



The Graphical Revolution



Readers of the September 10, 1945, issue of *Life* magazine encountered mostly the usual eclectic mix of articles and photographs: stories about the end of the Second World War, an account of dancer Vaslav Nijinsky's life in Vienna, a photo essay on the United Auto Workers. Also included in that issue was something unexpected: a provocative article by Vannevar Bush (1890–1974) about the future of scientific research. Van Bush (as he was called) had already made his mark in the history of computing by designing one of the most significant analog computers—the differential analyzer—between 1927 and 1931 while an engineering professor at MIT. At the time of the *Life* article in 1945, Bush was serving as Director of the Office of Scientific Research and Development, which had been responsible for coordinating U.S. scientific activities during the war, including the Manhattan Project.

Condensed somewhat from its first appearance two months earlier in *The Atlantic Monthly*, Bush's *Life* article "As We May Think" described some hypothetical inventions of the future ostensibly for the scientist and researcher who must deal with an ever-increasing number of technical journals and articles. Bush saw microfilm as the solution and imagined a device he called the *Memex* to store books, articles, records, and pictures inside a desk. The Memex also allowed the user to establish thematic connections among these works, according to the associations normally made by the human mind. He even imagined a new professional group of people who would forge these trails of association through massive bodies of information.

Although articles about the delights of the future have been common throughout the twentieth century, “As We May Think” is different. This isn’t a story about household laborsaving devices or futuristic transportation or robots. This is a story about *information* and how new technology can help us successfully deal with it.

Through the six and a half decades since the first relay calculators were built, computers have become smaller, faster, and cheaper all at the same time. This trend has changed the very nature of computing. As computers get cheaper, each person can have his or her own. As computers get smaller and faster, software can become more sophisticated and the machines can assume more and more work.

One way in which this extra power and speed can be put to good use is in improving the most crucial part of the computer system, which is the *user interface*—the point at which human and computer meet. People and computers are very different animals, and unfortunately it’s easier to persuade people to make adjustments to accommodate the peculiarities of computers than the other way around.

In the early days, digital computers weren’t interactive at all. Some of them were programmed using switches and cables, while others used punched paper tape or film. By the 1950s and 1960s (and even continuing into the 1970s), computers had evolved to the point where *batch processing* was the norm: Programs and data were punched on cards, which were then read into computer memory. The program analyzed the data, drew some conclusions, and printed the results on paper.

The earliest interactive computers used teletypewriters. Setups such as the Dartmouth time-sharing system (dating from the early 1960s) that I described in the preceding chapter supported multiple teletypewriters that could be used at the same time. In such a system, a user types a line at the teletypewriter, and the computer replies with one or more lines in response. The exchange of information between teletypewriter and computer consists entirely of streams of ASCII (or another character set), which are almost entirely character codes with some simple control codes, such as the carriage return and linefeed. The transaction proceeds only in one direction down the roll of paper.

The cathode-ray tube (which became more common during the 1970s) shouldn’t have such restrictions, however. Software can instead treat the entire screen in a more flexible manner—as a two-dimensional platform for information. Yet, possibly in an attempt to keep the display output logic of an operating system generalized, much early software written for small computers continued to treat the CRT as a “glass teletypewriter”—displaying output line by line going down the screen and scrolling the contents of the screen up when the text reached the bottom. All the utilities in CP/M and most utilities in MS-DOS used the video display in a teletypewriter mode. Perhaps the archetypal teletypewriter operating system is UNIX, which still proudly upholds that tradition.

Interestingly enough, the ASCII character set isn't entirely inadequate in dealing with the cathode-ray tube. When ASCII was originally designed, the code *1Bh* was labeled Escape and was specifically intended for handling extensions of the character set. In 1979, the American National Standards Institute (ANSI) published a standard entitled "Additional Controls for Use with American National Standard Code for Information Interchange." The purpose of this standard was "to accommodate the foreseeable needs for input/output control of two-dimensional character-imaging devices, including interactive terminals of both the cathode ray tube and printer types..."

Of course, the Escape code *1Bh* is just 1 byte and can mean only one thing. The Escape code works by prefacing variable-length sequences that perform a variety of functions. For example, the sequence

1Bh 5Bh 32h 4Ah

which is the Escape code followed by the characters *[2J*, is defined to erase the entire screen and move the cursor to the upper left corner. This isn't something that can be done on a teletypewriter. The sequence

1Bh 5Bh 35h 3Bh 32h 39h 48h

which is the Escape code followed by the characters *[5;29H*, moves the cursor to row 5 and column 29.

A combined keyboard and CRT that responds to ASCII codes (and possibly to a collection of Escape sequences) coming from a remote computer is sometimes called a *dumb terminal*. Such terminals are faster than teletypewriters and somewhat more flexible, but they're not quite fast enough for real innovations in the user interface. Such innovations came with small computers in the 1970s that—like the hypothetical computer we built in Chapter 21—included the video display memory as part of the microprocessor's address space.

The first indication that home computers were going to be much different from their larger and more expensive cousins was probably the application VisiCalc. Designed and programmed by Dan Bricklin (born 1951) and Bob Frankston (born 1949) and introduced in 1979 for the Apple II, VisiCalc used the screen to give the user a two-dimensional view of a spreadsheet. Prior to VisiCalc, a spreadsheet (or worksheet) was a piece of paper with rows and columns generally used for doing series of calculations. VisiCalc replaced the paper with the video display, allowing the user to move around the spreadsheet, enter numbers and formulas, and recalculate everything after a change.

