atoms, making them jump to higher quantum-energy levels. In returning to their previous quantum levels, these excited electrons give up their extra energy in the form of light, at frequencies (i.e., colors) predicted by quantum theory. Any given phosphor has several different quantum levels to which electrons can be excited, each corresponding to a color associated with the return to an unexcited state. Further, electrons on some levels are less stable and return to the unexcited state more rapidly than others. A phosphor's *fluorescence* is the light emitted as these very unstable electrons lose their excess energy while the phosphor is being struck by electrons. *Phosphorescence* is the light given off by the return of the relatively more stable excited electrons to their unexcited state once the electron beam excitation is removed. With typical phosphors, most of the light emitted is phosphorescence, since the excitation and hence the fluorescence usually last just a fraction of a microsecond. A phosphor's *persistence* is defined as the time from the removal of excitation to the moment when phosphorescence has decayed to 10 percent of the initial light output. The range of persistence of different phosphors can reach many seconds, but for most phosphors used in graphics equipment it is usually 10 to 60 microseconds. This light output decays exponentially with time. Characteristics of phosphors are detailed in [SHER79].

The *refresh rate* of a CRT is the number of times per second the image is redrawn; it is typically 60 per second for raster displays. As the refresh rate decreases, *flicker* develops because the eye can no longer integrate the individual light impulses coming from a pixel. The refresh rate above which a picture stops flickering and fuses into a steady image is called the *critical fusion frequency*, or *CFF*. The process of fusion is familiar to all of us; it occurs whenever we watch television or motion pictures. A flicker-free picture appears constant or steady to the viewer, even though, in fact, any given point is "off" much longer than it is "on."

One determinant of the CFF is the phosphor's persistence: The longer the persistence, the lower the CFF. The relation between fusion frequency and persistence is nonlinear: Doubling persistence does not halve the CFF. As persistence increases into the several-second range, the fusion frequency becomes quite small. At the other extreme, even a phosphor with absolutely no persistence at all can be used, since all the eye really requires is to see some light for a short period of time, repeated at a frequency above the CFF.

Persistence is not the only factor affecting CFF. CFF also increases with image intensity and with ambient room lighting, and varies with different wavelengths of emitted light. Finally, it depends on the observer. Fusion is, after all, a physiological phenomenon, and differences among viewers of up to 20 Hz in CFF have been reported [ROGO83]. Cited fusion frequencies are thus usually averages for a large number of observers. Eliminating flicker for 99 percent of viewers of very high-intensity images (especially prevalent with black-on-white raster displays) requires refresh rates of 80 to 90 Hz.

The *horizontal scan rate* is the number of scan lines per second that the circuitry driving a CRT is able to display. The rate is approximately the product of the refresh rate and the number of scan lines. For a given scan rate, an increase in the refresh rate means a decrease in the number of scan lines.

The *resolution* of a monochromatic CRT is defined just as is resolution for hardcopy devices. Resolution is usually measured with a *shrinking raster*: A known number of equally spaced parallel lines that alternate between black and white are displayed, and the interline spacing is uniformly decreased until the lines just begin to merge together into a uniform

field of gray. This merging happens at about the point where the interline spacing is equal to the diameter at which the spot intensity is 60 percent of the intensity at the center of the spot. The resolution is the distance between the two outermost lines, divided by the number of lines in the raster. There is a clear dependence between spot size and achievable resolution: The larger the spot size, the lower the achievable resolution.

In the shrinking-raster process, the interline spacing is decreased not by modifying the contents of a raster bitmap, but by changing the gain (amount of amplification) of the vertical or horizontal deflection amplifiers, depending on whether the vertical or horizontal resolution is being measured. These amplifiers control how large an area on the screen is covered by the bitmap image. Thus, CRT resolution is (properly) not a function of the bitmap resolution, but may be either higher or lower than that of the bitmap.

Resolution is not a constant. As the number of electrons in the beam increases, resolution tends to decrease, because a bright line is wider than a dim line. This effect is a result of *bloom*, the tendency of a phosphor's excitation to spread somewhat beyond the area being bombarded, and also occurs because the spot size of an intense electron beam is bigger than that of a weak beam. Vertical resolution on a raster monitor is determined primarily by spot size; if the vertical resolution is n lines per inch, the spot size needs to be about $1/n$ inches. Horizontal resolution (in which the line-pairs are vertical) is determined by both spot size and the speed with which the electron beam can be turned on and off as it moves horizontally across the screen. This rate is related to the bandwidth of the display, as discussed in the next paragraph. Research on defining the resolution of a display precisely and on our ability to perceive images is ongoing. The *modulation transfer function*, used extensively in this research, relates a device's input signal to its output signal [SNYD85].

The bandwidth of a monitor has to do with the speed with which the electron gun can be turned on or off. To achieve a horizontal resolution of n pixels per scan line, it must be possible to turn the electron gun on at least $n/2$ times and off another $n/2$ times in one scan line, in order to create alternating on and off lines. Consider a raster scan of 1000 lines by 1000 pixels, displayed at a 60-Hz refresh rate. One pixel is drawn in about 11 nanoseconds [WHIT84], so the period of an on–off cycle is about 22 nanoseconds, which corresponds to a frequency of 45 MHz. This frequency is the minimum bandwidth needed to achieve 1000 lines (500 line-pairs) of resolution, but is not the actual bandwidth because we have ignored the effect of spot size. The nonzero spot size must be compensated fo. with a higher bandwidth which causes the beam to turn on and off more quickly, giving the pixels sharper edges than they would have otherwise. It is not unusual for the actual bandwidth of a 1000 by 1000 monitor to be 100 MHz. The actual relationships among resolution, bandwidth, and spot size are complex, and only recently has progress been made in quantifying them [INFA85].

Color television sets and color raster displays use some form of *shadow-mask CRT*. Here, the inside of the tube's viewing surface is covered with closely spaced groups of red, green, and blue phosphor dots. The dot groups are so small that light emanating from the individual dots is perceived by the viewer as a mixture of the three colors. Thus, a wide range of colors can be produced by each group, depending on how strongly each individual phosphor dot is excited. A *shadow mask*, which is a thin metal plate perforated with many small holes and mounted close to the viewing surface, is carefully aligned so that each of

the three electron beams (one each for red, green, and blue) can hit only one type of phosphor dot. The dots thus can be excited selectively.

Figure 4.14 shows one of the most common types of shadow-mask CRT, a *delta–delta* CRT. The phosphor dots are arranged in a triangular *triad* pattern, as are the three electron guns. The guns are deflected together, and are aimed (converged) at the same point on the viewing surface. The shadow mask has one small hole for each triad. The holes are precisely aligned with respect to both the triads and the electron guns, so that each dot in the triad is exposed to electrons from only one gun. High-precision delta–delta CRTs are particularly difficult to keep in alignment. An alternative arrangement, the *precision in-line delta CRT* shown in Fig. 4.15, is easier to converge and is gaining in popularity for high-precision (1000-scan-lines) monitors. In this case, the three beams simultaneously expose three in-line phosphor dots. However, the in-line arrangement does slightly reduce image sharpness at the edges of the tube. Still in the research laboratory but likely to become commercially viable is the *flat-panel color CRT*, in which the electron beams move parallel to the viewing surface, and are then turned 90° to strike the surface.

The need for the shadow mask and triads imposes a limit on the resolution of color CRTs not present with monochrome CRTs. In very high-resolution tubes, the triads are placed on about 0.21-millimeter centers; those in home television tubes are on about 0.60-millimeter centers (this distance is also called the *pitch* of the tube). Because a finely focused beam cannot be guaranteed to hit exactly in the center of a shadow-mask hole, the beam diameter (the diameter at which the intensity is 50 percent of the maximum) must be about $\frac{7}{4}$ times the pitch. Thus, on a mask with a pitch of 0.25-millimeter (0.01 inches),
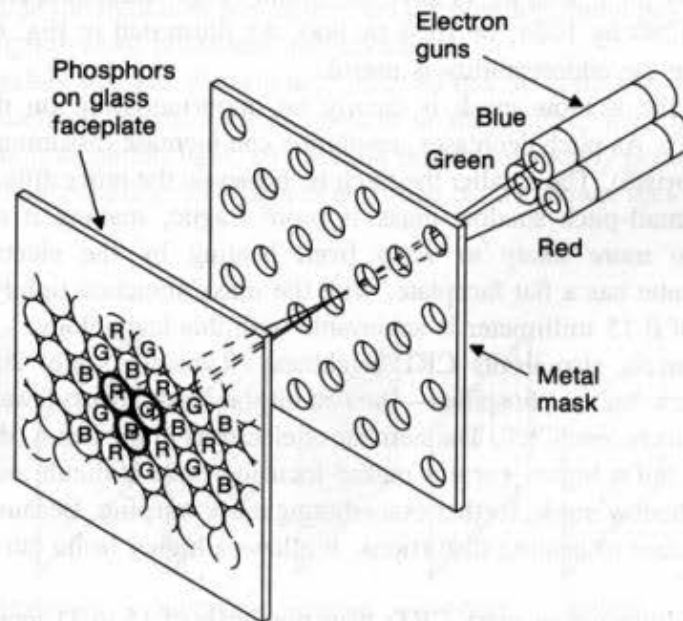

Fig. 4.14 Delta-delta shadow-mask CRT. The three guns and phosphor dots are arranged in a triangular (delta) pattern. The shadow mask allows electrons from each gun to hit only the corresponding phosphor dots.
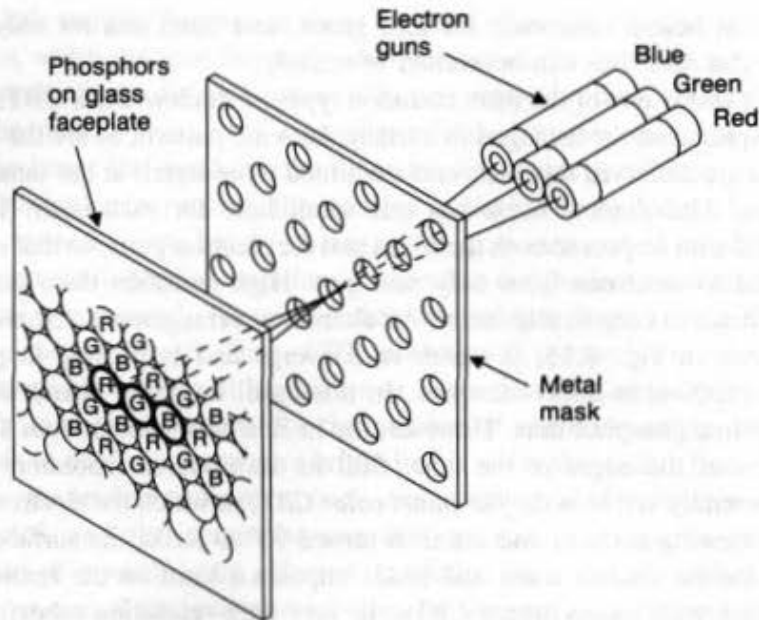
Fig. 4.15 A precision in-line CRT: the three electron guns are in a line.

the beam is about 0.018 inches across, and the resolution can be no more than about $\frac{1}{0.018} = 55$ lines per inch. On a 0.25-millimeter pitch, 19-inch (diagonal measure) monitor, which is about 15.5 inches wide by 11.6 inches high [CONR85], the resolution achievable is thus only $15.5 \times 55 = 850$ by $11.6 \times 55 = 638$. This value compares with a typical addressability of 1280 by 1024, or 1024 by 800. As illustrated in Fig. 4.2, a resolution somewhat less than the addressability is useful.

The pitch of the shadow mask is clearly an important limit on the resolution of shadow-mask CRTs. As pitch decreases, resolution can increase (assuming bandwidth and dot size are appropriate). The smaller the pitch is, however, the more difficult the tube is to manufacture. A small-pitch shadow mask is more fragile, making it more difficult to mount. It is also more likely to warp from heating by the electron beam. The *flat-tension-mask* tube has a flat faceplate, with the mask stretched tightly to maintain its position; a pitch of 0.15 millimeter is achievable with this technology.

The shadow mask also limits CRT brightness. Typically, only 20 percent of the electrons in the beam hit the phosphors—the rest hit the mask. Thus, fewer electrons make light than in a monochrome CRT. The number of electrons in the beam (the *beam current*) can be increased, but a higher current makes focusing more difficult and also generates more heat on the shadow mask, further exacerbating mask warping. Because the flat tension mask is more resistant to heating distortions, it allows a higher beam current and hence a brighter image.

Most high-quality shadow-mask CRTs have diagonals of 15 to 21 inches, with slightly curved faceplates that create optical distortions for the viewer. Several types of flat-face CRTs are becoming available, including a 29-inch–diagonal tube with a pitch of 0.31 millimeter. Of course, the price is high, but it will come down as demand develops.

The *direct-view storage tube* (DVST) is similar to the standard CRT, except that it does not need to be refreshed because the image is stored as a distribution of charges on the inside surface of the screen. Because no refresh is needed, the DVST can display complex images without the high scan rate and bandwidth required by a conventional CRT. The major disadvantage of the DVST is that modifying any part of an image requires redrawing the entire modified image to establish a new charge distribution in the DVST. This redraw can be unacceptably slow (many seconds for a complex image).

The ubiquitous Tektronix 4010 display terminal, based on the DVST, was the first low-cost, widely available interactive graphics terminal. It was the Model T of computer graphics, and was so pervasive that its instruction set became a defacto standard. Even today, many display systems include a Tektronix-compatibility feature, so that buyers can continue to run their (often large) libraries of older software developed for the 4010. Now, however, DVSTs have been superseded by raster displays and have essentially disappeared from the graphics scene.

A *liquid-crystal display* (LCD) is made up of six layers, as shown in Fig. 4.16. The front layer is a vertical polarizer plate. Next is a layer with thin grid wires electrodeposited on the surface adjoining the crystals. Next is a thin (about 0.0005-inch) liquid-crystal layer, then a layer with horizontal grid wires on the surface next to the crystals, then a horizontal polarizer, and finally a reflector.

The liquid-crystal material is made up of long crystalline molecules. The individual molecules normally are arranged in a spiral fashion such that the direction of polarization of polarized light passing through is rotated 90°. Light entering through the front layer is polarized vertically. As the light passes through the liquid crystal, the polarization is rotated 90° to horizontal, so the light now passes through the rear horizontal polarizer, is reflected, and returns through the two polarizers and crystal.

When the crystals are in an electric field, they all line up in the the same direction, and thus have no polarizing effect. Hence, crystals in the electric field do not change the polarization of the transmitted light, so the light remains vertically polarized and does not pass through the rear polarizer: The light is absorbed, so the viewer sees a dark spot on the display.
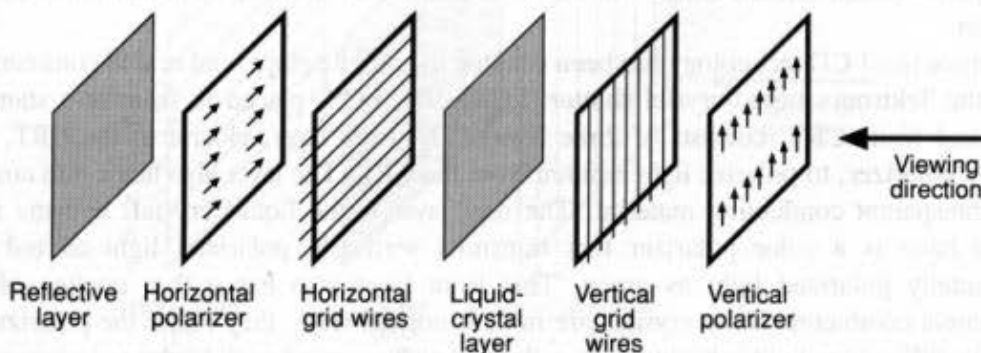

Fig. 4.16 The layers of a liquid-crystal display (LCD), all of which are sandwiched together to form a thin panel.

A dark spot at point (x_1, y_1) is created via *matrix addressing*. The point is selected by applying a negative voltage $-V$ to the horizontal grid wire x_1 and a positive voltage $+V$ to the vertical grid wire y_1: Neither $-V$ nor $+V$ is large enough to cause the crystals to line up, but their difference is large enough to do so. Now the crystals at (x_1, y_1) no longer rotate the direction of polarization of the transmitted light, so it remains vertically polarized and does not pass through the rear polarizer: The light is absorbed, so the viewer sees a dark spot on the display.

To display dots at (x_1, y_1) and (x_2, y_2), we cannot simply apply the positive voltage to x_1 and x_2 and the negative voltage to y_1 and y_2: that would cause dots to appear at (x_1, y_1), (x_1, y_2), (x_2, y_1), and (x_2, y_2), because all these points will be affected by the voltage. Rather, the points must be selected in succession, one after the other, and this selection process must be repeated, to refresh the activation of each point. Of course, if $y_1 = y_2$, then both points on the row can be selected at the same time.

The display is refreshed one row at a time, in raster-scan fashion. Those points in a row that are "on" (i.e., dark in the case of a black-on-white LCD display) are selected only about $1/N$ of the time, where N is the number of rows. Fortunately, once the crystals are lined up, they stay that way for several hundred milliseconds, even when the voltage is withdrawn (the crystals' equivalent of phosphors' persistence). But even so, the crystal is not switched on all the time.

Active matrix panels are LCD panels that have a transistor at each (x, y) grid point. The transistors are used to cause the crystals to change their state quickly, and also to control the degree to which the state has been changed. These two properties allow LCDs to be used in miniature television sets with continuous-tone images. The crystals can also be dyed to provide color. Most important, the transistor can serve as a memory for the state of a cell and can hold the cell in that state until it is changed. That is, the memory provided by the transistor enables a cell to remain on all the time and hence to be brighter than it would be if it had to be refreshed periodically. Color LCD panels with resolutions up to 800 by 1000 on a 14-inch diagonal panel have been built.

Advantages of LCDs are low cost, low weight, small size, and low power consumption. In the past, the major disadvantage was that LCDs were passive, reflecting only incident light and creating no light of their own (although this could be corrected with backlighting): Any glare washed out the image. In recent years, use of active panels has removed this concern.

Nonactive LCD technology has been adapted to color displays, and is sold commercially as the Tektronix liquid-crystal shutter (LCS). The LCS, placed in front of a standard black-and-white CRT, consists of three layers. The back layer, closest to the CRT, is a vertical polarizer, to polarize light emitted from the CRT. The layer also has a thin coating of a transparent conducting material. The next layer is the liquid crystal, and the third (front) layer is a color polarizer that transmits vertically polarized light as red and horizontally polarized light as green. This front layer also has a thin coating of the transparent conductor. If the crystals are in their normal state, they rotate the polarization plane by 90°, so the light is horizontally polarized as it approaches the color polarizer of the third layer, and is seen as green. If the appropriate voltage is applied to the conductive coatings on the front and back layers, then the crystals line up and do not affect the vertical polarization, so the light is seen as red.

The crystals are switched back and forth between their two states at a rate of 60 Hz. At the same time and in synchrony, images to be seen as red and green are alternated on the monochrome display. Mixtures of red and green are created by intensifying the same spot during the red and green phases, potentially with different intensities.

The LCS is an alternative to the shadow-mask CRT, but has limited color resolution. It is possible, however, that this technology can be extended to work with three colors. If it can be, the shadow mask will no longer be a limiting factor in achieving higher-resolution full-color displays. Spot size and bandwidth will be the major determinants, as with monochrome CRTs. Eliminating the shadow mask also will increase ruggedness. Because LCD displays are small and light, they can be used in head-mounted displays such as that discussed in Section 8.1.6.

The *plasma panel* is an array of tiny neon bulbs. Each bulb can be put into an ''on'' (intensified) state or an ''off'' state, and remains in the state until explicitly changed to the other. This memory property means that plasma panels need not be refreshed. Plasma panels typically have 50 to 125 cells per inch, with a 10- to 15-inch diagonal, but 40- by 40-inch panels with 50 cells per inch are sold commercially, and even larger and denser panels can be custom-made.

The neon bulbs are not discrete units, but rather are part of a single integrated panel made of three layers of glass, as seen in Fig. 4.17. The inside surface of the front layer has thin vertical strips of an electrical conductor; the center layer has a number of holes (the bulbs), and the inside surface of the rear layer has thin horizontal strips of an electrical conductor. Matrix addressing is used to turn bulbs on and off. To turn on a bulb, the system adjusts the voltages on the corresponding lines such that their difference is large enough to pull electrons from the neon molecules, thus firing the bulb and making it glow. Once the glow starts, a lower voltage is applied to sustain it. To turn off a bulb, the system momentarily decreases the voltages on the appropriate lines to less than the sustaining
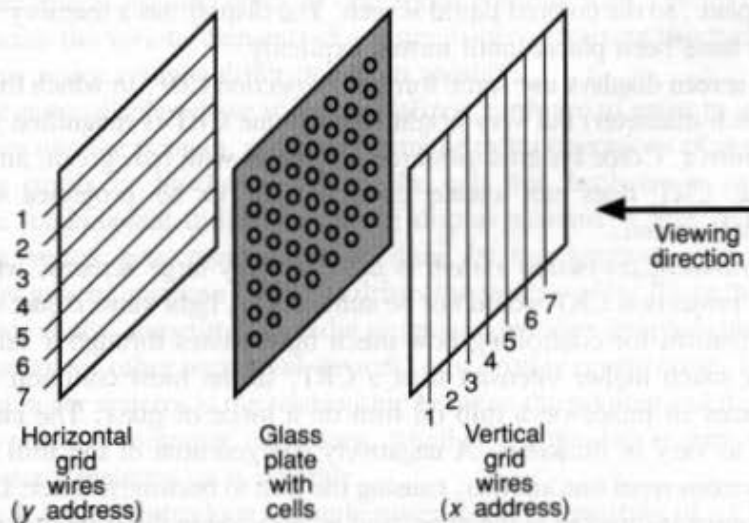

Horizontal grid wires (y address)   Glass plate with cells   Vertical grid wires (x address)

Viewing direction

Fig. 4.17 The layers of a plasma display, all of which are sandwiched together to form a thin panel.

voltage. Bulbs can be turned on or off in about 15 microseconds. In some panel designs, the individual cells are replaced with an open cavity, because the neon glow is contained in a localized area. In this case, the front and back glass layers are separated by spacers. Some plasma panels can also display multiple gray levels.

The plasma panel has the advantages of being flat, transparent, and rugged, and of not needing a bitmap refresh buffer. It can be used with a rear-projection system to mix photographic slides as static background for computer-generated dynamic graphics, but has found most use in military applications, where small size and ruggedness are important. However, its cost, although continually decreasing, is still relatively high for its limited resolution. Color has been demonstrated in the laboratory, but is not commercially available.

Electroluminescent (EL) displays consist of the same gridlike structure as used in LCD and plasma displays. Between the front and back panels is a thin (typically 500-nanometers) layer of an electroluminescent material, such as zinc sulfide doped with manganese, that emits light when in a high electric field (about 10^6 volts per centimeter). A point on the panel is illuminated via the matrix-addressing scheme, several hundred volts being placed across the horizontal and vertical selection lines. Color electroluminescent displays are also available.

These displays are bright and can be switched on and off quickly, and transistors at each pixel can be used to store the image. Typical panel sizes are 6 by 8 inches up to 12 by 16 inches, with 70 addressable dots per inch. These displays' major disadvantage is that their power consumption is higher than that of the LCD panel. However, their brightness has led to their use in some portable computers.

Electrophoretic displays use positively charged colored particles suspended in a solution of a contrasting color, sealed between two parallel, closely spaced plates that have matrix-addressing selection lines. A negative voltage on the front selection line and a positive voltage on the rear selection line pulls the colored particles toward the front plate, where they are seen instead of the colored liquid. Reversing the voltages pulls the particles toward the rear plate, so the colored liquid is seen. The display has a memory: The particles stay where they have been placed until moved explicitly.

Most large-screen displays use some form of *projection CRT*, in which the light from a small (several-inch-diameter) but very bright monochrome CRT is magnified and projected from a curved mirror. Color systems use three projectors with red, green, and blue filters. A shadow-mask CRT does not create enough light to be projected onto a large (2-meter-diagonal) screen.

The *GE light-valve projection system* is used for very large screens, where the light output from the projection CRT would not be sufficient. A light valve is just what its name implies: A mechanism for controlling how much light passes through a valve. The light source can have much higher intensity than a CRT. In the most common approach, an electron gun traces an image on a thin oil film on a piece of glass. The electron charge causes the film to vary in thickness: A negatively charged area of the film is "stretched out," as the electrons repel one another, causing the film to become thinner. Light from the high-intensity source is directed at the glass, and is refracted in different directions because of the variation in the thickness of the oil film. Optics involving Schlieren bars and lenses project light that is refracted in certain directions on the screen, while other light is not

TABLE 4.3 COMPARISON OF DISPLAY TECHNOLOGIES

	CRT	Electro-luminescent	Liquid crystal	Plasma panel
power consumption	fair	fair–good	excellent	fair
screen size	excellent	good	fair	excellent
depth	poor	excellent	excellent	good
weight	poor	excellent	excellent	excellent
ruggedness	fair–good	good–excellent	excellent	excellent
brightness	excellent	excellent	fair–good	excellent
addressability	good–excellent	good	fair–good	good
contrast	good–excellent	good	fair	good
intensity levels per dot	excellent	fair	fair	fair
viewing angle	excellent	good	poor	good–excellent
color capability	excellent	good	good	fair
relative cost range	low	medium–high	low	high

projected. Color is possible with these systems, through use of either three projectors or a more sophisticated set of optics with a single projector. More details are given in [SHER79]. Other similar light-valve systems use LCDs to modulate the light beam.

Table 4.3 summarizes the characteristics of the four major display technologies. The pace of technological innovation is such, however, that some of the relationships may change over the next few years. Also, note that the liquid-crystal comparisons are for passive addressing; with active matrix addressing; gray levels and colors are achievable. More detailed information on these display technologies is given in [APT85; BALD85; CONR85; PERR85; SHER79; and TANN85].

4.3 RASTER-SCAN DISPLAY SYSTEMS

The basic concepts of raster graphics systems were presented in Chapter 1, and Chapter 2 provided further insight into the types of operations possible with a raster display. In this section, we discuss the various elements of a raster display, stressing two fundamental ways in which various raster systems differ one from another.

First, most raster displays have some specialized hardware to assist in scan converting output primitives into the pixmap, and to perform the raster operations of moving, copying, and modifying pixels or blocks of pixels. We call this hardware a *graphics display processor*. The fundamental difference among display systems is how much the display processor does versus how much must be done by the graphics subroutine package executing on the general-purpose CPU that drives the raster display. Note that the graphics display processor is also sometimes called a *graphics controller* (emphasizing its similarity to the control units for other peripheral devices) or a *display coprocessor*. The second key differentiator in raster systems is the relationship between the pixmap and the address space of the general-purpose computer's memory, whether the pixmap is part of the general-purpose computer's memory or is separate.

In Section 4.3.1, we introduce a simple raster display consisting of a CPU containing the pixmap as part of its memory, and a video controller driving a CRT. There is no display processor, so the CPU does both the application and graphics work. In Section 4.3.2, a

graphics processor with a separate pixmap is introduced, and a wide range of graphics-processor functionalities is discussed in Section 4.3.3. Section 4.3.4 discusses ways in which the pixmap can be integrated back into the CPU's address space, given the existence of a graphics processor.

4.3.1   Simple Raster Display System

The simplest and most common raster display system organization is shown in Fig. 4.18. The relation between memory and the CPU is exactly the same as in a nongraphics computer system. However, a portion of the memory also serves as the pixmap. The video controller displays the image defined in the frame buffer, accessing the memory through a separate access port as often as the raster-scan rate dictates. In many systems, a fixed portion of memory is permanently allocated to the frame buffer, whereas some systems have several interchangeable memory areas (sometimes called *pages* in the personal-computer world). Yet other systems can designate (via a register) any part of memory for the frame buffer. In this case, the system may be organized as shown in Fig. 4.19, or the entire system memory may be dual-ported.

The application program and graphics subroutine package share the system memory and are executed by the CPU. The graphics package includes scan-conversion procedures, so that when the application program calls, say, SRGP_lineCoord (*x1*, *y1*, *x2*, *y2*), the graphics package can set the appropriate pixels in the frame buffer (details on scan-conversion procedures were given in Chapter 3). Because the frame buffer is in the address space of the CPU, the graphics package can easily access it to set pixels and to implement the PixBlt instructions described in Chapter 2.

