(excerpt For ECE660, Fall 1999) - F.S.Hill,Jr.

CHAPTER 1. Introduction to Computer Graphics

"Begin at the beginning," the King said gravely,
"and go on till you come to the end; then stop."
Lewis Carroll, Alice in Wonderland

The machine does not isolate
the man from the great
problems of nature
but plunges him more deeply
into them
Antoine de Saint-Exupéry

"Any sufficiently advanced
technology is indistinguishable from magic."
Arthur C. Clarke

1.1 What is computer graphics?

Good question. People use the term "computer graphics" to mean different things in different contexts. Most simply, computer graphics are **pictures** that are generated by a computer. Everywhere you look today there are examples to be found, especially in magazines and on television. This book was typeset using a computer: every character (even this one: **G**) was "drawn" from a library of character shapes stored in computer memory. Books and magazines abound with pictures created on a computer. Some look so natural you can't distinguish them from photographs of a "real" scene. Others have an artificial or surreal feeling, intentionally fashioned to achieve some visual effect. And movies today often show scenes that never existed, but were carefully crafted by computer, mixing the real and the imagined.

"Computer graphics" also refers to the **tools** used to make such pictures. The purpose of this book is to show what the tools are and how to apply them. There are both hardware and software tools. Hardware tools include video monitors and printers that display graphics, as well as input devices like a mouse or trackball that let a user point to items and draw figures. The computer itself, of course, is a hardware tool, along with its special circuitry to facilitate graphical display or image capture.

As for software tools, you are already familiar with the usual ones: the operating system, editor, compiler, and debugger, that are found in any programming environment. For graphics there must also be a collection of "graphics routines" that produce the pictures themselves. For example, all graphics libraries have functions to draw a simple line or circle (or characters such as **G**). Some go well beyond this, containing functions to draw and manage windows with pull-down menus and dialog boxes, or to set up a "camera" in a three-dimensional coordinate system and to make "snapshots" of objects stored in some data base.

In this book we show how to write programs that utilize graphics libraries, and how to add functionality to them. Not too long ago, programmers were compelled to use highly "device dependent" libraries, designed for use on one specific computer system with one specific display device type. This made it very difficult to "port" a program to another system, or to use it with another device: usually the programmer had to make substantial changes to the program to get it to work, and the process was time-consuming and highly error-prone. Happily the situation is far better today. Device independent graphics libraries are now available that allow the programmer to use a common set of functions within an application, and to run the same application on a variety of systems and displays. OpenGL is such a library, and serves as the main tool we use in this book. The OpenGL way of creating graphics is used widely in both universities and industry. We begin a detailed discussion of it in Chapter 2.

Finally, “computer graphics” often means the whole **field of study** that involves these tools and the pictures they produce. (So it’s also used in the singular form: “computer graphics is...”). The field is often acknowledged to have started in the early 1960’s with Ivan Sutherland’s pioneering doctoral thesis at MIT on ‘Sketchpad’ [ref]. Interest in graphics grew quickly, both in academia and industry, and there were rapid advances in display technology and in the algorithms used to manage pictorial information. The special interest group in graphics, SIGGRAPH[1], was formed in 1969, and is very active today around the world. (The must-not-miss annual SIGGRAPH meeting now attracts 30,000 participants a year.) More can be found at http://www.siggraph.org. Today there are hundreds of companies around the world having some aspect of computer graphics as their main source of revenue, and the subject of computer graphics is taught in most computer science or electrical engineering departments.

Computer graphics is a very appealing field of study. You learn to write programs that create pictures, rather than streams of text or numbers. Humans respond readily to pictorial information, and are able to absorb much more information from pictures than from a collection of numbers. Our eye-brain systems are highly attuned to recognizing visual patterns. Reading text is of course one form of pattern recognition: we instantly recognize character shapes, form them into words, and interpret their meaning. But we are even more acute when glancing at a picture. What might be an inscrutable blather of numbers when presented as text becomes an instantly recognizable shape or pattern when presented graphically. The amount of information in a picture can be enormous. We not only recognize what’s “in it”, but also glean a world of information from its subtle details and texture

People study computer graphics for many reasons. Some just want a better set of tools for plotting curves and presenting the data they encounter in their other studies or work. Some want to write computer-animated games, while others are looking for a new medium for artistic expression. Everyone wants to be more productive, and to communicate ideas better, and computer graphics can be a great help.

There is also the “input” side. A program generates output — pictures or otherwise — from a combination of the algorithms executed in the program and the data the user inputs to the program. Some programs accept input crudely through characters and numbers typed at the keyboard. Graphics program, on the other hand, emphasize more familiar types of input: the movement of a mouse on a desktop, the strokes of a pen on a drawing tablet, or the motion of the user’s head and hands in a virtual reality setting. We examine many techniques of “interactive computer graphics” in this book; that is, we combine the techniques of natural user input with those that produce pictorial output.

(Section 1.2 on uses of Computer Graphics deleted.)

1.3. Elements of Pictures Created in Computer Graphics.

What makes up a computer drawn picture? The basic objects out of which such pictures are composed are called **output primitives**. One useful categorization of these is:

- polylines
- text
- filled regions
- raster images

We will see that these types overlap somewhat, but this terminology provides a good starting point. We describe each type of primitive in turn, and hint at typical software routines that are used to draw it. More detail on these tools is given in later chapters, of course. We also discuss the various attributes of each output primitive**.** The **attributes** of a graphic primitive are the characteristics that affect how it appears, such as color and thickness.

[1] SIGGRAPH is a Special Interest Group in the ACM: the Association for Computing Machinery.

1.3.1. Polylines.

A polyline is a connected sequence of straight lines. Each of the examples in Figure 1.8 contain several polylines: a). one polyline extends from the nose of the dinosaur to its tail; the plot of the mathematical function is a single polyline, and the "wireframe" picture of a chess pawn contains many polylines that outline its shape .

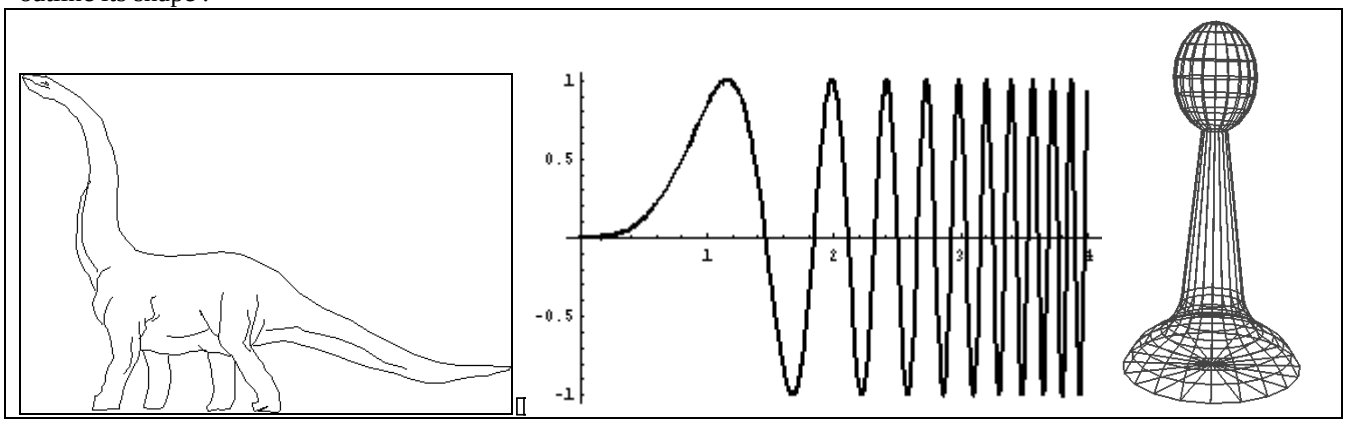


Figure 1.8. a). a polyline drawing of a dinosaur (courtesy of Susan Verbeck),
b). a plot of a mathematical function,
c). a wireframe rendering of a 3D object.

Note that a polyline can appear as a smooth curve. Figure 1.9 shows a blow-up of a curve revealing its underlying short line segments. The eye blends them into an apparently smooth curve.

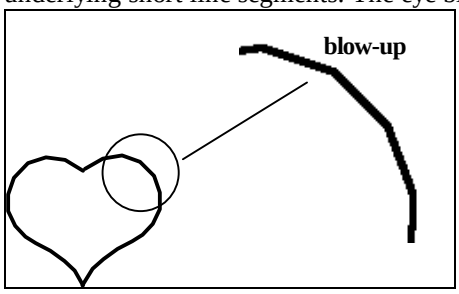


Figure 1.9. A curved line made up of straight line segments.

Pictures made up of polylines are sometimes called **line drawings.** Some devices, like a pen plotter, are specifically designed to produce line drawings.

The simplest polyline is a single straight line segment. A line segment is specified by its two endpoints, say (x_1, y_1) and (x_2, y_2). A drawing routine for a line might look like `drawLine(x1, y1, x2, y2);`

It draws a line between the two endpoints. We develop such a tool later, and show many examples of its use. At that point we get specific about how coordinates like x_1 are represented (by integers or by real numbers), and how colors can be represented in a program.