What was amazing about VisiCalc is that it was an application that *could not be duplicated on larger computers*. A program such as VisiCalc needs to update the screen very quickly. For this reason, it wrote directly to the random access memory used for the Apple II's video display. This memory is part of the address space of the microprocessor. The interface between a large time-shared computer and a dumb terminal is simply not fast enough to make a spreadsheet program usable.

The faster a computer can respond to the keyboard and alter the video display, the tighter the potential interaction between user and computer. Most

of the software written in the first decade of the IBM Personal Computer (through the 1980s) wrote directly to video display memory. Because IBM set a hardware standard that other computer manufacturers adhered to, software manufacturers could bypass the operating system and use the hardware directly without fear that their programs wouldn't run right (or at all) on some machines. If all the PC clones had different hardware interfaces to their video displays, it would have been too difficult for software manufacturers to accommodate all the different designs.

For the most part, early applications for the IBM PC used only text output and not graphics. The use of text output also helped the applications run as fast as possible. When a video display is designed like the one described in Chapter 21, a program can display a particular character on the screen by simply writing the character's ASCII code into memory. A program using a graphical video display usually needs to write 8 or more bytes into memory to draw the image of the text character.

The move from character displays to graphics was, however, an extremely important step in the evolution of computers. Yet the development of computer hardware and software that work with graphical images rather than just text and numbers evolved very slowly. As early as 1945, John von Neumann envisioned an oscilloscope-like display that could graph pictorial information. But it wasn't until the early 1950s that computer graphics were ready to become a reality when MIT (with help from IBM) set up the Lincoln Laboratory to develop computers for the Air Force's air defense system. This project was known as SAGE (Semi-Automatic Ground Environment) and included graphics display screens to help the operators analyze large amounts of data.

The early video displays used in systems such as SAGE weren't like those we use today on personal computers. Today's common PC displays are known as *raster* displays. Much like a TV, the total image is composed of a series of horizontal raster lines drawn by an electron gun shooting a beam that moves very rapidly back and forth across the screen. The screen can be visualized as a large rectangular array of dots called *pixels* (*picture elements*). Within the computer, a block of memory is devoted to the video display and contains 1 or more bits for each pixel on the screen. The values of these bits determine whether pixels are illuminated and what color they are.

For example, most computer displays nowadays have a resolution of at least 640 pixels horizontally and 480 pixels vertically. The total number of pixels is the product of these two numbers: 307,200. If only 1 bit of memory is devoted to each pixel, each pixel is limited to just two colors, usually black and white. A 0 pixel could be black and a 1 pixel could be white, for example. Such a video display requires 307,200 *bits* of memory, or 38,400 bytes.

Increasing the number of possible colors necessitates more bits per pixel and increases the memory requirements of the display adapter. For example, a byte could be used for each pixel to encode gray shades. In such an arrangement, the byte 00h is black, FFh is white, and the values in between are shades of gray.

Color on a CRT is achieved by means of three electron guns, one for each of the three additive primary colors, red, green, and blue. (You can examine a television or color computer screen with a magnifying glass to convince yourself that this is true. Printing uses a different set of primaries.) The combination of red and green is yellow, the combination of red and blue is magenta, the combination of green and blue is cyan, and the combination of all three primary colors is white.

The simplest type of color graphics display adapter requires 3 bits per pixel. The pixels could be encoded like this with 1 bit per primary color:

Bits	Color
000	Black
001	Blue
010	Green
011	Cyan
100	Red
101	Magenta
110	Yellow
111	White

But such a scheme would be suitable only for simple cartoonlike images. Most real-world colors are combinations of various *levels* of red, green, and blue. If you were willing to devote 2 bytes per pixel, you could allocate 5 bits for each primary color (with 1 bit left over). That gives you 32 levels of red, green, and blue and a total of 32,768 different colors. This scheme is often referred to as *high color* or *thousands of colors*.

The next step is to use 3 bytes per pixel, or 1 byte for each primary. This encoding scheme results in 256 levels of red, green, and blue for a total of 16,777,216 different colors, often referred to as *full color* or *millions of colors*. If the resolution of the video display is 640 pixels horizontally by 480 pixels vertically, the total amount of memory required is 921,600 bytes, or nearly a megabyte.

The number of bits per pixel is sometimes referred to as the *color depth* or *color resolution*. The number of different colors is related to the number of bits per pixel in this way:

$$\text{Number of colors} = 2^{\text{Number of bits per pixel}}$$

A video adapter board has only a certain amount of memory, so it's limited in the combinations of resolutions and color depths that are possible. For example, a video adapter board that has a megabyte of memory can do a 640-by-480 resolution with 3 bytes per pixel. But if you want to use a resolution of 800 by 600, there's not enough memory for 3 bytes per pixel. Instead, you'll need to use 2 bytes per pixel.

Although raster displays seem very natural to us now, in the early days they were not quite practical because they required what was then a great deal of memory. Instead, the SAGE video displays were *vector* displays, more like an oscilloscope than a TV. The electron gun could be electrically

positioned to point to any part of the display and draw lines and curves directly. The persistence of the image on the screen allowed assembling these lines and curves into rudimentary pictures.

The SAGE computers also supported *light pens* that let the operators alter images on the display. Light pens are peculiar devices that look like a stylus with a wire attached to one end. If the proper software is running, the computer can detect where the light pen is pointing on the screen and alter an image in response to the pen's movements.

