraw, including support for offscreen graphics worlds and the extended version 2 picture format. Basic QuickDraw is still used in more recent black-and-white Macintosh systems such as the Macintosh Classic® and PowerBook 100 computers.
- The original version of Color QuickDraw, which was introduced with the first Macintosh II systems. This first generation of Color QuickDraw could support up to 256 colors.
- The current version of Color QuickDraw, which was originally introduced as 32-Bit Color QuickDraw and is now part of System 7. This version has been expanded to support up to millions of colors.

Applications that use only basic QuickDraw routines are compatible with all Macintosh systems. However, applications that use routines specific to Color QuickDraw cannot run on computers supporting only basic QuickDraw.

Drawing Environments

The Macintosh computer was the first to popularize the bitmapped screen, as opposed to the character-oriented screen—common to terminals and early personal computers—on which only a single, built-in character set could be displayed. On a bitmapped screen every pixel can be manipulated. **Pixels** are the dots that make up a visible image on the screen. Drawing on the Macintosh screen consists of manipulating memory bits that QuickDraw translates into an analogous manipulation of pixels. This allows your application to create shapes and characters in differing sizes and differing styles. Such flexibility gives your application and its users many of the capabilities of a design studio and a print shop.

Your application performs all graphics operations in graphics ports. A **graphics port** is a drawing environment—defined by a `GrafPort` record for a basic graphics port or a `CGrafPort` record for a color graphics port—that contains the information QuickDraw needs to transmit drawing operations from bits in memory to onscreen pixels.

A **basic graphics port** is the drawing environment provided by basic QuickDraw; a basic graphics port contains the information that basic QuickDraw uses to create and manipulate onscreen either black-and-white images or color images that employ a basic eight-color system.

A **color graphics port** is the sophisticated color drawing environment provided by Color QuickDraw; a color graphics port contains the information that Color QuickDraw uses to create and manipulate grayscale and color images onscreen.

While your application can draw directly into basic and color graphics ports, you can improve your application's appearance and performance by constructing images in offscreen graphics worlds and then copying them to onscreen graphics ports. An **offscreen graphics world** is a sophisticated environment for preparing complex color, grayscale, or black-and-white images before displaying them on the screen. Defined in a private data structure referred to by a pointer of type `GWorldPtr`, an offscreen graphics world also contains a graphics port of its own.

Your application can print the images it prepares in graphics ports by drawing into a printing graphics port using QuickDraw drawing routines. A **printing graphics port** is the printing environment defined by a `TPrPort` record, which contains a graphics port plus additional information used by the printer driver and system software.

The visible image for a graphics port is contained in either a bitmap or a pixel map. A **bitmap** is defined by a data structure of type `BitMap`, and it represents the positions and states of a corresponding set of pixels, which can be either black and white or the eight predefined colors provided by basic QuickDraw. A bitmap is contained within a basic graphics port. A **pixel map** is defined by a data structure of type `PixMap`, and it represents the positions and states of a corresponding set of color pixels. A handle to a pixel map is contained within a color graphics port.

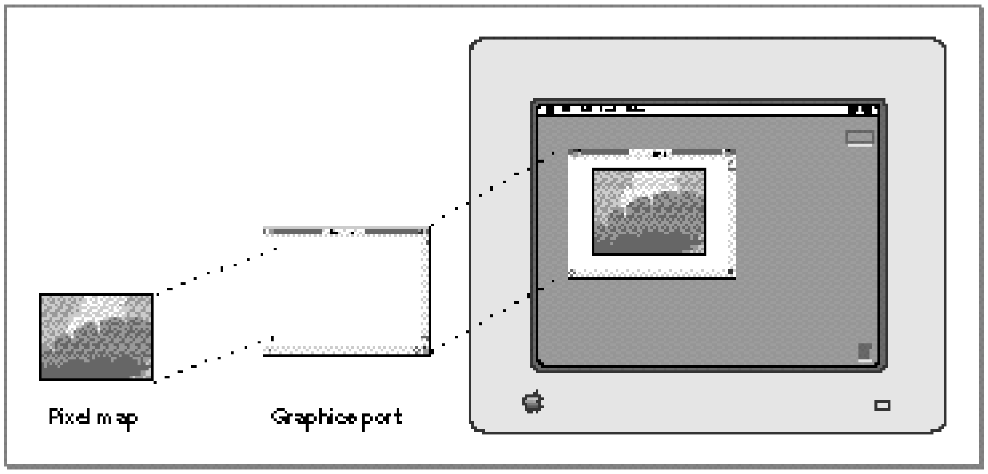
Because your application can typically deal with graphics ports instead of hardware devices, QuickDraw helps your application achieve device independence. A graphics port does the following:

- It specifies the bitmap or pixel map that points to the area of memory in which your drawing operations take place. The bitmap or pixel map contains information about how that memory should be arranged to map bits to the screen.
- It contains a metaphorical graphics pen with which to perform drawing operations. You can set this pen to different sizes, patterns, and colors.
- It holds information about text: if your application or its user writes characters, they will be styled and sized according to information in your application's graphics port.

The fields of a graphics port are maintained by QuickDraw, and you should never write directly into those fields. However, QuickDraw provides routines for changing the fields of a graphics port: you can point to an image in a different area of memory, reshape and resize the pen, change the pen's pattern and color, and switch fonts. You can, and often must, read the fields of a graphics port.

Figure 1-1 illustrates how an onscreen image represents bits in memory. In this figure, an application uses the Window Manager function `GetNewCWindow` to create a new window on a grayscale screen; `GetNewCWindow` automatically uses Color QuickDraw to create the graphics port for the empty window. The application uses Color QuickDraw procedures to fill the graphics port's pixel map with an image and to draw the image onscreen.

Figure 1-1 A grayscale image representing bits in memory

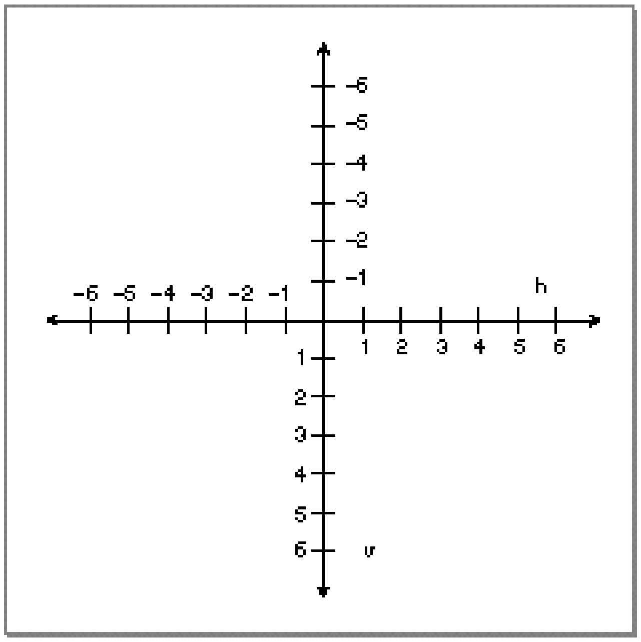


QuickDraw's Coordinate Plane

Your application typically uses Window Manager routines to create graphics ports in the form of windows. Your application can draw into a window without regard to its location on the screen—even if the window spans more than one screen. This is possible because QuickDraw maintains a global coordinate system for a computer's entire potential drawing space and a different local coordinate system for every window displayed in this space.

The Macintosh screen (or screens) on which QuickDraw displays your images represents a small part of a large global coordinate plane. Coordinates in the **global coordinate system** reflect the entire potential drawing space on this plane. The (0,0) origin point of the global coordinate plane is assigned to the upper-left corner of the main screen—that is, the one with the menu bar—while coordinate values increase to the right and (unlike a Cartesian plane) down. Any pixel on the screen can be specified by a vertical coordinate (ordinarily labeled v) and a horizontal coordinate (ordinarily labeled h). Figure 1-2 illustrates the QuickDraw global coordinate plane.