A special case arises when a line segment shrinks to a single point, and is drawn as a "dot". Even the lowly dot has important uses in computer graphics, as we see later. A dot might be programmed using the routine `drawDot(x1, y1);`

When there are several lines in a polyline each one is called an **edge**, and two adjacent lines meet at a **vertex**. The edges of a polyline can cross one another, as seen in the figures**.** Polylines are specified as a list of vertices, each given by a coordinate pair:

$$(x_0, y_0),\ (x_1, y_1), (x_2, y_2), ..., (x_n, y_n) \qquad (1.1)$$

For instance, the polyline shown in Figure 1.10 is given by the sequence (2, 4), (2, 11), (6, 14), (12, 11), (12, 4), ... (what are the remaining vertices in this polyline?).

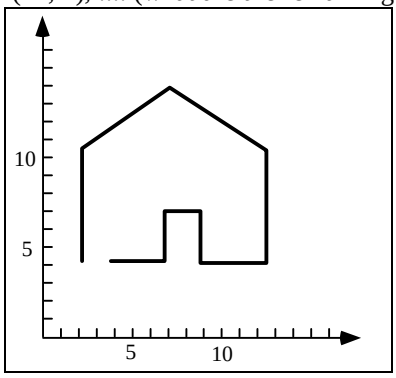


Figure 1.10. An example polyline.

To draw polylines we will need a tool such as: `drawPolyline( poly);`

where the variable `poly` is a list containing all the endpoints (x_i, y_i) in some fashion. There are various ways to capture a list in a program, each having its advantages and disadvantages.

A polyline need not form a closed figure, but if the first and last points are connected by an edge the polyline is a **polygon**. If in addition no two edges cross, the polygon is called **simple.** Figure 1.11 shows some interesting polygons; only *A* and *D* are simple. Polygons are fundamental in computer graphics, partly because they are so easy to define, and many drawing (rendering) algorithms have been finely tuned to operate optimally with polygons. Polygons are described in depth in Chapter 3.

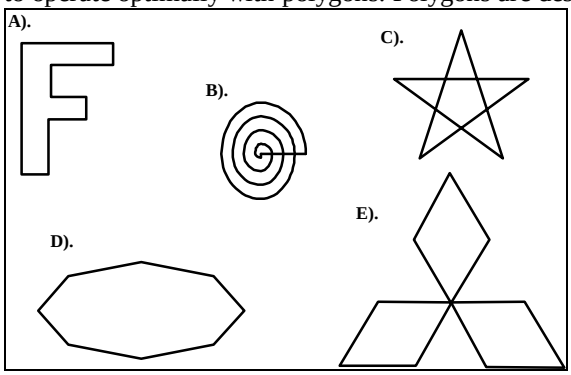


Figure 1.11. Examples of polygons..

Attributes of Lines and Polylines.
Important attributes of a polyline are the color and thickness of its edges, the manner in which the edges are dashed, and the manner in which thick edges blend together at their endpoints. Typically all of the edges of a polyline are given the same attributes.

The first two polylines in Figure 1.12 are distinguished by the line thickness attribute. The third polyline is drawn using dashed segments.

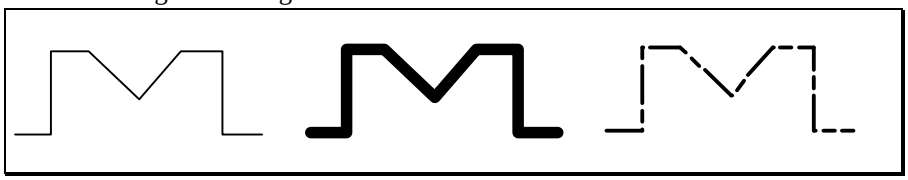


Figure 1.12. Polylines with different attributes.

When a line is thick its ends have shapes, and a user must decide how two adjacent edges "join". Figure 1.13 shows various possibilities. Case a) shows "butt-end" lines that leave an unseemly "crack" at the joint. Case b) shows rounded ends on the lines so they join smoothly, part c) shows a mitered joint, and part d) shows a trimmed mitered joint. Software tools are available in some packages to allow the user to choose the type of joining. Some methods are quite expensive computationally.

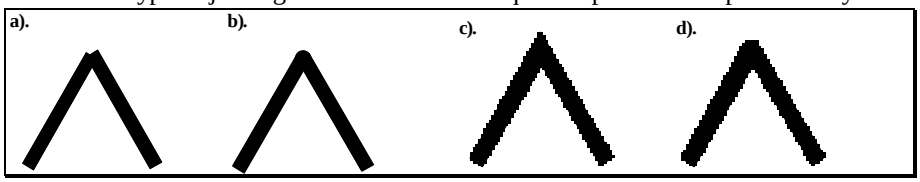


Figure 1.13. Some ways of joining two thick lines in a polyline.

The attributes of a polyline are sometimes set by calling routines such as `setDash(dash7)` or `setLineThickness(thickness)`.

1.3.2. Text.

Some graphics devices have two distinct display modes, a **text mode** and a **graphics mode**. The text mode is used for simple input/output of characters to control the operating system or edit the code in a program. Text displayed in this mode uses a built-in character generator. The character generator is capable of drawing alphabetic, numeric, and punctuation characters, and some selection of special symbols such as **♥, ð,** and ⊕ . Usually these characters can't be placed arbitrarily on the display but only in some row and column of a built-in grid.

A graphics mode offers a richer set of character shapes, and characters can be placed arbitrarily. Figure 1.14 shows some examples of text drawn graphically.

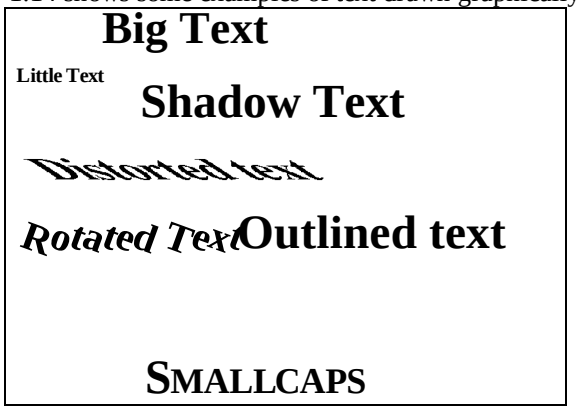


Figure 1.14. Some text drawn graphically.

A tool to draw a character string might look like: `drawString(x, y, string);` It places the starting point of the string at position (*x*, *y*), and draws the sequence of characters stored in the variable `string`.

Text Attributes.

There are many text attributes, the most important of which are typeface, color, size, spacing, and orientation.

Font. A font is a specific set of character shapes (a **typeface**) in a particular style and size. Figure 1.15 shows various character styles.

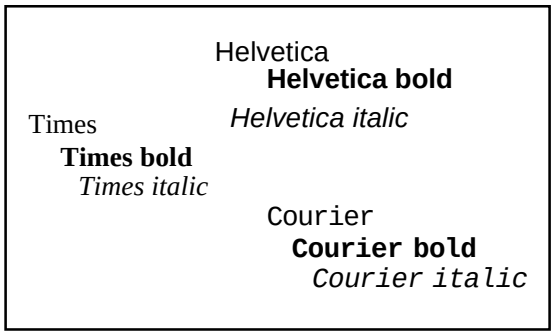


Figure 1.15. Some examples of fonts.

The shape of each character can be defined by a polyline (or more complicated curves such as Bezier curves – see Chapter 11), as shown in Figure 1.16a, or by an arrangement of dots, as shown in part b. Graphics packages come with a set of predefined fonts, and additional fonts can be purchased from companies that specialize in designing them.

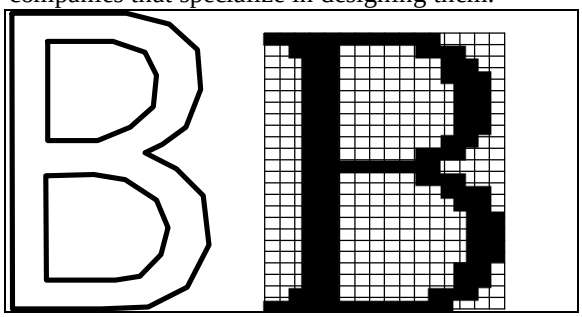


Figure 1.16. A character shape defined by a polyline and by a pattern of dots.

Orientation of characters and strings: Characters may also be drawn tilted along some direction. Tilted strings are often used to annotate parts of a graph. The graphic presentation of high-quality text is a complex subject. Barely perceptible differences in detail can change pleasing text into ugly text. Indeed, we see so much printed material in our daily lives that we subliminally expect characters to be displayed with certain shapes, spacings, and subtle balances.

1.3.3. Filled Regions

The **filled region** (sometimes called "fill area") primitive is a shape filled with some color or pattern. The boundary of a filled region is often a polygon (although more complex regions are considered in Chapter 4). Figure 1.17 shows several filled polygons. Polygon *A* is filled with its edges visible, whereas *B* is filled with its border left undrawn. Polygons *C* and *D* are non-simple. Polygon *D* even contains polygonal holes. Such shapes can still be filled, but one must specify exactly what is meant by a polygon's "interior", since filling algorithms differ depending on the definition. Algorithms for performing the filling action are discussed in Chapter 10.