How does this work? Even technological sophisticates are sometimes puzzled when they first encounter a light pen. The key is that a light pen doesn't *emit* light—it *detects* light. The circuitry that controls the movements of the electron gun in the CRT (regardless of whether a raster or vector display is used) can also determine when the light from the electron gun hits the light pen and hence where the light pen is pointing on the screen.

One of the first people to envision a new era of interactive computing was Ivan Sutherland (born 1938), who in 1963 demonstrated a revolutionary graphics program he had developed for the SAGE computers named Sketchpad. Sketchpad could store image descriptions in memory and display the images on the video display. In addition, you could use the light pen to draw images on the display and change them, and the computer would keep track of it all.

Another early visionary of interactive computing was Douglas Engelbart (born 1925), who read Vannevar Bush's article "As We May Think" when it was published in 1945 and five years later began a lifetime of work developing new ideas in computer interfaces. In the mid-1960s, while at the Sanford Research Institute, Engelbart completely rethought input devices and came up with a five-pronged keyboard for entering commands (which never caught on) and a smaller device with wheels and a button that he called a *mouse*. The mouse is now almost universally accepted for moving a pointer around the screen to select on-screen objects.

Many of the early enthusiasts of interactive graphical computing (although not Engelbart) came together at Xerox, fortunately at a time when raster displays became economically feasible. Xerox had founded the Palo Alto Research Center (PARC) in 1970 in part to help develop products that would allow the company to enter the computer industry. Perhaps the most famous visionary at PARC was Alan Kay (born 1940), who encountered Van Bush's microfilm library (in a short story by Robert Heinlein) when he was 14, and who had already conceived of a portable computer he called the Dynabook.

The first big project at PARC was the Alto, designed and built between 1972 and 1973. By the standards of those years, it was an impressive piece of work. The floor-standing system unit had 16-bit processing, two 3-MB disk drives, 128 KB of memory (expandable to 512 KB), and a mouse with three buttons. Because the Alto preceded the availability of 16-bit single-chip microprocessors, the Alto processor had to be built from about 200 integrated circuits.

The video display was one of the several unusual aspects of the Alto. The screen was approximately the size and shape of a sheet of paper—8 inches wide and 10 inches high. It ran in a raster graphics mode with 606 pixels horizontally by 808 pixels vertically, for a total of 489,648 pixels. One bit of memory was devoted to each pixel, which meant that each pixel could be either black or white. The total amount of memory devoted to the video display was 64 KB, which was part of the address space of the processor.

By writing into this video display memory, software could draw pictures on the screen or display text in different fonts and sizes. By rolling the mouse on the desk, the user of the Alto could position a pointer on the screen and interact with on-screen objects. Rather than treating the video display in the same way as the teletypewriter—linearly echoing user input and writing out program output—the screen became a two-dimensional high-density array of information and a more direct source of user input.

Over the remainder of the 1970s, programs written for the Alto developed some very interesting characteristics. Multiple programs were put into windows and displayed on the same screen simultaneously. The video graphics of the Alto allowed software to go beyond text and truly mirror the user's imagination. Graphical objects (such as buttons and menus and little pictures called *icons*) became part of the user interface. The mouse was used for selecting windows or triggering the graphical objects to perform program functions.

This was software that went beyond the user interface into user intimacy, software that facilitated the extension of the computer into realms beyond those of simple number crunching. This was software that was designed—to quote the title of a legendary paper written by Douglas Engelbart in 1963—“for the Augmentation of Man's Intellect.”

What PARC developed in the Alto was the beginnings of the *graphical user interface*, or GUI (pronounced *gooey*). But Xerox didn't sell the Alto (one would have cost over \$30,000 if they had), and over a decade passed before the ideas in the Alto would be embodied in a successful consumer product.

In 1979, Steve Jobs and a contingent from Apple Computer visited PARC and were quite impressed with what they saw. But it took them over three years to introduce a computer that had a graphical interface. This was the ill-fated Apple Lisa in January 1983. A year later, however, Apple introduced the much more successful Macintosh.

The original Macintosh had a Motorola 68000 microprocessor, 64 KB of ROM, 128 KB of RAM, a 3½-inch diskette drive (storing 400 KB per diskette), a keyboard, a mouse, and a video display capable of displaying 512 pixels horizontally by 342 pixels vertically. (The CRT itself measured only 9 inches diagonally.) That's a total of 175,104 pixels. Each pixel was associated with 1 bit of memory and could be colored either black or white, so about 22 KB were required for the video display RAM.

The hardware of the original Macintosh was elegant but hardly revolutionary. What made the Mac so different from other computers available in 1984 was the Macintosh operating system, generally referred to as the *system software* at the time and later known as the *Mac OS*.

A text-based single-user operating system such as CP/M or MS-DOS isn't very large and doesn't have an extensive application programming interface (API). As I explained in Chapter 22, mostly what's required in these text-based operating systems is a way for applications to use the file system. A graphical operating system such as the Mac OS, however, is much larger and has hundreds of API functions. Each of them is identified by a name that describes what the function does.

While a text-based operating system such as MS-DOS provides a couple of simple API functions to let application programs display text on the screen in a teletypewriter manner, a graphical operating system such as the Mac OS must provide a way for programs to display *graphics* on the screen. In theory, this can be accomplished by implementing a single API function that lets an application set the color of a pixel at a particular horizontal and vertical coordinate. But it turns out that this is inefficient and results in very slow graphics.