Figure 1-2 The QuickDraw global coordinate plane



Note
The orientation of the vertical axis, while convenient for computer graphics, differs from mathematical convention. Also, the coordinate plane is bounded by the limits of QuickDraw coordinates, which range from –32768 to 32767. ◆

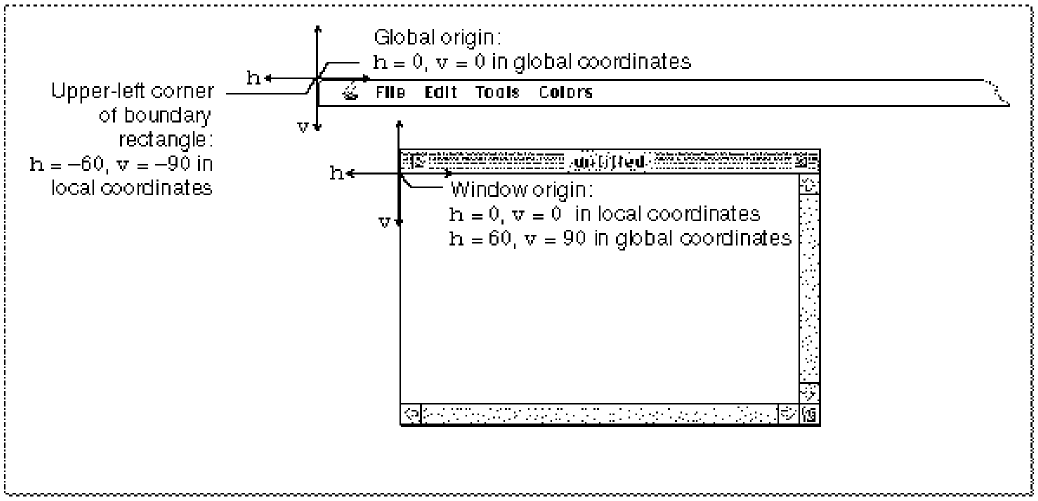
Windows are rectangular areas that are subsets of the global coordinate plane. Each window represents its own QuickDraw graphics port. When you create a window, the Window Manager uses QuickDraw to create a graphics port in which the window's contents are displayed. (See the chapter "Window Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for a complete description of creating and managing windows.)

When your application creates a new graphics port, QuickDraw defines a **boundary rectangle,** which, by default, is the entire main screen; this rectangle links the local coordinate system of a graphics port to QuickDraw's global coordinate system and defines the area of the pixel image or bit image into which QuickDraw can draw. A boundary rectangle is stored in the pixel map for a color graphics port or in the bitmap for a basic graphics port.

The graphics port includes a field called `portRect`, which defines a rectangle to be used for drawing. In a graphics port that represents a window, the `portRect` rectangle—or simply, the **port rectangle**—represents the window's content region.

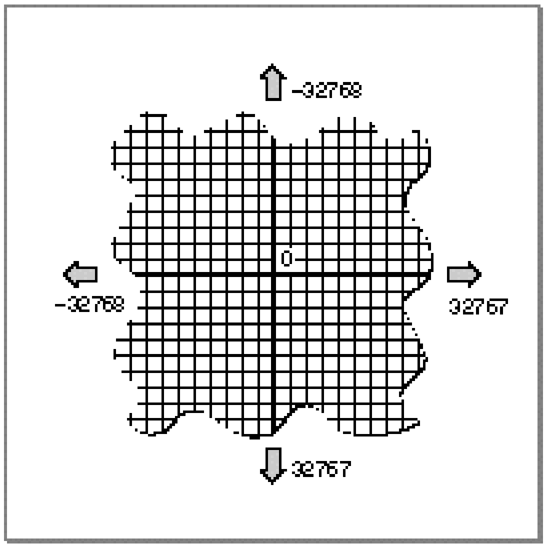
When you use the Window Manager to place a window on the screen, you specify the location of its port rectangle in global coordinates. However, within the port rectangle, the drawing area is described using a **local coordinate system.** You draw into a window in local coordinates, without regard to the window's location on the screen. Figure 1-3 illustrates the local and global coordinate systems for a sample window that is 180 pixels high by 300 pixels wide, placed with its window origin 90 pixels down and 60 pixels to the right of the upper-left corner of the main screen.

Figure 1-3 A window's local and global coordinate systems


Global origin:
h = 0, v = 0 in global coordinates
Upper-left corner of boundary rectangle:
h = –60, v = –90 in local coordinates
h
v
File Edit Tools Colors
untitled
h
v
Window origin:
h = 0, v = 0 in local coordinates
h = 60, v = 90 in global coordinates

It is helpful to conceptualize the global coordinate plane as a two-dimensional grid—one with integer coordinates ranging from –32768 to 32767—as illustrated in Figure 1-4.

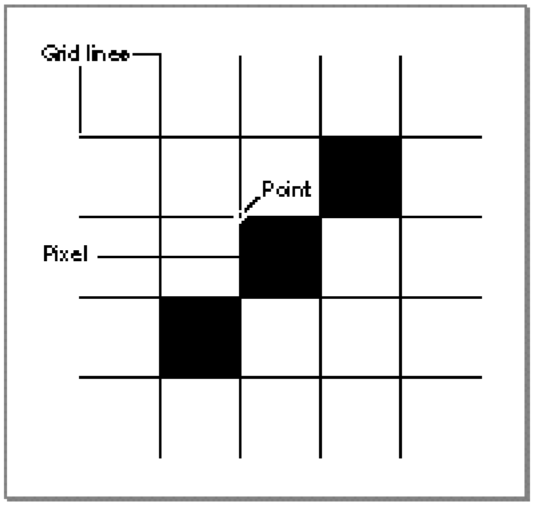
Figure 1-4 The coordinate plane



The intersection of a horizontal and a vertical grid line marks a **point** on the coordinate plane. Notice that, because all coordinates are limited to simple integers, the QuickDraw plane is finite, although very large: there are over four billion points on the grid, many more than there are dots on any screen. When using QuickDraw, you associate small parts of the grid with areas on the screen.

Note also the distinction between points on the coordinate grid and pixels, the dots that make up a visible image on the screen. Figure 1-5 illustrates the relationship between the two: the pixel is down and to the right of the point by which it is addressed.

Figure 1-5 Points and pixels



As the grid lines are infinitely thin, so a point is infinitely small. Pixels, by contrast, lie *between* the lines of the coordinate grid, not at their intersections. This gives them a definite physical extent, so that they can be seen on the screen.

After your application creates a window, the user can move or resize it within the global coordinate system. However, to draw images into the window, your application needs only to specify pixels within the local coordinate system for that window.

Your application uses a cursor to allow the user to select all or part of the content of a window. A **cursor** is a 16-by-16 pixel image that appears on the screen. Called the *pointer* in Macintosh user manuals, the user controls the cursor with the mouse. Basic QuickDraw supplies a predefined cursor for your application: the arrow cursor, which is familiar to all Macintosh users. Your application can also use other cursors. For example, when the user moves the cursor to any text in your application, your application should change the arrow cursor to an I-beam cursor and, when performing a lengthy process that precludes the user from interacting with your application, your application should change the cursor to a wristwatch cursor or an animated cursor.

One point in the cursor's image is designated as the **hot spot,** which in turn points to a location on the screen. The hot spot is the portion of the cursor that must be positioned over a screen object before mouse clicks can have an effect on that object. For example, when the user presses the mouse button, the Event Manager function `WaitNextEvent` reports the location of the cursor's hot spot in global coordinates. Your application can use the basic QuickDraw procedure `GlobalToLocal` to convert the global coordinates of that point to local coordinates for the window.

Images

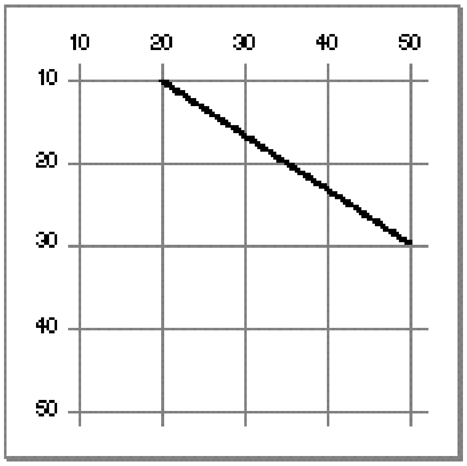
QuickDraw provides a plethora of routines for drawing different kinds of images. These routines typically require that you start at a particular location in a graphics port and then move the graphics pen. The **graphics pen** is a metaphorical device for performing drawing operations onscreen. Your application can set this pen to different sizes, patterns, and colors.