The video controller cycles through the frame buffer, one scan line at a time, typically 60 times per second. Memory reference addresses are generated in synchrony with the raster scan, and the contents of the memory are used to control the CRT beam's intensity or
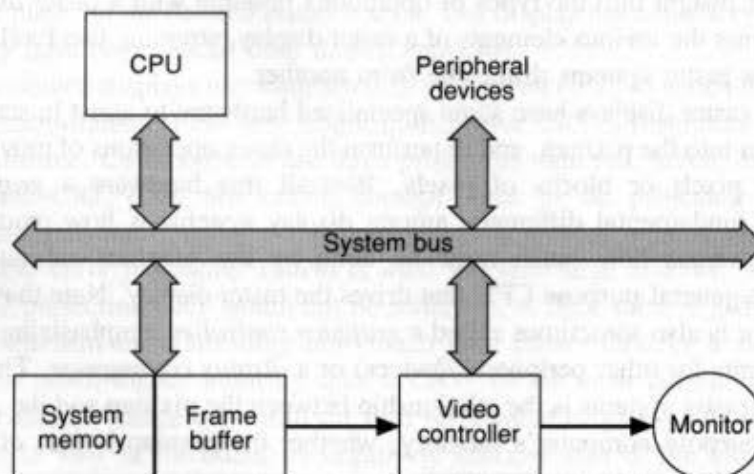

Fig. 4.18 A common raster display system architecture. A dedicated portion of the system memory is dual-ported, so that it can be accessed directly by the video controller, without the system bus being tied up.
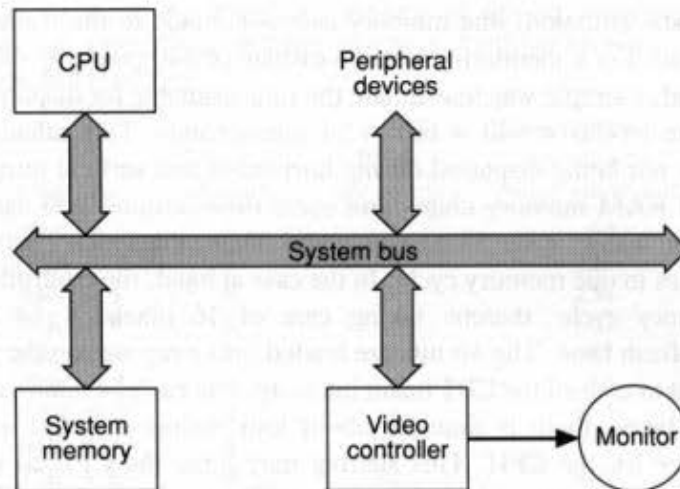
Fig. 4.19 A simple raster display system architecture. Because the frame buffer may be stored anywhere in system memory, the video controller accesses the memory via the system bus.

color. The video controller is organized as shown in Fig. 4.20. The raster-scan generator produces deflection signals that generate the raster scan; it also controls the X and Y address registers, which in turn define the memory location to be accessed next.

Assume that the frame buffer is addressed in x from 0 to x_{max} and in y from 0 to y_{max}; then, at the start of a refresh cycle, the X address register is set to zero and the Y register is set to y_{max} (the top scan line). As the first scan line is generated, the X address is incremented up through x_{max}. Each pixel value is fetched and is used to control the intensity of the CRT beam. After the first scan line, the X address is reset to zero and the Y address is decremented by one. The process continues until the last scan line ($y = 0$) is generated.
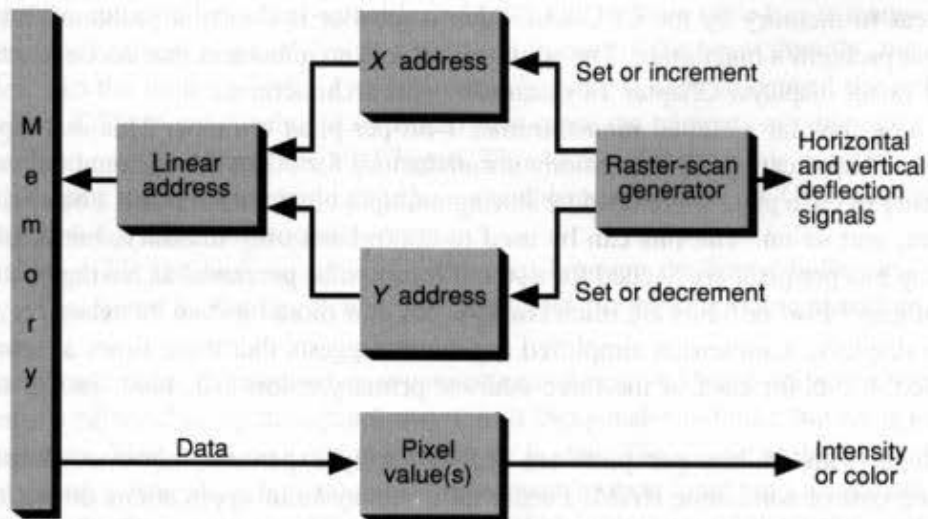
Fig. 4.20 Logical organization of the video controller.

In this simplistic situation, one memory access is made to the frame buffer for each pixel to be displayed. For a medium-resolution display of 640 pixels by 480 lines refreshed 60 times per second, a simple way to estimate the time available for displaying a single 1-bit pixel is to calculate $1/(480 \times 640 \times 60) = 54$ nanoseconds. This calculation ignores the fact that pixels are not being displayed during horizontal and vertical retrace (see Exercise 4.10). But typical RAM memory chips have cycle times around 200 nanoseconds: They cannot support one access each 54 nanoseconds! Thus, the video controller must fetch multiple pixel values in one memory cycle. In the case at hand, the controller might fetch 16 bits in one memory cycle, thereby taking care of 16 pixels $\times$ 54 ns/pixel = 864 nanoseconds of refresh time. The 16 bits are loaded into a register on the video controller, then are shifted out to control the CRT beam intensity, one each 54 nanoseconds. In the 864 nanoseconds this takes, there is time for about four memory cycles: one for the video controller and three for the CPU. This sharing may force the CPU to wait for memory accesses, potentially reducing the speed of the CPU by 25 percent. Of course, cache memory on the CPU chip can be used to ameliorate this problem.

It may not be possible to fetch 16 pixels in one memory cycle. Consider the situation when the pixmap is implemented with five 64-KB–memory chips, with each chip able to deliver 1 bit per cycle (this is called a 64-KB by 1 chip organization), for a total of 5 bits in the 200-nanoseconds cycle time. This is an average of 40 nanoseconds per bit (i.e., per pixel), which is not much faster than the 54 nanoseconds/pixel scan rate and leaves hardly any time for accesses to the memory by the CPU (except during the approximately 7-microsecond inter–scan-line retrace time and 1250-microsecond interframe vertical retrace time). With five 32-KB by 2 chips, however, 10 pixels are delivered in 200 nanoseconds, leaving slightly over half the time available for the CPU. With a 1600 by 1200 display, the pixel time is $1/(1600 \times 1200 \times 60) = 8.7$ nanoseconds. With a 200-nanoseconds memory cycle time, $200/8.7 = 23$ pixels must be fetched each cycle. A 1600 $\times$ 1200 display needs 1.92 MB of memory, which can be provided by eight 256-KB chips. Again, 256-KB by 1 chips can provide only 8 pixels per cycle: on the other hand, 32-KB by 8 chips can deliver 64 pixels, freeing two-thirds of the memory cycles for the CPU.

Access to memory by the CPU and video controller is clearly a problem: Table 4.4 shows that problem's magnitude. The solution is RAM architectures that accommodate the needs of raster displays. Chapter 18 discusses these architectures.

We have thus far assumed monochrome, 1-bit-per-pixel bitmaps. This assumption is fine for some applications, but is grossly unsatisfactory for others. Additional control over the intensity of each pixel is obtained by storing multiple bits for each pixel: 2 bits yield four intensities, and so on. The bits can be used to control not only intensity, but also color. How many bits per pixel are needed for a stored image to be perceived as having continuous shades of gray? Five or 6 bits are often enough, but 8 or more bits can be necessary. Thus, for color displays, a somewhat simplified argument suggests that three times as many bits are needed: 8 bits for each of the three additive primary colors red, blue, and green (see Chapter 13).

Systems with 24 bits per pixel are still relatively expensive, however, despite the decreasing cost of solid-state RAM. Furthermore, many color applications do not require 2^{24} different colors in a single picture (which typically has only 2^{18} to 2^{20} pixels). On the other hand, there is often need for both a small number of colors in a given picture or

TABLE 4.4 PERCENTAGE OF TIME AN IMAGE IS BEING TRACED DURING WHICH THE PROCESSOR CAN ACCESS THE MEMORY CONTAINING THE BITMAP*

Visible area pixels × lines	Chip size	Number of chips	Pixels per access	ns between accesses by video controller	Percent of time for processor accesses
512 × 512	256K × 1	1	1	64	0
512 × 512	128K × 2	1	2	127	0
512 × 512	64K × 4	1	4	254	20
512 × 512	32K × 8	1	8	507	60
512 × 512	16K × 16	1	16	1017	80
1024 × 1024	256K × 1	4	4	64	0
1024 × 1024	128K × 2	4	8	127	0
1024 × 1024	64K × 4	4	16	254	20
1024 × 1024	32K × 8	4	32	407	60
1024 × 1024	16K × 16	4	64	1017	80
1024 × 1024	1M × 1	1	1	16	0
1024 × 1024	64K × 16	1	16	254	21
1024 × 1024	32K × 32	1	32	509	61

*A 200-nanosecond memory cycle time and 60-Hz display rate are assumed throughout. The pixel time for a 512 × 512 display is assumed to be 64 nanoseconds; that for 1024 × 1024, 16 nanoseconds. These times are liberal, since they do not include the horizontal and vertical retrace times; the pixel times are actually about 45 and 11.5 nanoseconds, respectively.

application and the ability to change colors from picture to picture or from application to application. Also, in many image-analysis and image-enhancement applications, it is desirable to change the visual appearance of an image without changing the underlying data defining the image, in order, say, to display all pixels with values below some threshold as black, to expand an intensity range, or to create a pseudocolor display of a monochromatic image.

For these various reasons, the video controller of raster displays often includes a *video look-up table* (also called a *look-up table* or *LUT*). The look-up table has as many entries as there are pixel values. A pixel's value is used not to control the beam directly, but rather as an index into the look-up table. The table entry's value is used to control the intensity or color of the CRT. A pixel value of 67 would thus cause the contents of table entry 67 to be accessed and used to control the CRT beam. This look-up operation is done for each pixel on each display cycle, so the table must be accessible quickly, and the CPU must be able to load the look-up table on program command.

In Fig. 4.21, the look-up table is interposed between the frame buffer and the CRT display. The frame buffer has 8 bits per pixel, and the look-up table therefore has 256 entries.

The simple raster display system organizations of Figs. 4.18 and 4.19 are used in many inexpensive personal computers. Such a system is inexpensive to build, but has a number of disadvantages. First, scan conversion in software is slow. For instance, the (x, y) address of each pixel on a line must be calculated, then must be translated into a memory address consisting of a byte and bit-within-byte pair. Although each of the individual steps is simple, each is repeated many times. Software-based scan conversion slows down the
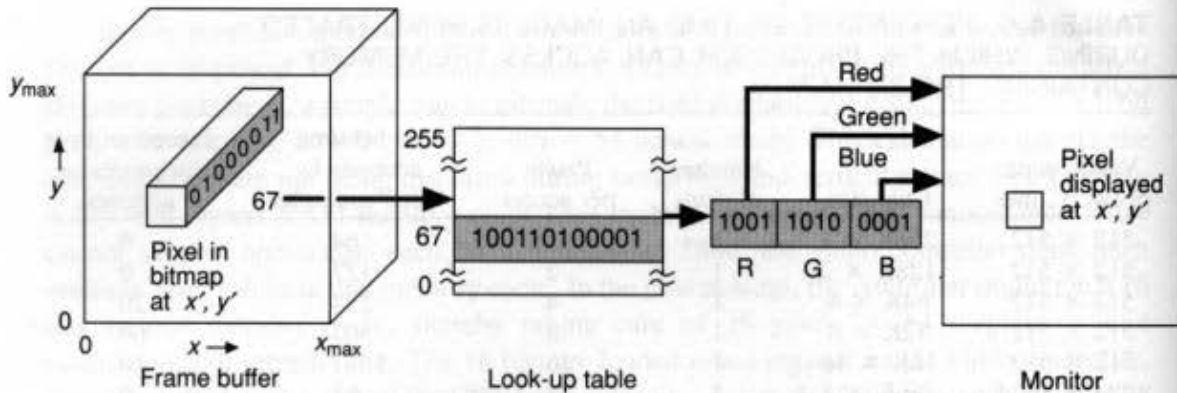
Fig. 4.21 Organization of a video look-up table. A pixel with value 67 (binary 01000011) is displayed on the screen with the red electron gun at $\frac{9}{15}$ of maximum, green at $\frac{10}{15}$, and blue at $\frac{1}{15}$. This look-up table is shown with 12 bits per entry. Up to 24 bits are common.

overall pace of user interaction with the application, potentially creating user dissatisfaction.

The second disadvantage of this architecture is that as the addressability or the refresh rate of the display increases, the number of memory accesses made by the video controller also increases, thus decreasing the number of memory cycles available to the CPU. The CPU is thus slowed down, especially with the architecture in Fig. 4.19. With the dual-porting of part of the system memory shown in Fig. 4.18, the slowdown occurs only when the CPU is accessing the frame buffer, usually for scan conversion or raster operations. These two disadvantages must be weighed against the ease with which the CPU can access the frame buffer and against the architectural simplicity of the system.

4.3.2 Raster Display System with Peripheral Display Processor

The raster display system with a peripheral display processor is a common architecture (see Fig. 4.22) that avoids the disadvantages of the simple raster display by introducing a separate graphics processor to perform graphics functions such as scan conversion and raster operations, and a separate frame buffer for image refresh. We now have two processors: the general-purpose CPU and the special-purpose display processor. We also have three memory areas: the system memory, the display-processor memory, and the frame buffer. The system memory holds data plus those programs that execute on the CPU: the application program, graphics package, and operating system. Similarly, the display-processor memory holds data plus the programs that perform scan conversion and raster operations. The frame buffer contains the displayable image created by the scan-conversion and raster operations.

In simple cases, the display processor can consist of specialized logic to perform the mapping from 2D (x, y) coordinates to a linear memory address. In this case, the scan-conversion and raster operations are still performed by the CPU, so the display-processor memory is not needed; only the frame buffer is present. Most peripheral display processors also perform scan conversion. In this section, we present a prototype system. Its
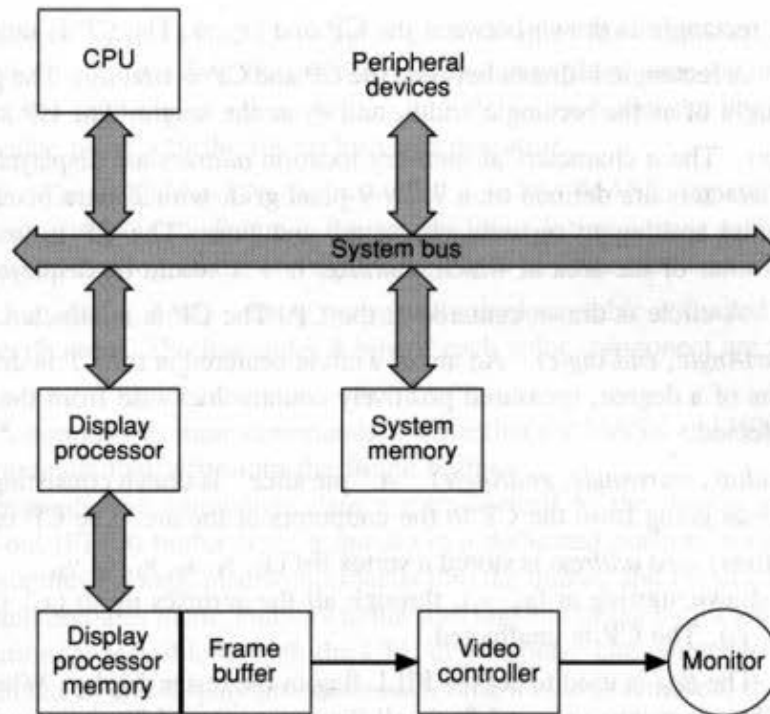
Fig. 4.22 Raster system architecture with a peripheral display processor.

features are a (sometimes simplified) composite of many typical commercially available systems, such as the plug-in graphics cards used with IBM's PC, XT, AT, PS, and compatible computers.

The frame buffer is 1024 by 1024 by 8 bits per pixel, and there is a 256-entry look-up table of 12 bits, 4 each for red, green, and blue. The origin is at lower left, but only the first 768 rows of the pixmap (y in the range of 0 to 767) are displayed. The display has six status registers, which are set by various instructions and affect the execution of other instructions. These are the CP (made up of the X and Y position registers), FILL, INDEX, WMODE, MASK, and PATTERN registers. Their operation is explained next.

The instructions for the simple raster display are as follows:

Move (x, y) The X and Y registers that define the current position (CP) are set to x and y. Because the pixmap is 1024 by 1024, x and y must be between 0 and 1023.

MoveR (dx, dy) The values dx and dy are added to the X and Y registers, thus defining a new CP. The dx and dy values must be between -1024 and $+1023$, and are represented in 2's-complement notation. The addition may cause overflow and hence a wraparound of the X and Y register values.

Line (x, y) A line is drawn from CP to (x, y), and this position becomes the new CP.

LineR (dx, dy) A line is drawn from CP to CP + (dx, dy), and this position becomes the new CP.

Point (x, y) The pixel at (x, y) is set, and this position becomes the new CP.

PointR (dx, dy) The pixel at CP + (dx, dy) is set, and this position becomes the new CP.

Rect (x, y) A rectangle is drawn between the CP and (x, y). The CP is unaffected.

RectR (dx, dy) A rectangle is drawn between the CP and CP + (dx, dy). The parameter dx can be thought of as the rectangle width, and dy as the height. The CP is unaffected.

Text $(n, address)$ The n characters at memory location *address* are displayed, starting at the CP. Characters are defined on a 7- by 9-pixel grid, with 2 extra pixels of vertical and horizontal spacing to separate characters and lines. The CP is updated to the lower-left corner of the area in which character $n + 1$ would be displayed.

Circle $(radius)$ A circle is drawn centered at the CP. The CP is unaffected.

Arc $(radius, startAngle, endAngle)$ An arc of a circle centered at the CP is drawn. Angles are in tenths of a degree, measured positively counterclockwise from the x axis. The CP is unaffected.

CircleSector $(radius, startAngle, endAngle)$ A "pie slice" is drawn consisting of a closed area with lines going from the CP to the endpoints of the arc. The CP is unaffected.

Polygon $(n, address)$ At *address* is stored a vertex list $(x_1, y_1, x_2, y_2, x_3, y_3, \ldots, x_n, y_n)$. A polygon is drawn starting at (x_1, y_1), through all the vertexes up to (x_n, y_n), and then back to (x_1, y_1). The CP is unaffected.

AreaFill $(flag)$ The *flag* is used to set the FILL flag in the raster display. When the flag is set to ON (by a nonzero value of *flag*), all the areas created by the commands Rect, RectR, Circle, CircleSector, Polygon are filled in as they are created, using the pattern defined with the Pattern command.

PixelValue $(index)$ The pixel value *index* is loaded into the INDEX register. Its value is loaded into the pixmap when any of the output primitives in the preceding list are scan-converted.

Pattern $(row1, row2, \ldots, row16)$ The 16 2-byte arguments define the pattern used to fill areas created by Rect, RectR, Circle, CircleSector, and Polygon. The pattern is a 16 by 16 array of bits, stored in the PATTERN register. When a filled area is being created and the FILL register is ON, then, if a bit in the PATTERN register is a 1, the pixel value in the INDEX register is loaded into the pixmap; otherwise, the pixmap is unaffected. A solid fill is created if all the bits of the PATTERN register are ON.

WBlockR $(dx, dy, address)$ The 8-bit pixel values stored starting at *address* in main memory are written into the rectangular region of the pixmap from CP to CP + (dx, dy). Writing begins at the upper-left corner of the region and proceeds row by row, top to bottom.

RBlockR $(dx, dy, address)$ A rectangular region of the pixmap, from CP to CP + (dx, dy), is read into the main memory area beginning at *address*. Reading begins at the upper-left corner of the region and proceeds row by row, top to bottom.

RasterOp $(dx, dy, xdest, ydest)$ A rectangular region of the frame buffer, from CP to CP + (dx, dy), is combined with the region of the same size with lower-left corner at $(xdest, ydest)$, overwriting that region. The combination is controlled by the WMODE register.

WMode $(mode)$ The value of *mode* is loaded into the WMODE register. This register controls how pixels in the frame buffer are combined with any pixel value being written into the frame buffer. There are four *mode* values: **replace**, **xor**, **and**, and **or**. The

modes behave as described in Chapter 2. Note that, for simplicity, the preceding command descriptions have been written as though **replace** were the only write-mode value. In **xor** mode, new values written into the frame buffer are combined with the current value using a bit-by-bit exclusive-or operation.

Mask (*mask*)   The 8-bit *mask* value is loaded into the MASK register. This register controls which bits of the frame buffer are modified during writing to the frame buffer: 1 allows modification of the corresponding bit; 0 inhibits modification.

LuT (*index, red, green, blue*)   Entry *index* in the look-up table is loaded with the given color specification. The low-order 4 bits of each color component are loaded into the table.

Table 4.5 summarizes these commands. Notice that the MASK and WMODE registers affect *all* commands that write into the frame buffer.

The commands and immediate data are transferred to the display processor via a first-in, first-out (FIFO) buffer (i.e., a queue) in a dedicated portion of the CPU address space. The graphics package places commands into the queue, and the display accesses the instructions and executes them. Pointers to the start and end of the buffer are also in specific memory locations, accessible to both the CPU and display. The pointer to the start of the buffer is modified by the display processor each time a byte is removed; the pointer to the end of the buffer is modified by the CPU each time a byte is added. Appropriate testing is done to ensure that an empty buffer is not read and that a full buffer is not written. Direct memory access is used to fetch the addressed data for the instructions.

A queue is more attractive for command passing than is a single instruction register or location accessed by the display. First, the variable length of the instructions favors the queue concept. Second, the CPU can get ahead of the display, and queue up a number of display commands. When the CPU has finished issuing display commands, it can proceed to do other work while the display empties out the queue.

Programming examples.   Programming the display is similar to using the SRGP package of Chapter 2, so only a few examples will be given here. A "Z" means that a hexadecimal value is being specified; an "A" means that the address of the following parenthetical list is used.

A white line on a completely black background is created as follows:

LuT	5, 0, 0, 0	Look-up–table entry 5 is black
LuT	6, Z'F', Z'F', Z'F'	Look-up–table entry 6 is white
WMode	replace	
AreaFill	true	Turn on the FILL flag
Pattern	32Z'FF'	32 bytes of all 1s, for solid pattern
Mask	Z'FF'	Enable writing to all planes
PixelValue	5	Scan convert using pixel value of 5
Move	0, 0	
Rect	1023, 767	Visible part of frame buffer now black
PixelValue	6	Scan convert using pixel value of 6
Move	100, 100	
LineR	500, 400	Draw the line

TABLE 4.5 SUMMARY OF THE RASTER DISPLAY COMMANDS*

Command mnemonic	Arguments and lengths	How CP is affected	Registers that affect command
Move	x(2), y(2)	CP := (x, y)	–
MoveR	dx(2), dy(2)	CP := CP + (dx, dy)	CP
Line	x(2), y(2)	CP := (x, y)	CP, INDEX, WMODE, MASK
LineR	dx(2), dy(2)	CP := CP + (dx, dy)	CP, INDEX, WMODE, MASK
Point	x(2), y(2)	CP := (x, y)	CP, INDEX, WMODE, MASK
PointR	dx(2), dy(2)	CP := CP + (dx, dy)	CP, INDEX, WMODE, MASK
Rect	x(2), y(2)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
RectR	dx(2), dy(2)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
Text	n(1), address(4)	CP := next char pos'n	CP, INDEX, WMODE, MASK
Circle	radius(2)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
Arc	radius(2), startAngle(2), endAngle(2)	–	CP, INDEX, WMODE, MASK
CircleSector	radius(2), startAngle(2), endAngle(2)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
Polygon	n(1), address(4)	–	CP, INDEX, WMODE, MASK, FILL, PATTERN
AreaFill	flag(1)	–	–
PixelValue	index(1)	–	–
Pattern	address(4)	–	–
WBlockR	dx(2), dy(2), address(4)	–	CP, INDEX, WMODE, MASK
RBlockR	dx(2), dy(2), address(4)	–	CP, INDEX, WMODE, MASK
RasterOp	dx(2), dy(2), xdest(2), dest(2)	–	CP, INDEX, WMODE, MASK
Mask	mask(1)	–	–
WMode	mode(1)	–	–
LuT	index(1), red(1), green(1), blue(1)	–	–

*The number in parentheses after each argument is the latter's length in bytes. Also indicated are the effect of the command on the CP, and which registers affect how the command operates.

A red circle overlapping a blue triangle, all on a black background, are created with:

LuT	5, 0, 0, 0	Look-up–table entry 5 is black
LuT	7, Z'F', 0, 0	Look-up–table entry 7 is red
LuT	8, 0, 0, Z'F'	Look-up–table entry 8 is blue
WMode	replace	
AreaFill	Z'FF'	On
Pattern	32Z'FF'	32 bytes of all 1s, for solid pattern
Mask	Z'FF'	Enable writing to all planes
PixelValue	5	Get ready for black rectangle
Move	0, 0	
Rect	1023, 767	Visible part of frame buffer now black
PixelValue	8	Next do blue triangle as a three-vertex polygon
Polygon	3, A(200, 200, 800, 200, 500, 700)	
PixelValue	7	Put red circle on top of triangle
Move	511, 383	Put CP at center of display
Circle	100	Radius-100 circle at CP

4.3.3 Additional Display-Processor Functionality

Our simple display processor performs only some of the graphics-related operations that might be implemented. The temptation faced by the system designer is to offload the main CPU more and more by adding functionality to the display processor, such as by using a local memory to store lists of display instructions, by doing clipping and the window-to-viewport transformation, and perhaps by providing pick-correlation logic and automatic feedback when a segment is picked. Ultimately, the display processor becomes another general-purpose CPU doing general interactive graphics work, and the designer is again tempted to provide special-function hardware to offload the display processor.

This *wheel of reincarnation* was identified by Myer and Sutherland in 1968 [MYER68]. These authors' point was that there is a tradeoff between special-purpose and general-purpose functionality. Special-purpose hardware usually does the job faster than does a general-purpose processor. On the other hand, special-purpose hardware is more expensive and cannot be used for other purposes. This tradeoff is an enduring theme in graphics system design.

If clipping (Chapter 3) is added to the display processor, then output primitives can be specified to the processor in coordinates other than device coordinates. This specification can be done in floating-point coordinates, although some display processors operate on only integers (this is changing rapidly as inexpensive floating-point chips become available). If only integers are used, the coordinates used by the application program must be integer, or the graphics package must map floating-point coordinates into integer coordinates. For this mapping to be possible, the application program must give the graphics package a rectangle guaranteed to enclose the coordinates of all output primitives specified to the package. The rectangle must then be mapped into the maximum integer range, so that everything within the rectangle is in the integer coordinate range.

If the subroutine package is 3D, then the display processor can perform the far more complex 3D geometric transformations and clipping described in Chapters 5 and 6. Also, if the package includes 3D surface primitives, such as polygonal areas, the display processor

can also perform the visible surface-determination and rendering steps discussed in Chapters 15 and 16. Chapter 18 discusses some of the fundamental approaches to organizing general- and special-purpose VLSI chips to perform these steps quickly. Many commercially available displays provide these features.

Another function that is often added to the display processor is *local segment storage*, also called *display list storage*. Display instructions, grouped into named segments and having unclipped integer coordinates, are stored in the display-processor memory, permitting the display processor to operate more autonomously from the CPU.