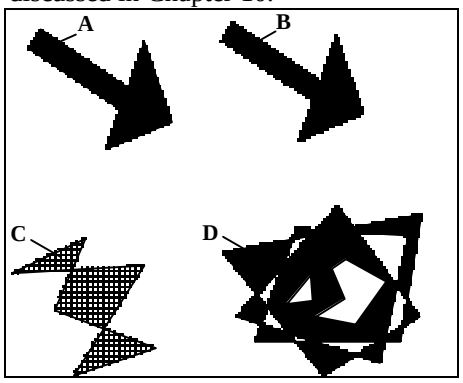


Figure 1.17. Examples of filled Polygons.

To draw a filled polygon one would use a routine like: `fillPolygon(poly, pattern);`

where the variable `poly` holds the data for the polygon - the same kind of list as for a polyline - and the variable `pattern` is some description of the pattern to be used for filling. We discuss details for this in Chapter 4.

Figure 1.18 shows the use of filled regions to shade the different faces of a 3D object. Each polygonal "face" of the object is filled with a certain shade of gray that corresponds to the amount of light that would reflect off that face. This makes the object appear to be bathed in light from a certain direction. Shading of 3D objects is discussed in Chapter 8.

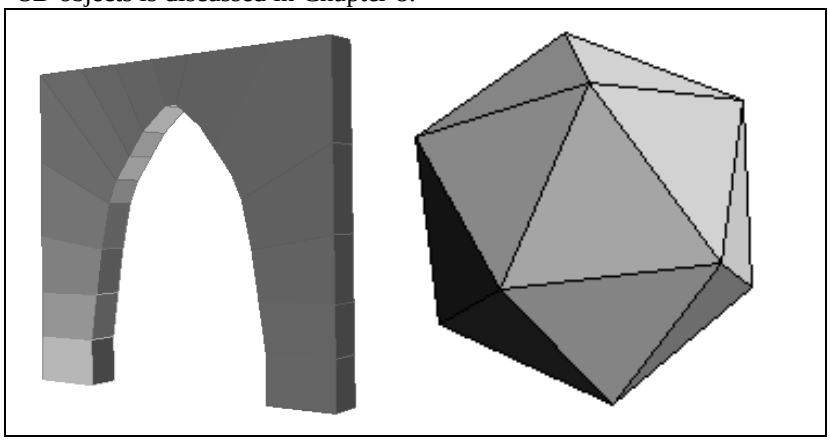


Figure 1.18. Filling polygonal faces of 3D objects to suggest proper shading.

The attributes of a filled region include the attributes of the enclosing border, as well as the pattern and color of the filling.

1.3.4. Raster Image.

Figure 1.19a shows a **raster image** of a chess piece. It is made up of many small "cells", in different shades of gray, as revealed in the blow-up shown in Figure 1.19b. The individual cells are often called "**pixels**" (short for "picture elements"). Normally your eye can't see the individual cells; it blends them together and synthesizes an overall picture.

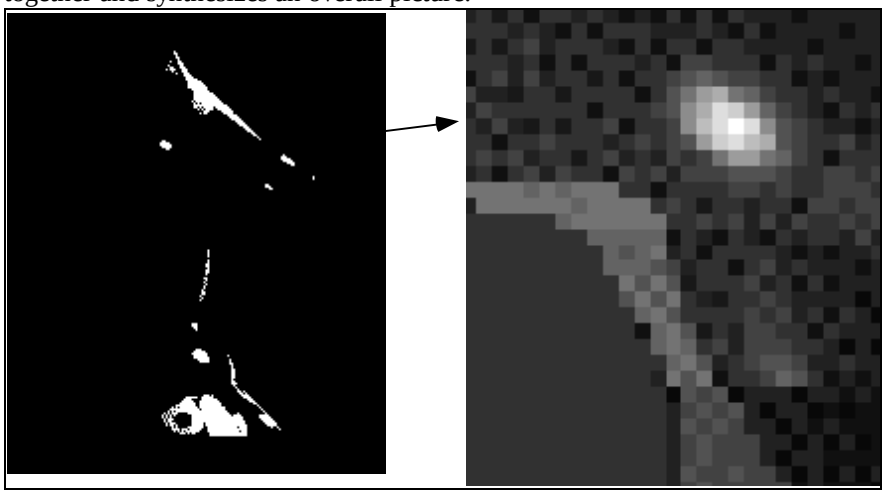
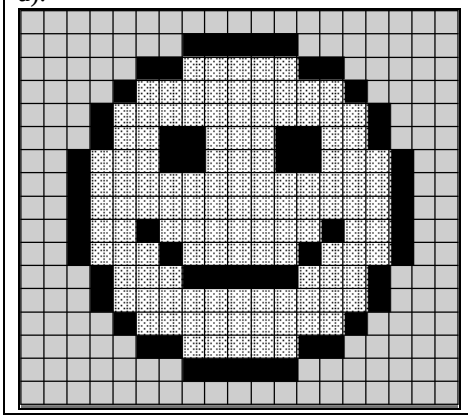


Figure 1.19. a). A raster image of a chess piece. b). A blow-up of the image. (Raytracing courtesy of Andrew Slater)

A raster image is stored in a computer as an array of numerical values. This array is thought of as being rectangular, with a certain number of rows and a certain number of columns. Each numerical value represents the value of the pixel stored there. The array as a whole is often called a "**pixel map"**. The term "**bitmap"** is also used, (although some people think this term should be reserved for pixel maps wherein each pixel is represented by a single bit, having the value 0 or 1.)

Figure 1.20 shows a simple example where a figure is represented by a 17 by 19 array (17 rows by 19 columns) of cells in three shades of gray. Suppose the three gray levels are encoded as the values 1, 2, and 7. Figure 1.20b shows the numerical values of the pixel map for the upper left 6 by 8 portion of the image.

a).



b).

2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	7
2	2	2	2	2	7	7	1
2	2	2	2	7	1	1	1
2	2	2	7	1	1	1	1
2	2	2	7	1	1	7	7

Figure 1.20. A simple figure represented as a bitmap.

How are raster images created? The three principal sources are:

1). Hand designed images.

A designer figures out what values are needed for each cell, and types them into memory. Sometimes a paint program can be used to help automate this: the designer can draw and manipulate various graphical shapes, viewing what has been made so far. When satisfied, the designer stores the result in a file. The icon above was created this way.

2). Computed Images.

An algorithm is used to "render" a scene, which might be modeled abstractly in computer memory. As a simple example, a scene might consist of a single yellow smooth sphere illuminated by a light source that emanates orange light. The model contains descriptions of the size and position of the sphere, the placement of the light source, and a description of the hypothetical "camera" that is to "take the picture". The raster image plays the role of the film in the camera. In order to create the raster image, an algorithm must calculate the color of light that falls on each pixel of the image in the camera. This is the way in which ray traced images such as the chess piece in Figure 1.20 are created; see Chapter 16.

Raster images also frequently contain images of straight lines. A line is created in an image by setting the proper pixels to the line's color. But it can require quite a bit of computation to determine the sequence of pixels that "best fit" the ideal line between two given end points. Bresenham's algorithm (see Chapter 2) provides a very efficient approach to determining these pixels.

Figure 1.21a shows a raster image featuring several straight lines, a circular arc, and some text characters. Figure 1.21b shows a close-up of the raster image in order to expose the individual pixels that are "on" the lines. For a horizontal or vertical line the black square pixels line up nicely forming a sharp line. But for the other lines and the arc the "best" collection of pixels produces only an approximation to the "true" line desired. In addition, the result shows the dread "**jaggies**" that have a relentless presence in raster images.

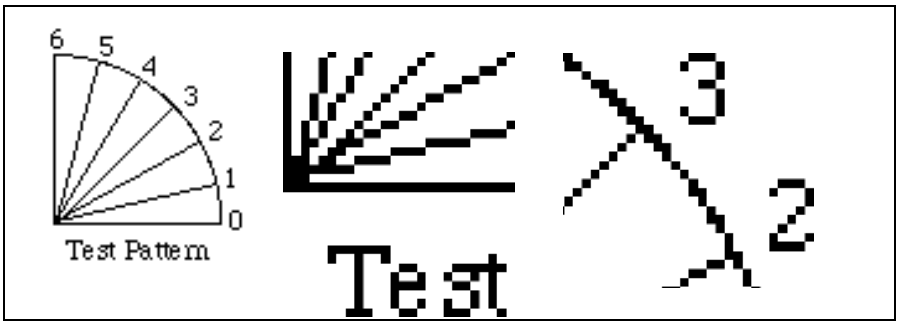


Figure 1.21. a). a collection of lines and text. b). Blow-up of part a, having “jaggies”.

3). Scanned images.
A photograph or television image can be digitized as described above. In effect a grid is placed over the original image, and at each grid point the digitizer reads into memory the “closest” color in its repertoire. The bitmap is then stored in a file for later use. The image of the kitten in Figure 1.22 was formed this way.