You specify where to begin drawing by placing the pen at some location in the window's local coordinate system, and then specifying an act of drawing, usually from there to another location. Take, for example, the following two lines of code:

```
MoveTo(20,10);
LineTo(50,30);
```

The `MoveTo` procedure places the graphics pen at a point with a horizontal coordinate of 20 and a vertical coordinate of 10 in the local coordinate system of the graphics port, and the `LineTo` procedure draws a line from there to a point with a horizontal coordinate of 50 and a vertical coordinate of 30, as shown in Figure 1-6.

Figure 1-6 Drawing a line



Whenever you draw into a graphics port, the characteristics of its graphics pen determine how the drawing looks. These characteristics are

- pen location, specified in a graphics port's local coordinates
- pen size, specified by a width and height in pixels
- pen pattern, which defines, in effect, the ink that the pen draws with, ranging from solid black to intricate, multicolor patterns
- pattern mode, also called *transfer mode,* which specifies how the pen pattern interacts with white or any existing drawing that the pattern overlays
- pen visibility, specified by an integer indicating whether drawing operations will actually appear—for example, for 0 or negative values, the pen draws with "invisible" ink

The pen's initial settings are a size of 1 pixel by 1 pixel, a solid black pattern, a pattern mode in which the black writes over everything, and visible ink.

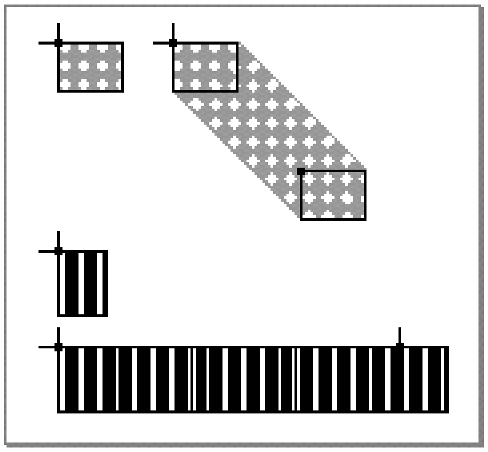
A **pattern** is an image that can be repeated indefinitely to form a repeating design, such as stripes, when drawing lines and shapes or when filling areas on the screen. There are two type of patterns: bit patterns and pixel patterns. A **bit pattern** is an 8-by-8 pixel image drawn by default in black and white, although any two colors can be used on a color screen. A **pixel pattern** can use additional colors and can be of any width and height that's a power of 2.

By starting at a particular position and moving the graphics pen, you can use QuickDraw procedures to define and directly draw a number of graphic shapes using the size and pattern of the graphics pen. Several procedures and the shapes they produce are listed here.

Procedure	**Resulting shape**
`LineTo`	Line
`DrawChar`	Character
`FrameRect`	Rectangle
`FrameOval`	Oval
`FrameRoundRect`	Rounded rectangle
`FrameArc`	Arc

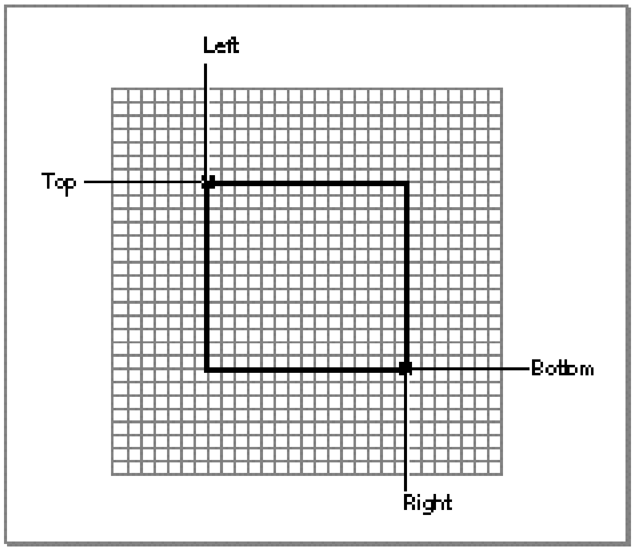
A **line** is defined by two points: the current location of the graphics pen and its destination. The pen hangs below and to the right of the defining points, as shown in Figure 1-7, where two lines are drawn with different bit patterns and pen sizes.

Figure 1-7 Lines drawn with different bit patterns and pen sizes



A **rectangle** can be defined by two points—its upper-left and its lower-right corners, as shown in Figure 1-8, or by four boundaries—its upper, left, lower, and right sides. Rectangles are used to define active areas on the screen, to assign coordinate systems to graphical entities, and to specify the locations and sizes for various graphics operations.

Figure 1-8 A rectangle



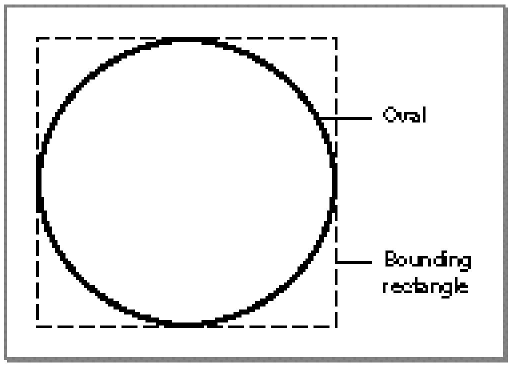
QuickDraw also provides routines that allow you to perform a variety of mathematical calculations on rectangles—changing their sizes, shifting them around, and so on.

Note
Purely speaking, rectangles and points are mathematical entities that have no direct representation on the screen. The borders of the rectangle are infinitely thin, lying along the lines of the coordinate grid. To give a rectangle a shape that can be drawn on the screen, you must use one of QuickDraw's drawing routines, such as `FrameRect` and `PaintRect`. ◆

You use rectangles known as **bounding rectangles** to define the outmost limits of other shapes, such as ovals and rounded rectangles. The lines of bounding rectangles completely enclose the shapes they bound; in other words, no pixels from these shapes lie outside the infinitely thin lines of the bounding rectangles.

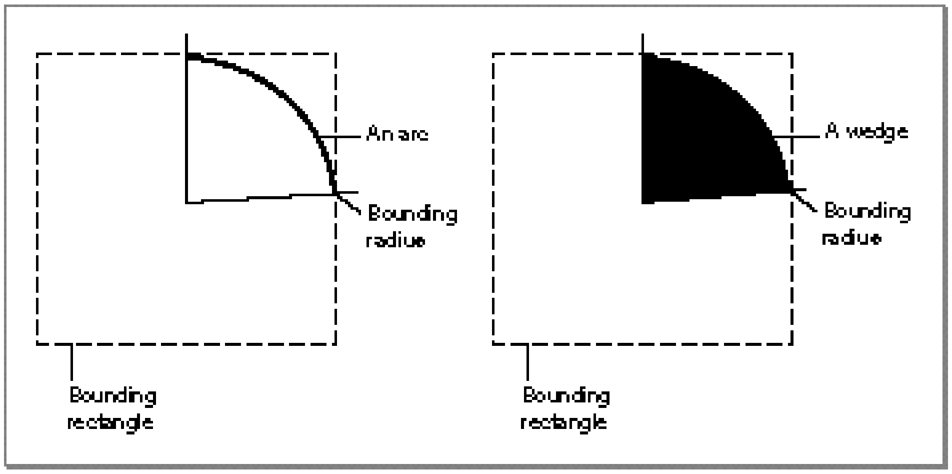
An **oval** is a circular or elliptical shape defined by the bounding rectangle that encloses it. If the bounding rectangle is square (that is, has equal width and height), then the oval is a circle, as shown in Figure 1-9.