What exactly can a display processor do with these stored segments? It can transform and redraw them, as in zooming or scrolling. Local dragging of segments into new positions can be provided. Local picking can be implemented by having the display processor compare the cursor position to all the graphics primitives (more efficient ways of doing this are discussed in Chapter 7). Regeneration, required to fill in the holes created when a segment is erased, can also be done from segment storage. Segments can be created, deleted, edited, and made visible or invisible.

Segments can also be copied or referenced, both reducing the amount of information that must be sent from the CPU to the display processor and economizing on storage in the display processor itself. For instance, the display instructions to create a VLSI chip pad configuration to be used many times in a drawing can be sent to the display processor just once and can be stored as a segment. Each occurrence of the pad is then sent as a display instruction referencing the segment. It is possible to build up a complex hierarchical data structure using this capability, and many commercial display processors with local segment memory can copy or reference other segments. When the segments are displayed, a reference to another segment must be preceded by saving the display processor's current state, just as a subroutine call is preceded by saving the CPU's current state. References can be nested, giving rise to a *structured display file* or *hierarchical display list*, as in PHIGS [ANSI88], which is discussed further in Chapter 7. In the GKS graphics package [ENDE87; HOPG86], which uses a linear, unnested display list, an existing segment can be copied into a segment that is being created.

The segment data structures need not be in the display-processor memory (Fig. 4.22): They can be built directly by the graphics package in system memory and accessed by the display processor. This option, of course, requires that the display processor be able to read from system memory, which implies that the display processor must be directly on the system bus—RS-232 and Ethernet-speed connections are not viable.

If all the information being displayed is represented in a segment data structure, then the display processor can also implement window-manager operations such as move, open, close, resize, scroll, push, and pop. When a window is panned, the segments are retraversed with a new viewing transformation in effect. At some point, the wheel of reincarnation will again come into play, but we must also keep in mind the startlingly low cost of special-purpose VLSI chips: By the time this book becomes outdated, window manager chips are likely to cost only a few dollars each. Indeed, the pace of technological innovation is such that the graphics functionality found in display processors will continue its dramatic increase in scope and decrease in cost.

Although this raster display system architecture with its graphics display and separate frame buffer has many advantages over the simple raster display system of Section 4.3.1, it

also has some disadvantages. If the display processor is accessed by the CPU as a peripheral on a direct-memory-access port or on an RS-232 interface, then there is considerable operating-system overhead each time an instruction is passed to it (this is not an issue for a display processor whose instruction register is memory-mapped into the CPU's address space, since then it is easy for the graphics package to set up the registers directly).

There is also a marked partitioning of the memories, as was shown in Fig. 4.22. Building the display list in the display-list memory is slow because of the need to issue a display-processor instruction to add or delete elements. The display list may have to be duplicated in the main processor's memory, because it cannot always be read back. An environment in which the display list is built directly in main memory by the graphics subroutine package can thus be more flexible, faster, and easier to program.

The raster-operation command is a particular difficulty. Conceptually, it should have four potential source–destination pairs: system memory to system memory, system memory to frame buffer, frame buffer to system memory, and frame buffer to frame buffer (here, the frame buffer and display processor memory of Fig. 4.22 are considered identical, since they are in the same address space). In display-processor systems, however, the different source–destination pairs are handled in different ways, and the system-memory-to-system-memory case may not exist. This lack of symmetry complicates the programmer's task and reduces flexibility. For example, if the offscreen portion of the pixmap becomes filled with menus, fonts, and so on, then it is difficult to use main memory as an overflow area. Furthermore, because the use of pixmaps is so pervasive, failure to support raster operations on pixmaps stored in main memory is not really viable.

Another problem is that the output of scan-conversion algorithms *must* go to the frame buffer. This requirement precludes *double-buffering*: scan converting a new image into system memory, then copying it into the pixmap to replace the image currently stored there. In addition, certain window-manager strategies and animation techniques require partially or completely obscured windows to be kept current in offscreen canvases, again requiring scan conversion into system memory (Chapters 10 and 19).

The display processor defined earlier in this section, like many real display processors, moves raster images between the system memory and frame buffer via I/O transfers on the system bus. Unfortunately, this movement can be too slow for real-time operations, such as animation, dragging, and popping up windows and menus: The time taken in the operating system to initiate the transfers and the transfer rate on the bus get in the way. This problem can be partially relieved by increasing the display processor's memory to hold more offscreen pixmaps, but then that memory is not available for other purposes—and there is almost never enough memory anyway!

4.3.4 Raster Display System with Integrated Display Processor

We can ameliorate many of the shortcomings of the peripheral display processor discussed in the previous section by making the frame buffer part of the system memory, thus creating the *single-address-space* (SAS) display system architecture shown in Fig. 4.23. Here the display processor, the CPU, and the video controller are all on the system bus and can thus all access system memory. The origin and, in some cases, the size of the frame buffer are held in registers, making double-buffering a simple matter of reloading the register: The
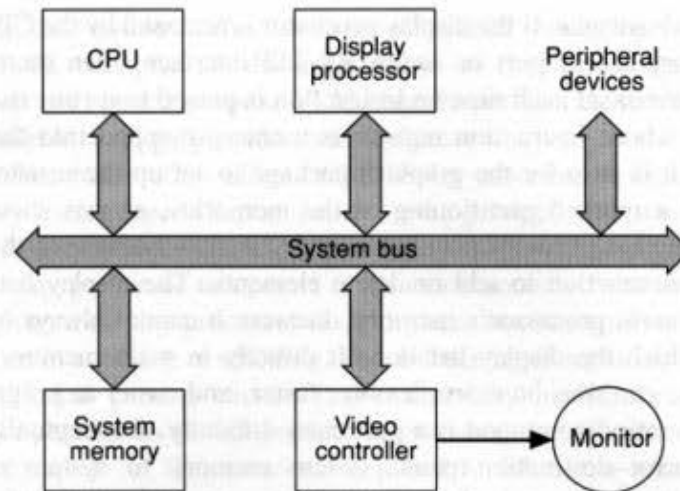
Fig. 4.23 A single-address-space (SAS) raster display system architecture with an integral display processor. The display processor may have a private memory for algorithms and working storage.

results of scan conversion can go either into the frame buffer for immediate display, or elsewhere in system memory for later display. Similarly, the source and destination for raster operations performed by the display processor can be anywhere in system memory (now the only memory of interest to us). This arrangement is also attractive because the CPU can directly manipulate pixels in the frame buffer simply by reading or writing the appropriate bits.

SAS architecture has, however, a number of shortcomings. Contention for access to the system memory is the most serious. We can solve this problem at least partially by dedicating a special portion of system memory to be the frame buffer and by providing a second access port to the frame buffer from the video controller, as shown in Fig. 4.24. Another solution is to use a CPU chip containing instruction- or data-cache memories, thus reducing the CPU's dependence on frequent and rapid access to the system memory. Of course, these and other solutions can be integrated in various ingenious ways, as discussed in more detail in Chapter 18. In the limit, the hardware PixBlt may work on only the frame buffer. What the application programmer sees as a single PixBlt instruction may be treated as several different cases, with software simulation if the source and destination are not supported by the hardware. Some processors are actually fast enough to do this, especially if they have an instruction-cache memory in which the tight inner loop of the software simulation can remain.

As suggested earlier, nontraditional memory-chip organizations for frame buffers also can help to avoid the memory-contention problem. One approach is to turn on all the pixels on a scan line in one access time, thus reducing the number of memory cycles needed to scan convert into memory, especially for filled areas. The video RAM (VRAM) organization, developed by Texas Instruments, can read out all the pixels on a scan line in one cycle, thus reducing the number of memory cycles needed to refresh the display. Again, Chapter 18 gives more detail on these issues.
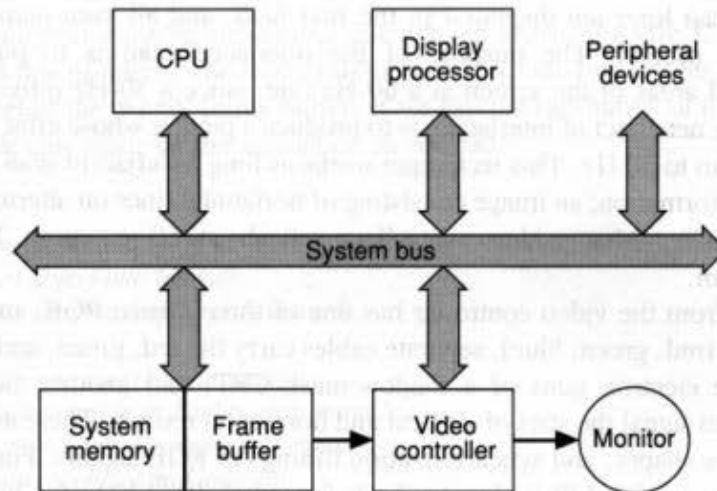
Fig. 4.24 A more common single-address-space raster display system architecture with an integral display processor (compare to Fig. 4.23). The display processor may have a private memory for algorithms and working storage. A dedicated portion of the system memory is dual-ported so that it can be accessed directly by the video controller, without the system bus being tied up.

Another design complication arises if the CPU has a virtual address space, as do the commonly used Motorola 680x0 and Intel 80x86 families, and various reduced-instruction-set-computer (RISC) processors. In this case memory addresses generated by the display processor must go through the same dynamic address translation as other memory addresses. In addition, many CPU architectures distinguish between a kernel operating system virtual address space and an application program virtual address space. It is often desirable for the frame buffer (canvas 0 in SRGP terminology) to be in the kernel space, so that the operating system's display device driver can access it directly. However, the canvases allocated by the application program must be in the application space. Therefore display instructions which access the frame buffer must distinguish between the kernel and application address spaces. If the kernel is to be accessed, then the display instruction must be invoked by a time-consuming operating system service call rather than by a simple subroutine call.

Despite these potential complications, more and more raster display systems do in fact have a single-address-space architecture, typically of the type in Fig. 4.24. The flexibility of allowing both the CPU and display processor to access any part of memory in a uniform and homogeneous way is very compelling, and does simplify programming.

4.4   THE VIDEO CONTROLLER

The most important task for the video controller is the constant refresh of the display. There are two fundamental types of refresh: *interlaced* and *noninterlaced*. The former is used in broadcast television and in raster displays designed to drive regular televisions. The refresh cycle is broken into two fields, each lasting $\frac{1}{60}$ second; thus, a full refresh lasts $\frac{1}{30}$ second. All

odd-numbered scan lines are displayed in the first field, and all even-numbered ones are displayed in the second. The purpose of the interlaced scan is to place some new information in all areas of the screen at a 60-Hz rate, since a 30-Hz refresh rate tends to cause flicker. The net effect of interlacing is to produce a picture whose effective refresh rate is closer to 60 than to 30 Hz. This technique works as long as adjacent scan lines do in fact display similar information; an image consisting of horizontal lines on alternating scan lines would flicker badly. Most video controllers refresh at 60 or more Hz and use a noninterlaced scan.

The output from the video controller has one of three forms: RGB, monochrome, or NTSC. For RGB (red, green, blue), separate cables carry the red, green, and blue signals to control the three electron guns of a shadow-mask CRT, and another cable carries the synchronization to signal the start of vertical and horizontal retrace. There are standards for the voltages, wave shapes, and synchronization timings of RGB signals. For 480–scan-line monochrome signals, RS-170 is the standard; for color, RS-170A; for higher-resolution monochrome signals, RS-343. Frequently, the synchronization timings are included on the same cable as the green signal, in which case the signals are called *composite video*. Monochrome signals use the same standards but have only intensity and synchronization cables, or merely a single cable carrying composite intensity and synchronization.

NTSC (National Television System Committee) video is the signal format used in North American commercial television. Color, intensity, and synchronization information is combined into a signal with a bandwidth of about 5 MHz, broadcast as 525 scan lines, in two fields of 262.5 lines each. Just 480 lines are visible; the rest occur during the vertical retrace periods at the end of each field. A monochrome television set uses the intensity and synchronization information; a color television set also uses the color information to control the three color guns. The bandwidth limit allows many different television channels to broadcast over the frequency range allocated to television. Unfortunately, this bandwidth limits picture quality to an effective resolution of about 350 by 350. Nevertheless, NTSC is the standard for videotape-recording equipment. Matters may improve, however, with increasing interest in 1000-line high-definition television (HDTV) for videotaping and satellite broadcasting. European and Soviet television broadcast and videotape standards are two 625–scan-line, 50-Hz standards, SECAM and PAL.

Some video controllers superimpose a programmable cursor, stored in a 16 by 16 or 32 by 32 pixmap, on top of the frame buffer. This avoids the need to PixBlt the cursor shape into the frame buffer each refresh cycle, slightly reducing CPU overhead. Similarly, some video controllers superimpose multiple small, fixed-size pixmaps (called *sprites*) on top of the frame buffer. This feature is used often in video games.

4.4.1 Animation with the Lookup Table

Raster images can be animated in several ways. To show a rotating object, we can scan convert into the pixmap successive views of the object from slightly different locations, one after the other. This scan conversion must be done at least 10 (preferably 15 to 20) times per second to give a reasonably smooth effect, and hence a new image must be created in no more than 100 milliseconds. However, if scan-converting the object takes much of this 100 milliseconds—say 75 milliseconds—then the complete object can be displayed for only 25

milliseconds before it must be erased and redrawn—a distracting effect. Double-buffering is used to avoid this problem. The frame buffer is divided into two images, each with half of the bits per pixel of the overall frame buffer. If we call the two halves of the pixmap *image0* and *image1*, we can describe the animation as follows:

> *Load look-up table to display all pixel value as background color;*
> *Scan convert object into image0;*
> *Load look-up table to display only image0;*
> **do** {
>   *Scan convert object into image1;*
>   *Load look-up table to display only image1;*
>   *Rotate object data structure description;*
>   *Scan convert object into image0;*
>   *Load look-up table to display only image0;*
>   *Rotate object data structure description;*
> } **while** (*not terminated*);

Of course, if rotating and scan converting the object takes more than 100 milliseconds, the animation is quite jerky, but the transition from one image to the next appears to be instantaneous, as loading the look-up table typically takes less than 1 millisecond.

Another form of look-up–table animation displays a short, repetitive sequence of images [SHOU79]. Suppose we want to display a bouncing ball. Figure 4.25 shows how the frame buffer is loaded; the numbers indicate the pixel values placed in each region of the frame buffer. Figure 4.26 shows how to load the look-up table at each step to display all but one of the balls at background color 0. By cycling the contents of the look-up table, we can achieve motion effects. Not only can balls be bounced, but the effect of moving lights on movie marquees can be simulated, as can the motion of fluids through pipes and the rotation of wheels. This is discussed further in Section 21.1.4.

For more complicated cyclic animation, such as rotating a complex wire-frame object, it may be impossible to keep the separate images from overlapping in the frame buffer. In that case, some of the displayed images will have "holes" in them. A few such holes are not especially distracting, especially if realism is not essential. As more holes appear, however, the animation loses its effectiveness and double-buffering becomes more attractive.
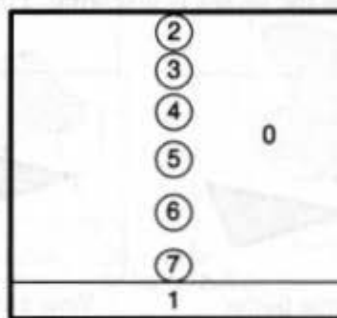

Fig. 4.25 Contents of the frame buffer for a bouncing-ball animation.

Entry number	Colors loaded in table at each step in animation										
	1	2	3	4	5	6	7	8	9	10	11
0	white	white	white	white	white	white	white	white	white	white	white
1	black	black	black	black	black	black	black	black	black	black	black
2	red										red
3		red								red	
4			red						red		
5				red				red			
6					red		red				
7						red					

Fig. 4.26 Look-up table to bounce a red ball on a black surface against a white background. Unspecified entries are all white.

4.4.2　Bitmap Transformations and Windowing

With some video controllers, the pixmap is decoupled from the view surface; that is, the direct, fixed correspondence between positions in the frame buffer and positions on the display is removed. An *image transformation* then defines the correspondence. The image transformation transforms from the frame buffer to the view surface. The transformation can, in general, include a translation, scaling, rotation, and an additional clipping operation.

Figure 4.27 shows the type of transformation found on some raster displays. Part of the frame buffer, defined by a rectangular clip region, is enlarged to fill the entire view surface. The ratio between the size of the window and the view surface must be an integer (3, in the figure). Pixels in the frame buffer outside the window are not used, and none of the pixel values are modified; the transformations are applied by the video controller at the refresh rate.

The image transformation can be changed many times per second so as to give the real-time–dynamics effect of scrolling over or zooming into an image. Also, a sequence of arbitrary images can be shown rapidly by loading each image into a different area of the frame buffer. The image transformation is changed periodically to display a full-screen view first of one area, then of the next, and so on.

The scaling needed to enlarge an image is trivially accomplished by repeating pixel values from within the window as the image is displayed. For a scale factor of 2, each pixel
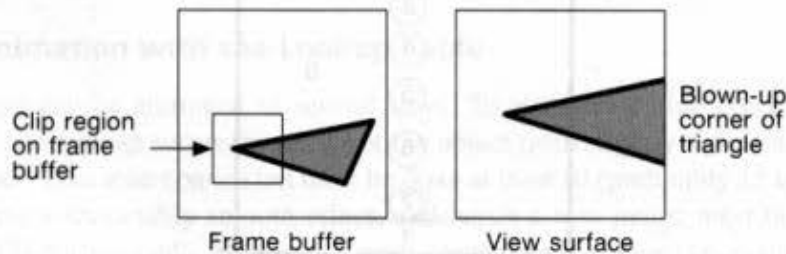

Clip region on frame buffer

Frame buffer

Blown-up corner of triangle

View surface

Fig. 4.27 Portion of the frame buffer enlarged on the view surface.

Before scaling, as defined
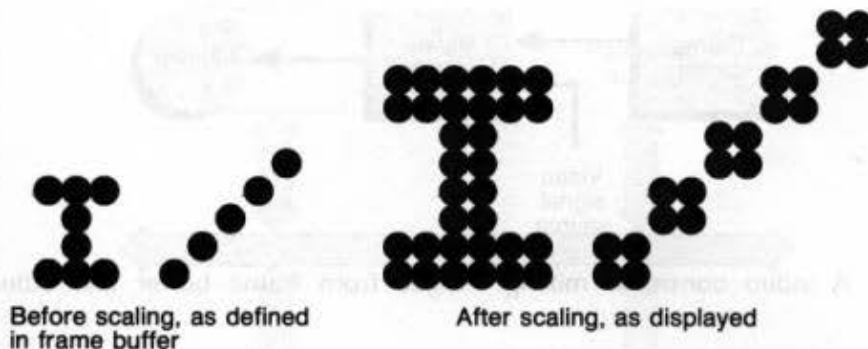in frame buffer

After scaling, as displayed

Fig. 4.28 Effect of scaling up a pixmap by a factor of 2.

value is used four times, twice on each of two successive scan lines. Figure 4.28 shows the
effect of scaling up a letter and adjoining line by a factor of 2. Unless the image's storage
has higher resolution than does its display, scaling up does not reveal more detail: the image
is enlarged but has a more jagged appearance. Thus, this animation effect sacrifices spatial
resolution but maintains a full range of colors, whereas the double-buffering described in
the previous section maintains spatial resolution but decreases the number of colors
available in any one image.

In a more general application of image transformations, the scaled image covers only
the part of the view surface defined by a viewport, as in Fig. 4.29. Now we must define to
the system what is to appear on the view surface outside of the viewport. One possibility is
to display some constant color or intensity; another, shown in the figure, is to display the
frame buffer itself. The hardware implementation for the latter option is simple. Registers
containing the viewport boundary coordinates are compared to the X, Y registers defining
the raster scan's current position. If the beam is in the viewport, pixels are fetched from
within the window area of the frame buffer and are replicated as required. Otherwise, pixels
are fetched from the position in the frame buffer with the same (x, y) coordinates as the
beam.

Frame buffer   Clip region   Monitor
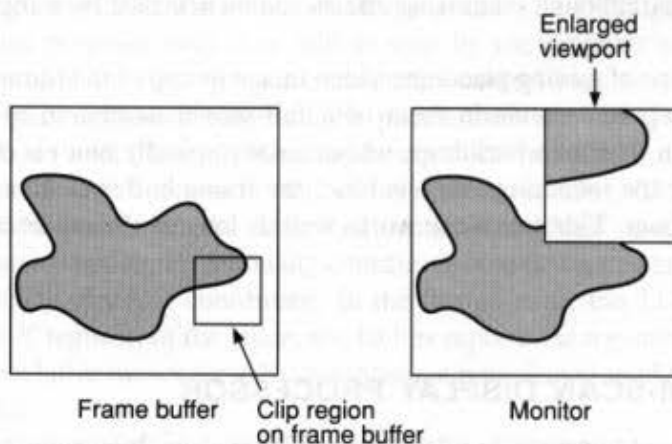   on frame buffer

Fig. 4.29 A portion of the frame buffer, specified by the clip region on the frame
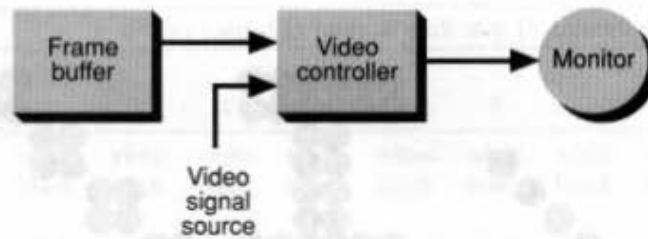buffer, scaled up by a factor of 2 and overlaid on the unscaled frame buffer.

Fig. 4.30 A video controller mixing images from frame buffer and video-signal source.

There are VLSI chips that implement many of these image display functions. These various transformations are special cases of general window-manager operations that might be desired. Window-manager chips are available for use in the video controller [SHIR86]. Each window has a separate pixmap in the system memory, and a data structure in the memory defines window sizes and origins. As a scan line is displayed, the chip knows which window is visible and hence knows from which pixmap to fetch pixels. These and other more advanced hardware issues are discussed in Chapter 18.

4.4.3 Video Mixing

Another useful video-controller function is video mixing. Two images, one defined in the frame buffer and the other defined by a video signal coming from a television camera, recorder, or other source, can be merged to form a composite image. Examples of this merging are seen regularly on television news, sports, and weather shows. Figure 4.30 shows the generic system organization.

There are two types of mixing. In one, a graphics image is set into a video image. The chart or graph displayed over the shoulder of a newscaster is typical of this style. The mixing is accomplished with hardware that treats a designated pixel value in the frame buffer as a flag to indicate that the video signal should be shown instead of the signal from the frame buffer. Normally, the designated pixel value corresponds to the background color of the frame-buffer image, although interesting effects can be achieved by using some other pixel value instead.

The second type of mixing places the video image on top of the frame-buffer image, as when a weather reporter stands in front of a full-screen weather map. The reporter is actually standing in front of a backdrop, whose color (typically blue) is used to control the mixing: Whenever the incoming video is blue, the frame buffer is shown; otherwise, the video image is shown. This technique works well as long as the reporter is not wearing a blue tie or shirt!

4.5   RANDOM-SCAN DISPLAY PROCESSOR

Figure 4.31 shows a typical random (vector) display system. It is generically similar to the display-processor–based raster system architecture discussed in Section 4.3.2. There is of course no pixmap for refreshing the display, and the display processor has no local memory
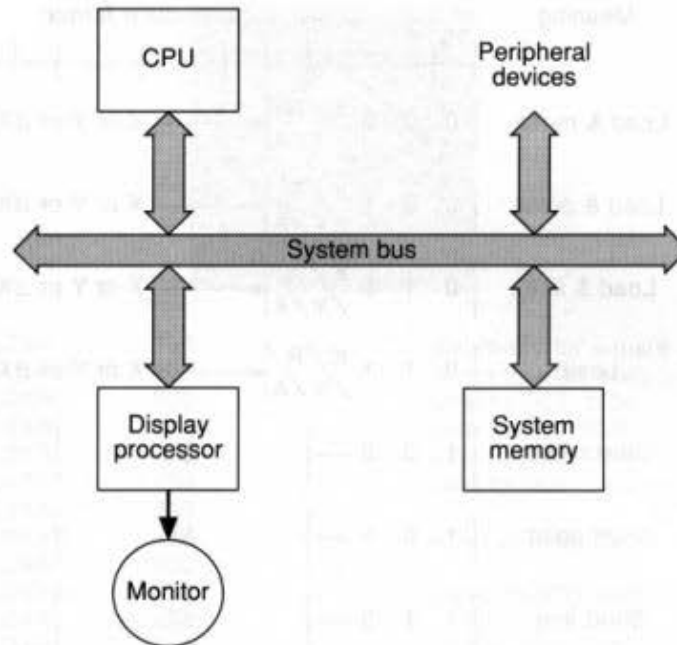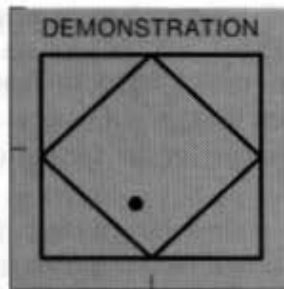
Fig. 4.31 Architecture of a random display system.

for scan-conversion algorithms, since that functionality is typically implemented using programmed logic arrays or microcode.

The random-scan graphics-display processor is often called a *display processing unit* (DPU), or a *graphics controller*. The DPU has an instruction set and instruction address register, and goes through the classic instruction fetch, decode, execute cycle found in any computer. Because there is no pixmap, the display processor must execute its program 30 to 60 times per second in order to provide a flicker-free display. The program executed by the DPU is in main memory, which is shared by the general CPU and the DPU.

The application program and graphics subroutine package also reside in main memory and execute on the general CPU. The graphics package creates a display program of DPU instructions and tells the DPU where to start the program. The DPU then asynchronously executes the display program until it is told to stop by the graphics package. A JUMP instruction at the end of the display program transfers control back to its start, so that the display continues to be refreshed without CPU intervention.

Figure 4.32 shows a set of instructions and mnemonics for a simple random-scan DPU. The processor has X and Y registers and an instruction counter. The instructions are defined for a 16-bit word length. The R/A (relative/absolute) modifier on the LD instructions indicates whether the following coordinate is to be treated as an 11-bit relative coordinate or a 10-bit absolute coordinate. In the former case, the 11 bits are added to either the X or the Y register; in the latter, the 10 bits replace the register contents. (Eleven bits are needed for relative moves for a 2's-complement representation of values in the range -1024 to $+1023$.)

The same R/A interpretation is used for the JUMP instruction, except the instruction counter is modified, thereby effecting a change in the flow of control. The SM, SP, and SL instructions afford compact representation of contour lines, scatter plots, and the like.

Mnemonic	Meaning	Instruction format
LD {X/Y} {R/A} M	Load & move	`0 0 0 X/Y R/A ←——— X or Y or ΔX or ΔY ———→`
LD {X/Y} {R/A} P	Load & point	`0 0 1 X/Y R/A ←——— X or Y or ΔX or ΔY ———→`
LD {X/Y} {R/A} L	Load & line	`0 1 0 X/Y R/A ←——— X or Y or ΔX or ΔY ———→`
LD {X/Y} {R/A}	Load	`0 1 1 X/Y R/A ←——— X or Y or ΔX or ΔY ———→`
SM	Short move	`1 0 0 —  ΔX   ΔY`
SP	Short point	`1 0 1 —  ΔX   ΔY`
SL	Short line	`1 1 0 —  ΔX   ΔY`
CHAR	Characters	`1 1 1 0 ———  Char 1`
		Char 2 Char 3
		etc.
		to terminate code
JUMP {R/A} LI	Jump	`1 1 1 1 R/A L I ——————`
		Absolute or relative address

Key to notation

X/Y: 0 ⇒ Load X, 1 ⇒ Load Y
R/A: 0 ⇒ 11 bits of ΔX or ΔY, 1 ⇒ 10 bits of X or Y
{ }: Choose one of, for use in mnemonic code
L: Frame lock bit, 1 ⇒ delay jump until next clock tick
I: Interrupt bit, 1 ⇒ interrupt CPU

Fig. 4.32 Instruction set for the random-scan display system.