It makes more sense for the operating system to provide a complete graphics programming system, which means that the operating system includes API functions to draw lines, rectangles, and ellipses (including circles) as well as text. Lines can be either solid or composed of dashes or dots. Rectangles and ellipses can be filled with various patterns. Text can be displayed in various fonts and sizes and with effects such as boldfacing and underlining. The graphics system is responsible for determining how to render these graphical objects as a collection of dots on the display.

Programs running under a graphical operating system use the same APIs to draw graphics on both the computer's video display and the printer. A word processing application can thus display a document on the screen so that it looks very similar to the document later printed, a feature known as WYSIWYG (pronounced *wizzy wig*). This is an acronym for "What you see is what you get," the contribution to computer lingo of the comedian Flip Wilson in his Geraldine persona.

Part of the appeal of a graphical user interface is that different applications work roughly the same and leverage a user's experience. This means that the operating system must also support API functions that let applications implement the various components of the user interface, such as buttons and menus. Although the GUI is generally viewed as an easy environment for users, it's also just as importantly an environment for programmers. Programmers can implement a modern user interface without re-inventing the wheel.

Even before the introduction of the Macintosh, several companies had begun to create a graphical operating system for the IBM PC and compatibles. In one sense, the Apple developers had an easier job because they were designing the hardware and software together. The Macintosh system software had to support only one type of diskette drive, one type of video display, and two printers. Implementing a graphical operating system for the PC, however, required supporting many different pieces of hardware.

Moreover, although the IBM PC had been introduced just a few years earlier (in 1981), many people had grown accustomed to using their favorite MS-DOS applications and weren't ready to give them up. It was considered very important for a graphical operating system for the PC to run MS-DOS applications as well as applications designed expressly for the new operating system. (The Macintosh didn't run Apple II software primarily because it used a different microprocessor.)

In 1985, Digital Research (the company behind CP/M) introduced GEM (the Graphical Environment Manager), VisiCorp (the company marketing VisiCalc) introduced VisiOn, and Microsoft released Windows version 1.0, which was quickly perceived as being the probable winner in the "windows wars." It wasn't until the May 1990 release of Windows 3.0, however, that Windows began to attract a lot of users. Its popularity has increased since then, and today Windows is the operating system used on about 90 percent of small computers. Despite the similar appearances of the Macintosh and Windows, the APIs for the two systems are very different.

In theory, aside from the graphics display, a graphical operating system doesn't require much more in the way of hardware than a text-based operating system. In theory, not even a hard disk drive is required: The original Macintosh didn't have one, and Windows 1.0 didn't require one. Windows 1.0 didn't even require a mouse, although everyone agreed that it was much easier to use with a mouse.

Still, however, it's not surprising that graphical user interfaces have become more popular as microprocessors have grown faster and as memory and storage have become more plentiful. As more and more features are added to graphical operating systems, they have grown large. Today's graphical operating systems generally require a couple hundred megabytes of hard disk space and upwards of 32 megabytes of memory.

Applications for graphical operating systems are almost never written in assembly language. In the early days, the popular language for Macintosh applications was Pascal. For Windows applications, it was C. But once again, PARC had demonstrated a different approach. Beginning about 1972, the researchers at PARC were developing a language named Smalltalk that embodied the concept of *object-oriented programming*, or OOP (pronounced *oop*).

Traditionally, high-level programming languages differentiate between code (which is statements generally beginning with a keyword such as *set* or *for* or *if*) and data, which is numbers represented by variables. This distinction no doubt originates from the architecture of von Neumann computers, in which something is either machine code or is data acted upon by machine code.

In object-oriented programming, however, an *object* is a combination of code and data. The actual way in which the data in an object is stored is understood only by code associated with the object. Objects communicate with one another by sending and receiving *messages*, which give instructions to an object or ask for information from it.

Object-oriented languages are often helpful for programming applications for graphical operating systems because the programmer can treat objects on the screen (such as windows and buttons) in much the same way that a user perceives them. A button is an example of an object in an object-oriented language. A button has a certain dimension and position on the screen and displays some text or a little picture, all of which is data associated with the object. Code associated with the object determines when the user “presses” the button with the keyboard or the mouse and sends a message indicating the button has been triggered.

The most popular object-oriented languages for small computers, however, are extensions of traditional ALGOL-like languages, such as C and Pascal. The most popular object-oriented extension of C is called C++. (As you might recall, two plus signs in C is an increment operator.) Largely the brainchild of Bjarne Stroustrup (born 1950) of Bell Telephone Laboratories, C++ was implemented first as a translator that converted a program written in C++ to one written in C (although very ugly and virtually unreadable C). The C program could then be compiled normally.

Object-oriented languages can’t do anything more than traditional languages can do, of course. But programming is a problem-solving activity, and object-oriented languages allow the programmer to consider different solutions that are often structurally superior. It’s also possible—although not exactly easy—to write a single program using an object-oriented language that can be compiled to run either on the Macintosh or under Windows. Such a program doesn’t refer to the APIs directly but rather uses objects that in turn call the API functions. Two different object definitions are used to compile the program for the Macintosh or Windows API.

Most programmers working on small computers no longer run a compiler from a command line. Instead, programmers use an *integrated development environment* (IDE), which combines all the tools they need in one convenient program that runs like other graphical applications. Programmers also take advantage of a technique called *visual programming*, in which windows are designed interactively by using the mouse to assemble buttons and other components.