Figure 1-9 An oval



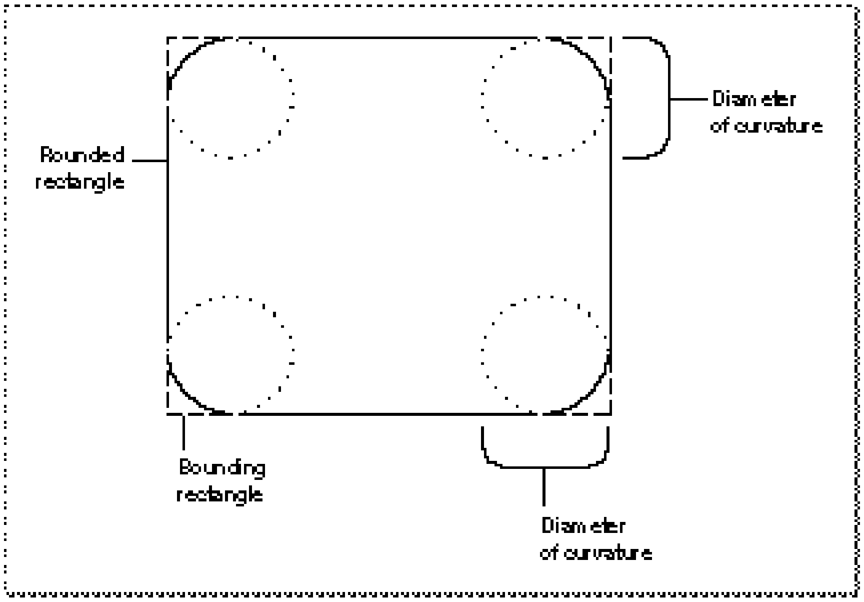
An **arc** is a portion of the circumference of an oval bounded by a pair of radii joining at the oval's center; an arc does not include the bounding radii or any part of the oval's interior. A **wedge** is a pie-shaped segment of an oval, bounded by a pair of radii joining at the oval's center; a wedge does include part of the oval's interior. Arcs and wedges are defined by the bounding rectangle that encloses the oval, along with a pair of angles marking the positions of the bounding radii, as shown in Figure 1-10.

Figure 1-10 An arc and a wedge



A **rounded rectangle** is a rectangle with rounded corners. The figure is defined by the rectangle itself, along with the width and height of the ovals forming the corners (called the *diameters of curvature*), as shown in Figure 1-11. The corner width and corner height are limited to the width and height of the rectangle itself; if they are larger, the rounded rectangle becomes an oval.

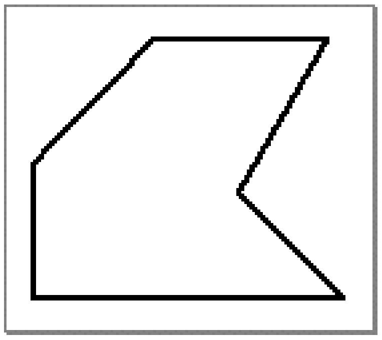
Figure 1-11 A rounded rectangle



Three types of graphic objects—polygons, regions, and pictures—require you to call several routines to create and draw them. You begin by calling a routine that collects drawing commands into a definition for the object. You use a series of drawing routines to define the object. Then you use a routine that signals the end of the object definition. Finally, you use a routine that draws your newly defined object.

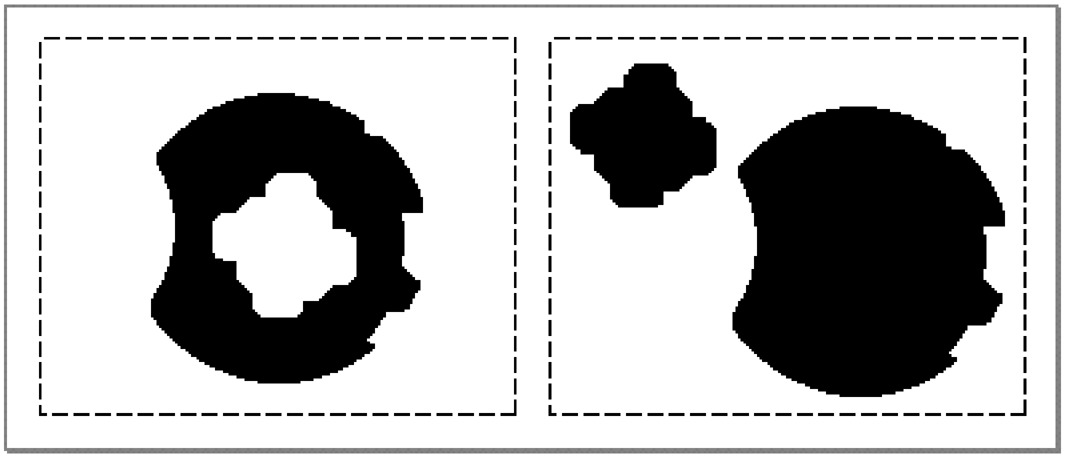
A **polygon** is defined by any sequence of points representing the polygon's vertices, connected by straight lines from one point to the next. You define a polygon by drawing the lines with QuickDraw line-drawing operations: you move to the first vertex point in the sequence, draw a line from there to the second vertex, from there to the third, and so on. When you finish, QuickDraw automatically completes the figure with a closing line from the last vertex back to the first. Figure 1-12 shows an example of a polygon.

Figure 1-12 A polygon



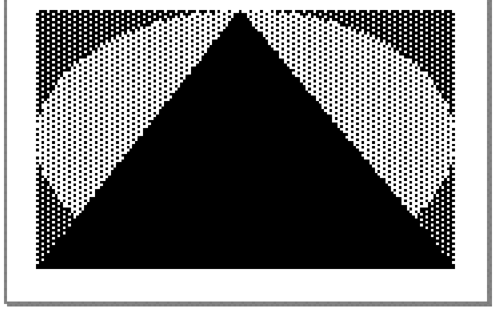
A **region** is an arbitrary area or set of areas, the outline of which is one or more closed loops. One of QuickDraw's most powerful capabilities is the ability to work with regions of arbitrary size, shape, and complexity. You define a region by drawing its boundary with QuickDraw operations. The boundary can be any set of lines and shapes (even including other regions) forming one or more closed loops. A region can be concave or convex, can consist of one connected area or many separate ones, and can even have holes in the middle. In Figure 1-13 the region on the left has a hole and the one on the right consists of two unconnected areas.

Figure 1-13 Two regions



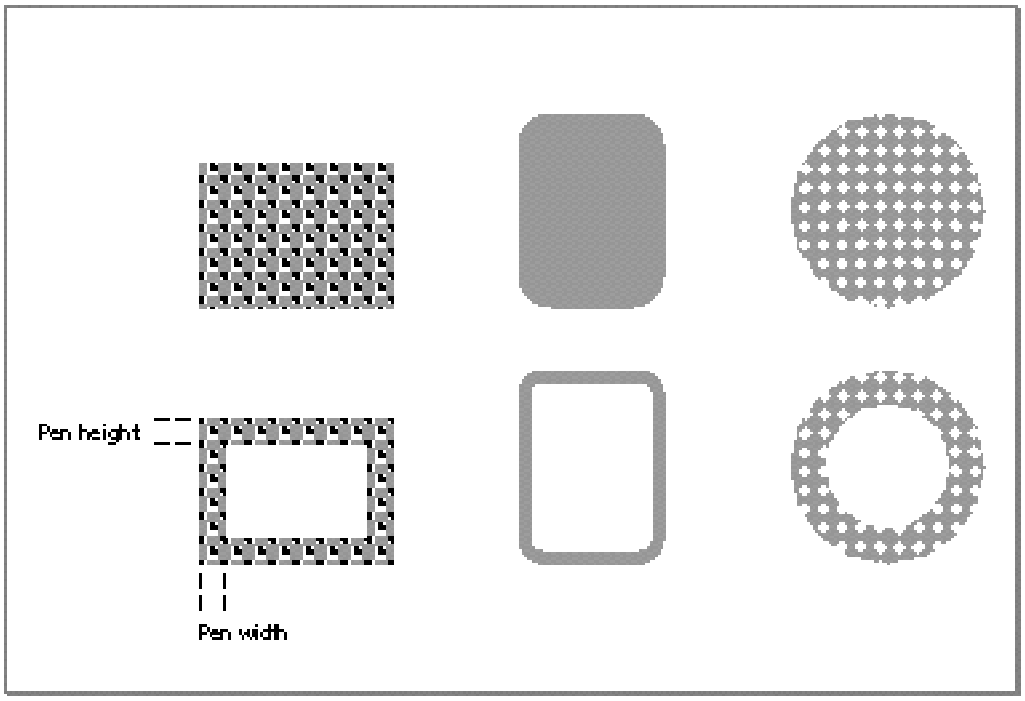
Your application can record a sequence of QuickDraw drawing operations in a **picture** and play its image back later. Pictures provide a form of graphic data exchange: one program can draw something that was defined in another program, with great flexibility and without having to know any details about what's being drawn. Figure 1-14 shows an example of a picture containing a rectangle, an oval, and a rectangle.