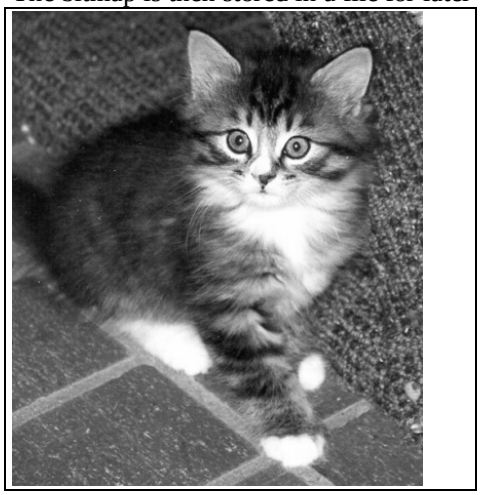


Figure 1.22. A scanned image.

Because raster images are simply arrays of numbers, they can be subsequently processed to good effect by a computer. For instance, Figure 1.23 shows three successive enlargements of the kitten image above. These are formed by “pixel replication” (discussed in detail in Chapter 10). Each pixel has been replicated three times in each direction in part a; by six times in part b, and by 12 times in part c.

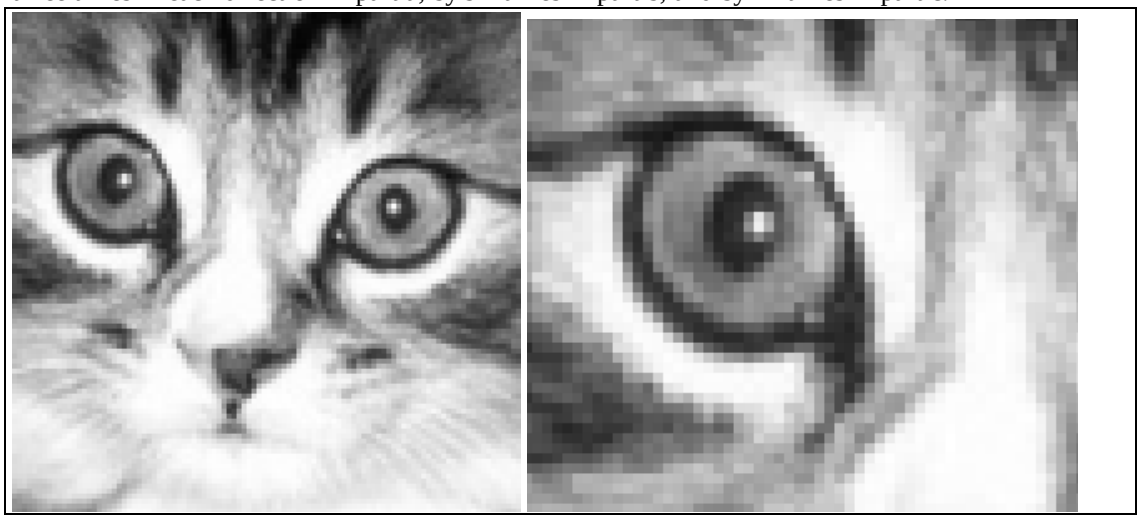


Figure 1.23. Three successive blow-ups of the kitten image. a). three times enlargement, b). six times enlargement.

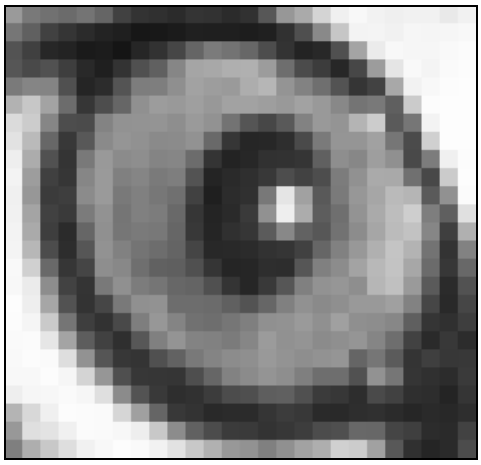


Figure 1.23c. Twelve times enlargement.

As another example, one often needs to “clean up” a scanned image, for instance to remove specks of noise or to reveal important details. Figure 1.24a shows the kitten image with gray levels altered to increase the contrast and make details more evident, and Figure 1.24b shows the effect of “edge enhancement”, achieved by a form of filtering the image.

Figure 1.24. Examples of image enhancement

Figure 1.25 shows two examples of editing an image to accomplish some visual effect. Part a shows the kitten image “embossed”, and part b shows it distorted geometrically.

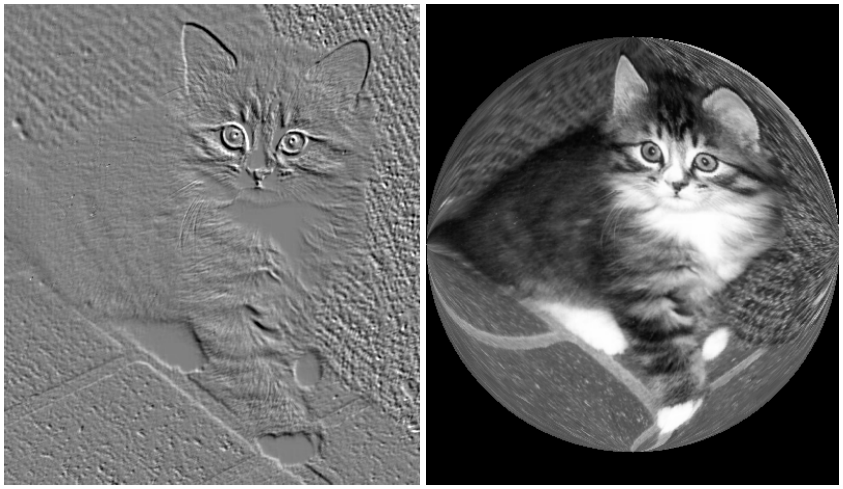


Figure 1.25. Examples of altering an image for visual effect.

1.3.5. Representation of gray shades and color for Raster Images

An important aspect of a raster image is the manner in which the various colors or shades of gray are represented in the bitmap. We briefly survey the most common methods here.

1.3.5.1. Gray-scale Raster Images.

If there are only two pixel values in a raster image it is called **bi-level.** Figure 1.26a shows a simple bi-level image, representing a familiar arrow-shaped cursor frequently seen on a computer screen. Its raster consists of 16 rows of 8 pixels each. Figure 1.26b shows the bitmap of this image as an array of 1's and 0's. The image shown at the left associates black with a 1 and white with a 0, but this association might just as easily be reversed. Since one bit of information is sufficient to distinguish two values, a bilevel image is often referred to as a "**1 bit per pixel**" image.

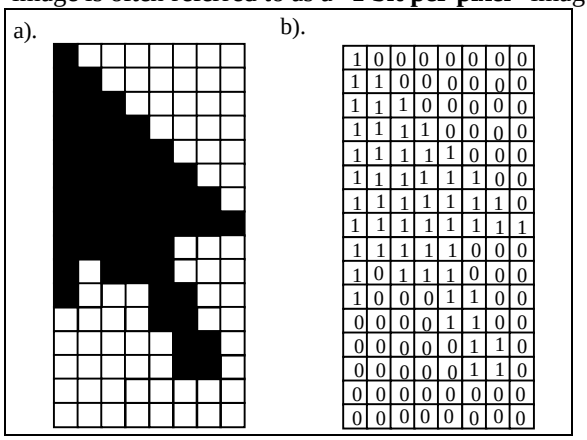


Figure 1.26. A bilevel image of a cursor, and its bitmap.

When the pixels in a gray-scale image take on more than two values, each pixel requires more than a single bit to represent it in memory. Gray-scale images are often classified in terms of their **pixel depth,** the number of bits needed to represent their gray levels. Since an *n*-bit quantity has 2^n possible values, there can be 2^n gray levels in an image with pixel depth *n*. The most common values are:

- 2 bits/pixel produce 4 gray levels
- 4 bits/pixel produce 16 gray levels
- 8 bits/pixel produce 256 gray levels

Figure 1.27 shows 16 gray levels ranging from black to white. Each of the sixteen possible pixel values is associated with a binary **4-tuple** such as 0110 or 1110. Here 0000 represents black, 1111 denotes white, and the other 14 values represent gray levels in between.

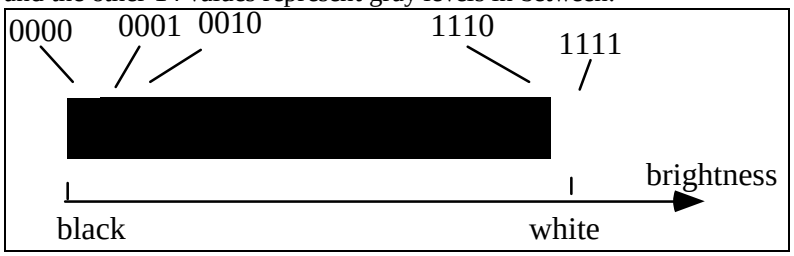


Figure 1.27. Sixteen levels of gray.

Many gray scale images[2] employ 256 gray levels, since this usually gives a scanned image acceptable quality. Each pixel is represented by some 8-bit values such as 01101110. The pixel value usually represents "brightness", where black is represented by 00000000, white by 11111111, and a medium gray by 10000000. Figure 1.23 seen earlier uses 256 gray levels.