In Chapter 22, I described text files, which are files that contain only ASCII characters and which are readable by human beings like you and me. Back in the days of text-based operating systems, text files were ideal to exchange information among applications. One big advantage of text files is that they’re searchable—that is, a program can look at many text files and determine which of them contains a particular text string. But once you have a facility in the operating system to display text using various fonts and sizes and effects such as italics, boldfacing, and underlining, the text file suddenly seems woefully inadequate. Indeed, most word processing programs save documents in a proprietary binary format. Text files are also not suitable for pictorial information.

But it’s possible to encode information (such as font specifications and paragraph layout) along with text and still have a readable text file. The

key is to choose an escape character to denote this information. In the Rich Text Format (RTF) designed by Microsoft as a means to exchange formatted text among applications, the curly brackets { and } and the backslash character \ are used to enclose information that indicates how the text is to be formatted.

PostScript is a text file format that takes this concept to extremes. Designed by John Warnock (born 1940), cofounder of Adobe Systems, PostScript is an entire general-purpose graphics programming language used today mostly to draw text and graphics on high-end computer printers.

The incorporation of graphical images into the personal computing environment is the direct result of better and cheaper hardware. As microprocessors have become faster, as memory has become cheaper, as video displays and printers have increased in resolution and blossomed in full color, that power has been exploited through computer graphics.

Computer graphics comes in two flavors, which are referred to by the same words I used earlier to differentiate graphical video displays: vector and raster.

Vector graphics involves creating images algorithmically using straight lines, curves, and filled areas. This is the province of the *computer-assisted drawing* (or CAD) program. Vector graphics finds its most important application in engineering and architectural design. A vector graphics image can be stored in a file in a format referred to as a *metafile*. A metafile is simply a collection of vector graphics drawing commands usually encoded in binary form.

The use of lines, curves, and filled areas of vector graphics is entirely appropriate when you're designing a bridge but hopelessly inadequate when you want to show what the actual constructed bridge looks like. That bridge is a real-world image. It's simply too complex to be represented by vector graphics.

Raster graphics (also known as *bitmap graphics*) comes to the rescue. A bitmap encodes an image as a rectangular array of bits that correspond to the pixels of an output device. Just like a video display, a bitmap has a spatial dimension (or resolution), which is the width and height of the image in pixels. Bitmaps also have a color dimension (or color resolution, or color depth), which is the number of bits associated with each pixel. Each pixel in a bitmap has the same number of bits.

Although a bitmap image is two dimensional, the bitmap itself is just a single stream of bytes—usually the top row of pixels, followed by the second row, followed by the third row, and so on.

Some bitmap images are created “manually” by someone using a paint program designed for a graphical operating system. Other bitmap images are created algorithmically by computer code. These days, however, bitmaps are very often used for images from the real world (such as photographs), and there are several different pieces of hardware that allow you to move images from the real world into the computer. These devices generally use something called a *charge-coupled device* (CCD), which is a semiconductor

that releases an electrical charge when exposed to light. One CCD cell is required for each pixel to be sampled.

The *scanner* is the oldest of these devices. Much like a photocopy machine, it uses a row of CCDs that sweep along the surface of a printed image, such as a photograph. The CCDs generate electrical charges based on the intensity of light. Software that works with the scanner translates the image into a bitmap that's stored in a file.

Video camcorders use a two-dimensional array of CCD cells to capture images. Generally these images are recorded on videotape. But the video output might be fed directly into a *video frame grabber*, which is a board that converts an analog video signal to an array of pixel values. These frame grabbers can be used with any common video source, such as that from a VCR or a laser disc player, or even directly from a cable television box.

Most recently, digital cameras have become financially viable for the home user. These often look very much like normal cameras. But instead of film, an array of CCDs is used to capture an image that's stored directly in memory within the camera and later transferred into the computer.

A graphical operating system often supports the storage of bitmaps in files in a particular format. The Macintosh uses the Paint format, the name of which is a reference to the MacPaint program that inaugurated the format. (The Macintosh PICT format that combines bitmaps and vector graphics is actually the preferred format.) In Windows, the native format is referred to as BMP, which is the filename extension used for bitmaps.

Bitmaps can be quite large, and it's beneficial to figure out some way to make them smaller. This effort falls under an area of computer science known as *data compression*.

Suppose we were dealing with an image with 3 bits per pixel such as I described earlier. You have a picture of sky and a house and a lawn. This picture probably has large patches of blue and green. Maybe the very top row of the bitmap has 72 blue pixels in a row. The bitmap file could be made smaller if there were some way to actually encode the number 72 in the file to mean that the blue pixel repeats 72 times. This type of compression is known as *run-length encoding*, or RLE.

The common office fax machine uses RLE compression to reduce the size of an image before sending it over the telephone line. Because a fax interprets an image as black and white with no gray shades or colors, there are generally long stretches of white pixels.

A bitmap file format that's been popular for over a decade is the Graphics Interchange Format, or GIF (pronounced *jif* like the peanut butter), developed by CompuServe in 1987. GIF files use a compression technique called LZW, which stands for its creators, Lempel, Ziv, and Welch. LZW is more powerful than RLE because it detects *patterns* of differently valued pixels rather than just consecutive strings of same-value pixels.

Both RLE and LZW are referred to as *lossless* compression techniques because the original file can be entirely re-created from the compressed data. In other words, the compression is *reversible*. It's fairly easy to prove that

reversible compression doesn't work for every type of file. In some cases, the "compressed" file is actually larger than the original file!