Figure 1-14 A simple QuickDraw picture



As you see in Figure 1-14, QuickDraw shapes may be drawn using various pen patterns. Painting a shape fills both its outline and its interior with the current pen pattern. Filling a shape fills both its outline and its interior with any specified pattern (not necessarily the current pen pattern). The three figures in the top row of Figure 1-15 are filled.

Figure 1-15 Filling and framing various shapes



Framing a shape draws just its outline, using the current pen size, pen pattern, and pattern mode. The interior of the shape is unaffected, allowing previously existing pixels to “show through.” The three figures in the bottom row of Figure 1-15 are framed. Erasing a shape fills both its outline and its interior with the current background pattern (typically solid white on a black-and-white monitor or a solid background color on a color monitor). Inverting a shape reverses the colors of all pixels within its boundary—for example, all white pixels become black, and all black pixels become white. With color pixels, the results of inverting are less predictable.

Colors

The earliest Macintosh models all used basic QuickDraw to draw to built-in screens with known characteristics. The Macintosh II computer introduced Color QuickDraw, which supports a variety of screens of differing sizes and color capabilities. With Color QuickDraw, users can choose from a wide range of screen options, from simple 12-inch black-and-white screens to full-page grayscale monitors to large two-page displays capable of presenting millions of colors. Users can even connect two or more separate screens to the same computer and simultaneously view different portions of the system's global coordinate plane.

A pixel, which is short for *picture element,* is the smallest dot that QuickDraw can draw. On a black-and-white monitor, a pixel is a single-color phosphor dot that displays in two states—black and white. On a color screen, three phosphor dots (red, green, and blue) compose each color pixel.

A pair of fields in a graphics port, `fgColor` and `bkColor`, specify a foreground and background color. The **foreground color** is the color used for bit patterns and for the graphics pen when drawing. By default, the foreground color is black. The **background color** is the color of the pixels in the bitmap or pixel map wherever no drawing has taken place. By default, the background color is white. However, when there is a color screen your application can draw with a color other than black by changing the foreground color, and your application can draw into a background other than white by changing the background color. For example, by changing the foreground color to red and the background color to blue before drawing a rectangle, your application can draw a red rectangle against a blue background.

On a color screen, you can draw in color even when you are using a basic graphics port. Although basic QuickDraw graphics routines were designed for black-and-white drawing, they also support an eight-color system that basic QuickDraw predefines for display on color screens and color printers. Because Color QuickDraw also supports this eight-color system, it is compatible across all Macintosh platforms.

The basic QuickDraw color values consist of 1 bit for normal black-and-white drawing (black on white), 1 bit for inverted black-and-white drawing (white on black), 3 bits for the additive primary colors (red, green, blue) used in video display, and 4 bits for the subtractive primary colors (cyan, magenta, yellow, black) used in printing. Basic QuickDraw defines a set of constants for those standard colors:

```
CONST
  blackColor   = 33;
  whiteColor   = 30;
  redColor     = 205;
  greenColor   = 341;
  blueColor    = 409;
  cyanColor    = 273;
  magentaColor = 137;
  yellowColor  = 69;
```

These are the only colors available in basic QuickDraw (or with Color QuickDraw drawing into a basic graphics port).

In Color QuickDraw, however, a color pixel represents up to 48 bits in memory. On a grayscale screen, a white phosphor dot whose intensity can vary is a pixel that usually represents 1, 2, 4, or 8 bits in memory.

To remove (for most applications) the burden of worrying about screen capabilities, Color QuickDraw is device-independent. Your application can use an `RGBColor` **record,** a data structure of type `RGBColor`, to specify a color by its red, green, and blue components, with each component defined as a 16-bit integer. Color QuickDraw compares the resulting 48-bit value with the colors actually available on a video device—such as a plug-in video card or a built-in video interface—at execution time and then chooses the closest match.

What the user finally sees depends on the characteristics of the actual video device and screen. Screens may display color or black and white; the video devices that control them may have indexed colors that support pixels of 1-bit, 2-bit, 4-bit, or 8-bit depths, or direct colors that support pixels of 16-bit or 32-bit depths. Color QuickDraw automatically determines which method is used by the video device and matches your requested color with the closest available color.

Indexed Colors

Some video devices use **indexed colors** to support a maximum of 256 colors at any one time. The indexed color system was created when the Macintosh II computer was introduced, at a time when memory was scarce and moving megabyte images around was impractical. With indexed color, the maximum value of a pixel in a `PixMap` record is limited to a single byte. Each pixel's byte can specify one of 256 (2^8) different values. Video devices implementing indexed color contain a data structure called a **color lookup table** (or, more commonly, a CLUT). The CLUT, in turn, contains entries for all possible color values.

For example, an indexed video device supporting 2 bits per pixel provides indexes into a table of four colors; if two colors are black and white, however, only two other hues can be shown.

An indexed video device supporting 4 bits per pixel can provide indexes into a table of 16 colors, a number sufficient for many straightforward graphics, such as those used in charts, presentations, or games.

An indexed video device supporting a byte (8 bits) for each pixel allows 256 colors to be displayed, which for many images is enough to produce near-photographic quality. The problem is that the colors needed for one near-photographic image may not be appropriate for another. The prevailing shades of browns necessary for displaying a painting by Rembrandt aren't appropriate for the prevailing shades of blues in a Monet painting. Because most indexed video devices use a variable CLUT (rather than a fixed one), you can display a Rembrandt painting with one set of 256 colors, then use system software to reload the CLUT with a different set of 256 colors for a Monet painting. If your application needs this sort of control on indexed video devices, you can use the Palette Manager (as described in the chapter "Palette Manager" in *Inside Macintosh: Advanced Color Imaging*) to arrange palettes—that is, sets of colors—for particular images and for video devices of differing color capabilities.

Note

Some Macintosh computers, such as grayscale PowerBook computers, have a fixed CLUT, which your application cannot change. ◆

If your application uses a 48-bit `RGBColor` record to specify a color, the Color Manager examines the colors available in the CLUT on the video device. If the video device supports 8 bits per pixel, for example, it contains a CLUT with 256 entries. Comparing these entries to the `RGBColor` record you specify, the Color Manager determines which color in the CLUT is closest, and the Color Manager gives Color QuickDraw the index to this color. Color QuickDraw then draws with this color.

Direct Colors

Video devices that implement **direct color** eliminate the competition for limited table spaces and remove the need for color table matching. By using direct color, video devices may support a maximum of thousands or millions of colors. When you specify a 48-bit `RGBColor` record, Color QuickDraw truncates the least significant bits of its red, green, and blue components to either 16 bits (5 bits each for red, green, and blue, with 1 bit unused) or 32 bits (8 bits each for red, green, and blue, with 8 bits unused). Using 16 bits, direct video devices can display over 32,000 colors; using 32 bits, direct video devices can display as many as 16 million different colors.