Figure 4.33 is a simple DPU program, written in assembler style, that uses many of these instructions. Notice how the square and diamond are drawn: The first move instruction is absolute, but the rest are relative, so as to facilitate dragging the objects around the screen. If relative instructions were not used, then dragging would require modifying the coordinates of all the instructions used to display the object. The final instruction jumps back to the start of the DPU program. Because the frame lock bit is set, as indicated by the L modifier, execution of the jump is delayed until the next tick of a 30-Hz clock in order to allow the DPU to refresh at 30 Hz but to prevent more frequent refreshing of small programs, which could burn the phosphor.

Notice that, with this instruction set, only one load command (mnemonic LD) is needed to draw or move either horizontally or vertically, because the other coordinate is held constant; oblique movements, however, require two load commands. The two loads can be in either x-then-y or y-then-x order, with the second of the two always specifying a

SQUARE:	LDXA	100	Get ready for square
	LDYAM	100	Move to (100, 100)
	LDXRL	800	Line to (900, 100)
	LDYRL	700	Line to (900, 800)
	LDXRL	−800	Line to (100, 800)
	LDYRL	−700	Line to (100, 100), the starting point
POINT:	LDXA	300	for square
	LDYAP	450	Point at (300, 450)
DIAMOND:	LDXA	100	
	LDYAM	450	Move to (100, 450)
	LDXR	400	
	LDYRL	−350	Line to (500, 100)
	LDXR	400	
	LDYRL	350	Line to (900, 450)
	LDYR	350	
	LDXRL	−400	Line to (500, 800)
	LDXR	−400	
	LDYRL	−350	Line to (100, 450), the starting point
TEXT:	LDXA	200	for diamond
	LDYAM	900	Move to (200, 900) for text
	CHAR	'DEMONSTRATION *t*'	*t* is terminate code
	JUMPRL	SQUARE	Regenerate picture, frame lock

Fig. 4.33 Program for the random-scan display processor.

move, draw-point, or draw-line operation. For the character command (mnemonic CHAR), a character string follows, ended by a nondisplayable termination code.

There are two main differences between these DPU instructions and the instruction set for a general-purpose computer. With the exception of the JUMP command, all the instructions here are rather special-purpose. A register can be loaded and added to, but the result cannot be stored; the register controls only the position of the CRT beam. The second difference, again excluding JUMP, is that all data are immediate; that is, they are part of the instruction. LDXA 100 means ''load the data value 100 into the X register,'' not ''load the contents of address 100 into the X register,'' as in a computer instruction. This restriction is removed in some of the more advanced DPUs described in Chapter 18.

There are only a few differences between the instruction sets for typical random and raster display processors. The random processor lacks area-filling, bit-manipulation, and look-up table commands. But because of the instruction counter, the random processor does have transfer of control commands. Random displays can work at higher resolutions than can raster displays and can draw smooth lines lacking the jagged edges found on raster displays. The fastest random displays can draw about 100,000 short vectors in a refresh cycle, allowing real-time animation of extremely complex shapes.

4.6 INPUT DEVICES FOR OPERATOR INTERACTION

In this section, we describe the workings of the most common input devices. We present a brief and high-level discussion of how the types of devices available work. In Chapter 8, we discuss the advantages and disadvantages of the various devices, and also describe some more advanced devices.

Our presentation is organized around the concept of *logical devices*, introduced in Chapter 2 as part of SRGP and discussed further in Chapters 7 and 8. There are five basic logical devices: the *locator*, to indicate a position or orientation; the *pick*, to select a displayed entity; the *valuator*, to input a single real number; the *keyboard*, to input a character string; and the *choice*, to select from a set of possible actions or choices. The logical-device concept defines equivalence classes of devices on the basis of the type of information the devices provide to the application program.

4.6.1 Locator Devices

Tablet. A *tablet* (or *data tablet*) is a flat surface, ranging in size from about 6 by 6 inches up to 48 by 72 inches or more, which can detect the position of a movable stylus or puck held in the user's hand. Figure 4.34 shows a small tablet with both a stylus and puck (hereafter, we generally refer only to a stylus, although the discussion is relevant to either). Most tablets use an electrical sensing mechanism to determine the position of the stylus. In one such arrangement, a grid of wires on $\frac{1}{4}$- to $\frac{1}{2}$-inch centers is embedded in the tablet surface. Electromagnetic signals generated by electrical pulses applied in sequence to the wires in the grid induce an electrical signal in a wire coil in the stylus. The strength of the

Fig. 4.34 A data tablet with both a stylus and a puck. The stylus has a pressure-sensitive switch in the tip, which closes when the stylus is pressed. The puck has several pushbuttons for command entry, and a cross-hair cursor for accuracy in digitizing drawings that are placed on the tablet. (Courtesy of Summagraphics Corporation.)

signal induced by each pulse is used to determine the position of the stylus. The signal strength is also used to determine roughly how far the stylus or cursor is from the tablet ("far," "near," (i.e., within about $\frac{1}{2}$ inch of the tablet), or "touching"). When the answer is "near" or "touching," a cursor is usually shown on the display to provide visual feedback to the user. A signal is sent to the computer when the stylus tip is pressed against the tablet, or when any button on the puck (pucks have up to 16 buttons) is pressed.

The tablet's (x, y) position, button status, and nearness state (if the nearness state is "far," then no (x, y) position is available) is normally obtained 30 to 60 times per second. Some tablets can generate an interrupt when any of the following events occur:

- t units of time have passed (thus, the tablet serves as a clock to trigger the updating of the cursor position).

- The puck or stylus has moved more than some distance d. This distance-interval sampling is useful in digitizing drawings in order to avoid recording an excessive number of points.

- The nearness state has changed (some tablets do not report a change from "near" to "far").

- The tip switch or a puck button has been pressed or released. It is important to have available both *button-down* and *button-up* events from the tablet and other such devices. For instance, a button-down event might cause a displayed object to start growing larger, and the button-up event then might stop the size change.

- The cursor enters a specified rectangular area of the tablet surface.

Relevant parameters of tablets and other locator devices are their resolution (number of distinguishable points per inch), linearity, repeatability, and size or range. These parameters are particularly crucial for digitizing maps and drawings; they are of less concern when the device is used only to position a screen cursor, because the user has the feedback of the screen cursor position to guide his hand movements, and because the resolution of a typical display is much less than that of even inexpensive tablets.

Other tablet technologies use sound (sonic) coupling and resistive coupling. The *sonic tablet* uses sound waves to couple the stylus to microphones positioned on the periphery of the digitizing area. Sound bursts are created by an electrical spark at the tip of the stylus. The delay between when the spark occurs and when its sound arrives at each microphone is proportional to the distance from the stylus to each microphone (see Fig. 4.35). The sonic tablet is advantageous for digitizing drawings bound in thick books, where a normal tablet stylus does not get close enough to the tablet to record an accurate position. Also, it does not require a dedicated working area, as the other tablets do. Sonic coupling is also useful in 3D positioning devices, as discussed in Chapter 8.

One type of *resistive tablet* uses a battery-powered stylus that emits high-frequency radio signals. The tablet, which is just a piece of glass, is coated with a thin layer of conducting material in which an electrical potential is induced by the radio signals. The strength of the signals at the edges of the tablet is inversely proportional to the distance to the stylus and can thus be used to calculate the stylus position. Another similar tablet uses a resistive polymer mesh stretched across the faceplate of a CRT [SUNF86]. The stylus
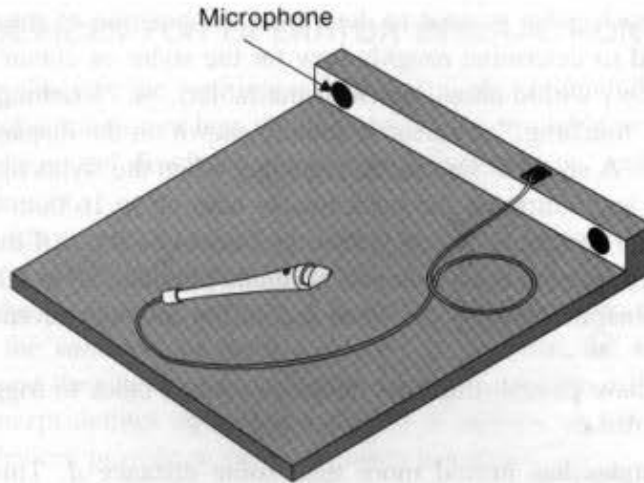
Microphone

Fig. 4.35 A 2D sound tablet. Sound waves emitted from the stylus are received by the two microphones at the rear of the tablet.

applies an electrical potential to the mesh, and the stylus's position is determined from the voltage drop across the mesh.

Most tablet styluses have to be connected to the tablet controller by a wire. The stylus of the resistive tablet can be battery-powered, as shown in Fig. 4.36. Such a stylus is attractive because there is no cord to get in the way. On the other hand, walking away with it in your pocket is easy to do!

Several types of tablets are transparent, and thus can be back-lit for digitizing X-ray films and photographic negatives, and can also be mounted directly over a CRT. The resistive tablet is especially well suited for this, as it can be curved to the shape of the CRT.
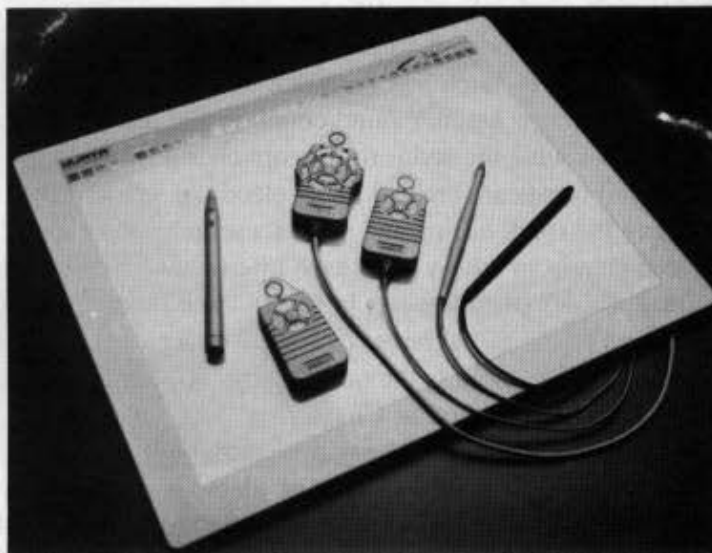
Fig. 4.36 The Penmouse tablet with its cordless three-button, battery-powered stylus and puck. (Courtesy of Kurta Corporation.)

Mouse.　A mouse is a small hand-held device whose relative motion across a surface can be measured. Mice differ in the number of buttons and in how relative motion is detected. Other important differences between various types of mice are discussed in Section 8.1.7. The motion of the roller in the base of a *mechanical mouse* is converted to digital values that are used to determine the direction and magnitude of movement. The *optical mouse* is used on a special pad having a grid of alternating light and dark lines. A light-emitting diode (LED) on the bottom of the mouse directs a beam of light down onto the pad, from which it is reflected and sensed by detectors on the bottom of the mouse. As the mouse is moved, the reflected light beam is broken each time a dark line is crossed. The number of pulses so generated, which is equal to the number of lines crossed, are used to report mouse movements to the computer.

Because mice are relative devices, they can be picked up, moved, and then put down again without any change in reported position. (A series of such movements is often called "stroking" the mouse.) The relative nature of the mouse means that the computer must maintain a "current mouse position," which is incremented or decremented by mouse movements.

Trackball.　The *trackball*, one of which is shown in Fig. 4.37, is often described as an upside-down mechanical mouse. The motion of the trackball, which rotates freely within its housing, is sensed by potentiometers or shaft encoders. The user typically rotates the trackball by drawing the palm of her hand across the ball. Various switches are usually mounted within finger reach of the trackball itself and are used in ways analogous to the use of mouse and tablet-puck buttons.

Joystick.　The *joystick* (Fig. 4.38) can be moved left or right, forward or backward; again, potentiometers sense the movements. Springs are often used to return the joystick to its home center position. Some joysticks, including the one pictured, have a third degree of freedom: the stick can be twisted clockwise and counterclockwise. The *isometric joystick*,

Fig. 4.37 Trackball with several nearby switches. (Courtesy of Measurement Systems, Inc.)

Fig. 4.38 A joystick with a third degree of freedom. The joystick can be twisted clockwise and counterclockwise. (Courtesy of Measurement Systems, Inc.)

shown in Fig. 4.39, is rigid: strain gauges on the shaft measure slight deflections caused by force applied to the shaft.

It is difficult to use a joystick to control the absolute position of a screen cursor directly, because a slight movement of the (usually) short shaft is amplified five or ten times in the movement of the cursor. This makes the screen cursor's movements quite jerky and does not allow quick and accurate fine positioning. Thus, the joystick is often used to control the velocity of the cursor movement rather than the absolute cursor position. This means that the current position of the screen cursor is changed at rates determined by the joystick.

The *joyswitch*, a variant of the joystick, is found in some home and arcade computer games. The stick can be moved in any of eight directions: up, down, left, right, and the four

Fig. 4.39 An isometric joystick. (Courtesy of Measurement Systems, Inc.)

diagonal directions. Small switches sense in which of the eight directions the stick is being pushed.

Touch panel. Mice, trackballs, and joysticks all take up work-surface area. The *touch panel* allows the user to point at the screen directly with a finger to move the cursor around on the screen. Several different technologies are used for touch panels. Low-resolution panels (from 10 to 50 resolvable positions in each direction) use a series of infrared LEDs and light sensors (photodiodes or phototransistors) to form a grid of invisible light beams over the display area. Touching the screen breaks one or two vertical and horizontal light beams, thereby indicating the finger's position. If two parallel beams are broken, the finger is presumed to be centered between them; if one is broken, the finger is presumed to be on the beam.

A capacitively coupled touch panel can provide about 100 resolvable positions in each direction. When the user touches the conductively coated glass panel, electronic circuits detect the touch position from the impedance change across the conductive coating [INTE85].

One high-resolution panel (about 500 resolvable positions in each direction) uses sonar-style ranging. Bursts of high-frequency sound waves traveling alternately horizontally and vertically are introduced at the edge of a glass plate. The touch of a finger on the glass causes part of the wave to be reflected back to its source. The distance to the finger can be calculated from the interval between emission of the wave burst and its arrival back at the source. Another high-resolution panel uses two slightly separated layers of transparent material, one coated with a thin layer of conducting material and the other with resistive material. Fingertip pressure forces the layers to touch, and the voltage drop across the resistive substrate is measured and is used to calculate the coordinates of the touched position. Low-resolution variations of this method use bars of conducting material or thin wires embedded in the transparent material. Touch-panel technologies have been used to make small positioning pads for keyboards.

The most significant touch-panel parameters are resolution, the amount of pressure required for activation (not an issue for the light-beam panel), and transparency (again, not an issue for the light-beam panel). An important issue with some of the technologies is parallax: If the panel is $\frac{1}{2}$ inch away from the display, then users touch the position on the panel that is aligned with their eyes and the desired point on the display, not at the position on the panel directly perpendicular to the desired point on the display.

Users are accustomed to some type of tactile feedback, but touch panels of course offer none. It is thus especially important that other forms of immediate feedback be provided, such as an audible tone or highlighting of the designated target or position.

Light pen. *Light pens* were developed early in the history of interactive computer graphics. The pen is misnamed; it *detects* light pulses, rather than emitting light as its name implies. The event caused when the light pen sees a light pulse on a raster display can be used to save the video controller's X and Y registers and interrupt the computer. By reading the saved values, the graphics package can determine the coordinates of the pixel seen by the light pen. The light pen cannot report the coordinates of a point that is completely black, so special techniques are needed to use the light pen to indicate an arbitrary position: one is to display, for a single frame time, a dark blue field in place of the regular image.

The light pen, when used with a vector display, acts as a pick rather than a positioning

device. When the light pen senses light, the DPU is stopped and the CPU is interrupted. The CPU then reads the DPU's instruction address register, the contents of which can be used to determine which output primitive was being drawn when the interrupt occurred. As with the raster display, a special technique, called *light-pen tracking*, is needed to indicate positions within a single vector with the light pen [FOLE82].

The light pen is an aging technology with limited use. Unless properly adjusted, light pens sometimes detect false targets, such as fluorescent lights or other nearby graphics primitives (e.g., adjacent characters), and fail to detect intended targets. When used over several hours, a light pen can be tiring for inexperienced users, because it must be picked up, pointed, and set down for each use.

4.6.2 Keyboard Devices

The *alphanumeric keyboard* is the prototypical text input device. Several different technologies are used to detect a key depression, including mechanical contact closure, change in capacitance, and magnetic coupling. The important functional characteristic of a keyboard device is that it creates a code (ASCII, EBCDIC, etc.) uniquely corresponding to a pressed key. It is sometimes desirable to allow *chording* (pressing several keys at once) on an alphanumeric keyboard, to give experienced users rapid access to many different commands. This is in general not possible with the standard *coded keyboard*, which returns an ASCII code per keystroke and returns nothing if two keys are pressed simultaneously (unless the additional keys were shift, control, or other special keys). In contrast, an *unencoded keyboard* returns the identity of all keys that are pressed simultaneously, thereby allowing chording.

4.6.3 Valuator Devices

Most valuator devices that provide scalar values are based on potentiometers, like the volume and tone controls of a stereo set. Valuators are usually rotary potentiometers (dials), typically mounted in a group of eight or ten, as in Fig. 4.40. Simple rotary potentiometers

Fig. 4.40 A bank of eight rotary potentiometers. The readouts on the light-emitting diodes (LED) above each dial can be used to label each dial, or to give the current setting. (Courtesy of Evans and Sutherland Computer Corporation.)

Fig. 4.41 Function keys with light-emitting diode (LED) labels, integrated with a keyboard unit. (Courtesy of Evans and Sutherland Computer Corporation.)

can be rotated through about 330°; this may not be enough to provide both adequate range and resolution. Continuous-turn potentiometers can be rotated freely in either direction, and hence are unbounded in range. Linear potentiometers, which are of necessity bounded devices, are used infrequently in graphics systems.

4.6.4   Choice Devices

Function keys are the most common choice device. They are sometimes built as a separate unit, but more often are integrated with a keyboard. Other choice devices are the *buttons* found on many tablet pucks and on the mouse. Choice devices are generally used to enter commands or menu options a graphics program. Dedicated-purpose systems can use function keys with permanent key-cap labels. So that labels can be changeable or "soft," function keys can include a small LCD or LED display next to each button or in the key caps themselves, as shown in Fig. 4.41. Yet another alternative is to place buttons on the bezel of the display, so that button labels can be shown on the display itself, right next to the physical button.

4.7   IMAGE SCANNERS

Although data tablets can be used to digitize existing line drawings manually, this is a slow and tedious process, unsuitable for more than a few simple drawings—and it does not work at all for half-tone images. Image scanners provide an efficient solution. A television camera used in conjunction with a digital frame grabber is an inexpensive way to obtain moderate-resolution (1000 by 1000, with multiple intensity levels) raster images of black-and-white or color photographs. Slow-scan charge-coupled–device (CCD) television cameras can create a 2000 by 2000 image in about 30 seconds. An even lower-cost approach uses a scan head, consisting of a grid of light-sensing cells, mounted on the print head of a printer; it scans images at a resolution of about 80 units per inch. These resolutions are not acceptable for high-quality publication work, however. In such cases, a *photo scanner* is used. The photograph is mounted on a rotating drum. A finely collimated light beam is directed at the photo, and the amount of light reflected is measured by a photocell. For a negative, transmitted light is measured by a photocell inside the drum,
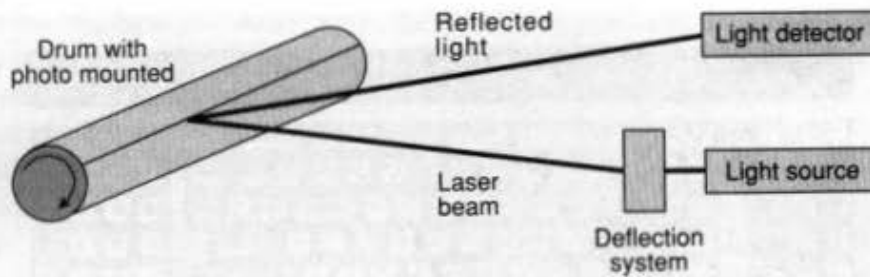
Fig. 4.42 A photo scanner. The light source is deflected along the drum axis, and the amount of reflected light is measured.

which is transparent. As the drum rotates, the light source slowly moves from one end to the other, thus doing a raster scan of the entire photograph (Fig. 4.42). For colored photographs, multiple passes are made, using filters in front of the photocell to separate out various colors. The highest-resolution scanners use laser light sources, and have resolutions greater then 2000 units per inch.

Another class of scanner uses a long thin strip of CCDs, called a *CCD array*. A drawing is digitized by passing it under the CCD array, incrementing the drawing's movement by whatever resolution is required. Thus, a single pass, taking 1 or 2 minutes, is sufficient to digitize a large drawing. Resolution of the CCD array is 200 to 1000 units per inch, which is less than that of the photo scanner. Such a scanner is shown in Fig. 4.43.

Line drawings can easily be scanned using any of the approaches we have described. The difficult part is distilling some meaning from the collection of pixels that results. *Vectorizing* is the process of extracting lines, characters, and other geometric primitives
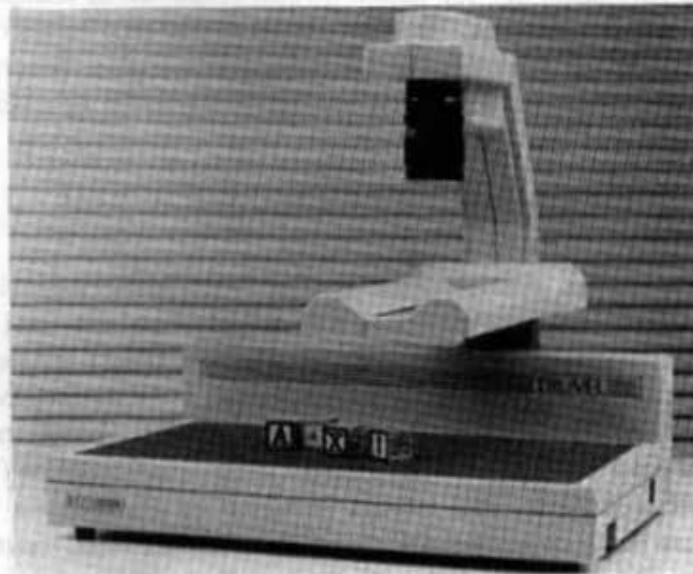
Fig. 4.43 A liquid-crystal display (LCD) scanner. The lens and linear LCD array in the scanning head (top) move left to right to scan a photo or object. (Photographs provided by Truvel Corporation.)

from a raster image. This task requires appropriate algorithms, not scanning hardware, and is essentially an image-processing problem involving several steps. First, thresholding and edge enhancement are used to clean up the raster image—to eliminate smudges and smears and to fill in gaps. Feature-extraction algorithms are then used to combine adjacent "on" pixels into geometric primitives such as straight lines. At a second level of complexity, pattern-recognition algorithms are used to combine the simple primitives into arcs, letters, symbols, and so on. User interaction may be necessary to resolve ambiguities caused by breaks in lines, dark smudges, and multiple lines intersecting near one another.

A more difficult problem is organizing a collection of geometric primitives into meaningful data structures. A disorganized collection of lines is not particularly useful as input to a CAD or topographic (mapping) application program. The higher-level geometric constructs represented in the drawings need to be recognized. Thus, the lines defining the outline of a county should be organized into a polygon primitive, and the small "+" representing the center of an arc should be grouped with the arc itself. There are partial solutions to these problems. Commercial systems depend on user intervention when the going gets tough, although algorithms are improving continually.

EXERCISES

4.1 For an electrostatic plotter with 18-inch-wide paper, a resolution of 200 units to the inch in each direction, and a paper speed of 3 inches per second, how many bits per second must be provided to allow the paper to move at full speed?

4.2 If long-persistence phosphors decrease the fusion frequency, why not use them routinely?

4.3 Write a program to display test patterns on a raster display. Three different patterns should be provided: (1) horizontal lines 1 pixel wide, separated by 0, 1, 2, or 3 pixels; (2) vertical lines 1 pixel wide, separated by 0, 1, 2, or 3 pixels; and (3) a grid of 1-pixel dots on a grid spaced at 5-pixel intervals. Each pattern should be displayable in white, red, green, or blue, as well as alternating color bars. How does what you observe when the patterns are displayed relate to the discussion of raster resolution?

4.4 Prepare a report on technologies for large-screen displays.

4.5 How long would it take to load a 512 by 512 by 1 bitmap, assuming that the pixels are packed 8 to a byte and that bytes can be transferred and unpacked at the rate of 100,000 bytes per second? How long would it take to load a 1024 by 1280 by 1 bitmap?

4.6 Design the logic of a hardware unit to convert from 2D raster addresses to byte plus bit-within-byte addresses. The inputs to the unit are as follows: (1) (x, y), a raster address; (2) *base*, the address of the memory byte that has raster address $(0, 0)$ in bit 0; and (3) x_{max}, the maximum raster x address (0 is the minimum). The outputs from the unit are as follows: (1) *byte*, the address of the byte that contains (x, y) in one of its bits; and (2) *bit*, the bit within *byte* which contains (x, y). What simplifications are possible if $x_{max} + 1$ is a power of 2?

4.7 Program a raster copy operation that goes from a bitmap into a virtual-memory area that is scattered across several pages. The hardware raster copy on your display works only in physical address space, so you must set up a scatter-write situation, with multiple moves invoked, one for each logical page contained in the destination.

4.8 Design an efficient instruction-set encoding for the simple raster display instruction set of Section 4.3.2. By "efficient" is meant "minimizing the number of bits used per instruction."

4.9 Using the instruction set for the simple raster display of Section 4.3.2, write program segments to do the following.

  a. Double-buffer a moving object. You need not actually write the code that moves and draws the object; just indicate its placement in the overall code sequence.

  b. Draw a pie chart, given as input a list of data values. Use different colors for each slice of the pie, choosing colors you believe will look good together.

  c. Animate the left-to-right movement of circles arranged in a horizontal row. Each circle is five units in radius and the circles are placed on 15-unit centers. Use the animation technique discussed in Section 4.4.1. At each step in the animation, every fourth circle should be visible.

  d. Write a program that uses RasterOp to drag a 25 by 25 icon around on the screen, using an offscreen area of the bitmap to store both the icon and the part of the bitmap currently obscured by the icon.

4.10 In a raster scan of n lines with m pixels per line displayed at r cycles per second, there is a certain amount of time during which no image is being traced: the horizontal retrace time t_h, which occurs once per scan line, and the vertical retrace time t_v, which occurs once per frame.

  a. Derive equations for the percentage of time that no image is being traced, remembering that different equations are needed for the interlaced and noninterlaced cases, because the two fields per frame of an interlaced scan each require a vertical retrace time.

  b. Evaluate the equation for the following values, taken from [WHIT84]:

Visible area pixels × lines	Refresh rate, Hz	Interlace	Vertical retrace time, microseconds	Horizontal retrace time, microseconds
640 × 485	30	yes	1271	11
1280 × 1024	30	yes	1250	7
640 × 485	60	no	1250	7
1280 × 1024	60	no	600	4

  c. What is the meaning of the percentages you have calculated?

4.11 Develop a design strategy for a video controller that can display a 960 by x image on a 480 by $x/2$ display, where x is the horizontal resolution. (This is the type of device needed to drive a video recorder from a high-resolution display system.)

4.12 A raster display has 4 bits per pixel and a look-up table with 12 bits per entry (4 bits each for red, green, and blue). Think of the four planes of the pixmap as being partitioned to hold two images: image 1 in the two high-order planes, image 0 in the two low-order planes. The color assignments for each of the 2-bit pixel values in each image are 00 = red, 01 = green, 10 = blue, and 11 = white.

  a. Show how to load the look-up table so that just image 1 is displayed.

  b. Show how to load the look-up table so that just image 0 is displayed.

Note that these are the look-up tables needed for the two phases of double-buffering, as discussed in Section 4.4.1.

4.13 Given a 16-entry by 12-bit-per-entry look-up table with two 2-bit images, show how it should be loaded to produce the lap-dissolve given by the expression $image1 * t + image0 * (1 - t)$ for values of $t = 0.0, 0.25$, and 0.5. The look-up table color assignments are the same as in Exercise 4.12. (See Chapter 17.)