In recent years, *lossy* compression techniques have become popular. A lossy compression isn't reversible because some of the original data is effectively discarded. You wouldn't want to use lossy compression on your spreadsheets or word processing documents. Presumably every number and word is important. But you probably wouldn't mind lossy compression for images, just as long as the data that's discarded doesn't make much of a difference in the overall picture. That's why lossy compression techniques are based on psychovisual research that investigates human vision to determine what's important and what's not.

The most significant lossy compression techniques used for bitmaps are collectively referred to as JPEG (pronounced *jay peg*). JPEG stands for the Joint Photography Experts Group and actually describes several compression techniques, some lossless and some lossy.

It's fairly straightforward to convert a metafile to a bitmap. Because video display memory and bitmaps are conceptually identical, if a program knows how to draw a metafile in video display memory, it knows how to draw a metafile on a bitmap.

But converting a bitmap to a metafile isn't so easy, and for some complex images might well be impossible. One technique related to this job is *optical character recognition*, or OCR. OCR is used when you have a bitmap of some text (from a fax machine, perhaps, or scanned from typed pages) and need to convert it to ASCII character codes. The OCR software needs to analyze the patterns of bits and determine what characters they represent. Due to the algorithmic complexity of this job, OCR software is usually not 100 percent accurate. Even less accurate is software that attempts to convert handwriting to ASCII text.

Bitmaps and metafiles are the digital representations of visual information. Audio information can also be converted to bits and bytes.

Digitized sound made a big consumer splash in 1983 with the compact disc, which became the biggest consumer electronics success story ever. The CD was developed by Philips and Sony to store 74 minutes of digitized sound on one side of a disk 12 centimeters in diameter. The length of 74 minutes was chosen so that Beethoven's Ninth Symphony could fit on one CD.

Sound is encoded on a CD using a technique called *pulse code modulation*, or PCM. Despite the fancy name, PCM is conceptually a fairly simple process.

Sound is vibration. Human vocal cords vibrate, a tuba vibrates, a tree falling in a forest vibrates, and these objects cause air molecules to move. The air alternately pushes and pulls, compresses and thins, back and forth some hundreds of times or thousands of times a second. The air in turn vibrates our eardrums, and we sense sound.

Analogous to these waves of sound are the little hills and valleys in the surface of the tin foil cylinder used to record and play back sound in Thomas Edison's first phonograph in 1877. Until the compact disc, this

technique of recording sound barely changed, although cylinders were replaced by disks, and tin foil by wax and eventually plastic. Early phonographs were entirely mechanical, but eventually electrical amplification was used to strengthen the sound. The variable resistor in a microphone converts sound to electricity, and the electromagnet in a loudspeaker converts electricity back to sound.

An electrical current that represents sound isn't like the on-off digital signals that we've encountered throughout this book. Sound waves vary continuously, and so does the voltage of such a current. The electrical current is an *analog* of the sound waves. A device known as an *analog-to-digital converter* (ADC)—generally implemented in a chip—converts an analog voltage to a binary number. The output of an ADC is a certain number of digital signals—usually 8, 12, or 16—that together indicate the relative level of the voltage. A 12-bit ADC, for example, converts a voltage to a number between 000h and FFFh and can differentiate 4096 different voltage levels.

In the technique known as *pulse code modulation*, the voltage representing a sound wave is converted to digital values at a constant rate. These numbers are stored on the CD in the form of little holes carved into the surface of the disc. They're read with a laser light reflected from the surface of the CD. During playback, the numbers are converted to an electrical current again using a *digital-to-analog converter*, or DAC. (A DAC is also used in color graphics boards to convert a pixel value to analog color signals that go to the monitor.)

The voltage of the sound wave is converted to numbers at a constant rate, known as the *sampling rate*. In 1928, Harry Nyquist of Bell Telephone Laboratories showed that a sampling rate must be at least twice the maximum frequency that needs to be recorded and played back. It's commonly assumed that humans hear sounds ranging from 20 Hz to 20,000 Hz. The sampling frequency used for CDs is a bit more than double that maximum, specifically 44,100 samples per second.

The number of bits per sample determines the dynamic range of the CD, which is the difference between the loudest and the softest sound that can be recorded and played back. This is somewhat complicated: As the electrical current varies back and forth as an analog of the sound waves, the peaks it hits represent the waveform's *amplitude*. What we perceive as the *intensity* of the sound is proportional to twice the amplitude. A *bel* (which is three-quarters of Alexander Graham Bell's last name) is a tenfold increase in intensity; a *decibel* is one-tenth of a bel. One decibel represents approximately the smallest increase in loudness that a person can perceive.

It turns out that the use of 16 bits per sample allows a dynamic range of 96 decibels, which is approximately the difference between the threshold of hearing (below which we can't hear anything) and the threshold of pain. The compact disk uses 16 bits per sample.

So for each second of sound, a compact disk contains 44,100 samples of 2 bytes each. But you probably want stereo as well. So double that for

a total of 176,400 bytes per second. That's 10,584,000 bytes per minute of sound. (Now you know why digital recording of sound wasn't common before the 1980s.) The full 74 minutes of stereo sound on the CD requires 783,216,000 bytes.