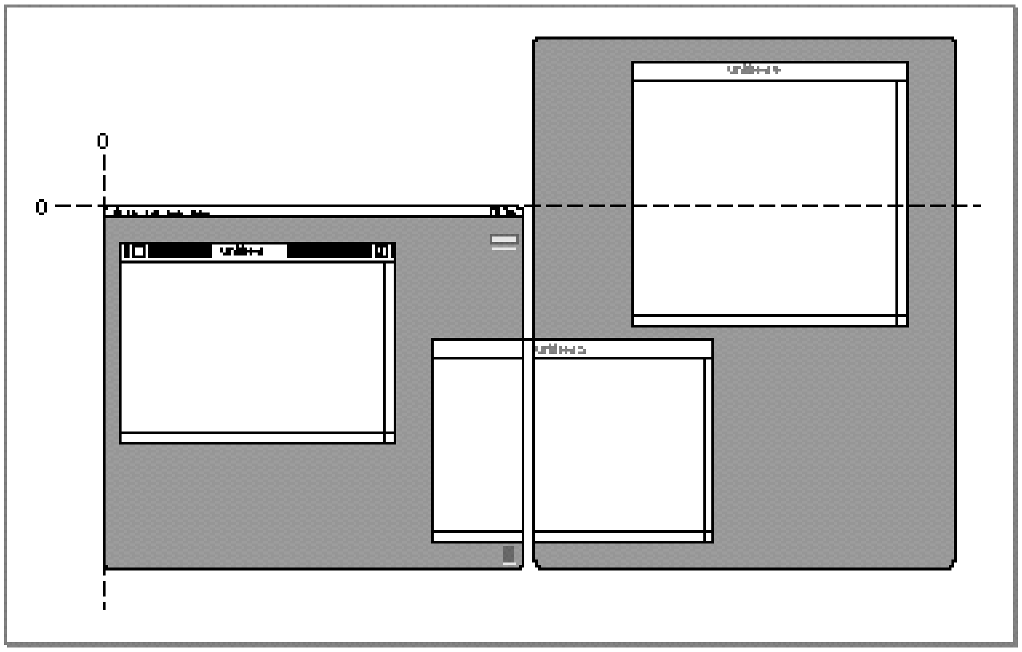
Using direct color not only removes much of the complexity of the CLUT mechanism for video device developers, but it also allows the display of thousands or millions of colors simultaneously, resulting in near-photographic realism.

Multiple Screens

A **video device** is a piece of hardware, such as a plug-in video card or a built-in video interface, that controls a screen. To use more than one screen, a user may have more than one video device installed on his or her computer.

In a drawing environment with multiple screens, the one with the menu bar is the **main screen.** Color QuickDraw maps the (0,0) origin point of the global coordinate plane to the main screen's upper-left corner, and other screens are positioned adjacent to it. In Figure 1-16, a full-page screen sits next to the main screen. Remember that each window—even a window that overlaps two screens—has its own local coordinate system with a (0,0) point at its upper-left corner.

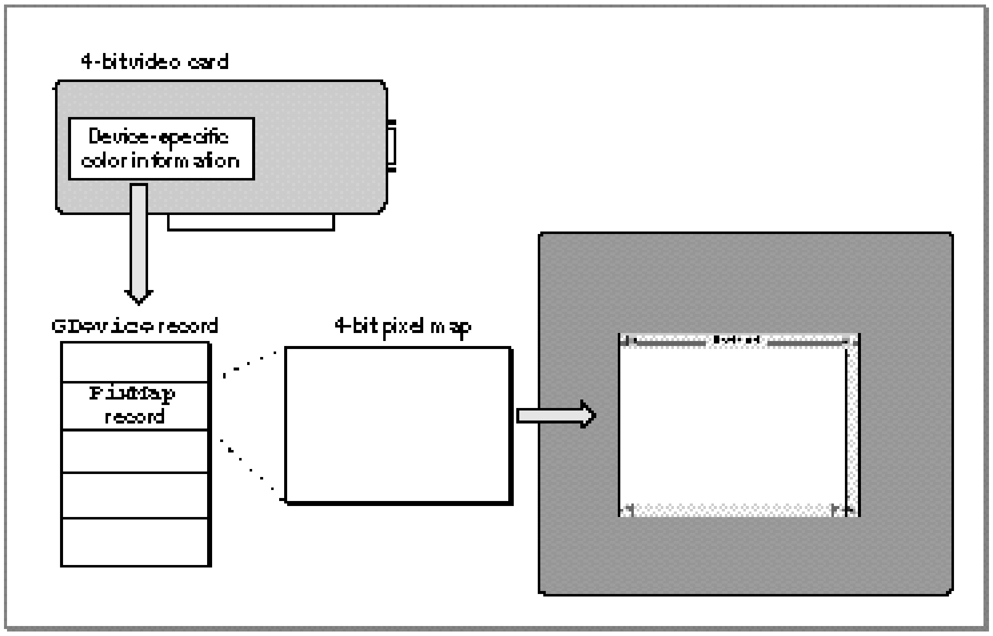
Figure 1-16 A two-screen system



Color QuickDraw stores state information for a video device in a **`GDevice` record.** (Color QuickDraw creates `GDevice` records—basic QuickDraw does not, nor does basic QuickDraw support multiple screens.) When a computer supporting Color QuickDraw starts up, it allocates and initializes a handle to a `GDevice` record for each video device it finds. The firmware in the ROM for each video device supplies information about whether the device uses indexed or direct colors, how much video RAM is available, and so on. Some of this information is stored in the `GDevice` record, where it is available to the entire graphics system.

As illustrated in Figure 1-17, when your application opens a color window, the `CGrafPort` record for the window contains a handle to a `PixMap` record contained in the main screen's `GDevice` record. The `PixMap` record for your window thereby contains the correct pixel specifications for the main screen. Color QuickDraw internally calculates the changes required for drawing to any other screens.

Figure 1-17 The `GDevice` record and pixel map for a 4-bit video card



When a multiscreen system starts up, one of the screens is the **startup screen,** the screen on which the "happy Macintosh" appears. By default, the main screen is the startup screen. However, by using the Monitors control panel, the user can specify a different startup screen.

During the startup of a multiscreen environment, system software calls the Window Manager procedure `InitWindows` to create a region that is the union of all the active screens (minus the menu bar and the rounded corners on the outermost screens). The Window Manager saves this region, called the **gray region,** as the global variable `GrayRgn`. The gray region describes and defines the desktop: the area in which the user can drag windows.

Users can drag windows from one screen to another and even across multiple screens. Color QuickDraw calculates the global coordinates of the rectangle into which it must draw and issues the drawing command to each video device that the rectangle intersects.

For many applications, Color QuickDraw provides a device-independent interface; your application can draw images in a color graphics port for a window, and Color QuickDraw automatically manages the screen display—even if the user has multiple screens. Your application generally never needs to create `GDevice` records. However, you may find it useful for your application to examine `GDevice` records to determine the capabilities of the user's screens. When zooming a window, for example, your application can use `GDevice` records to determine which screen contains the largest area of a window, and then determine the ideal window size for that screen.