Effect of Pixel depth: Gray-scale Quantization.
Sometimes an image that initially uses 8 bits per pixel is altered so that fewer bits per pixel are used. This might occur if a particular display device is incapable of displaying so many levels, or if the full image takes up too much memory. Figures 1.28 through 1.30 show the effect on the kitten image if pixel values are simply truncated to fewer bits. The loss in fidelity is hardly noticeable for the images in Figure 1.28, which use 6 and 5 bits/pixel (providing 64 and 32 different shades of gray, respectively).

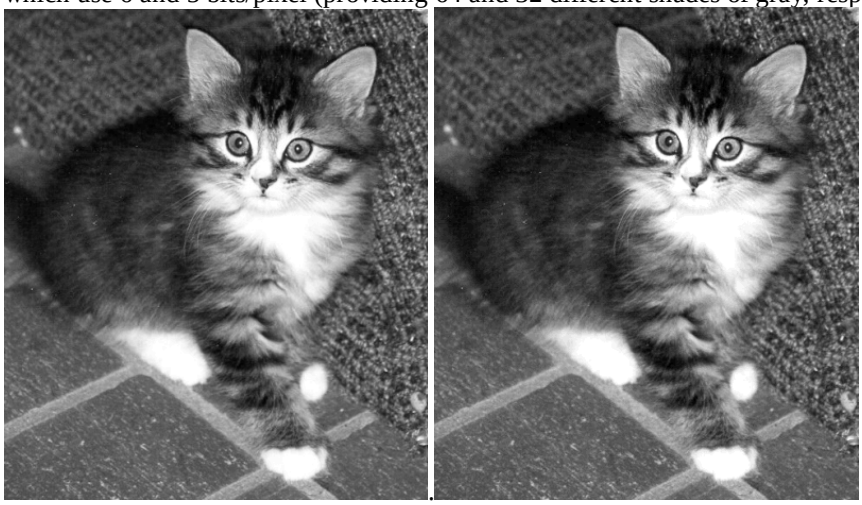


Figure 1.28. The image reduced to 6 bits/pixel and 5 bits/pixel.

But there is a significant loss in quality in the images of Figure 1.29. Part a shows the effect of truncating each pixel value to 4 bits, so there are only 16 possible shades of gray. For example, pixel value 01110100 is replaced with 0111. In part b the eight possible levels of gray are clearly visible. Note that some areas of the figure that show gradations of gray in the original now show a “lake” of uniform gray. This is often called **banding**, since areas that should show a gradual shift in the gray level instead show a sequence of uniform gray “bands”.

[2] Thousands are available on the Internet, frequently as Gif, Jpeg, or Tiff images.

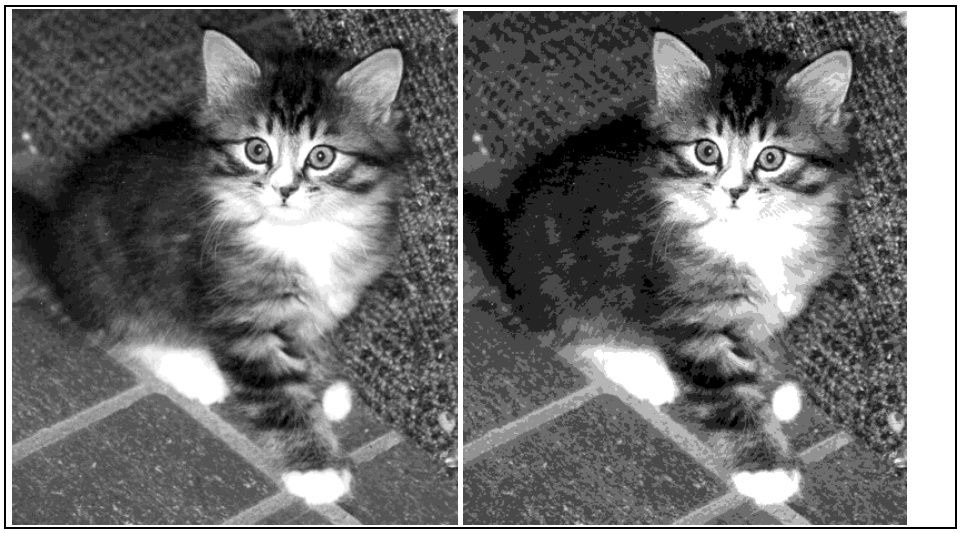


Figure 1.29. The image reduced to 4 bits/pixel and to 3 bits/pixel.

Figure 1.30 shows the cases of 2 and 1 bits/pixel. In part a the four levels are clearly visible and there is a great deal of banding. In part b there is only black and white and much of the original image information has been lost. In Chapter 10 we show techniques such as dithering for improving the quality of an image when two few bits are used for each pixel.

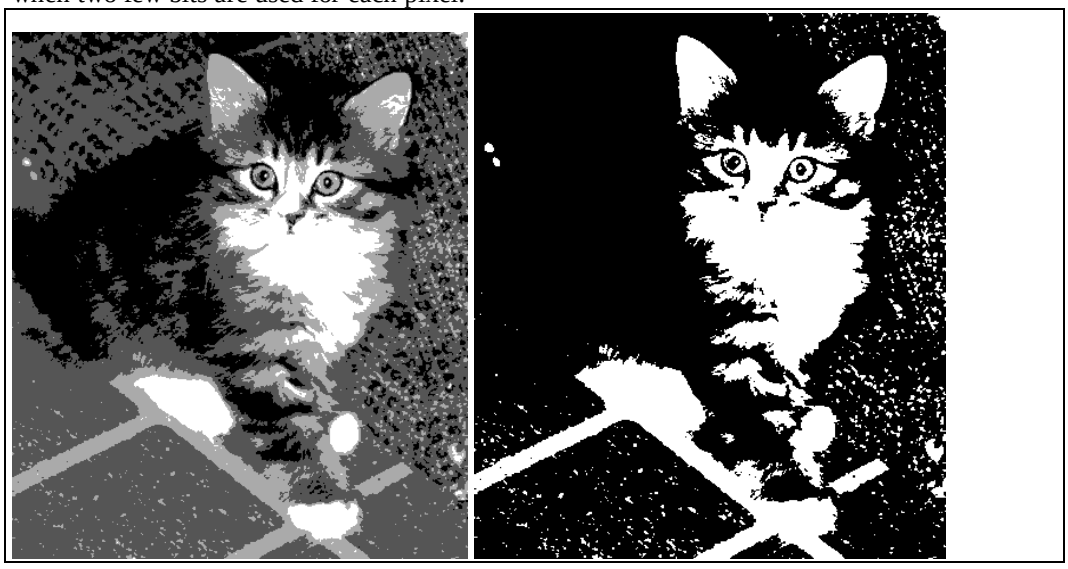


Figure 1.30. The image reduced to 2 bits/pixel and 1 bit/pixel.

1.3.5.2. Color Raster Images.

Color images are desirable because they match our daily experience more closely than do gray-scale images. Color raster images have become more common in recent years as the cost of high quality color displays has come down. The cost of scanners that digitize color photos has also become reasonable.

Each pixel in a color image has a "color value", a numerical value that somehow represents a color. There are a number of ways to associate numbers and colors (see Chapter 12 for a detailed discussion), but one of the most common is to describe a color as a combination of amounts of red, green, and blue light. Each pixel value is a **3-tuple**, such as (23, 14, 51), that prescribes the intensities of the red, green, and blue light components in that order.

The number of bits used to represent the color of each pixel is often called its **color depth.** Each value in the (red, green, blue) 3-tuple has a certain number of bits, and the color depth is the sum of these values. A color depth of 3 allows one bit for each component. For instance the pixel value (0, 1, 1) means that the red component is "off", but both green and blue are "on". In most displays the contributions from each component are added together (see Chapter 12 for exceptions such as in printing), so (0,1,1) would represent the addition of green and blue light, which is perceived as cyan. Since each component can be on or off there are eight possible colors, as tabulated in Figure 1.31. As expected, equal amounts of red, green, and blue, (1, 1, 1), produce white.

color value	displayed
0,0,0	black
0,0,1	blue
0,1,0	green
0,1,1	cyan
1,0,0	red
1,0,1	magenta
1,1,0	yellow
1,1,1	white

Figure 1.31. A common correspondence between color value and perceived color.

A color depth of 3 rarely offers enough precision for specifying the value of each component, so larger color depths are used. Because a byte is such a natural quantity to manipulate on a computer, many images have a color depth of eight. Each pixel then has one of 256 possible colors. A simple approach allows 3 bits for each of the red and the green components, and 2 bits for the blue component. But more commonly the association of each byte value to a particular color is more complicated, and uses a "color look-up" table, as discussed in the next section.

The highest quality images, known as **true color** images, have a color depth of 24, and so use a byte for each component. This seems to achieve as good color reproduction as the eye can perceive: more bits don't improve an image. But such images require a great deal of memory: three bytes for every pixel. A high quality image of 1080 by 1024 pixels requires over three million bytes!