Digitized sound has many well-known advantages over analog sound. In particular, whenever analog sound is copied (for example, when a phonograph record is created from a master recording tape) some fidelity is lost. Digitized sound is numbers, however, and numbers can always be faithfully transcribed and copied. It used to be that the longer a telephone signal had to travel in a wire, the worse it would sound. This is no longer the case. Because much of the telephone system is now digital, calls from across the country sound as clear as those from across the street.

CDs can store data as well as sound. When used exclusively for data, they're called CD-ROM (CD Read-Only Memory). A CD-ROM is generally limited to about 660 megabytes. Most computers these days have CD-ROM drives installed, and much application and game software is distributed on CD-ROM.

The introduction of sound, music, and video into the personal computer was known as *multimedia* just a decade ago and is now so common that it doesn't need a special name. Most home computers sold these days have a sound board that includes an ADC for digitally recording sound through a microphone and a DAC for playing back recorded sound through speakers. Sounds can be stored on a disk in *waveform files*.

Because you don't always need CD quality sound when recording and playing back sound on home computers, the Macintosh and Windows offer lower sampling rates, specifically 22,050 Hz, 11,025 Hz, and 8000 Hz; a lower sample size of 8 bits; and monophonic recording. Sound can be recorded using as few as 8000 bytes per second, which is 480,000 bytes per minute.

Everybody knows from science fiction movies and television shows that computers of the future converse with their users in spoken English. Once a computer is equipped with hardware to digitally record and play back sound, everything else involved in this goal is a software problem.

There are a couple of ways that computers can be made to talk in recognizable words and sentences. One approach is to have a human being record sentence fragments, phrases, words, and numbers that can then be stored in files and strung together in different ways. This approach is often used for information systems accessed over the telephone, and it works fine when there are only a limited number of combinations of words and numbers that must be played back.

A more general form of voice synthesis involves a process that converts arbitrary ASCII text to waveform data. Because English spelling, for example, isn't always consistent, such a software system uses a dictionary or complex algorithms to determine the actual pronunciation of words. Basic vocal sounds (called phonemes) are combined to form whole words. Often the software must make other adjustments. For example, if a sentence is followed by a question mark, the sound of the last word must be increased in frequency.

Voice recognition—the conversion of waveform data to ASCII text—is a much more complex problem. Indeed, many humans have problems understanding regional variations in spoken language. While dictation software for the personal computer is available, it usually requires some training so that it can reasonably transcribe what a particular person is saying. Far beyond the conversion to ASCII text is the problem of programming the computer so that it actually “understands” what is said. Such a problem is in the realm of the field of *artificial intelligence*.

The sound boards in today’s computers are also supplied with small electronic music synthesizers that can imitate the sounds of 128 different musical instruments and 47 different percussion instruments. These are referred to as MIDI (pronounced *middy*) synthesizers. MIDI is the Musical Instrument Digital Interface, a specification developed in the early 1980s by a consortium of manufacturers of electronic music synthesizers to connect these electronic instruments to one another and to computers.

Various types of MIDI synthesizers use a variety of methods for synthesizing instrument sounds, some of which are more realistic than others. The overall quality of a particular MIDI synthesizer is quite outside the province of the MIDI specification. All that’s required is that the synthesizer respond to short messages—usually 1, 2, or 3 bytes in length—by playing sounds. MIDI messages mostly indicate what instrument is desired, that a particular note should begin playing, or that a note currently playing should stop playing.

A MIDI file is a collection of MIDI messages with timing information. A MIDI file usually contains an entire musical composition that can be played back on the computer’s MIDI synthesizer. A MIDI file is usually much smaller than a waveform file containing the same music. In terms of relative size, if a waveform file is like a bitmap file, a MIDI file is like a vector graphics metafile. The downside is that the music encoded in a MIDI file could sound great on one MIDI synthesizer and quite horrid on another.

Another feature of multimedia is digitized movies. The apparent motion of movie and television images is achieved by quickly displaying a sequence of individual still images. These individual images are called *frames*. Movies proceed at the rate of 24 frames per second, North American television at 30 frames per second, and television in most other places in the world at 25 frames per second.

A movie file on a computer is simply a series of bitmaps with sound. But without compression, a movie file requires a huge amount of data. For example, consider a movie with each frame the size of a 640-by-480-pixel computer screen with 24-bit color. That’s 921,600 bytes per frame. At 30 frames per second, we’re up to 27,648,000 bytes per second. Keep multiplying and you get 1,658,880,000 bytes per minute, and 199,065,600,000 bytes—just about 200 gigabytes—for a two-hour movie. This is why most movies displayed on the personal computer are short, small, and jumpy.

Just as JPEG compression is used to reduce the amount of data required to store still images, MPEG compression is used for movies. MPEG (pronounced *em peg*) stands for Moving Pictures Expert Group. Compression

techniques for moving images take advantage of the fact that a particular frame usually contains much information that's duplicated from the previous frame.

There are different MPEG standards for different media. MPEG-2 is for high-definition television (HDTV) and for *digital video discs* (DVDs), also called *digital versatile discs*. DVDs are the same size as CDs, but they can be recorded on both sides and in two layers per side. On DVDs, video is compressed by a factor of about 50, so a two-hour movie requires only 4 gigabytes, which can fit on one layer of one side. The use of both layers and both sides increases the capacity of DVDs to about 16 gigabytes, which is about 25 times the capacity of a CD. It's expected that DVD-ROM will eventually replace CD-ROM for the distribution of software.