4.14 In Section 4.4.1, two animation methods are discussed. What are the advantages and disadvantages of each?

4.15 Redesign the simple DPU instruction set of Fig. 4.32 for a 16-bit word, assuming that all opcodes are equal in length.

4.16 Consider means for picking (as with a light pen) a line on a raster display when all that can be detected is a single pixel.

Plate I.1 Radiation therapy planning simulation. Volume rendering is used to display the interaction of radiation beams and a child's head. (Image by R. Drebin, Silicon Graphics. Copyright © 1988 Pixar. CT scans provided by F. Zonneveld, N.V. Philips.)

Plate I.2 Local map automatically generated by the IGD hypermedia system, showing the currently visited node in the center, possible source nodes in the left column and possible destination nodes in the right column. (Courtesy of S. Feiner, S. Nagy, and A. van Dam, Brown University Computer Graphics Group.)
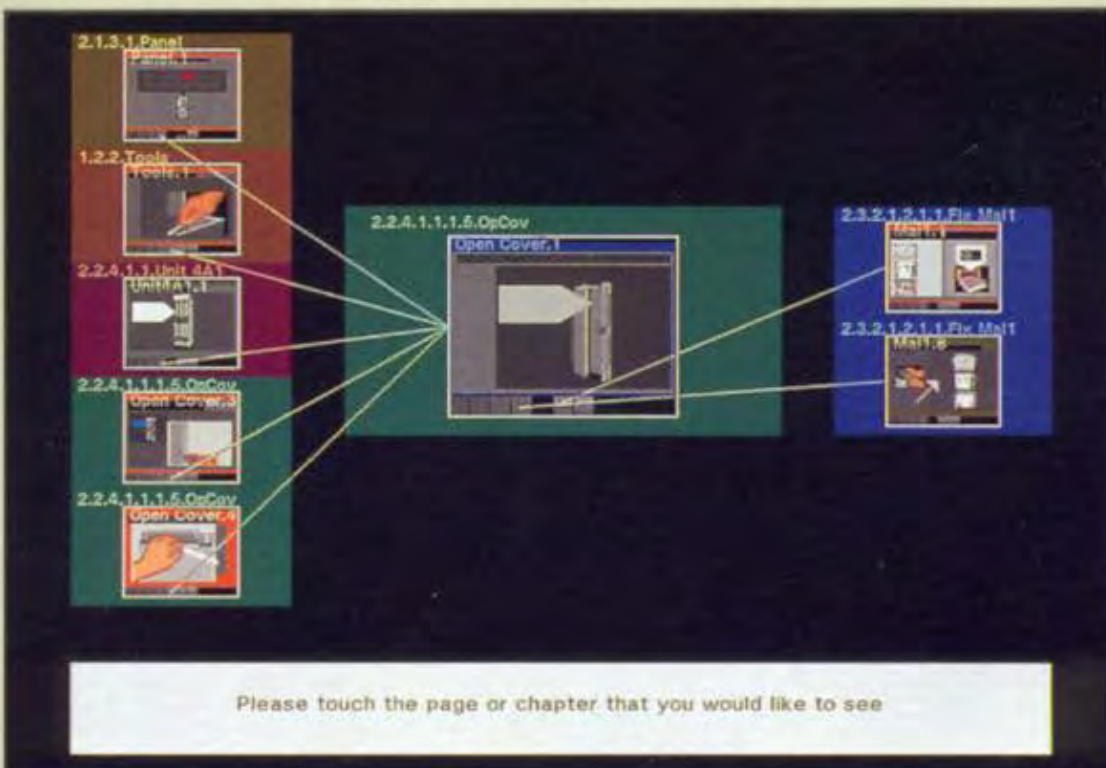
Plate I.3 One stage in the process of smoothly turning a sphere inside out. The sphere is sliced to show interior structure. (Copyright © 1989 John Hughes, Brown University Computer Graphics Group.)

Plate I.4 Animated sequences of stereographic projections of the sphere and hypersphere as they are rotated in 3- and 4-space, respectively. For further information, see [KOCA87]. (Image by D. Laidlaw and H. Kocak.)

Plate I.5 (a) The cockpit of an F5 flight simulator; the pilot's view is projected onto a dome surrounding the cockpit. (b) The view from the cockpit of a flight simulator. The fighter jet is modeled geometrically, whereas the terrain is photo-textured. (Courtesy of R. Economy, General Electric Company.)

(a)

(b)

Plate I.6 A video game in which the player must pack 3D shapes together in a small space. The deep perspective and wireframe drawing of the piece aid in this task. (Courtesy of Larry Lee, California Dreams.)

▼
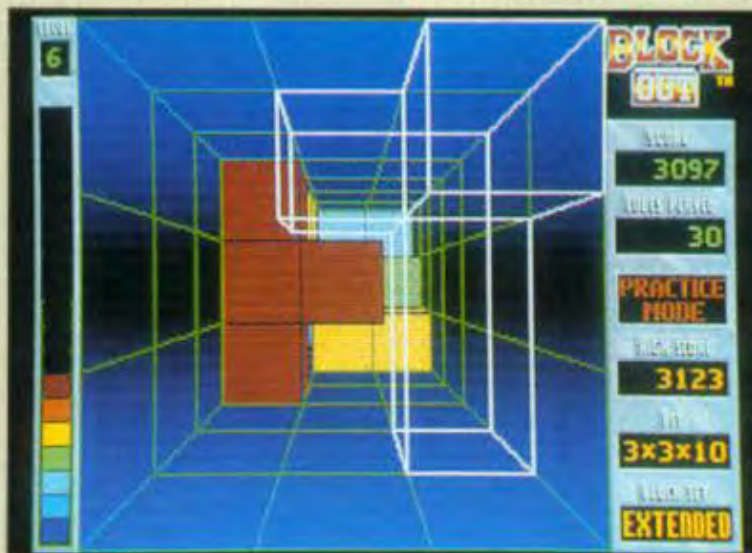
Plate I.7 *Hard Drivin'* arcade video game. (Courtesy of Atari Games Corporation, copyright © 1988 Atari Games Corporation.)

Plate I.8 Home improvement design center from Innovis. (Courtesy of Innovis Interactive Technologies, Tacoma, WA.)

Plate I.9 "Dutch Interior," after Vermeer, by J. Wallace, M. Cohen, and D. Greenberg, Cornell University. (Copyright © 1987 Cornell University, Program of Computer Graphics.)

Plate I.10 Severe tornadic storm, by R. Wilhelmson, L. Wicker, and C. Shaw, NCSA, University of Illinois. (Application Visualization System by Stardent Computer.)

▼
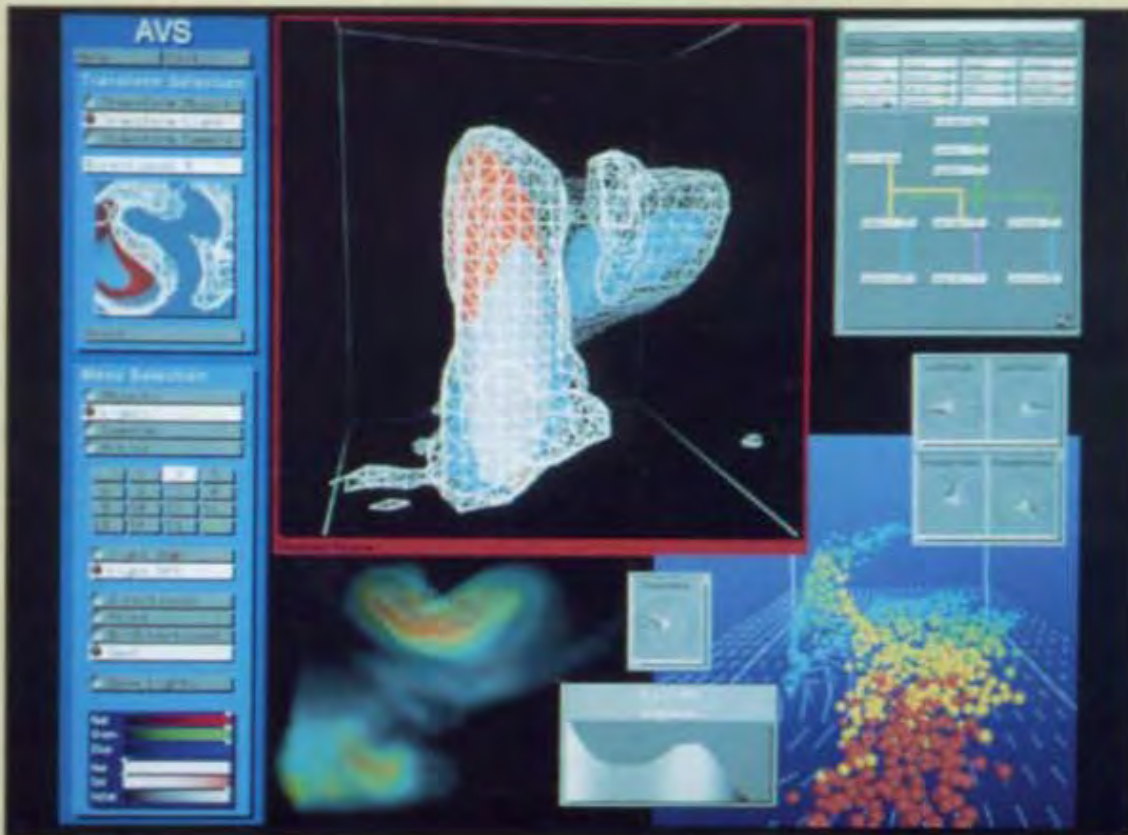
Plate I.11 (a) Chevrolet Astro Van, product launch material. (Copyright © 1985 Digital Productions.) (b) Castle interior. (Copyright © 1985 Digital Productions. Both images courtesy of G. Demos.)

(a)

(b)

Plate I.12 *The Abyss* — Pseudopod sequence. (Copyright © 1989 Twentieth Century Fox. All rights reserved. Courtesy of Industrial Light & Magic, Computer Graphics Division.)

Plate I.13 Command ship from *The Last Starfighter*. The texture-mapped ship has 450,000 polygons. (Copyright © 1984 Digital Productions. Courtesy of G. Demos.)

Plate I.14 A spaceball six-degree of freedom positioning device. (Courtesy of Spatial Systems, Inc.)

Plate I.15 A DataGlove (right) and computer image of the glove. The DataGlove measures finger movements and hand orientation and position. The computer image of the hand tracks the changes. (Courtesy of Jaron Lanier, VPL.)

Plate I.16 A user wearing a head-mounted stereo display, DataGloves, and microphone for issuing commands. These devices are used to create a virtual reality for the user, by changing the stereo display presentation as the head is moved, with the DataGloves used to manipulate computer-generated objects. (Courtesy of Michael McGreevy and Scott Fisher, NASA Ames Research Center, Moffett Field, CA.)

▲
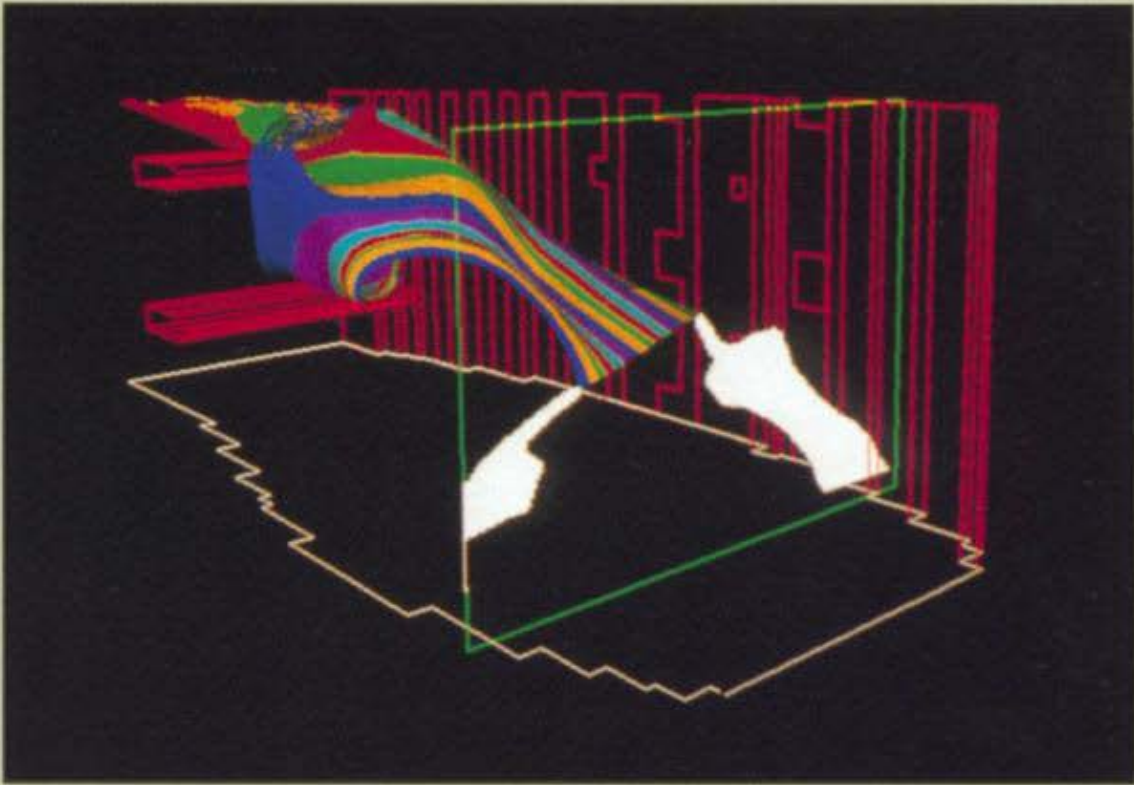
Plate I.17 Krueger's Videotouch system, in which a user's hand movements are used to manipulate an object. The hands' outlines are displayed along with the objects to provide natural feedback. (Courtesy of Myron Krueger, Artificial Reality Corp.)
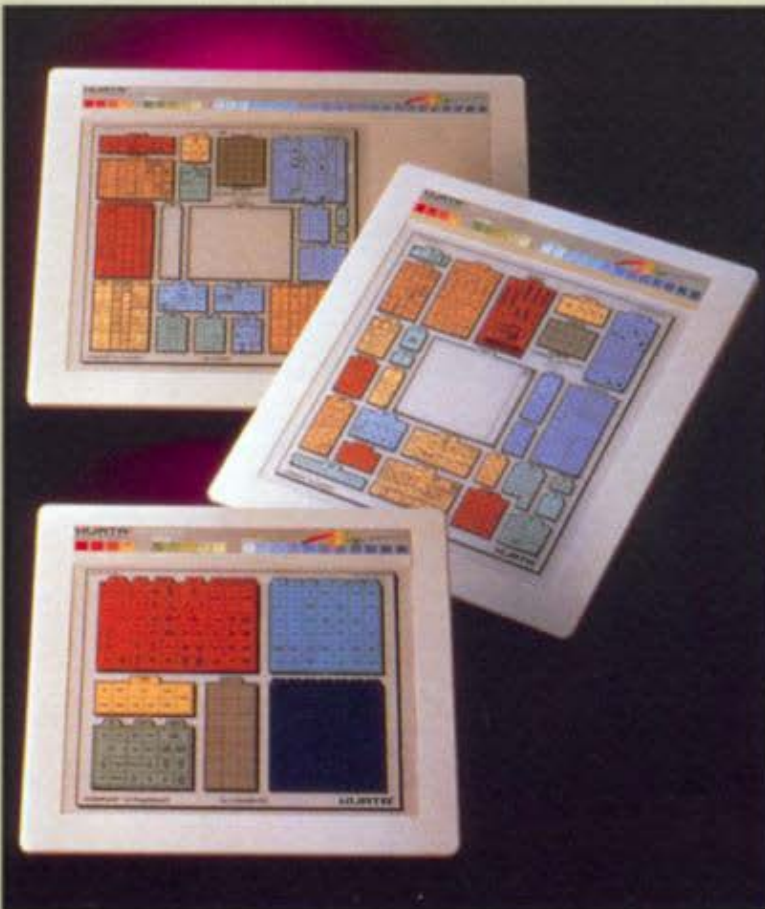
Plate I.18 Tablet-based menus. Color is used to group related menu items. (Courtesy of Kurta Corporation.)

Plate I.19 A menu using a hand to show operations. The second and third rows use before and after representations of the object operated upon to indicate the meaning. From left to right, top to bottom, the commands are: File, Delete, Shear, Rotate, Move, and Copy. (Courtesy of Peter Tierney, © 1988 Cybermation, Inc.)

Plate I.20 A menu of operations on text. From left to right, top to bottom, the meanings are: select font, set height, set width (using before and after representation), slant (using before and after representation), set letter spacing, set line spacing. (Courtesy of Peter Tierney, © 1988 Cybermation, Inc.)

Plate I.21 A menu of operations on geometric objects, all showing before and after representations. From left to right, top to bottom, the meanings are: move point, change radius of an arc (or change a line to an arc), add vertex (one segment becomes two), remove vertex (two line segments become one), fillet corner, and chamfer corner. (Courtesy of Peter Tierney, © 1988 Cybermation, Inc.)
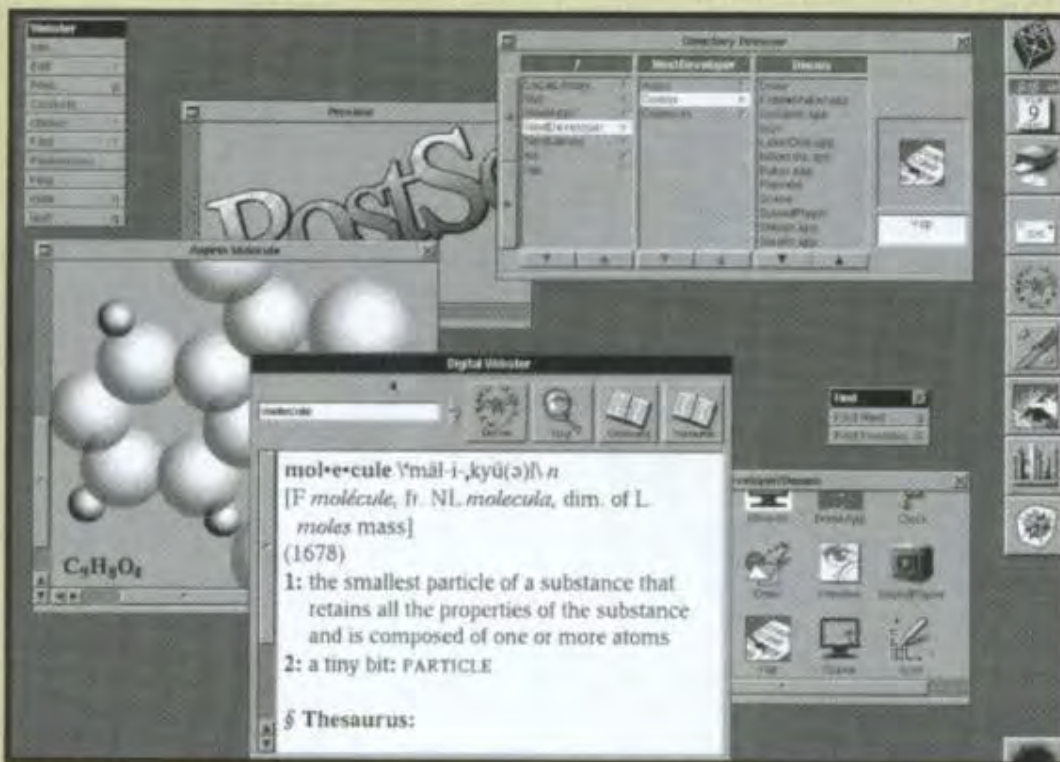
Plate I.22 The NeXT user interface. (Courtesy of NeXT, Inc. © 1989 NeXT, Inc.)

Plate I.23 Editable graphical history in Chimera. (a) Editor window shows picture created by Chimera's user. (b) History window shows sequence of before-after panel pairs created by Chimera. (Courtesy of David Kurlander and Steven Feiner, Columbia University.)

(a)

(b)
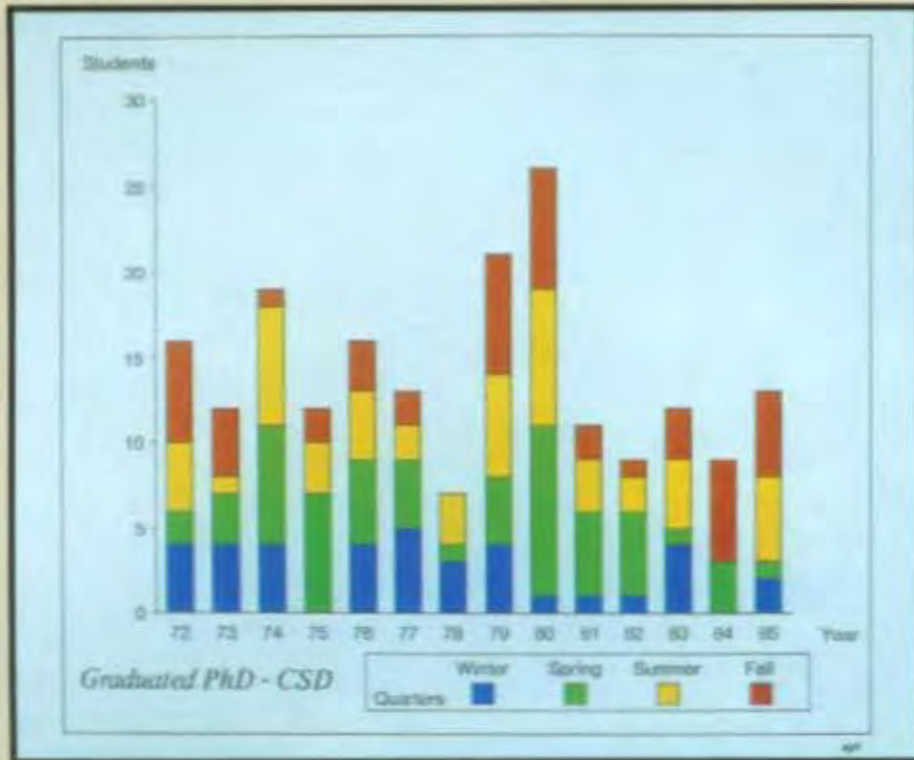
Plate I.24 A stacked bar chart created automatically by A Presentation Tool (APT) in response to a request to plot the number of graduating students per quarter from 1972 to 1985. APT generated this presentation as being more effective than many others it considered. (Courtesy of Jock Mackinlay.)

Plate I.25 Picture of a radio generated by IBIS to satisfy input communicative goals to show location of function dial and its change of state. IBIS determines the objects to include, rendering style, viewing and lighting specs, and picture composition. (Courtesy of Dorée Duncan Seligmann and Steven Feiner, Columbia University.)

Plate I.26 The OSF/Motif user interface. In this image, different shades of blue are used to distinguish visual elements. (Courtesy of Open Software Foundation.)

Plate I.27 The OSF/Motif user interface. The color slider bars are used to define colors for use in windows. Notice the use of shading on the edges of buttons, menus, and so forth, to create a 3D effect. (Courtesy of Open Software Foundation.)
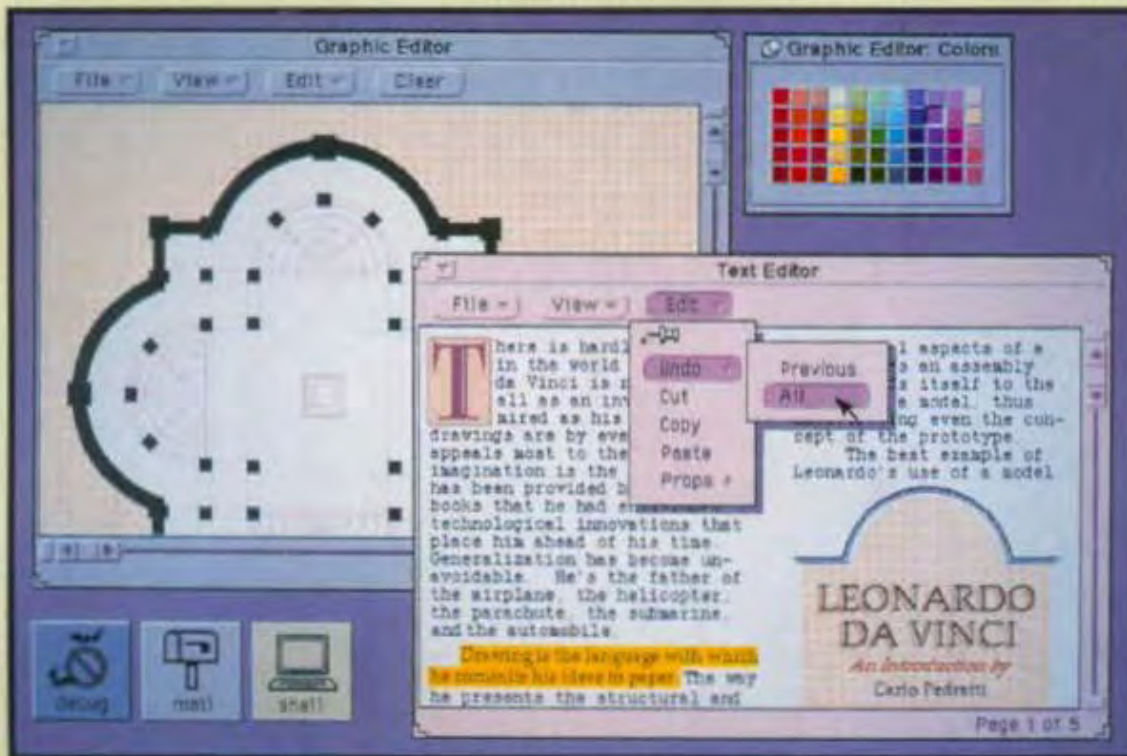
Plate I.28 The OPEN LOOK user interface. Yellow is used to highlight selected text. Subdued shades are used for the background and window borders. (Courtesy of Sun Microsystems.)