Plates 19 through 21 show some color raster images having different color depths. Plate 19 shows a full color image with a color depth of 24 bits. Plate 20 shows the degradation this image suffers when the color depth is reduced to 8 by simply truncating the red and green components to 3 bits each, and the blue component to 2 bits. Plate 21 also has a color depth of 8, so its pixels contain only 256 colors, but the 256 particular colors used have been carefully chosen for best reproduction. Methods to do this are discussed in Chapter 12.

author-supplied

Plate 19. Image with 24 bits/pixel.

author-supplied

Plate 20. Image with 3 bits for red and green pixels, and two bits for blue pixels.

author-supplied

Plate 1.21. Image with 256 carefully chosen colors.

1.4. Graphics Display Devices

We present an overview of some hardware devices that are used to display computer graphics. The devices include video monitors, plotters, and printers. A rich variety of graphics displays have been developed over the last thirty years, and new ones are appearing all the time. The quest is to display pictures of ever higher quality, that recreate more faithfully what is in the artist's or engineer's mind. In this section we look over the types of pictures that are being produced today, how they are being used, and the kinds of devices used to display them. In the process we look at ways to measure the "quality" of an image, and see how different kinds of display devices measure up.

1.4.1. Line Drawing Displays.

Some devices are naturally line-drawers. Because of the technology of the time, most early computer graphics were generated by line-drawing devices. The classic example is the **pen plotter**. A pen plotter moves a pen invisibly over a piece of paper to some spot that is specified by the computer, puts the pen down, and then sweeps the pen across to another spot, leaving a trail of ink of some color. Some plotters have a carousel that holds several pens which the program can exchange automatically in order to draw in different colors. Usually the choice of available colors is very limited: a separate pen is used for each color. The "quality" of a line-drawing is related to the precision with which the pen is positioned, and the sharpness of the lines drawn.

There are various kinds of pen plotters. **Flatbed plotters** move the pen in two dimensions over a stationary sheet of paper. **Drum plotters** move the paper back and forth on a drum to provide one direction of motion, while the pen moves back and forth at the top of the drum to provide the other direction.

There are also video displays called "vector", "random-scan", or "calligraphic" displays, that produce line-drawings. They have internal circuitry specially designed to sweep an electronic beam from point to point across the face of a cathode ray tube, leaving a glowing trail.

Figure 1.32 shows an example of a vector display, used by a flight controller to track the positions of many aircraft. Since each line segment to be displayed takes only a little data (two end points and perhaps a color), vector displays can draw a picture very rapidly (hundreds of thousands of vectors per second).

author-supplied

Figure 1.32. Example of a vector display (Courtesy Evans & Sutherland)

Vector displays, however, cannot show smoothly shaded regions or scanned images. Region filling is usually simulated by **cross hatching** with different line patterns, as suggested in Figure 1.33. Today raster displays have largely replaced vector displays except in very specialized applications.

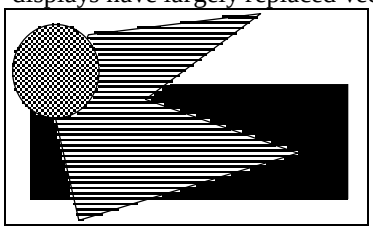


Figure 1.33. Cross-hatching to simulate filling a region.

1.4.2. Raster Displays.

Most displays used today for computer graphics are raster displays. The most familiar raster displays are the **video monitor** connected to personal computers and workstations (see Figure 1.34a), and the **flat panel** display common to portable personal computers (see Figure 1.34b). Other common examples produce **hard** copy of an image: the **laser printer**, **dot matrix printer**, **ink jet plotter**, and **film recorder.** We describe the most important of these below.

author-supplied

Figure 1.34. a). video monitors on PC, b). flat panel display.

Raster devices have a **display surface** on which the image is presented. The display surface has a certain number of pixels that it can show, such as 480 rows, where each row contains 640 pixels. So this display surface can show 480 x 640 • 307,000 pixels simultaneously. All such displays have a built-in coordinate system that associates a given pixel in an image with a given physical position on the display surface. Figure 1.35 shows an example. Here the horizontal coordinate *sx* increases from left to right, and the

vertical coordinate *sy* increases from top to bottom. This "upside-down" coordinate system is typical of raster devices.

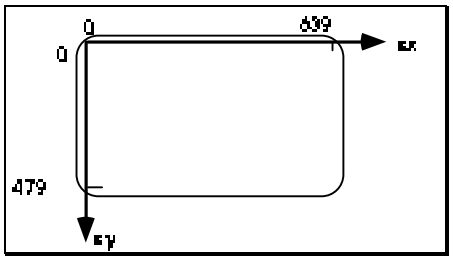


Figure 1.35. The built-in coordinate system for the surface of a raster display.

Raster displays are always connected one way or another to a **frame buffer**, a region of memory sufficiently large to hold all of the pixel values for the display (i.e. to hold the bitmap). The frame buffer may be physical memory on-board the display, or it may reside in the host computer**.** For example, a graphics card that is installed in a personal computer actually houses the memory required for the frame buffer.

Figure 1.36 suggests how an image is created and displayed. The graphics program is stored in system memory and is executed instruction by instruction by the central processing unit (CPU). The program computes appropriate values for each pixel in the desired image and loads them into the frame buffer. (This is the part we focus on later when it comes to programming: building tools that write the "correct" pixel values into the frame buffer.) A "scan controller" takes care of the actual display process. It runs autonomously (rather than under program control), and does the same thing pixel after pixel. It causes the frame buffer to "send" each pixel through a converter to the appropriate physical spot on the display surface. The converter takes a pixel value such as 01001011 and converts it to the corresponding quantity that produces a spot of color on the display.

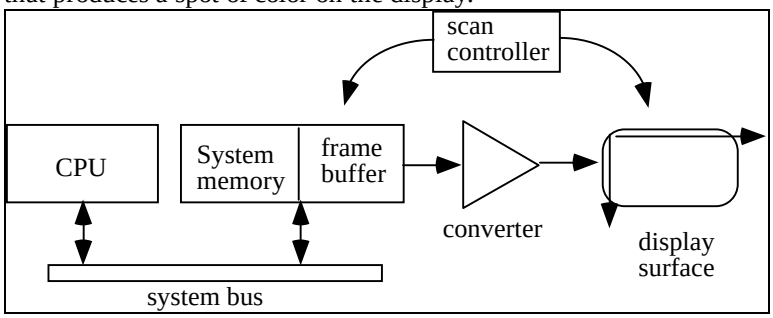


Figure 1.36. Block diagram of a computer with raster display.

The scanning process.
Figure 1.37 provides more detail on the scanning process. The main issue is how each pixel value in the frame buffer is "sent" to the right place on the display surface. Think of each of the pixels in the frame buffer as having a two-dimensional address (*x*, *y*). For address (136, 252), for instance, there is a specific memory location that holds the pixel value. Call it `mem`[136][252].

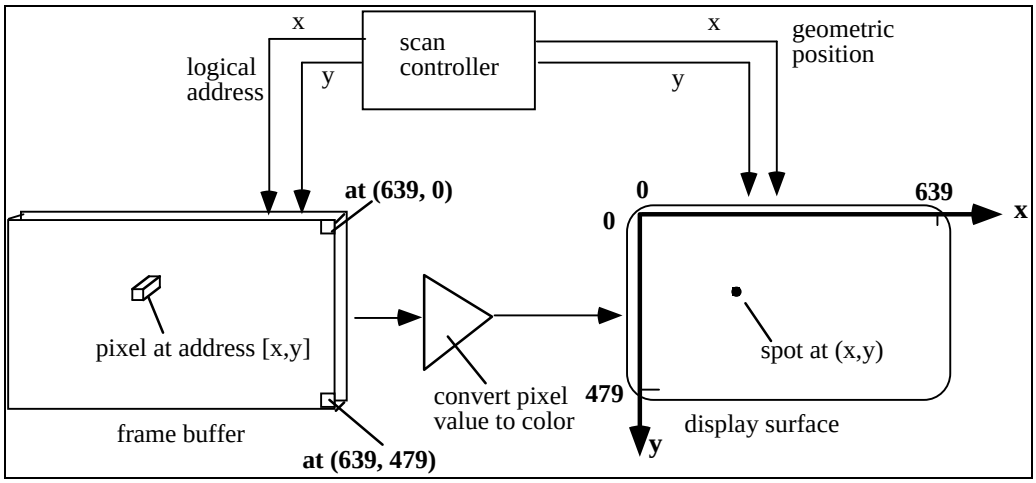


Figure 1.37. Scanning out an image from the frame buffer to the display surface.

The scan controller sends logical address (136, 252) to the frame buffer, which emits the value `mem`[136][252]. The controller also simultaneously "addresses" a physical (geometric) position (136, 252) on the display surface. Position (136, 252) corresponds to a certain physical distance of 136 units horizontally, and 252 units vertically, from the upper left hand corner of the display surface. Different raster displays use different units.

The value `mem`[136][252] is converted to a corresponding intensity or color in the conversion circuit, and the intensity or color is sent to the proper physical position (136, 252) on the display surface.

To scan out the image in the entire frame buffer, every pixel value is visited once, and its corresponding spot on the display surface is "excited" with the proper intensity or color.

In some devices this scanning must be repeated many times per second, in order to "refresh" the picture. The video monitor to be described next is such a device.