Are CD-ROM and DVD-ROM the modern day realization of Vannevar Bush's Memex? He originally conceived of Memex as using microfilm, but CD-ROM and DVD-ROM make much more sense for such a device. Electronic media have an advantage over physical media by being easily searchable. Unfortunately, few people have simultaneous access to multiple CD or DVD drives. The closest that we've come to Bush's concept doesn't involve storing all the information you'll need at your desk. It involves *interconnecting* computers to give them the ability to share information and use storage much more efficiently.

The first person to publicly operate a computer from a remote location was George Stibitz, the same man who designed the Bell Labs relay computer in the 1930s. The remote operation of a relay computer occurred at a demonstration at Dartmouth in 1940.

The telephone system is built to transmit sound, not bits, over wires. Sending bits over telephone wires requires that the bits be converted to sound and then back again. A continuous sound wave of a single frequency and a single amplitude (called a *carrier*) doesn't convey any substantial information at all. But change something about that sound wave—in other words, *modulate* that sound wave between two different states—and you can represent 0s and 1s. The conversion between bits and sound occurs in a device called the *modem* (which stands for modulator/demodulator). The modem is a form of *serial* interface because the individual bits in a byte are sent one after another rather than all at once. (Printers are often connected to computers with a parallel interface: Eight wires allow an entire byte to be transmitted at the same time.)

In early modems, a technique called *frequency-shift keying* (FSK) was used. A modem operating at 300 bits per second (for example) might convert a 0 bit to a frequency of 1070 Hz and a 1 bit to a frequency of 1270 Hz. Each byte is prefaced by a start bit and concluded with a stop bit, so each byte requires 10 bits. At 300 bits per second, the transmission speed is only 30 bytes per second. More modern modems use more sophisticated techniques to achieve speeds over 100 times that.

An early home computer enthusiast could set up a computer and a modem as a *bulletin board system* (BBS), to which other computers could call in and *download* files, which means transferring files from a remote computer

to one's own computer. This concept was extended into large information services such as CompuServe. In most cases, communication was entirely in the form of ASCII text.

The Internet is qualitatively different from these early efforts because it's decentralized. The Internet really exists as a collection of protocols for computers to talk to one another. Of major importance is TCP/IP, which consists of the *Transmission Control Protocol* and the *Internet Protocol*. Rather than just sending ASCII characters through the wires, TCP/IP-based transmitters divide larger blocks of data into smaller *packets*, which are sent separately over the transmission line (often a telephone line) and reassembled on the other end.

The popular graphical part of the Internet is the World Wide Web, which makes use of HTTP, the *Hypertext Transfer Protocol*. The actual data viewed on Web pages is defined by a text format called HTML, or *Hypertext Markup Language*. The *hypertext* part of these names is a word used to describe the linking of associated information, much like that proposed by Vannevar Bush for the Memex. An HTML file can contain links to other Web pages that can be easily invoked.

HTML is similar to the Rich Text Format that I described earlier, in that it contains ASCII text with formatting information. HTML also allows referencing pictures in the form of GIF files, PNG (Portable Network Graphics) files, and JFIF (JPEG File Interchange Format) files. Most World Wide Web browsers allow you to look at the HTML files, which is an advantage of their text format. Another advantage of defining HTML as a text file is that it's more easily searchable. Despite its name, HTML isn't really a *programming* language such as we've explored in Chapters 19 and 24. The Web browser reads the HTML file and formats the text and graphics accordingly.

It's sometimes helpful if some special program code runs while you are viewing and working with particular Web pages. Such code can run on either the *server* (which is the computer on which the original Web pages are stored) or the *client*, which is your computer. On the server side, usually all necessary work (such as interpreting online forms that a client fills out) can be handled with Common Gateway Interface (CGI) scripts. On the client side, HTML files can contain a simple programming language known as *JavaScript*. Your Web browser interprets the JavaScript statements just as it interprets HTML text.

Why can't a Web site simply provide an executable program that can run on your computer? Well, for one thing, what is your computer? If it's a Macintosh, it needs an executable that contains PowerPC machine code and uses the Mac OS API. A PC-compatible needs an executable that contains Intel Pentium machine code and probably uses the Windows API. But there are other computers and other graphical operating systems as well. Moreover, you don't want to be indiscriminately downloading executable files. They could originate from an untrustworthy source and might be malicious in some way.

An answer to these problems was provided by Sun Microsystems in the language Java (not to be confused with JavaScript). Java is a full-fledged

object-oriented programming language much like C++. In the preceding chapter, I explained the difference between compiled languages (which result in an executable that contains machine code), and interpreted languages (which don't). Java is somewhere in between. Java programs must be compiled, but the result of the compilation usually isn't machine code. It's instead *Java byte codes*. These are similar in structure to machine code, but they're for an imaginary computer called the *Java virtual machine* (JVM). A computer running the compiled Java program emulates the JVM by interpreting the Java byte codes. The Java program uses whatever graphical operating system is on the machine, thus allowing *platform-independent* programming.

While much of this book has focused on using electricity to send signals and information through a wire, a more efficient medium is light transmitted through optical fiber—thin tubes made of glass or polymer that guide the light around corners. Light passing through such optical fibers can achieve data transmission rates in the gigahertz region—some billion of bits per second.

So it seems that photons, not electrons, will be responsible for delivering much of the information of the future into our homes and offices; they'll be like faster dots and dashes of Morse code and those careful pulses of blinking light we once used to communicate late-night wisdom to our best friend across the way.