Plate I.29 The OPEN LOOK user interface. (Courtesy of Sun Microsystems.)
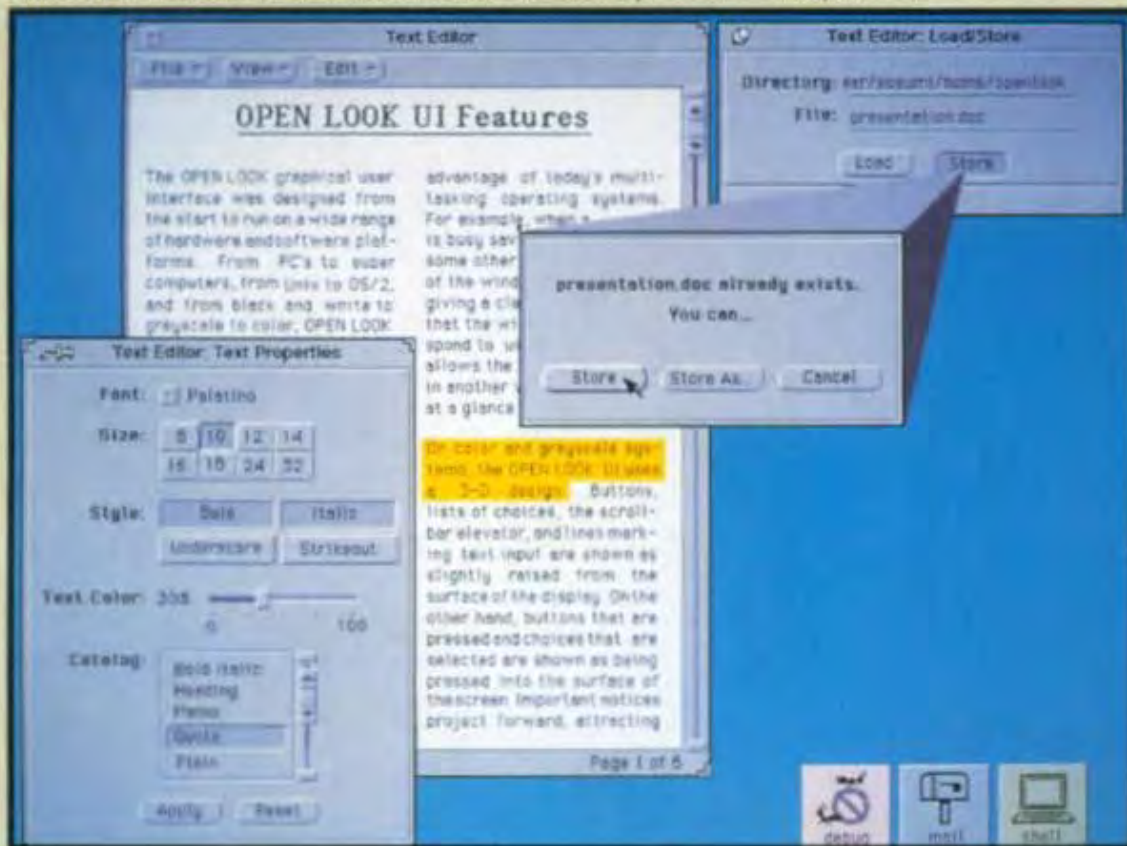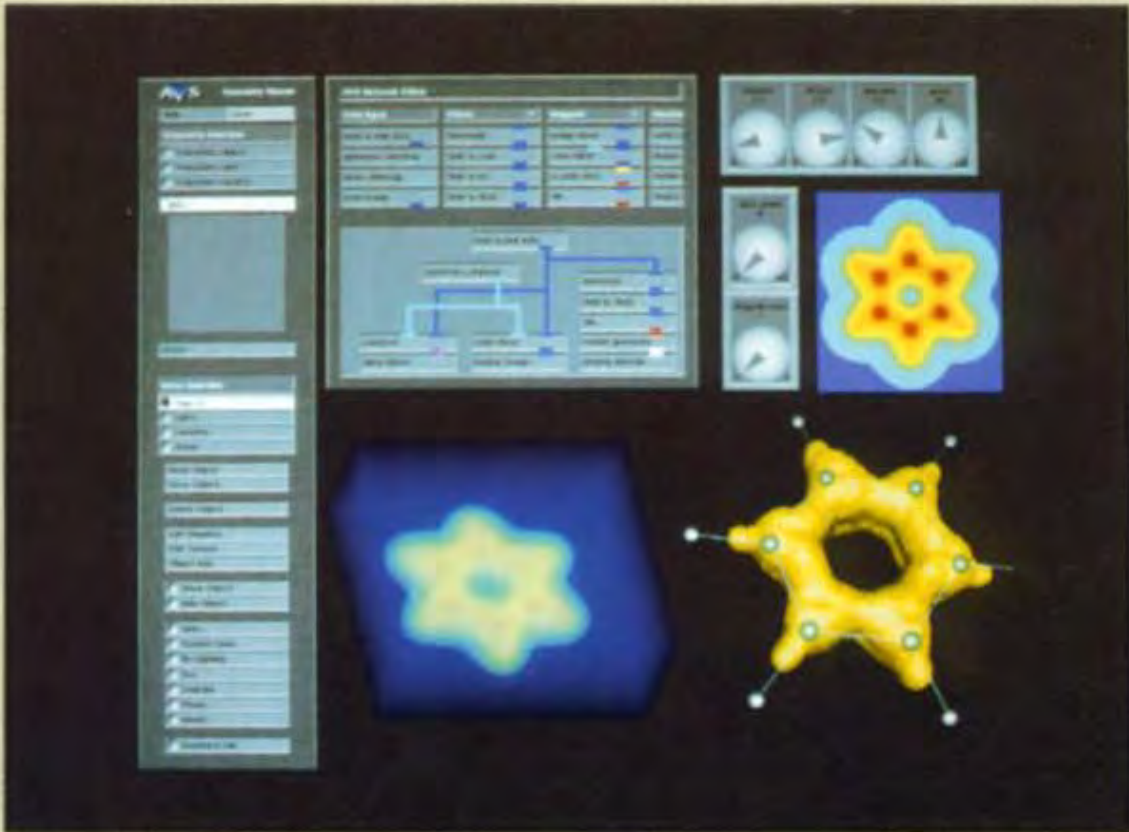
Plate I.30 The Application Visualization System (AVS) interface. The data flow diagram (middle center) is constructed interactively by the user from the menu of processing elements (top center). In this example, inputs from the the control dials (to the right) control the display of XYZ. (Courtesy of Stardent, Inc.)

Plate I.31 Objects modeled with NURBS, using the Alpha_1 modeling system. (Courtesy of University of Utah.)
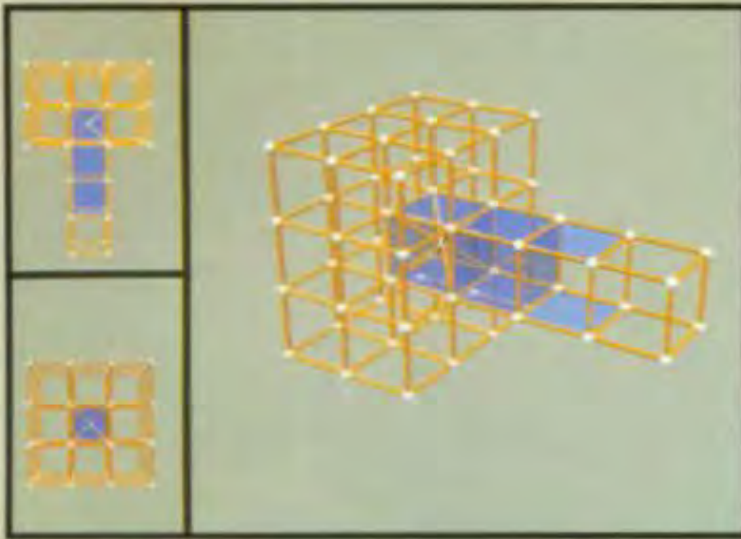
Plate I.32 Three views of a 3D nonmanifold representation with blue faces, yellow edges, and white vertices. In model shown, two solid cubes share a common face, rightmost cube has two dangling faces connected to a cubic wireframe, and leftmost cube has interior and exterior finite–element-method mesh wireframes. (Courtesy of K. Weiler, D. McLachlan, H. Thorvaldsdóttir; created using Dóre, © 1989 Ardent Computer Corporation.)

Plate I.33 A portion of a PANTONE® Color Specifier 747XR page. Color names (obscured in this reproduction) are shown along with the color samples. The color names are keyed to mixtures of standard inks that will reproduce the color. (Courtesy of Pantone, Inc. PANTONE® is Pantone, Inc.'s check-standard trademark for color reproduction and color reproduction materials. Process color reproduction may not match PANTONE®-identified solid color standards. Refer to current PANTONE® Color Publications for the accurate color.)

Plate I.34 The colors of the spectrum, from violet on the left to red on the right. The height of the curve is the spectral power distribution of illuminant C. (Courtesy of Barbara Meier, Brown University.)
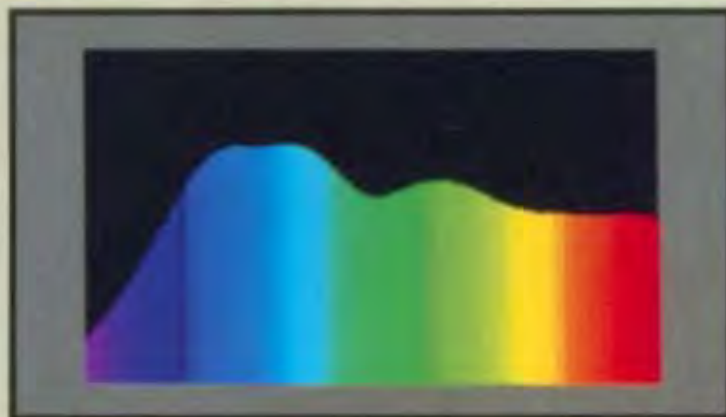
5

Geometrical
Transformations

This chapter introduces the basic 2D and 3D geometrical transformations used in computer graphics. The translation, scaling, and rotation transformations discussed here are essential to many graphics applications and will be referred to extensively in succeeding chapters.

The transformations are used directly by application programs and within many graphics subroutine packages. A city-planning application program would use translation to place symbols for buildings and trees at appropriate positions, rotation to orient the symbols, and scaling to size the symbols. In general, many applications use the geometric transformations to change the position, orientation, and size of objects (also called *symbols* or *templates*) in a drawing. In Chapter 6, 3D rotation, translation, and scaling will be used as part of the process of creating 2D renditions of 3D objects. In Chapter 7, we see how a contemporary graphics package uses transformations as part of its implementation and also makes them available to application programs.

5.1   2D TRANSFORMATIONS

We can *translate* points in the (x, y) plane to new positions by adding translation amounts to the coordinates of the points. For each point $P(x, y)$ to be moved by d_x units parallel to the x axis and by d_y units parallel to the y axis to the new point $P'(x', y')$, we can write

$$x' = x + d_x, \qquad y' = y + d_y. \qquad (5.1)$$

If we define the column vectors

$$P = \begin{bmatrix} x \\ y \end{bmatrix}, P' = \begin{bmatrix} x' \\ y' \end{bmatrix}, T = \begin{bmatrix} d_x \\ d_y \end{bmatrix}, \qquad (5.2)$$
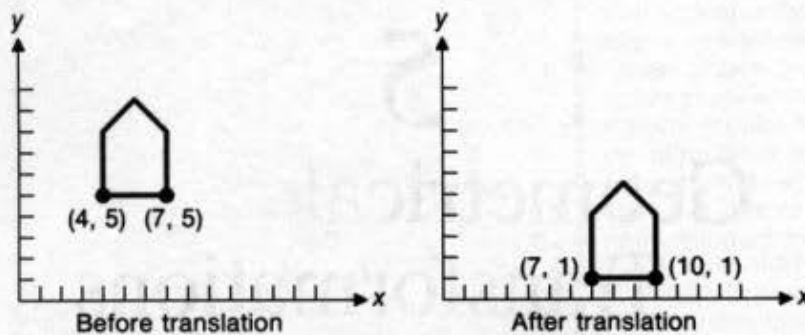
Fig. **5.1** Translation of a house.

then (5.1) can be expressed more concisely as

$$P' = P + T. \tag{5.3}$$

We could translate an object by applying Eq. (5.1) to every point of the object. Because each line in an object is made up of an infinite number of points, however, this process would take an infinitely long time. Fortunately, we can translate all the points on a line by translating only the line's endpoints and by drawing a new line between the translated endpoints; this is also true of scaling (stretching) and rotation. Figure 5.1 shows the effect of translating the outline of a house by (3, −4).

Points can be *scaled* (stretched) by s_x along the x axis and by s_y along the y axis into new points by the multiplications

$$x' = s_x \cdot x, \quad y' = s_y \cdot y. \tag{5.4}$$

In matrix form, this is

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \text{ or } P' = S \cdot P, \tag{5.5}$$

where S is the matrix in Eq. (5.5).

In Fig. 5.2, the house is scaled by $\frac{1}{2}$ in x and $\frac{1}{4}$ in y. Notice that the scaling is about the origin: The house is smaller *and* is closer to the origin. If the scale factors were greater than 1, the house would be both larger and further from the origin. Techniques for scaling about some point other than the origin are discussed in Section 5.2. The proportions of the house have also changed: a *differential* scaling, in which $s_x \neq s_y$, has been used. With a *uniform* scaling, in which $s_x = s_y$, the proportions are unaffected.
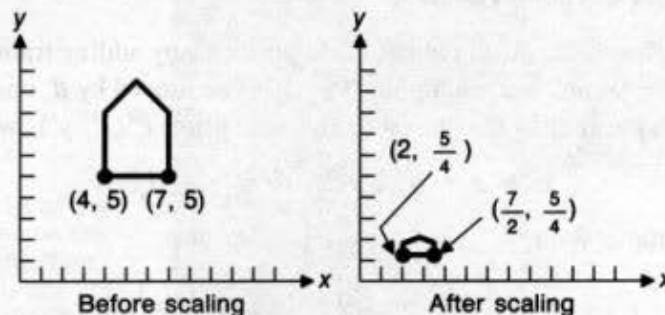
Fig. **5.2** Scaling of a house. The scaling is nonuniform, and the house changes position.
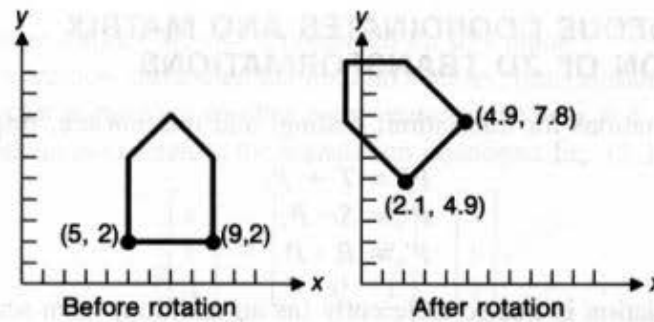
Fig. 5.3 Rotation of a house. The house also changes position.

Points can be *rotated* through an angle θ about the origin. A rotation is defined mathematically by

$$x' = x \cdot \cos\theta - y \cdot \sin\theta, \quad y' = x \cdot \sin\theta + y \cdot \cos\theta. \tag{5.6}$$

In matrix form, we have

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \text{ or } P' = R \cdot P, \tag{5.7}$$

where R is the rotation matrix in Eq. (5.7). Figure 5.3 shows the rotation of the house by 45°. As with scaling, rotation is about the origin; rotation about an arbitrary point is discussed in Section 5.2.

Positive angles are measured *counterclockwise* from x toward y. For negative (clockwise) angles, the identities $\cos(-\theta) = \cos\theta$ and $\sin(-\theta) = -\sin\theta$ can be used to modify Eqs. (5.6) and (5.7).

Equation (5.6) is easily derived from Fig. 5.4, in which a rotation by θ transforms $P(x, y)$ into $P'(x', y')$. Because the rotation is about the origin, the distances from the origin to P and to P', labeled r in the figure, are equal. By simple trigonometry, we find that

$$x = r \cdot \cos\phi, \quad y = r \cdot \sin\phi \tag{5.8}$$

and

$$x' = r \cdot \cos(\theta + \phi) = r \cdot \cos\phi \cdot \cos\theta - r \cdot \sin\phi \cdot \sin\theta,$$
$$y' = r \cdot \sin(\theta + \phi) = r \cdot \cos\phi \cdot \sin\theta + r \cdot \sin\phi \cdot \cos\theta. \tag{5.9}$$

Substituting Eq. (5.8) into Eq. (5.9) yields Eq. (5.6).
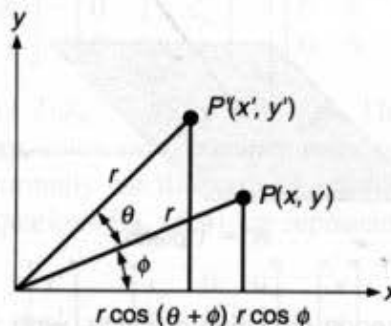
Fig. 5.4 Derivation of the rotation equation.

5.2 HOMOGENEOUS COORDINATES AND MATRIX REPRESENTATION OF 2D TRANSFORMATIONS

The matrix representations for translation, scaling, and rotation are, respectively,

$$P' = T + P, \tag{5.3}$$
$$P' = S \cdot P, \tag{5.5}$$
$$P' = R \cdot P. \tag{5.7}$$

Unfortunately, translation is treated differently (as an addition) from scaling and rotation (as multiplications). We would like to be able to treat all three transformations in a consistent way, so that they can be combined easily.

If points are expressed in *homogeneous coordinates*, all three transformations can be treated as multiplications. Homogeneous coordinates were first developed in geometry [MAXW46; MAXW51] and have been applied subsequently in graphics [ROBE65; BLIN77b; BLIN78a]. Numerous graphics subroutine packages and display processors work with homogeneous coordinates and transformations.

In homogeneous coordinates, we add a third coordinate to a point. Instead of being represented by a pair of numbers (x, y), each point is represented by a triple (x, y, W). At the same time, we say that two sets of homogeneous coordinates (x, y, W) and (x', y', W') represent the same point if and only if one is a multiple of the other. Thus, $(2, 3, 6)$ and $(4, 6, 12)$ are the same points represented by different coordinate triples. That is, each point has many different homogeneous coordinate representations. Also, at least one of the homogeneous coordinates must be nonzero: $(0, 0, 0)$ is not allowed. If the W coordinate is nonzero, we can divide through by it: (x, y, W) represents the same point as $(x/W, y/W, 1)$. When W is nonzero, we normally do this division, and the numbers x/W and y/W are called the Cartesian coordinates of the homogeneous point. The points with $W = 0$ are called points at infinity, and will not appear very often in our discussions.

Triples of coordinates typically represent points in 3-space, but here we are using them to represent points in 2-space. The connection is this: If we take all the triples representing the same point—that is, all triples of the form (tx, ty, tW), with $t \neq 0$—we get a line in 3-space. Thus, each homogeneous *point* represents a *line* in 3-space. If we *homogenize* the point (divide by W), we get a point of the form $(x, y, 1)$. Thus, the homogenized points form the plane defined by the equation $W = 1$ in (x, y, W)-space. Figure 5.5 shows this
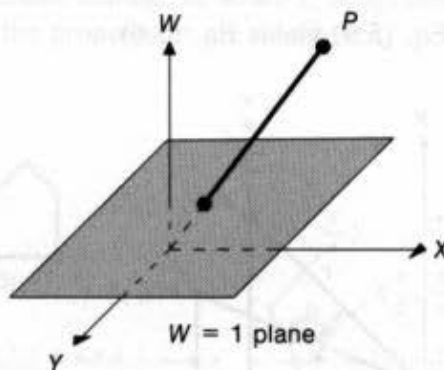

Fig. 5.5 The *XYW* homogeneous coordinate space, with the $W = 1$ plane and point $P(X, Y, W)$ projected onto the $W = 1$ plane.

relationship. Points at infinity are not represented on this plane.

Because points are now three-element column vectors, transformation matrices, which multiply a point vector to produce another point vector, must be 3×3. In the 3×3 matrix form for homogeneous coordinates, the translation equations Eq. (5.1) are

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.10}$$

We caution the reader that some graphics textbooks, including [FOLE82], use a convention of premultiplying matrices by row vectors, rather than postmultiplying by column vectors. Matrices must be transposed to go from one convention to the other, just as the row and column vectors are transposed:

$$(P \cdot M)^T = M^T \cdot P^T.$$

Equation (5.10) can be expressed differently as

$$P' = T(d_x, d_y) \cdot P, \tag{5.11}$$

where

$$T(d_x, d_y) = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.12}$$

What happens if a point P is translated by $T(d_{x1}, d_{y1})$ to P' and then translated by $T(d_{x2}, d_{y2})$ to P''? The result we expect intuitively is a net translation $T(d_{x1} + d_{x2}, d_{y1} + d_{y2})$. To confirm this intuition, we start with the givens:

$$P' = T(d_{x1}, d_{y1}) \cdot P, \tag{5.13}$$
$$P'' = T(d_{x2}, d_{y2}) \cdot P'. \tag{5.14}$$

Now, substituting Eq. (5.13) into Eq. (5.14), we obtain

$$P'' = T(d_{x2}, d_{y2}) \cdot (T(d_{x1}, d_{y1}) \cdot P) = (T(d_{x2}, d_{y2}) \cdot T(d_{x1}, d_{y1})) \cdot P. \tag{5.15}$$

The matrix product $T(d_{x2}, d_{y2}) \cdot T(d_{x1}, d_{y1})$ is

$$\begin{bmatrix} 1 & 0 & d_{x2} \\ 0 & 1 & d_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & d_{x1} \\ 0 & 1 & d_{y1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{x1} + d_{x2} \\ 0 & 1 & d_{y1} + d_{y2} \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.16}$$

The net translation is indeed $T(d_{x1} + d_{x2}, d_{y1} + d_{y2})$. The matrix product is variously referred to as the *compounding, catenation, concatenation,* or *composition* of $T(d_{x1}, d_{y1})$ and $T(d_{x2}, d_{y2})$. Here, we shall normally use the term *composition*.

Similarly, the scaling equations Eq. (5.4) are represented in matrix form as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.17}$$

Defining

$$S(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.18}$$

we have

$$P' = S(s_x, s_y) \cdot P. \tag{5.19}$$

Just as successive translations are additive, we expect that successive scalings should be multiplicative. Given

$$P' = S(s_{x1}, s_{y1}) \cdot P, \tag{5.20}$$
$$P'' = S(s_{x2}, s_{y2}) \cdot P', \tag{5.21}$$

then, substituting Eq. (5.20) into Eq. (5.21), we get

$$P'' = S(s_{x2}, s_{y2}) \cdot (S(s_{x1}, s_{y1}) \cdot P) = (S(s_{x2}, s_{y2}) \cdot S(s_{x1}, s_{y1})) \cdot P. \tag{5.22}$$

The matrix product $S(s_{x2}, s_{y2}) \cdot S(s_{x1}, s_{y1})$ is

$$\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{5.23}$$

Thus, the scalings are indeed multiplicative.

Finally, the rotation equations Eq. (5.6) can be represented as

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.24}$$

Letting

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.25}$$

we have

$$P' = R(\theta) \cdot P. \tag{5.26}$$

Showing that two successive rotations are additive is left as Exercise 5.2.

In the upper-left 2×2 submatrix of Eq. (5.25), consider each of the two rows as vectors. The vectors can be shown to have three properties:

1. Each is a unit vector

2. Each is perpendicular to the other (their dot product is zero)

3. The first and second vectors will be rotated by $R(\theta)$ to lie on the positive x and y axes, respectively (in the presence of conditions 1 and 2, this is equivalent to the submatrix having a determinant of 1).

The first two properties are also true of the columns of the 2 × 2 submatrix. The two directions are those into which vectors along the positive x and y axes are rotated. These properties suggest two useful ways to go about deriving a rotation matrix when we know the effect desired from the rotation. A matrix having these properties is called *special orthogonal*.

A transformation matrix of the form

$$\begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix},$$ (5.27)

where the upper 2 × 2 submatrix is orthogonal, preserves angles and lengths. That is, a unit square remains a unit square, and becomes neither a rhombus with unit sides, nor a square with nonunit sides. Such transformations are also called *rigid-body* transformations, because the body or object being transformed is not distorted in any way. An arbitrary sequence of rotation and translation matrices creates a matrix of this form.

What can be said about the product of an arbitrary sequence of rotation, translation, and scale matrices? They are called *affine* transformations, and have the property of preserving parallelism of lines, but not lengths and angles. Figure 5.6 shows the results of applying a −45° rotation and then a nonuniform scaling to the unit cube. It is clear that neither angles nor lengths have been preserved by this sequence, but parallel lines have remained parallel. Further rotation, scale, and translation operations will not cause the parallel lines to cease being parallel. $R(\theta)$, $S(s_x, s_y)$, and $T(d_x, d_y)$ are also affine.

Another type of primitive transformation, *shear transformations*, are also affine. Two-dimensional shear transformations are of two kinds: a shear along the x axis and a shear along the y axis. Figure 5.7 shows the effect of shearing the unit cube along each axis. The operation is represented by the matrix

$$SH_z = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$ (5.28)

The term a in the shear matrix is the proportionality constant. Notice that the product $SH_z [x \quad y \quad 1]^T$ is $[x + ay \quad y \quad 1]^T$, clearly demonstrating the proportional change in x as a function of y.

Unit cube   45°   Scale in x,
                            not in y
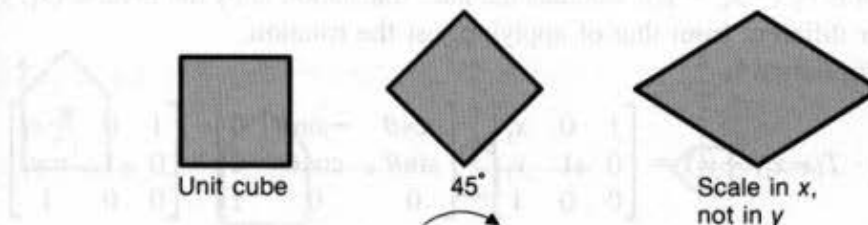
Fig. 5.6 A unit cube is rotated by −45° and is nonuniformly scaled. The result is an affine transformation of the unit cube, in which parallelism of lines is maintained, but neither angles nor lengths are maintained.
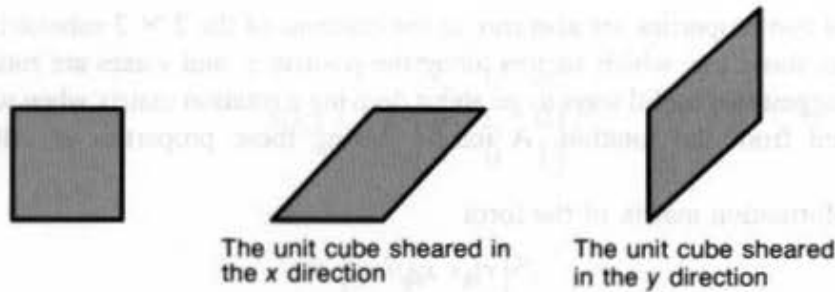
The unit cube sheared in
the x direction

The unit cube sheared
in the y direction

Fig. 5.7 The primitive-shear operations applied to the unit cube. In each case, the lengths of the oblique lines are now greater than 1.

Similarly, the matrix

$$SH_y = \begin{bmatrix} 1 & 0 & 0 \\ b & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (5.29)$$

shears along the y axis.

5.3 COMPOSITION OF 2D TRANSFORMATIONS

The idea of composition was introduced in the preceding section. Here, we use composition to combine the fundamental R, S, and T matrices to produce desired general results. The basic purpose of composing transformations is to gain efficiency by applying a single composed transformation to a point, rather than applying a series of transformations, one after the other.

Consider the rotation of an object about some arbitrary point P_1. Because we know how to rotate only about the origin, we convert our original (difficult) problem into three separate (easy) problems. Thus, to rotate about P_1, we need a sequence of three fundamental transformations:

1. Translate such that P_1 is at the origin
2. Rotate
3. Translate such that the point at the origin returns to P_1.

This sequence is illustrated in Fig. 5.8, in which our house is rotated about $P_1(x_1, y_1)$. The first translation is by $(-x_1, -y_1)$, whereas the later translation is by the inverse (x_1, y_1). The result is rather different from that of applying just the rotation.

The net transformation is

$$T(x_1, y_1) \cdot R(\theta) \cdot T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_1(1 - \cos\theta) + y_1\sin\theta \\ \sin\theta & \cos\theta & y_1(1 - \cos\theta) - x_1\sin\theta \\ 0 & 0 & 1 \end{bmatrix}. \qquad (5.30)$$
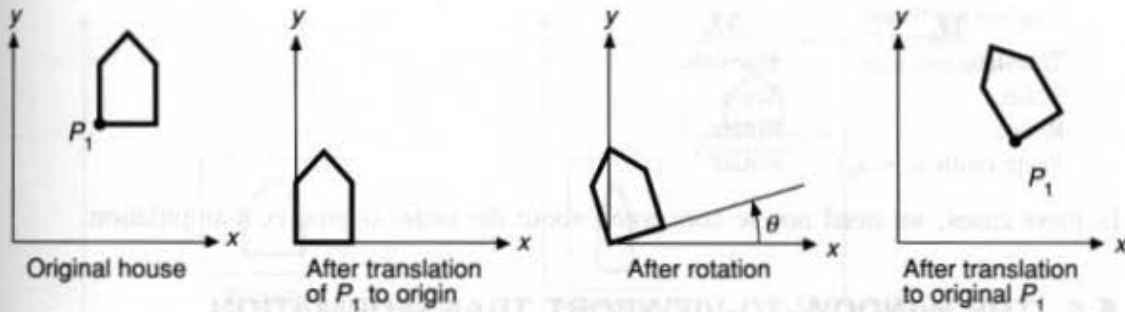
Fig. 5.8 Rotation of a house about the point P_1 by an angle θ.