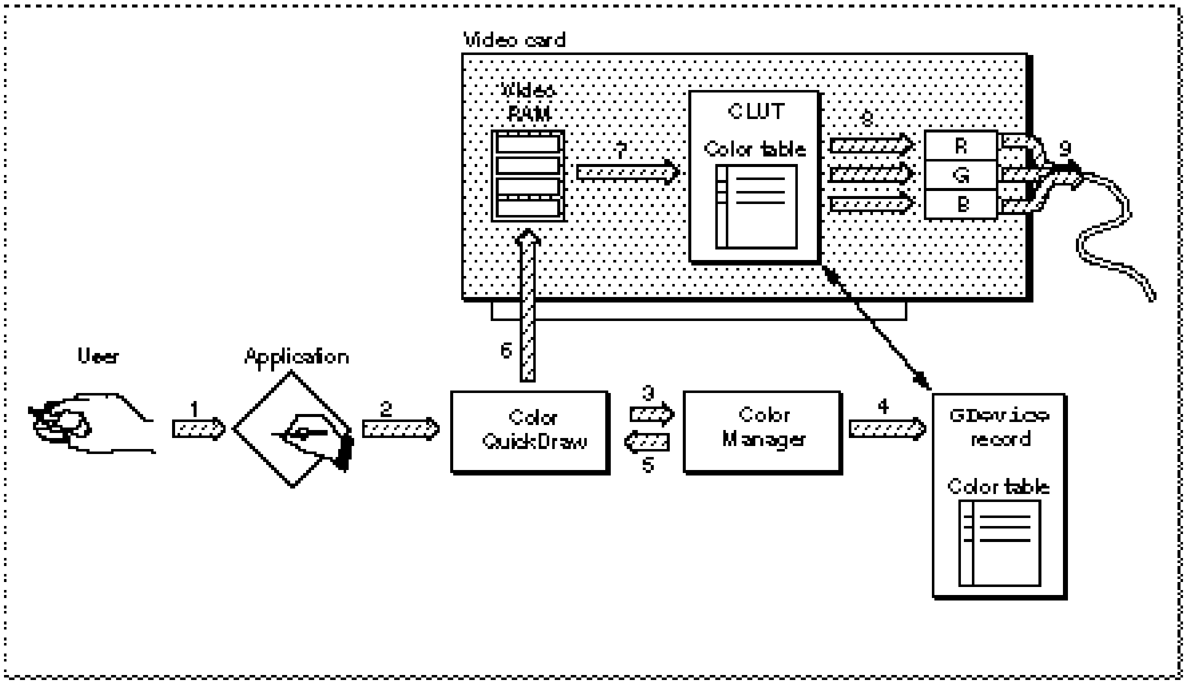
You may also wish to use the `DeviceLoop` procedure to optimize your application's drawing for screens with different capabilities. The `DeviceLoop` procedure searches for video devices that intersect your graphics port's drawing region, and it informs your application of each video device it finds. The `DeviceLoop` procedure provides your application with information about the pixel depth and other attributes of the video device on which drawing is currently taking place. Your application can then choose what drawing technique to use for the current device. When highlighting, for example, your application might invert black and white when drawing onto a 1-bit video device but use magenta as the highlight color on a color screen.

From Memory Bits to Onscreen Pixels

Tracing the path from data in memory to a pixel on one of several connected screens traverses the major elements of QuickDraw and recapitulates much of the discussion in this chapter.

In Figure 1-18, the user of an indexed color system selects a color for some object from an application (1). Using a 48-bit `RGBColor` record to specify the color, the application calls a Color QuickDraw routine to draw the object in that color (2). Color QuickDraw uses the Color Manager to determine what color in the video device's CLUT comes closest to the requested color (3).

Figure 1-18 The indexed-pixel path



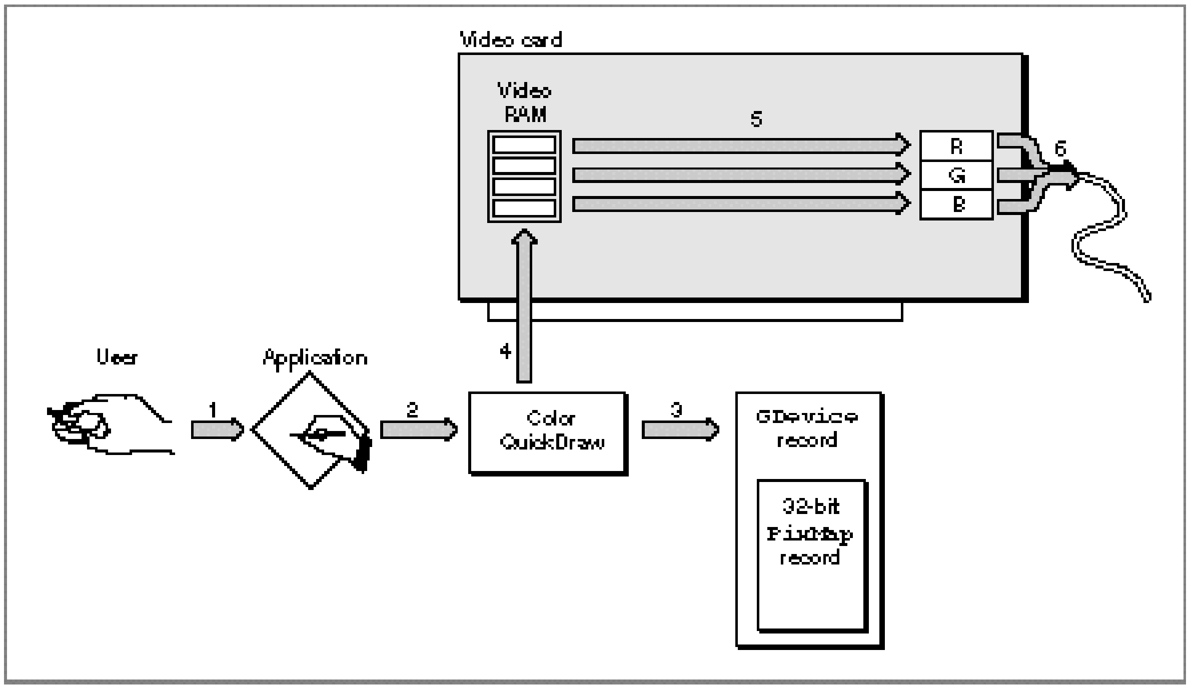
At startup, the video device's declaration ROM supplies information for the creation of the `GDevice` record that describes the characteristics of the device. The resulting `GDevice` record contains a `ColorTable` record that is kept synchronized with the card's CLUT. The Color Manager examines that `GDevice` record to find what colors are currently available (4) and to decide which color comes closest to the one requested by the application. The Color Manager gets the index value for the best match and returns that value to Color QuickDraw (5), which puts the index value into those places in video RAM that store the object (6).

The video device continuously displays video RAM by taking the index values, converting them to colors according to CLUT entries at those indexes (7), and sending them to digital-to-analog converters (8) that produce a signal for the screen (9).

For video devices that support direct color, the Color Manager's selection algorithm isn't needed. When an application specifies a color in an `RGBColor` record, Color QuickDraw uses the most significant 5 or 8 bits of each of the 16-bit red, green, and blue components of the specified color.

Figure 1-19 illustrates a user choosing a color for some object (1). Using a 48-bit `RGBColor` record to specify the color, the application uses a Color QuickDraw routine to draw the object in that color (2). Color QuickDraw knows from the `GDevice` record (3) that the screen is controlled by a direct device in which pixels are 32 bits deep, which means that 8 bits are used for each of the red, green, and blue components of the requested color. Color QuickDraw passes the high 8 bits from each 16-bit component of the 48-bit `RGBColor` record to the video device (4), which stores the resulting 24-bit value in video RAM for the object. The video device continuously displays video RAM by sending the three 8-bit red, green, and blue values for the color to digital-to-analog converters (5) that produce a signal for the screen (6).

Figure 1-19 The direct-pixel path

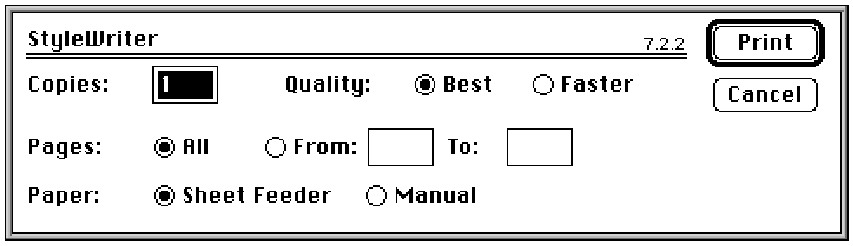


From Memory Bits to Printers

Available on all Macintosh computers, the Printing Manager is a collection of system software routines that your application can use to print to any type of connected printer by using the same QuickDraw routines for printing that your application uses for screen display. You can use the Printing Manager to print documents, to display and alter printing-related dialog boxes, and to handle printing errors. The Printing Manager takes much of the work out of coming up with a single way to handle all possible printer environments.

Your application uses the Printing Manager procedure `PrOpen` to open the current printer. (The **current printer** is the printer that the user last selected from the Chooser.) Before printing, your application should display the **job dialog box,** which solicits printing information from the user, such as the number of copies to print, the print quality, and the range of pages to print. Your application can use the `PrJobDialog` procedure to display a job dialog box. The `PrJobDialog` procedure handles all user interaction in the standard dialog items until the user clicks the Print or Cancel button. Figure 1-20 shows an example of a job dialog box. Your application prints the document in the active window if the user clicks the Print button.