With these generalities laid down, we look briefly at some specific raster devices, and see the different forms that arise.

- **Video Monitors.**

Video monitors are based on a **CRT**, or cathode-ray tube, similar to the display in a television set. Figure 1.38 adds some details to the general description above for a system using a video monitor as the display device. In particular, the conversion process from pixel value to "spot of light" is illustrated. The system shown has a color depth of 6 bits; the frame buffer is shown as having six bit "planes". Each pixel uses one bit from each of the planes.

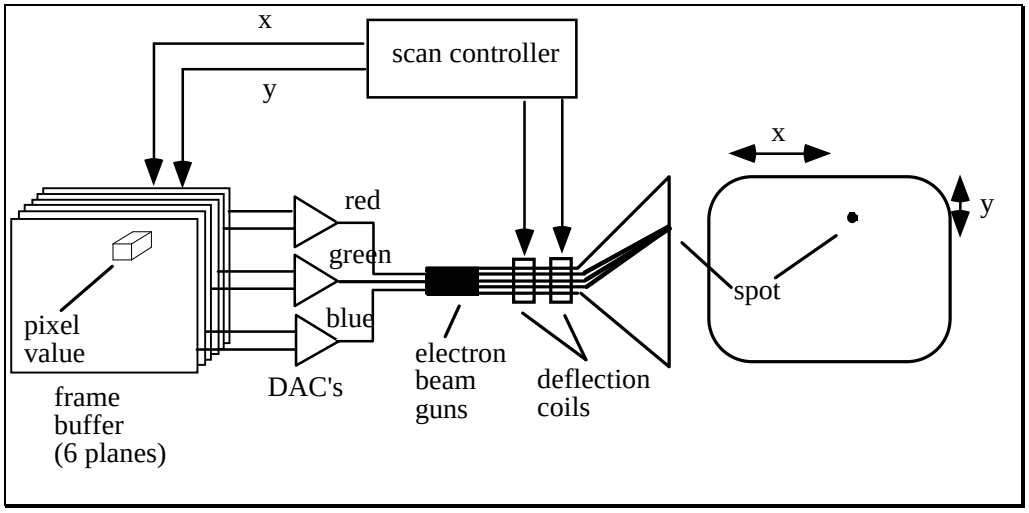


Figure 1.38. Operation of a color video monitor display system.

The red, green, and blue components of a pixel each use a pair of bits. These pairs are fed to three **digital-to-analog converters** (DAC's), which convert logical values like 01 into actual voltages. The correspondence between digital input values and output voltages is shown in Figure 1.39, where *Max* is the largest voltage level the DAC can produce.

input	voltage/brightness
00	0 * Max
01	0.333 * Max
10	0.666 * Max
11	1 * Max

Figure 1.39. Input-output characteristic of a two-bit DAC.

The three voltage levels drive three "guns" inside the CRT, which in turn excite three electron beams with intensities proportional to the voltages. The deflection coils divert the three beams so they stimulate three tiny phosphor dots at the proper place (*x*, *y*) on the inside of the cathode ray tube. Because of the phosphor materials used, one dot glows red when stimulated, one glows green, and one glows blue. The dots are so close together your eye sees one composite dot, and perceives a color that is the sum of the three component colors. Thus the composite dot can be made to glow in a total of 4 x 4 x 4 = 64 different colors.

As described earlier the scan controller addresses one pixel value `mem[x][y]` in the frame buffer at the same time it "addresses" one position (*x*, *y*) on the face of the CRT by sending the proper signal to the deflection coils. Because the glow of a phosphor dot quickly fades when the stimulation is removed, a CRT image must be **refreshed** rapidly (typically 60 times a second) to prevent disturbing **flicker**. During each "refresh interval" the scan controller scans quickly through the entire frame buffer memory, sending each pixel value to its proper spot on the screen's surface.

Scanning proceeds row by row through the frame buffer, each row providing pixel values for one **scanline** across the face of the CRT. The order of scanning is usually left to right along a **scanline** and from top to bottom by scanline. (Historians say this convention has given rise to terms like scanline, as well as the habit of numbering scanlines downward with 0 at the top, resulting in upside down coordinate systems.)

Some more expensive systems have a frame buffer that supports 24 planes of memory. Each of the DAC's has eight input bits, so there are 256 levels of red, 256 of green, and 256 of blue, for a total of 2^{24} = 16 million colors.

At the other extreme, there are **monochrome** video displays, which display a single color in different intensities. A single DAC converts pixel values in the frame buffer to voltage levels, which drive a single electron beam gun. The CRT has only one type of phosphor so it can produce various intensities of only one color. Note that 6 planes of memory in the frame buffer gives $2^6 = 64$ levels of gray.

The color display of Figure 1.39 has a *fixed* association with a displayed color. For instance, the pixel value 001101 sends 00 to the "red DAC", 11 to the "green DAC", and 01 to the "blue DAC", producing a mix of bright green and dark blue — a bluish-green. Similarly, 110011 is displayed as a bright magenta, and 000010 as a medium bright blue.

1.4.3. Indexed Color and the LUT.

Some systems are built using an alternative method of associating pixel values with colors. They use a **color lookup table** (or **LUT)**, which offers a *programmable* association between pixel value and final color. Figure 1.40 shows a simple example. The color depth is again six, but the six bits stored in each pixel go through an intermediate step before they drive the CRT. They are used as an *index* into a table of 64 values, say `LUT[0]...LUT[63]`. (Why are there exactly 64 entries in this LUT?) For instance, if a pixel value is 39, the values stored in `LUT[39]` are used to drive the DAC's, as opposed to having the bits in the value 39 itself drive them. As shown `LUT[39]` contains the 15 bit value 01010 11001 10010. Five of these bits (01010) are routed to drive the "red DAC", five others drive the "green DAC", and the last five drive the "blue DAC".

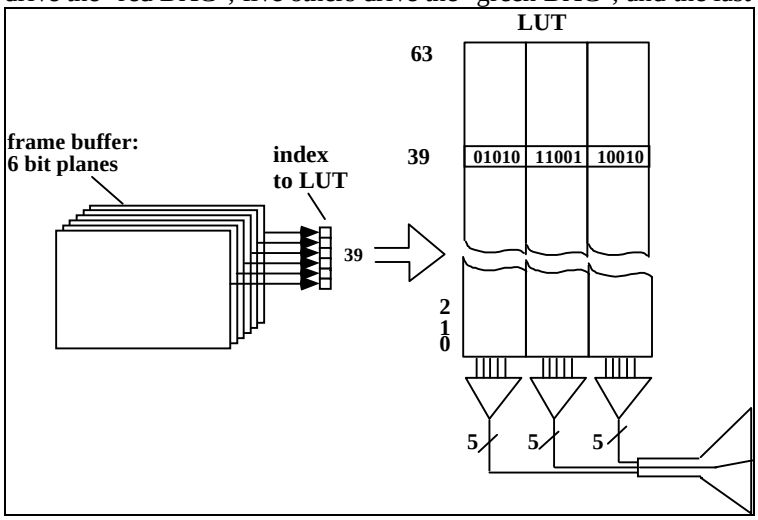


Figure 1.40. A color display system that incorporates a LUT.

Each of the `LUT[ ]` entries can be set under program control, using some system routine such as `setPalette()`. For example, the instruction: `setPalette(39, 17, 25, 4);`

would set the value in LUT[39] to the fifteen bit quantity 10001 11001 00100 (since 17 is 10001 in binary, 25 is 11001, and 4 is 00100).

To make a particular pixel glow in this color, say the pixel at location $(x, y) = (479, 532)$, the value 39 is stored in the frame buffer, using `drawDot()` defined earlier:

```
drawDot(479, 532, 39);          // set pixel at (479, 532) to value 39
```

Each time the frame buffer is "scanned out" to the display, this pixel is read as value 39, which causes the value stored in `LUT`[39] to be sent to the DAC's.

This programmability offers a great deal of flexibility in choosing colors, but of course it comes at a price: the program (or programmer) has to figure out which colors to use! We consider this further in Chapter 10.

What is the potential of this system for displaying colors? In the system of Figure 1.41 each entry of the LUT consists of 15 bits, so each color can be set to one of 2^{15} = 32K = 32,768 possible colors. The set of 2^{15} possible colors displayable by the system is called its **palette**, so we say this display "has a palette of 32K colors".

The problem is that each pixel value lies in the range 0..63, and only 64 different colors can be stored in the LUT at one time. Therefore this system can display a maximum of 64 different colors *at one time*. "At one time" here means during one scan-out of the entire frame buffer — something like 1/60-th of a second. The contents of the LUT are not changed in the middle of a scan-out of the image, so one whole scan-out uses a fixed set of 64 palette colors. Usually the LUT contents remain fixed for many scan-outs, although a program can change the contents of a small LUT during the brief dormant period between two successive scan-outs.

In more general terms, suppose that a raster display system has a color depth of b bits (so there are b bit planes in its frame buffer), and that each LUT entry is w bits wide. Then we have that:

The system can display 2^w colors, any 2^b at one time.