A similar approach is used to scale an object about an arbitrary point P_1. First, translate such that P_1 goes to the origin, then scale, then translate back to P_1. In this case, the net transformation is

$$T(x_1, y_1) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1) = \begin{bmatrix} 1 & 0 & x_1 \\ 0 & 1 & y_1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_1 \\ 0 & 1 & -y_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} s_x & 0 & x_1(1 - s_x) \\ 0 & s_y & y_1(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix}. \qquad (5.31)$$

Suppose we wish to scale, rotate, and position the house shown in Fig. 5.9 with P_1 as the center for the rotation and scaling. The sequence is to translate P_1 to the origin, to perform the scaling and rotation, and then to translate from the origin to the new position P_2 where the house is to be placed. A data structure that records this transformation might contain the scale factor(s), rotation angle, and translation amounts, and the order in which the transformations were applied, or it might simply record the composite transformation matrix:

$$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1). \qquad (5.32)$$

If M_1 and M_2 each represent a fundamental translation, scaling, or rotation, when is $M_1 \cdot M_2 = M_2 \cdot M_1$? That is, when do M_1 and M_2 commute? In general, of course, matrix multiplication is *not* commutative. However, it is easy to show that, in the following special cases, commutativity holds:
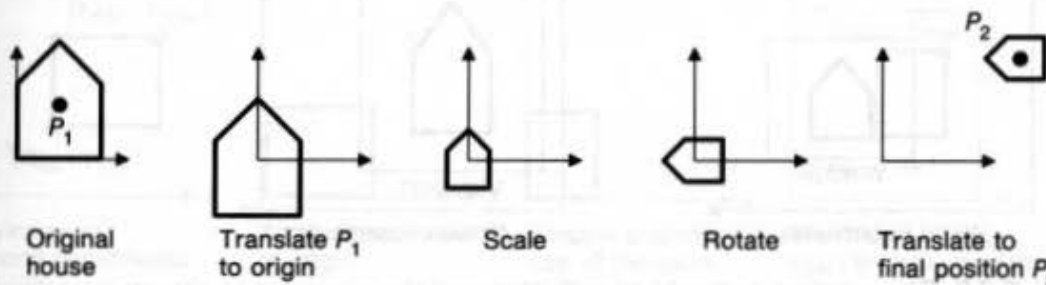
Fig. 5.9 Rotation of a house about the point P_1, and placement such that what was at P_1 is at P_2.

	M_1		M_2
	Translate		Translate
	Scale		Scale
	Rotate		Rotate
	Scale (with $s_x = s_y$)		Rotate

In these cases, we need not be concerned about the *order* of matrix manipulation.

5.4 THE WINDOW-TO-VIEWPORT TRANSFORMATION

Some graphics packages allow the programmer to specify output primitive coordinates in a floating-point *world-coordinate* system, using whatever units are meaningful to the application program: angstroms, microns, meters, miles, light-years, and so on. The term *world* is used because the application program is representing a world that is being interactively created or displayed to the user.

Given that output primitives are specified in world coordinates, the graphics subroutine package must be told how to map world coordinates onto screen coordinates (we use the specific term *screen coordinates* to relate this discussion specifically to SRGP, but that hardcopy output devices might be used, in which case the term *device coordinates* would be more appropriate). We could do this mapping by having the application programmer provide the graphics package with a transformation matrix to effect the mapping. Another way is to have the application programmer specify a rectangular region in world coordinates, called the *world-coordinate window*, and a corresponding rectangular region in screen coordinates, called the *viewport*, into which the world-coordinate window is to be mapped. The transformation that maps the window into the viewport is applied to all of the output primitives in world coordinates, thus mapping them into screen coordinates. Figure 5.10 shows this concept. As seen in this figure, if the window and viewport do not have the same height-to-width ratio, a *non*uniform scaling occurs. If the application program changes the window or viewport, then new output primitives drawn onto the screen will be affected by the change. Existing output primitives are not affected by such a change.

The modifier *world-coordinate* is used with *window* to emphasize that we are not discussing a *window-manager window*, which is a different and more recent concept, and
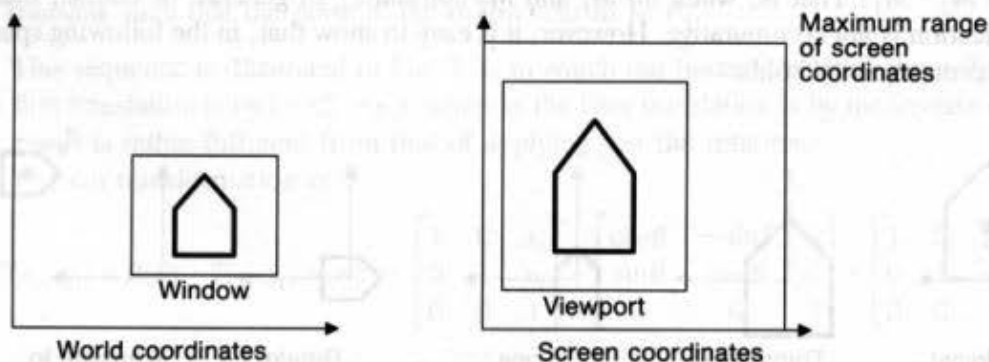

Fig. 5.10 The window in world coordinates and the viewport in screen coordinates determine the mapping that is applied to all the output primitives in world coordinates.
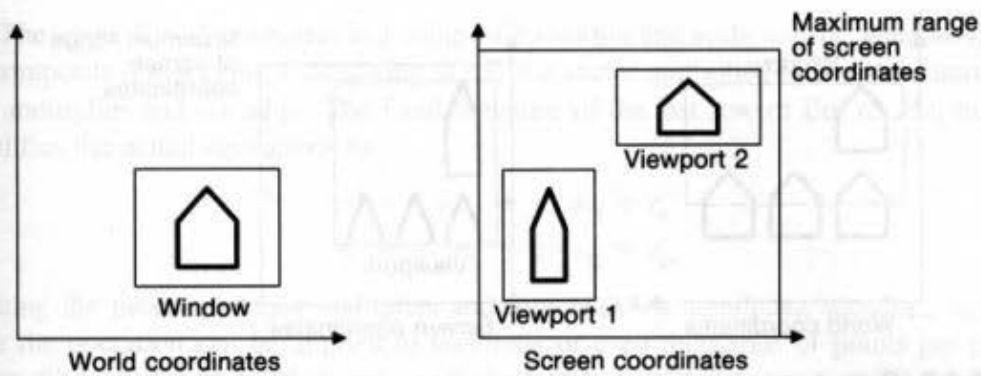
Fig. 5.11 The effect of drawing output primitives with two viewports. Output primitives specifying the house were first drawn with viewport 1, the viewport was changed to viewport 2, and then the application program again called the graphics package to draw the output primitives.

which unfortunately has the same name. Whenever there is no ambiguity as to which type of window is meant, we will drop the modifier.

If SRGP were to provide world-coordinate output primitives, the viewport would be on the current canvas, which defaults to canvas 0, the screen. The application program would be able to change the window or the viewport at any time, in which case subsequently specified output primitives would be subjected to a new transformation. If the change included a different viewport, then the new output primitves would be located on the canvas in positions different from those of the old ones, as shown in Fig. 5.11.

A window manager might map SRGP's canvas 0 into less than a full-screen window, in which case not all of the canvas or even of the viewport would necessarily be visible. In Chapter 10, we further discuss the relationships among world-coordinate windows, viewports, and window-manager windows.

Given a window and viewport, what is the transformation matrix that maps the window from world coordinates into the viewport in screen coordinates? This matrix can be developed as a three-step transformation composition, as suggested in Fig. 5.12. The
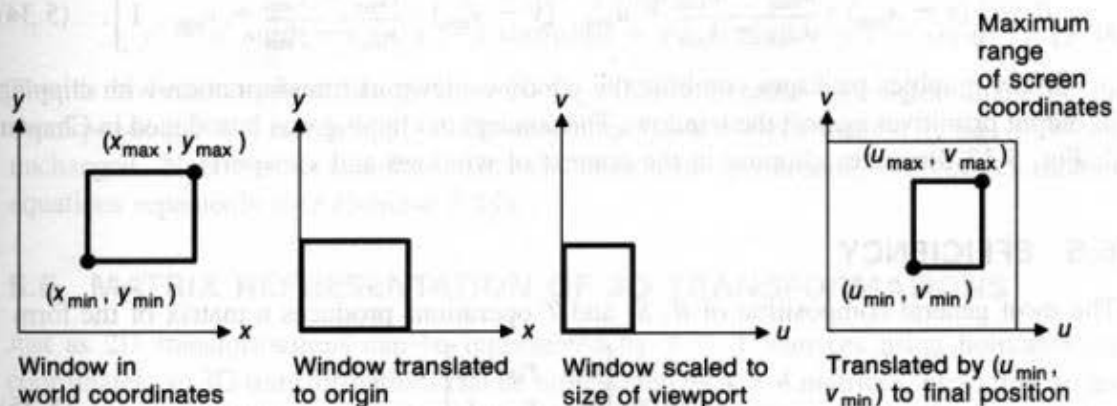
Fig. 5.12 The steps in transforming a world-coordinate window into a viewport.
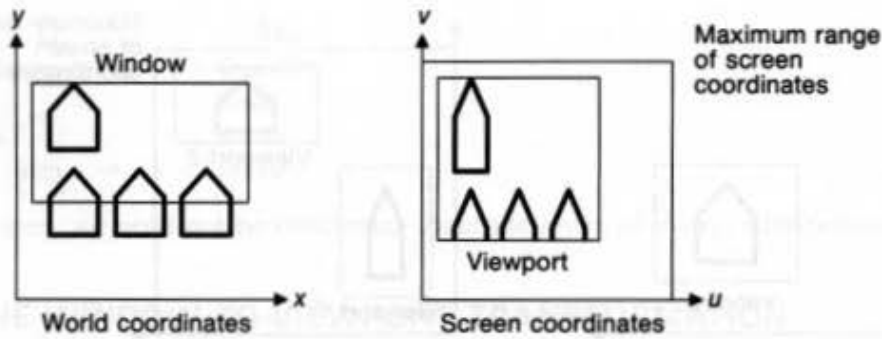
Fig. 5.13 Output primitives in world coordinates are clipped against the window. Those that remain are displayed in the viewport.

window, specified by its lower-left and upper-right corners, is first translated to the origin of world coordinates. Next, the size of the window is scaled to be equal to the size of the viewport. Finally, a translation is used to position the viewport. The overall matrix M_{wv} is:

$$M_{wv} = T(u_{min}, v_{min}) \cdot S\left(\frac{u_{max} - u_{min}}{x_{max} - x_{min}}, \frac{v_{max} - v_{min}}{y_{max} - y_{min}}\right) \cdot T(-x_{min}, -y_{min})$$

$$= \begin{bmatrix} 1 & 0 & u_{min} \\ 0 & 1 & v_{min} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \dfrac{u_{max} - u_{min}}{x_{max} - x_{min}} & 0 & 0 \\ 0 & \dfrac{v_{max} - v_{min}}{y_{max} - y_{min}} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_{min} \\ 0 & 1 & -y_{min} \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \dfrac{u_{max} - u_{min}}{x_{max} - x_{min}} & 0 & -x_{min} \cdot \dfrac{u_{max} - u_{min}}{x_{max} - x_{min}} + u_{min} \\ 0 & \dfrac{v_{max} - v_{min}}{y_{max} - y_{min}} & -y_{min} \cdot \dfrac{v_{max} - v_{min}}{y_{max} - y_{min}} + v_{min} \\ 0 & 0 & 1 \end{bmatrix} . \tag{5.33}$$

Multiplying $P = M_{wv} [x \quad y \quad 1]^T$ gives the expected result:

$$P = \left[(x - x_{min}) \cdot \frac{u_{max} - u_{min}}{x_{max} - x_{min}} + u_{min} \quad (y - y_{min}) \cdot \frac{v_{max} - v_{min}}{y_{max} - y_{min}} + v_{min} \quad 1 \right]. \tag{5.34}$$

Many graphics packages combine the window–viewport transformation with clipping of output primitives against the window. The concept of clipping was introduced in Chapter 3; Fig. 5.13 illustrates clipping in the context of windows and viewports.

5.5 EFFICIENCY

The most general composition of R, S, and T operations produces a matrix of the form

$$M = \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} . \tag{5.35}$$

The upper 2×2 submatrix is a composite rotation and scale matrix, whereas t_x and t_y are composite translations. Calculating $M \cdot P$ as a vector multiplied by a 3×3 matrix takes nine multiplies and six adds. The fixed structure of the last row of Eq. (5.35), however, simplifies the actual operations to

$$x' = x \cdot r_{11} + y \cdot r_{12} + t_x \qquad (5.36)$$
$$y' = x \cdot r_{21} + y \cdot r_{22} + t_y,$$

reducing the process to four multiplies and four adds—a significant speedup, especially since the operation can be applied to hundreds or even thousands of points per picture. Thus, although 3×3 matrices are convenient and useful for composing 2D transformations, we can use the final matrix most efficiently in a program by exploiting its special structure. Some hardware matrix multipliers have parallel adders and multipliers, thereby diminishing or removing this concern.

Another area where efficiency is important is creating successive views of an object, such as a molecule or airplane, rotated a few degrees between each successive view. If each view can be created and displayed quickly enough (30 to 100 milliseconds each), then the object will appear to be rotating dynamically. To achieve this speed, we must transform each individual point and line of the object as quickly as possible. The rotation equations (Eq. (5.6)) require four multiplies and two adds. We can decrease the operation count by recognizing that, because θ is small (just a few degrees), $\cos\theta$ is very close to 1. In this approximation, Eq. (5.6) becomes

$$x' = x - y \sin\theta, \qquad y' = x \sin\theta + y, \qquad (5.37)$$

which requires just two multiplies and two adds. The savings of two multiplies can be significant on computers lacking hardware multipliers.

Equation (5.37), however, is only an approximation to the correct values of x' and y': a small error is built in. Each time the formulae are applied to the new values of x and y, the error gets a bit larger. If we repeat the formulae indefinitely, the error will overwhelm the correct values, and the rotating image will begin to look like a collection of randomly drawn lines.

A better approximation is to use x' instead of x in the second equation:

$$x' = x - y \sin\theta,$$
$$y' = x' \sin\theta + y = (x - y \sin\theta)\sin\theta + y = x \sin\theta + y(1 - \sin^2\theta) \qquad (5.38)$$

This is a better approximation than is Eq. (5.37) because the determinant of the corresponding 2×2 matrix is 1, which means that the areas transformed by Eq. (5.38) are unchanged. Note that cumulative errors can also arise when using the correct rotation equations repeatedly (see Exercise 5.19).

5.6 MATRIX REPRESENTATION OF 3D TRANSFORMATIONS

Just as 2D transformations can be represented by 3×3 matrices using homogeneous coordinates, so 3D transformations can be represented by 4×4 matrices, providing we use homogeneous coordinate representations of points in 3-space as well. Thus, instead of representing a point as (x, y, z), we represent it as (x, y, z, W), where two of these

quadruples represent the same point if one is a nonzero multiple of the other; the quadruple $(0, 0, 0, 0)$ is not allowed. As in 2D, a standard representation of a point (x, y, z, W) with $W \neq 0$ is given by $(x/W, y/W, z/W, 1)$. Transforming the point to this form is called *homogenizing*, as before. Also, points whose W coordinate is zero are called points at infinity. There is a geometrical interpretation as well. Each point in 3-space is being represented by a line through the origin in 4-space, and the homogenized representations of these points form a 3D subspace of 4-space which is defined by the single equation $W = 1$.

The 3D coordinate system used in this text is *right-handed*, as shown in Fig. 5.14. By convention, positive rotations in a right-handed system are such that, when looking from a positive axis toward the origin, a 90° *counterclockwise* rotation will transform one positive axis into the other. This table follows from this convention:

If axis of rotation is	Direction of positive rotation is
x	y to z
y	z to x
z	x to y

These positive directions are also depicted in Fig. 5.14. The reader is warned that not all graphics texts follow this convention.

We use a right-handed system here because it is the standard mathematical convention, even though it is convenient in 3D graphics to think of a left-handed system superimposed on the face of a display (see Fig. 5.15), since a left-handed system gives the natural interpretation that larger z values are further from the viewer. Notice that, in a left-handed system, positive rotations are *clockwise* when looking from a positive axis toward the origin. This definition of positive rotations allows the same rotation matrices given in this section to be used for either right- or left-hand coordinate systems. Conversion from right to left and left to right is discussed in Section 5.8.

Translation in 3D is a simple extension from that in 2D:

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.39}$$

That is, $T(d_x, d_y, d_z) \cdot [x \quad y \quad z \quad 1]^T = [x + d_x \quad y + d_y \quad z + d_z \quad 1]^T$.
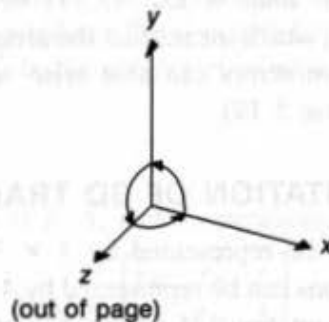

(out of page)
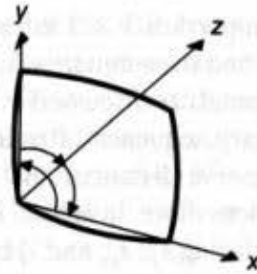
Fig. 5.14 The right-handed coordinate system.

Fig. 5.15 The left-handed coordinate system, with a superimposed display screen.

Scaling is similarly extended:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.40}$$

Checking, we see that $S(s_x, s_y, s_z) \cdot [x \quad y \quad z \quad 1]^T = [s_x \cdot x \quad s_y \cdot y \quad s_z \cdot z \quad 1]^T$.
The 2D rotation of Eq. (5.26) is just a 3D rotation about the z axis, which is

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.41}$$

This is easily verified: A 90° rotation of $[1\ 0\ 0\ 1]^T$, which is the unit vector along the x axis, should produce the unit vector $[0\ 1\ 0\ 1]^T$ along the y axis. Evaluating the product

$$\begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{5.42}$$

gives the predicted result of $[0\ 1\ 0\ 1]^T$.
The x-axis rotation matrix is

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.43}$$

The y-axis rotation matrix is

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.44}$$

The columns (and the rows) of the upper-left 3×3 submatrix of $R_z(\theta)$, $R_x(\theta)$, and $R_y(\theta)$ are mutually perpendicular unit vectors and the submatrix has a determinant of 1, which means the three matrices are special orthogonal, as discussed in Section 5.2. Also, the upper-left 3×3 submatrix formed by an arbitrary sequence of rotations is special orthogonal. Recall that orthogonal transformations preserve distances and angles.

All these transformation matrices have inverses. The inverse for T is obtained by negating d_x, d_y, and d_z; for S, by replacing s_x, s_y, and s_z by their reciprocals; that for each of the three rotation matrices is obtained by negating the angle of rotation.

The inverse of any orthogonal matrix B is just B's transpose: $B^{-1} = B^T$. In fact, taking the transpose does not need to involve even exchanging elements in the array that stores the matrix—it is necessary only to exchange row and column indexes when accessing the array. Notice that this method of finding an inverse is consistent with the result of negating θ to find the inverse of R_z, R_y, and R_z.

Any number of rotation, scaling, and translation matrices can be multiplied together. The result always has the form

$$M = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.45}$$

As in the 2D case, the 3×3 upper-left submatrix R gives the aggregate rotation and scaling, whereas T gives the subsequent aggregate translation. Some computational efficiency is achieved by performing the transformation explicitly as

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = R \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + T, \tag{5.46}$$

where R and T are submatrices from Eq. (5.45).

Corresponding to the two-dimensional shear matrices in Section 5.2 are three 3D shear matrices. The (x, y) shear is

$$SH_{xy}(sh_x, sh_y) = \begin{bmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.47}$$

Applying SH_{xy} to the point $[x \quad y \quad z \quad 1]^T$, we have $[x + sh_x \cdot z \quad y + sh_y \cdot z \quad z \quad 1]^T$. Shears along the x and y axes have a similar form.

So far, we have focused on transforming individual points. We transform lines, these being defined by two points, by transforming the endpoints. Planes, if they are defined by three points, may be handled the same way, but usually they are defined by a plane equation, and the coefficients of this plane equation must be transformed differently. We may also need to transform the plane normal. Let a plane be represented as the column vector of plane-equation coefficients $N = [A \quad B \quad C \quad D]^T$. Then a plane is defined by all points P such that $N \cdot P = 0$, where $\cdot$ is the vector dot product and $P = [x \quad y \quad z \quad 1]^T$. This dot product gives rise to the familiar plane equation $Ax + By + Cz + D = 0$, which

can also be expressed as the product of the row vector of plane-equation coefficients times the column vector P: $N^T \cdot P = 0$. Now suppose we transform all points P on the plane by some matrix M. To maintain $N^T \cdot P = 0$ for the transformed points, we would like to transform N by some (to be determined) matrix Q, giving rise to the equation $(Q \cdot N)^T \cdot M \cdot P = 0$. This expression can in turn be rewritten as $N^T \cdot Q^T \cdot M \cdot P = 0$, using the identity $(Q \cdot N)^T = N^T \cdot Q^T$. The equation will hold if $Q^T \cdot M$ is a multiple of the identity matrix. If the multiplier is 1, this leads to $Q^T = M^{-1}$, or $Q = (M^{-1})^T$. This means that the column vector N' of coefficients for a plane transformed by M is given by

$$N' = (M^{-1})^T \cdot N.$$

The matrix $(M^{-1})^T$ need not in general exist, because the determinant of M might be zero. This would happen if M includes a projection (we might want to investigate the effect of a perspective projection on a plane). It is possible to use, instead of $(M^{-1})^T$, the matrix of cofactors of M used in finding the inverse of M using Cramer's rule. See the Appendix for more details.

If just the normal of the plane is to be transformed (for example, to perform the shading calculations discussed in Chapter 16) and if M consists of only the composition of translation, rotation, and uniform scaling matrices, then the mathematics is even simpler. The N' of Eq. (5.48) can be simplified to $[A'\ B'\ C'\ 0]^T$. (With a zero W component, a homogeneous point represents a point at infinity, which can be thought of as a direction.)

5.7 COMPOSITION OF 3D TRANSFORMATIONS

In this section, we discuss how to compose 3D transformation matrices, using an example that will be useful in Section 6.4. The objective is to transform the directed line segments P_1P_2 and P_1P_3 in Fig. 5.16 from their starting position in part (a) to their ending position in part (b). Thus, point P_1 is to be translated to the origin, P_1P_2 is to lie on the positive z axis, and P_1P_3 is to lie in the positive y axis half of the (y, z) plane. The lengths of the lines are to be unaffected by the transformation.

Two ways to achieve the desired transformation are presented. The first approach is to compose the primitive transformations T, R_z, R_y, and R_z. This approach, although somewhat tedious, is easy to illustrate, and understanding it will help us to build an understanding of transformations. The second approach, using the properties of orthogonal matrices described in the previous section, is explained more briefly but is more abstract.
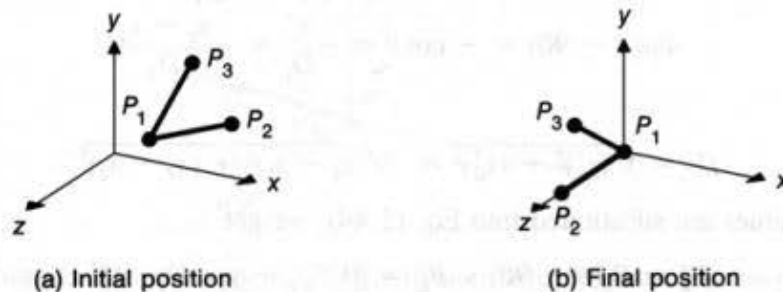

(a) Initial position   (b) Final position

Fig. 5.16 Transforming P_1, P_2, and P_3 from their initial (a) to their final (b) position.

To work with the primitive transformations, we again break a difficult problem into simpler subproblems. In this case, the desired transformation can be done in four steps:

1. Translate P_1 to the origin
2. Rotate about the y axis such that P_1P_2 lies in the (y, z) plane
3. Rotate about the x axis such that P_1P_2 lies on the z axis
4. Rotate about the z axis such that P_1P_3 lies in the (y, z) plane.

Step 1: Translate P_1 to the origin.   The translation is

$$T(-x_1, -y_1, -z_1) = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.49}$$

Applying T to P_1, P_2, and P_3 gives

$$P'_1 = T(-x_1, -y_1, -z_1) \cdot P_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \tag{5.50}$$

$$P'_2 = T(-x_1, -y_1, -z_1) \cdot P_2 = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \\ 1 \end{bmatrix}, \tag{5.51}$$

$$P'_3 = T(-x_1, -y_1, -z_1) \cdot P_3 = \begin{bmatrix} x_3 - x_1 \\ y_3 - y_1 \\ z_3 - z_1 \\ 1 \end{bmatrix}, \tag{5.52}$$

Step 2: Rotate about the y axis.   Figure 5.17 shows P_1P_2 after step 1, along with the projection of P_1P_2 onto the (x, z) plane. The angle of rotation is $-(90 - \theta) = \theta - 90$. Then

$$\cos(\theta - 90) = \sin\theta = \frac{z'_2}{D_1} = \frac{z_2 - z_1}{D_1},$$

$$\sin(\theta - 90) = -\cos\theta = -\frac{x'_2}{D_1} = -\frac{x_2 - x_1}{D_1}, \tag{5.53}$$

where

$$D_1 = \sqrt{(z'_2)^2 + (x'_2)^2} = \sqrt{(z_2 - z_1)^2 + (x_2 - x_1)^2}. \tag{5.54}$$

When these values are substituted into Eq. (5.44), we get

$$P''_2 = R_y(\theta - 90) \cdot P'_2 = [0 \quad y_2 - y_1 \quad D_1 \quad 1]^T. \tag{5.55}$$

As expected, the x component of P''_2 is zero, and the z component is the length D_1.
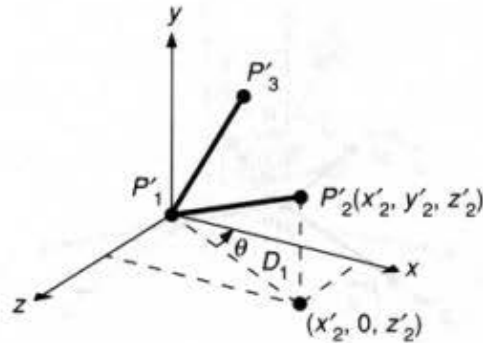
Fig. 5.17 Rotation about the y axis: The projection of $P_1'P_2'$, which has length D_1, is rotated into the z axis. The angle θ shows the positive direction of rotation about the y axis: The actual angle used is $-(90 - \theta)$.

Step 3: Rotate about the x axis. Figure 5.18 shows P_1P_2 after step 2. The angle of rotation is ϕ, for which

$$\cos\phi = \frac{z_2''}{D_2}, \quad \sin\phi = \frac{y_2''}{D_2}, \tag{5.56}$$

where $D_2 = |P_1''P_2''|$, the length of the line $P_1''P_2''$. But the length of line $P_1''P_2''$ is the same as the length of line P_1P_2, because rotation and translation transformations preserve length, so

$$D_2 = |P_1''P_2''| = |P_1P_2| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}. \tag{5.57}$$

The result of the rotation in step 3 is

$$
\begin{aligned}
P_2''' &= R_x(\phi) \cdot P_2'' = R_x(\phi) \cdot R_y(\theta - 90) \cdot P_2' \\
&= R_x(\phi) \cdot R_y(\theta - 90) \cdot T \cdot P_2 = [0 \quad 0 \quad |P_1P_2| \quad 1]^T. \tag{5.58}
\end{aligned}
$$

That is, P_1P_2 now coincides with the positive z axis.