Figure 1-20 The job dialog box for a StyleWriter printer



StyleWriter 7.2.2

Copies: 1

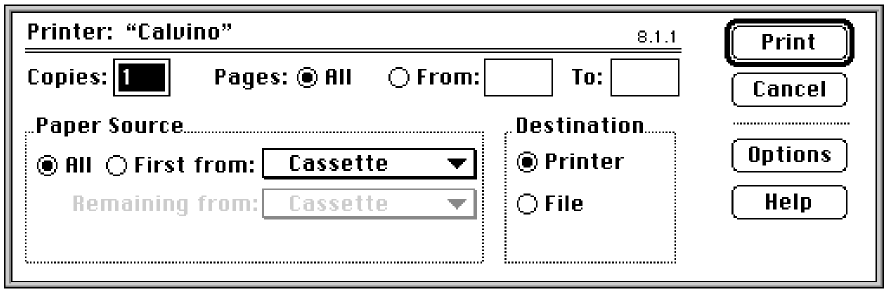
Quality: [✓ Best] Faster

Pages: [✓ All] From: To:

Paper: [✓ Sheet Feeder] Manual

Each printer has its own job dialog box. Thus, a style dialog box for one printer may differ slightly from that of another printer. Figure 1-21 shows a sample job dialog box for a LaserWriter printer.

Figure 1-21 The job dialog box for a LaserWriter printer



A TPrint record stores information about the choices made by the user in the print job dialog box. Your application can also customize the job dialog box to ask for additional information.

When the user clicks the Print button in the job dialog box, your application then uses the Printing Manager function PrOpenDoc to open a printing graphics port, which consists of a graphics port (either a GrafPort or CGrafPort record) plus additional information.

To set up the printing graphics port to print a page, your application should call the PrOpenPage procedure. Your application then prints by using the QuickDraw routines described in this book to draw into the printing graphics port. The Printing Manager uses a printer driver to do the actual printing. A **printer driver** does any necessary translating of QuickDraw drawing routines and sends the translated instructions and data to the printer. Each type of printer has its own printer driver, which is stored in a resource file in the Extensions folder inside the System Folder. Because your application does not communicate with any of the multitude of available printer drivers but instead uses the Printing Manager to handle this communication, the Printing Manager gives your application device-independent control over the printing process.

When your application has finished drawing into the page set up for the printing graphics port, your application closes the page by using the PrClosePage procedure. For every page that the user selects to be printed in a document, your application uses PrOpenPage and PrClosePage. When your application has finished printing, your application closes the printing graphics port by using the PrCloseDoc procedure; your application should then close the Printing Manager by using the PrClose procedure.

There are two main types of printer drivers for Macintosh computers: QuickDraw printer drivers and PostScript™ printer drivers. Using QuickDraw drawing operations, **QuickDraw printer drivers** render images on the Macintosh computer and then send the rendered images to the printer in the form of bitmaps or pixel maps. **PostScript printer drivers,** on the other hand, convert QuickDraw drawing operations into equivalent PostScript drawing operations, as necessary. PostScript printers have their own rendering capabilities. PostScript printer drivers typically send drawing operations to the printer, which itself renders images on the page.

For most applications, sending QuickDraw's picture-drawing routines to the printer driver is sufficient: the driver either uses QuickDraw or converts the drawing routines to PostScript. For some applications, such as page-layout programs, this may not be sufficient; such applications may rely on printer drivers to provide several features that are not available, or are difficult to achieve, using QuickDraw.

Using picture comments, your application can instruct printer drivers to perform operations that QuickDraw does not support. Created with the QuickDraw procedure `PicComment`, **picture comments** are data or commands for special processing that can be included in the code an application sends to a printer driver.

A number of picture comments have been given special definitions in printer drivers. The drivers for PostScript printers and even some QuickDraw printers support features unavailable with QuickDraw. When a printer driver encounters one of these comments, it converts it to its own printing code. Other picture comments signal the driver that PostScript code is enclosed, so your application can even include PostScript code directly in the definition of a picture.

Other Graphics Managers

In addition to the QuickDraw routines described in this book, several other collections of system software routines are available to assist you in drawing and printing images.

Your application can use QuickDraw's text-handling routines to measure and draw text ranging in complexity from a single glyph to a line of justified text containing multiple languages and styles. In addition to measuring and drawing text, QuickDraw's text-handling routines also help you to determine which characters to highlight and where to mark the insertion point. These routines are described in the chapter "QuickDraw Text" in *Inside Macintosh: Text*.

To provide more sophisticated color support on indexed graphics devices, your application can use the Palette Manager. The **Palette Manager** allows your application to specify sets of colors that it needs on a window-by-window basis. An indexed device supporting a byte for each pixel allows 256 colors to be displayed. On a video device that uses a variable CLUT, your application can use the Palette Manager to display tens of thousands of palettes—that is, sets of colors—consisting of 256 colors each, so that your application has up to 16 million colors at its disposal (although only 256 different colors can appear at once). For example, your application can use the Palette Manager to load the CLUT with a set of prevailingly brown colors to display a Rembrandt painting, then reload the CLUT with a set of prevailingly blue colors for a Monet painting. For information about the Palette Manager, see *Inside Macintosh: Advanced Color Imaging.*

To solicit color choices from users, your application can use the Color Picker Utilities. The **Color Picker Utilities** provide your application with a standard dialog box for soliciting a color choice from users. The Color Picker Utilities also provide routines that allow your application to convert between colors specified in `RGBColor` records and colors specified for other color models, such as the CMYK model used by many color printers. Most applications use the Color Picker Utilities only for soliciting color choices. To learn how to use the Color Picker Utilities, see *Inside Macintosh: Advanced Color Imaging*.

As color devices for input and output proliferate, so do the problems of moving images between them with good results. Different device types use different color models, which produce different gamuts, or ranges of colors. Screens, for example, typically display colors as combinations of red, green, and blue—combinations that your application specifies with `RGBColor` records when using Color QuickDraw. Screens by different manufacturers may be capable of displaying different intensities of red, green and blue, so that even though the screens work with RGB colors, their gamuts may be different. Color printers typically use a CMYK color model to work with varying intensities of cyan, magenta, yellow, and black. Print technologies vary drastically, and the gamut that an ink jet color printer can display may be quite different from one based on another technology. A single printer may be able to produce different gamuts depending on the paper or ink in use at the time of printing.

Two devices with differing color gamuts cannot reproduce each other's colors exactly, but shifting the colors of one device may improve the visual match. To match colors between screens and input and output devices such as scanners and printers, Macintosh system software provides a set of routines and algorithms called the **ColorSync Utilities.** Developers writing device drivers use the ColorSync Utilities to support color matching between devices. You can use the ColorSync Utilities in your application to communicate with a driver and present users with color-matching information—such as a device's color capabilities. For an image that your application prepares, for example, your application can present a print preview dialog box that signifies those colors within the image that the printer cannot accurately reproduce. Your application can also allow users to choose whether and how to match colors in the image with those available on the printer. The ColorSync Utilities are described in *Inside Macintosh: Advanced Color Imaging.*

The **Color Manager** assists Color QuickDraw in mapping your application's color requests to the actual colors available. Most applications never need to call the Color Manager directly. However, for completeness, the routines and data structures of the Color Manager are described in *Inside Macintosh: Advanced Color Imaging.*

Apple has also developed a new, object-based graphics architecture called **QuickDraw GX.** This new architecture provides applications with sophisticated color publishing capabilities. Your application can use QuickDraw GX instead of QuickDraw to create and draw objects on the screen. Rather than provide a set of drawing commands, as QuickDraw does, QuickDraw GX is built around graphics objects that your application can use as needed.

Your application can also use QuickDraw GX for drawing text. QuickDraw GX provides many sophisticated font and line layout capabilities, such as ligatures, style variations, kerning, and resolution-independent type manipulation.

For printing, QuickDraw GX offers flexible new capabilities to users and an architecture that streamlines development time for developers who write printing drivers. Even if your application uses QuickDraw and the Font Manager instead of QuickDraw GX to create images and text, your application can use the printing capabilities of QuickDraw GX.

See the *Inside Macintosh: QuickDraw GX* suite of books for information about programming with QuickDraw GX imaging technology.