Examples.
(1). A system with b = 8 bit planes and a LUT width w = 12 can display 4096 colors, any 256 of them at a time.
(2). A system with b = 8 bitplanes and a LUT width w = 24 can display 2^{24} = 16,777,216 colors, any 256 at a time.
(3). If b = 12 and w = 18, the system can display 256k = 262,144 colors, 4096 at a time.

There is no enforced relationship between the number of bit planes, b, and the width of the LUT, w. Normally w is a multiple of 3, so the same number of bits (w/3) drives each of the three DAC's. Also, b never exceeds w, so the palette is at least as large as the number of colors that can be displayed at one time. (Why would you never design a system with $w < b$?)

Note that the LUT itself requires very little memory, only 2^b words of w bits each. For example, if b = 12 and w = 18 there are only 9,216 bytes of storage in the LUT.

So what is the motivation for having a LUT in a raster display system? It is usually a need to reduce the cost of memory. Increasing b increases significantly the amount of memory needed for the frame buffer, mainly because there are so many pixels. The tremendous amount of memory needed can add significantly to the cost of the overall system.

To compare the costs of two systems, one with a LUT and one without, Figure 1.41 shows an example of two 1024 by 1280 pixel displays, (so each of them supports about 1.3 million pixels). Both systems allow colors to be defined with a precision of 24 bits, often called "true color".

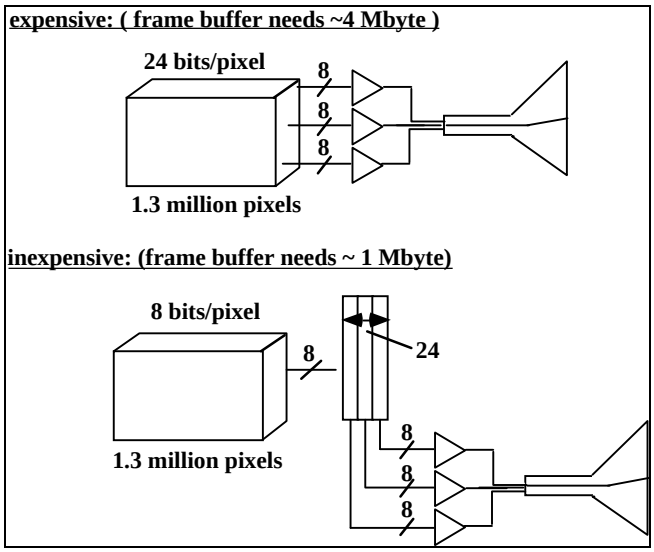


Figure 1.41. Comparison of two raster display systems.

System #1 (expensive): The first has a 24 bit/pixel frame buffer and no LUT, so each of its 1.3 million pixels can be set to any one of 2^{24} colors. Eight bits drive each of the DAC's. (The number '8' and the slash through the line into each DAC indicate the presence of eight bit lines feeding into the DAC.) The amount of memory required for the frame buffer here is 1024 x 1280 x 24 bits which is almost 4 megabytes.

System #2 (inexpensive): The second has an 8 bit/pixel frame buffer along with a LUT, and the LUT is 24 bits wide. The system can display 2^{24} different colors, but only 256 at a time. The amount of memory required for the frame buffer here is 1024 x 1280 x 8 which is about 1 megabyte. (The LUT requires a trivial 768 bytes of memory.) If memory costs a significant amount per megabyte, this system is much less expensive than system #1.

Putting a LUT in an inexpensive system attempts to compensate for the small number of different pixel values possible. The LUT allows the programmer to create a full set of colors, even though a given image can contain only a restricted set of them.

Displays with LUT's are still quite common today because memory costs are still high. The situation is changing rapidly, however, as memory prices plummet. Many reasonably priced personal computers today have 24 bit frame buffers.

Practice Exercises.

1.4.1. Why not always have a LUT? Since a LUT is inexpensive, and offers the advantage of flexibility, why not have a LUT even in a system with 24 bits per pixel?

1.4.2. Configure your own system. For each of the systems described below:

a). draw the circuit diagram, similar to Figure 1.41;

b). Label the number of bits associated with the frame buffer, the DAC's, and the LUT (if present);

c). Calculate (in bytes) the amount of storage required for the frame buffer and the LUT (if present):

i). b = 15, no LUT;

ii). b = 15, w = 24;

iii). b = 8, w = 18;

iv). b = 12, no LUT.

1.5. Graphics Input Primitives and Devices.

Many input devices are available that let the user control a computer. Whereas typing a command might be awkward, it is natural to "point" to a particular object displayed on the screen to make a choice for the next action.

You can look at an input device in two ways: what it *is*, and what it *does*. Each device is physically some piece of machinery like a mouse, keyboard, or trackball. It fits in the hand in a certain way and is natural for the user to manipulate. It measures these manipulations and sends corresponding numerical information back to the graphics program.

We first look at what input devices do, by examining the kinds of data each sends to the program. We then look at a number of input devices in common use today.

1.5.1. Types of Input Graphics Primitives.

Each device transmits a particular kind of data (e.g. a number, a string of characters, or a position) to the program. The different types of data are called **input primitives**. Two different physical devices may transmit the same type of data, so logically they generate the same graphics primitives.

The important input primitives are:

String. The **string** "device" is the most familiar, producing a **string of characters** and thus modeling the action of a keyboard. When an application requests a string, the program pauses while the user types it in followed by a termination character. The program then resumes with the string stored in memory.

Choice. A **choice** device reports a **selection** from a fixed number of choices. The programmer's model is a bank of buttons, or a set of buttons on a mouse.

Valuator. A **valuator** produces a real value between 0.0 and 1.0, which can be used to fix the length of a line, the speed of an action, or perhaps the size of a picture. The model in the programmer's mind is a knob that can be turned from 0 to 1 in smooth gradations.

Locator. A basic requirement in interactive graphics is to allow the user to point to a position on the display. The **locator** input device performs this function, because it produces a **coordinate pair** (x, y). The user manipulates an input device (usually a mouse) in order to position a visible cursor to some spot and then triggers the choice. This returns to the application the values of x and y, along with the trigger value.

Pick. The **pick** input device is used to identify a portion of a picture for further processing. Some graphics packages allow a picture to be defined in terms of **segments**, which are groups of related graphics. The package provides tools to define segments and to give them identifying names. When using `pick`(), the user "points" to a part of a picture with some physical input device, and the package figures out which segment is being pointed to. `pick()` returns the name of the segment to the application, enabling the user to erase, move, or otherwise manipulate the segment.

The graphics workstation is initialized when an application starts running: among other things each logical input function is associated with one of the installed physical devices.

1.5.2. Types of Physical Input Devices.

We look at the other side of input devices: the physical machine that is connected to the personal computer or workstation.

Keyboard. All workstations are equipped with a keyboard, which sends strings of characters to the application upon request. Hence a keyboard is usually used to obtain a `string` device. Some keyboards have cursor keys or function keys, which are often used to produce `choice` input primitives.

Buttons. Sometimes a separate bank of buttons is installed on a workstation. The user presses one of the buttons to perform a `choice` input function.

Mouse. The **mouse** is perhaps the most familiar input device of all, as it is easy and comfortable to operate. As the user slides the mouse over the desktop, the mouse sends the changes in its position to the workstation. Software within the workstation keeps track of the mouse's position and moves a **graphics cursor** — a small dot or cross — on the screen accordingly. The mouse is most often used to perform a `locate` or a `pick` function. There are usually some buttons on the mouse that the user can press to trigger the action.

Tablet. Like a mouse, a tablet is used to generate `locate` or `pick` input primitives. A **tablet** provides an area on which the user can slide a stylus. The tip of the stylus contains a microswitch. By pressing down on the stylus the user can trigger the logical function.

The tablet is particularly handy for digitizing drawings: the user can tape a picture onto the tablet surface and then move the stylus over it, pressing down to send each new point to the workstation. A menu area is sometimes printed on the tablet surface, and the user *Pick*'s a menu item by pressing down the stylus inside one of the menu item boxes. Suitable software associates each menu item box with the desired function for the application that is running.

Space Ball and Data Glove. The Space Ball and Data Glove are relatively new input devices. Both are designed to give a user explicit control over several variables at once, by performing hand and finger motions. Sensors inside each device pick up subtle hand motions and translate them into *Valuator* values that get passed back to the application. They are particularly suited to situations where the hand movements themselves make sense in the context of the program, such as when the user is controlling a virtual robot hand, and watching the effects of such motions simulated on the screen.

(Section 1.6 Summary - deleted.)

1.7. For Further Reading.

A number of books provide a good introduction to the field of computer graphics. Hearn and Baker [hearn94] gives a leisurely and interesting overview of the field with lots of examples. Foley and Van Dam [foley93] and David Rogers[rogers98] give additional technical detail on the many kinds of graphics input and output devices. An excellent series of five books known as “Graphics Gems” [gems], first published in 1990, brought together many new ideas and “gems” from graphics researchers and practitioners around the world.

There are also a number of journals and magazines that give good insight into new techniques in computer graphics. The most accessible is the IEEE Computer Graphics and Applications, which often features survey articles on new areas of effort with graphics. The classic repositories of new results in graphics are the annual Proceedings of SIGGRAPH [SIGGRAPH], and the ACM Transactions on Graphics [TOGS]. Another more recent arrival is the Journal of Graphics Tools [jgt].