Step 4: Rotate about the z axis. Figure 5.19 shows P_1P_2 and P_1P_3 after step 3, with P_2''' on the z axis and P_3''' at the position

$$P_3''' = [x_3''' \quad y_3''' \quad z_3''' \quad 1]^T = R_x(\phi) \cdot R_y(\theta - 90) \cdot T(-x_1, -y_1, -z_1) \cdot P_3. \tag{5.59}$$
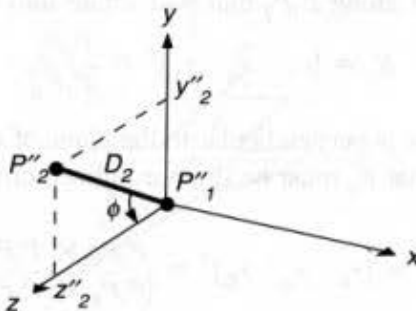
Fig. 5.18 Rotation about the x axis: $P_1''P_2''$ is rotated into the z axis by the positive angle ϕ. D_2 is the length of the line segment. The line segment $P_1''P_3''$ is not shown, because it is not used to determine the angles of rotation. Both lines are rotated by $R_x(\phi)$.
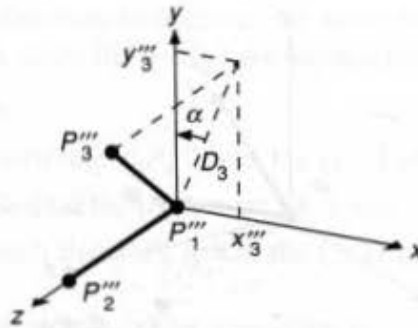
Fig. 5.19 Rotation about the z axis: The projection of $P_1'P_3'$, whose length is D_3, is rotated by the positive angle α into the y axis, bringing the line itself into the (y, z) plane. D_3 is the length of the projection.

The rotation is through the positive angle α, with

$$\cos\alpha = y_3'''/D_3, \quad \sin\alpha = x_3'''/D_3, \quad D_3 = \sqrt{x_3'''^2 + y_3'''^2}. \quad (5.60)$$

Step 4 achieves the result shown in Fig. 5.16(b).

The composite matrix

$$R_z(\alpha) \cdot R_z(\phi) \cdot R_y(\theta - 90) \cdot T(-x_1, -y_1, -z_1) = R \cdot T \quad (5.61)$$

is the required transformation, with $R = R_z(\alpha) \cdot R_z(\phi) \cdot R_y(\theta - 90)$. We leave it to the reader to apply this transformation to P_1, P_2, and P_3 to verify that P_1 is transformed to the origin, P_2 is transformed to the positive z axis, and P_3 is transformed to the positive y half of the (y, z) plane.

The second way to obtain the matrix R is to use the properties of orthogonal matrices discussed in Section 5.2. Recall that the unit row vectors of R rotate into the principal axes. Replacing the second subscripts of Eq. (5.45) with x, y, and z for notational convenience

$$R = \begin{bmatrix} r_{1_x} & r_{2_x} & r_{3_x} \\ r_{1_y} & r_{2_y} & r_{3_y} \\ r_{1_z} & r_{2_z} & r_{3_z} \end{bmatrix}. \quad (5.62)$$

Because R_z is the unit vector along P_1P_2 that will rotate into the positive z axis,

$$R_z = [r_{1_z} \quad r_{2_z} \quad r_{3_z}]^T = \frac{P_1P_2}{|P_1P_2|}. \quad (5.63)$$

In addition, the R_x unit vector is perpendicular to the plane of P_1, P_2, and P_3 and will rotate into the positive x axis, so that R_x must be the normalized cross-product of two vectors in the plane:

$$R_x = [r_{1_x} \quad r_{2_x} \quad r_{3_x}]^T = \frac{P_1P_3 \times P_1P_2}{|P_1P_3 \times P_1P_2|}. \quad (5.64)$$

Finally,

$$R_y = [r_{1_y} \quad r_{2_y} \quad r_{3_y}]^T = R_z \times R_x \quad (5.65)$$
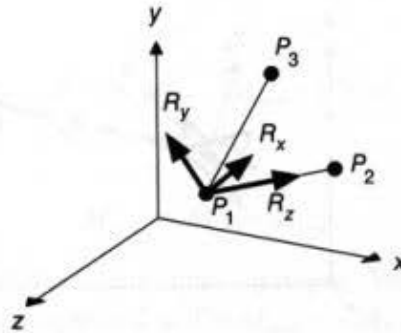
Fig. 5.20 The unit vectors R_x, R_y, and R_z, which are transformed into the principal axes.

will rotate into the positive y axis. The composite matrix is given by

$$
\begin{bmatrix}
r_{1x} & r_{2x} & r_{3x} & 0 \\
r_{1y} & r_{2y} & r_{3y} & 0 \\
r_{1z} & r_{2z} & r_{3z} & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \cdot T(-x_1, -y_1, -z_1) = R \cdot T, \tag{5.66}
$$

where R and T are as in Eq. (5.61). Figure 5.20 shows the individual vectors R_x, R_y, and R_z.

Let's consider another example. Figure 5.21 shows an airplane defined in the x_p, y_p, z_p coordinate system and centered at the origin. We want to transform the airplane so that it heads in the direction given by the vector DOF (direction of flight), is centered at P, and is not banked, as shown in Fig. 5.22. The transformation to do this consists of a rotation to head the airplane in the proper direction, followed by a translation from the origin to P. To find the rotation matrix, we just determine in what direction each of the x_p, y_p, and z_p axes is heading in Fig. 5.22, make sure the directions are normalized, and use them as column vectors in a rotation matrix.

The z_p axis must be transformed to the DOF direction, and the x_p axis must be transformed into a horizontal vector perpendicular to DOF—that is, in the direction of $y \times DOF$, the cross-product of y and DOF. The y_p direction is given by $z_p \times x_p = DOF \times (y \times$
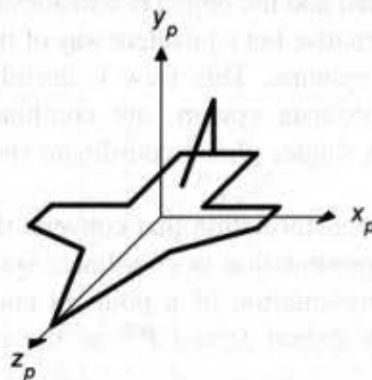
Fig. 5.21 An airplane in the (x_p, y_p, z_p) coordinate system.
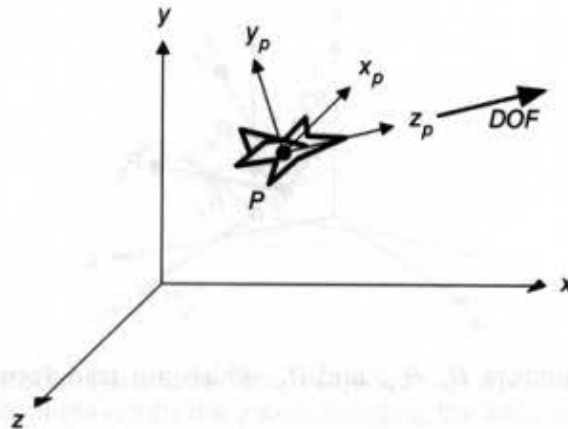
Fig. 5.22 The airplane of Figure 5.21 positioned at point P, and headed in direction DOF.

DOF), the cross-product of z_p and x_p; hence, the three columns of the rotation matrix are the normalized vectors $|y \times DOF|$, $|DOF \times (y \times DOF)|$, and $|DOF|$:

$$R = \begin{bmatrix} & & & 0 \\ |y \times DOF| & |DOF \times (y \times DOF)| & |DOF| & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (5.67)$$

The situation if DOF is in the direction of the y axis is degenerate, because there is an infinite set of possible vectors for the horizontal vector. This degeneracy is reflected in the algebra, because the cross-products $y \times DOF$ and $DOF \times (y \times DOF)$ are zero. In this special case, R is not a rotation matrix.

5.8 TRANSFORMATIONS AS A CHANGE IN COORDINATE SYSTEM

We have been discussing transforming a set of points belonging to an object into another set of points, when both sets are in the same coordinate system. With this approach, the coordinate system stays unaltered and the object is transformed with respect to the origin of the coordinate system. An alternative but equivalent way of thinking about a transformation is as a change of coordinate systems. This view is useful when multiple objects, each defined in its own local coordinate system, are combined, and we wish to express these objects' coordinates in a single, global coordinate system. This will be the case in Chapter 7.

Let us define $M_{i \leftarrow j}$ as the transformation that converts the representation of a point in coordinate system j into its representation in coordinate system i.

We define $P^{(i)}$ as the representation of a point in coordinate system i, $P^{(j)}$ as the representation of the point in system j, and $P^{(k)}$ as the representation of the point in coordinate system k; then,

$$P^{(i)} = M_{i \leftarrow j} \cdot P^{(j)} \text{ and } P^{(j)} = M_{j \leftarrow k} \cdot P^{(k)}. \quad (5.68)$$

Substituting, we obtain

$$P^{(i)} = M_{i\leftarrow j} \cdot P^{(j)} = M_{i\leftarrow j} \cdot M_{j\leftarrow k} \cdot P^{(k)} = M_{i\leftarrow k} \cdot P^{(k)}, \tag{5.69}$$

so

$$M_{i\leftarrow k} = M_{i\leftarrow j} \cdot M_{j\leftarrow k}. \tag{5.70}$$

Figure 5.23 shows four different coordinate systems. We see by inspection that the transformation from coordinate system 2 to 1 is $M_{1\leftarrow 2} = T(4, 2)$ (finding this transformation by inspection is not always simple—see the Appendix). Similarly, $M_{2\leftarrow 3} = T(2, 3) \cdot S(0.5, 0.5)$ and $M_{3\leftarrow 4} = T(6.7, 1.8) \cdot R(-45°)$. Then $M_{1\leftarrow 3} = M_{1\leftarrow 2} \cdot M_{2\leftarrow 3} = T(4, 2) \cdot T(2, 3) \cdot S(0.5, 0.5)$. The figure also shows a point that is $P^{(1)} = (10, 8)$, $P^{(2)} = (6, 6)$, $P^{(3)} = (8, 6)$, and $P^{(4)} = (4, 2)$ in coordinate systems 1 through 4, respectively. It is easy to verify that $P^{(i)} = M_{i\leftarrow j} \cdot P^{(j)}$ for $1 \le i, j \le 4$.

We also notice that $M_{i\leftarrow j} = M_{j\leftarrow i}^{-1}$. Thus, $M_{2\leftarrow 1} = M_{1\leftarrow 2}^{-1} = T(-4, -2)$. Because $M_{1\leftarrow 3} = M_{1\leftarrow 2} \cdot M_{2\leftarrow 3}$, $M_{1\leftarrow 3}^{-1} = M_{2\leftarrow 3}^{-1} \cdot M_{1\leftarrow 2}^{-1} = M_{3\leftarrow 2} \cdot M_{2\leftarrow 1}$.

In Section 5.6, we discussed left- and right-handed coordinate systems. The matrix that converts from points represented in one to points represented in the other is its own inverse, and is

$$M_{R\leftarrow L} = M_{L\leftarrow R} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.71}$$

The approach used in previous sections—defining all objects in the world-coordinate system, then transforming them to the desired place—implies the somewhat unrealistic notion that all objects are initially defined on top of one another in the same world-coordinate system. It is more natural to think of each object as being defined in its own coordinate system and then being scaled, rotated, and translated by redefinition of its coordinates in the new world-coordinate system. In this second point of view, one thinks naturally of separate pieces of paper, each with an object on it, being shrunk or stretched, rotated or placed on the world-coordinate plane. One can also, of course, imagine that the
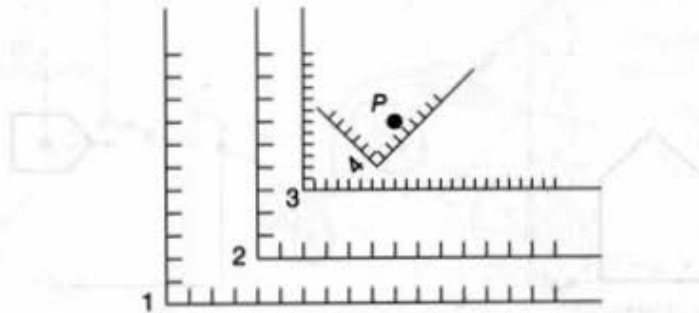

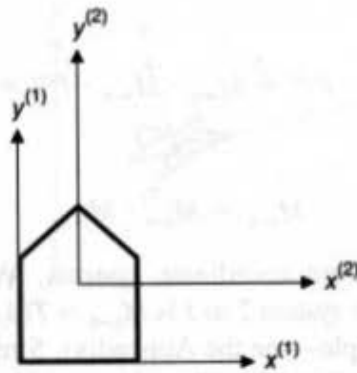**Fig. 5.23** The point P and coordinate systems 1, 2, 3, and 4.

Fig. 5.24 The house and two coordinate systems. Coordinates of points on the house can be represented in either coordinate system.

plane is being shrunk or stretched, tilted, or slid relative to each piece of paper. Mathematically, all these views are identical.

Consider the simple case of translating the set of points that define the house shown in Fig. 5.8 to the origin. This transformation is $T(-x_1, -y_1)$. Labeling the two coordinate systems as in Fig. 5.24, we see that the transformation that maps coordinate system 1 into 2—that is, $M_{2\leftarrow1}$—is $T(x_1, y_1)$, which is just $T(-x_1, -y_1)^{-1}$. Indeed, the general rule is that the transformation that transforms a set of points in a single coordinate system is just the inverse of the corresponding transformation to change the coordinate system in which a point is represented. This relation can be seen in Fig. 5.25, which is derived directly from Fig. 5.9. The transformation for the points represented in a single coordinate system is just

$$T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1). \tag{5.32}$$

In Fig. 5.25, the coordinate-system transformation is just

$$\begin{aligned} M_{5\leftarrow1} &= M_{5\leftarrow4}\, M_{4\leftarrow3}\, M_{3\leftarrow2}\, M_{2\leftarrow1} \\ &= (T(x_2, y_2) \cdot R(\theta) \cdot S(s_x, s_y) \cdot T(-x_1, -y_1))^{-1} \\ &= T(x_1, y_1) \cdot S(s_x^{-1}, s_y^{-1}) \cdot R(-\theta) \cdot T(-x_2, -y_2), \tag{5.72} \end{aligned}$$

so that

$$P^{(5)} = M_{5\leftarrow1}P^{(1)} = T(x_1, y_1) \cdot S(s_x^{-1}, s_y^{-1}) \cdot R(-\theta) \cdot T(-x_2, -y_2) \cdot P^{(1)}. \tag{5.73}$$

(a)   (b)
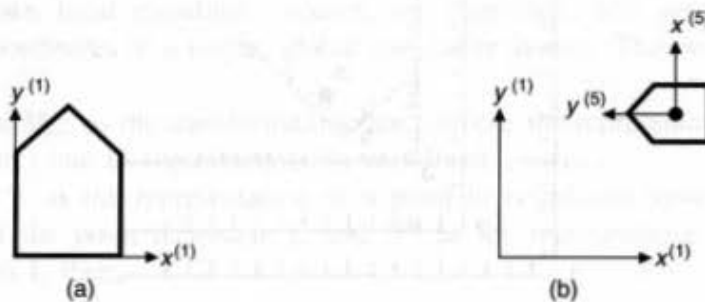
Fig. 5.25 The original house (a) in its coordinate system and the transformed house (b) in its coordinate system with respect to the original coordinate system.

An important question related to changing coordinate systems is changing transformations. Suppose $Q^{(j)}$ is a transformation in coordinate system j. It might, for example, be one of the composite transformations derived in previous sections. Suppose we wanted to find the transformation $Q^{(i)}$ in coordinate system i that could be applied to points $P^{(i)}$ in system i and produce exactly the same results as though $Q^{(j)}$ were applied to the corresponding points $P^{(j)}$ in system j. This equality is represented by $Q^{(i)} \cdot P^{(i)} = M_{i \leftarrow j} \cdot Q^{(j)} \cdot P^{(j)}$. Substituting $P^{(i)} = M_{i \leftarrow j} \cdot P^{(j)}$, this expression becomes $Q^{(i)} \cdot M_{i \leftarrow j} \cdot P^{(j)} = M_{i \leftarrow j} \cdot Q^{(j)} \cdot P^{(j)}$. Simplifying, we have $Q^{(i)} = M_{i \leftarrow j} \cdot Q^{(j)} \cdot M_{i \leftarrow j}^{-1}$.

The change-of-coordinate-system point of view is useful when additional information for subobjects is specified in the latters' own local coordinate systems. For example, if the front wheel of the tricycle in Fig. 5.26 is made to rotate about its z_{wh} coordinate, all wheels must be rotated appropriately, and we need to know how the tricycle as a whole moves in the world-coordinate system. This problem is complex because several successive changes of coordinate systems occur. First, the tricycle and front-wheel coordinate systems have initial positions in the world-coordinate system. As the bike moves forward, the front wheel rotates about the z axis of the wheel-coordinate system, and simultaneously the wheel- and tricycle-coordinate systems move relative to the world-coordinate system. The wheel- and tricycle-coordinate systems are related to the world-coordinate system by time-varying translations in x and z plus a rotation about y. The tricycle- and wheel-coordinate systems are related to each other by a time-varying rotation about y as the handlebars are turned. (The tricycle-coordinate system is fixed to the frame, not to the handlebars.)

To make the problem a bit easier, we assume that the wheel and tricycle axes are parallel to the world-coordinate axes and that the wheel moves in a straight line parallel to the world-coordinate x axis. As the wheel rotates by an angle α, a point P on the wheel rotates through the distance αr, where r is the radius of the wheel. Since the wheel is on the ground, the tricycle moves forward αr units. Therefore, the rim point P on the wheel moves and rotates with respect to the initial wheel-coordinate system with a net effect of translation by αr and rotation by α. Its new coordinates P' in the original wheel-coordinate system are thus

$$P'^{(wh)} = T(\alpha r, 0, 0) \cdot R_z(\alpha) \cdot P^{(wh)}, \qquad (5.74)$$
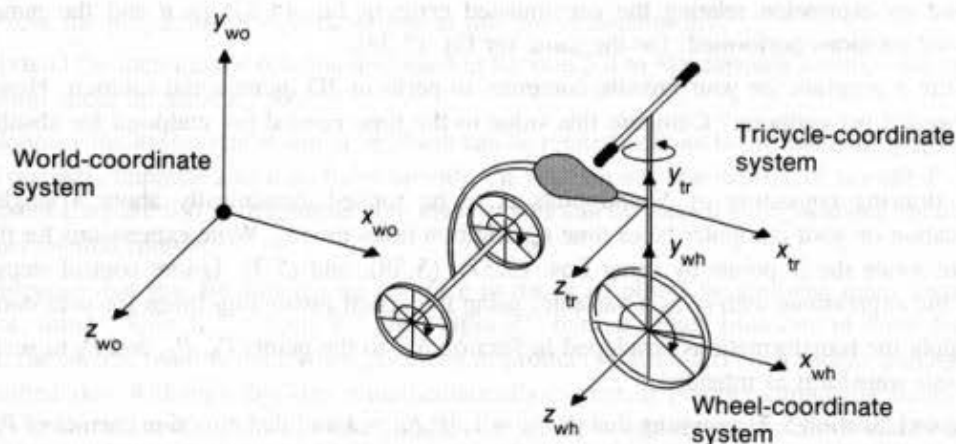

Fig. 5.26 A stylized tricycle with three coordinate systems.

and its coordinates in the new (translated) wheel-coordinate system are given by just the rotation

$$P'^{(wh')} = R_z(\alpha) \cdot P^{(wh)}. \tag{5.75}$$

To find the points $P^{(wo)}$ and $P'^{(wo)}$ in the world-coordinate system, we transform from the wheel to the world-coordinate system:

$$P^{(wo)} = M_{wo\leftarrow wh} \cdot P^{(wh)} = M_{wo\leftarrow tr} \cdot M_{tr\leftarrow wh} \cdot P^{(wh)}. \tag{5.76}$$

$M_{wo\leftarrow tr}$ and $M_{tr\leftarrow wh}$ are translations given by the initial positions of the tricycle and wheel. $P'^{(wo)}$ is computed with Eqs. (5.74) and (5.76):

$$P'^{(wo)} = M_{wo\leftarrow wh} \cdot P'^{(wh)} = M_{wo\leftarrow wh} \cdot T(\alpha r, 0, 0) \cdot R_z(\alpha) \cdot P^{(wh)}. \tag{5.77}$$

Alternatively, we recognize that $M_{wo\leftarrow wh}$ has been changed to $M_{wo\leftarrow wh'}$ by the translation of the wheel-coordinate system, and get the same result as Eq. (5.77), but in a different way:

$$P'^{(wo)} = M_{wo\leftarrow wh'} \cdot P'^{(wh')} = (M_{wo\leftarrow wh} \cdot M_{wh\leftarrow wh'}) \cdot (R_z(\alpha) \cdot P^{wh}). \tag{5.78}$$

In general, then, we derive the new $M_{wo\leftarrow wh'}$ and $M_{tr\leftarrow wh'}$ from their previous values by applying the appropriate transformations from the equations of motion of the tricycle parts. We then apply these updated transformations to updated points in local coordinate systems and derive the equivalent points in world-coordinate systems. We leave to the reader the problem of turning the tricycle's front wheel to change direction and of computing rotation angles for the rear wheels, using the wheels' radius and the tricycle's trajectory.

EXERCISES

5.1 Prove that we can transform a line by transforming its endpoints and then constructing a new line between the transformed endpoints.

5.2 Prove that two successive 2D rotations are additive: $R(\theta_1) \cdot R(\theta_2) = R(\theta_1 + \theta_2)$.

5.3 Prove that 2D rotation and scaling commute if $s_x = s_y$ or if $\theta = n\pi$ for integral n, and that otherwise they do not.

5.4 Find an expression relating the accumulated error in Eq. (5.37) to θ and the number of incremental rotations performed. Do the same for Eq. (5.38).

5.5 Write a program for your favorite computer to perform 2D incremental rotation. How much time is needed per endpoint? Compare this value to the time needed per endpoint for absolute 2D rotation.

5.6 A drawing consisting of N endpoints is to be rotated dynamically about a single axis. Multiplication on your computer takes time t_m; addition takes time t_a. Write expressions for the time needed to rotate the N points by using Eqs. (5.37), (5.38), and (5.7). Ignore control steps. Now evaluate the expressions with N as a variable, using the actual instruction times for your computer.

5.7 Apply the transformations developed in Section 5.7 to the points P_1, P_2, and P_3 to verify that these points transform as intended.

5.8 Rework Section 5.7, assuming that $|P_1P_2| = 1$, $|P_1P_3| = 1$ and that direction cosines of P_1P_2 and P_1P_3 are given (direction cosines of a line are the cosines of the angles between the line and the x, y, and z axes). For a line from the origin to (x, y, z), the direction cosines are $(x/d, y/d, z/d)$, where d is the length of the line.

5.9 Another reason that homogeneous coordinates are attractive is that 3D points at infinity in Cartesian coordinates can be represented explicitly in homogeneous coordinates. How can this be done?

5.10 Show that Eqs. (5.61) and (5.66) are equivalent.

5.11 Given a unit cube with one corner at $(0, 0, 0)$ and the opposite corner at $(1, 1, 1)$, derive the transformations necessary to rotate the cube by θ degrees about the main diagonal (from $(0, 0, 0)$ to $(1, 1, 1)$) in the counterclockwise direction when looking along the diagonal toward the origin.

5.12 Suppose that the base of the window is rotated at an angle θ from the x axis, as in the Core System [GSPC79]. What is the window-to-viewport mapping? Verify your answer by applying the transformation to each corner of the window, to see that these corners are transformed to the appropriate corners of the viewport.

5.13 Consider a line from the origin of a right-handed coordinate system to the point $P(x, y, z)$. Find the transformation matrices needed to rotate the line into the positive z axis in three different ways, and show by algebraic manipulation that, in each case, the P does go to the z axis. For each method, calculate the sines and cosines of the angles of rotation.

    a. Rotate about the x axis into the (x, y) plane, then rotate about the y axis into the z axis.
    b. Rotate about the y axis into the (y, z) plane, then rotate about the x axis into the z axis.
    c. Rotate about the z axis into the (x, z) plane, then rotate about the y axis into the z axis.

5.14 An object is to be scaled by a factor S in the direction whose direction cosines are (α, β, γ). Derive the transformation matrix.

5.15 Find the 4×4 transformation matrix that rotates by an angle θ about an arbitrary direction given by the direction vector $U = (u_x, u_y, u_z)$. Do this by composing the transformation matrix that rotates U into the z axis (call this M) with a rotation by $R_z(\theta)$, then composing this result with M^{-1}. The result should be

$$\begin{bmatrix} u_x^2 + \cos\theta(1 - u_x^2) & u_x u_y(1 - \cos\theta) - u_z\sin\theta & u_x u_z(1 - \cos\theta) + u_y\sin\theta & 0 \\ u_x u_y(1 - \cos\theta) + u_z\sin\theta & u_y^2 + \cos\theta(1 - u_y^2) & u_y u_z(1 - \cos\theta) - u_x\sin\theta & 0 \\ u_x u_z(1 - \cos\theta) - u_y\sin\theta & u_y u_z(1 - \cos\theta) + u_x\sin\theta & u_z^2 + \cos\theta(1 - u_z^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .(5.79)$$

Verify that, if U is a principal axis, the matrix reduces to R_x, R_y, or R_z. See [FAUX79] for a derivation based on vector operations. Note that negating both U and θ leave the result unchanged. Explain why this is true.

5.16 Prove the properties of $R(\theta)$ described at the end of Section 5.2.

5.17 Extend the incremental rotation discussed in Section 5.4 to 3D, forming a composite operation for rotation about an arbitrary axis.

5.18 Suppose the lowest rate at which an object can be rotated without being annoyingly slow is $360°$ over 30 seconds. Suppose also that, to be smooth, the rotation must be in steps of at most $4°$. Use the results from Exercise 5.6 to determine how many points can be rotated using absolute rotation, and using incremental rotation.

5.19 Suppose that you are creating an interface to rotate an object by applying many incremental rotations, using "Spin X," "Spin Y" and "Spin Z" buttons. Each time one of these buttons is pressed, the current rotation matrix is replaced by its product with a matrix that rotates slightly around the specified axis. Although this idea is mathematically correct, in practice cumulative floating-point roundoff errors will result that will cause points to be transformed incorrectly. Show that by applying the Gram-Schmidt process to the columns of the matrix (see Section A.3.6) you can convert the new matrix back to an orthonormal matrix. Also explain why, if it is already orthonormal, applying this process will not change it.

6

Viewing
in 3D

The 3D viewing process is inherently more complex than is the 2D viewing process. In 2D, we simply specify a window on the 2D world and a viewport on the 2D view surface. Conceptually, objects in the world are clipped against the window and are then transformed into the viewport for display. The extra complexity of 3D viewing is caused in part by the added dimension and in part by the fact that display devices are only 2D.

The solution to the mismatch between 3D objects and 2D displays is accomplished by introducing *projections*, which transform 3D objects onto a 2D projection plane. Much of this chapter is devoted to projections: what they are, their mathematics, and how they are used in a current graphics subroutine package, PHIGS [ANSI88]. Their use is also discussed further in Chapter 7.

In 3D viewing, we specify a *view volume* in the world, a projection onto a projection plane, and a viewport on the view surface. Conceptually, objects in the 3D world are clipped against the 3D view volume and are then projected. The contents of the projection of the view volume onto the projection plane, called the *window*, are then transformed (mapped) into the viewport for display. Figure 6.1 shows this conceptual model of the 3D viewing process, which is the model presented to the users of numerous 3D graphics subroutine packages. Just as with 2D viewing, a variety of strategies can be used to implement the viewing process. The strategies need not be identical to the conceptual model, so long as the results are those defined by the model. A typical implementation strategy for wire-frame line drawings is described in Section 6.5. For graphics systems that perform visible-surface determination and shading, a somewhat different pipeline, discussed in Chapter 16, is used.
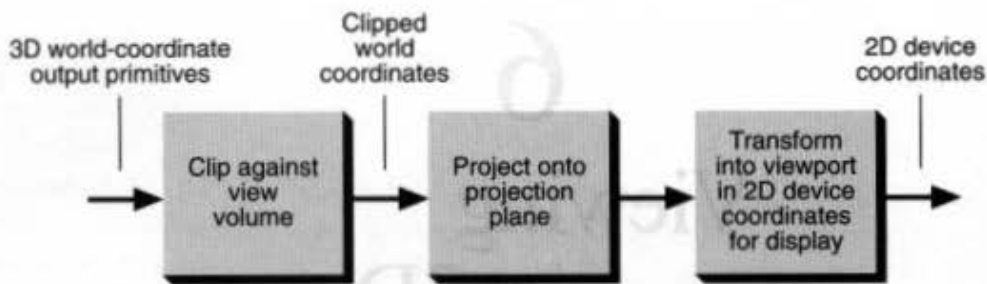
Fig. 6.1 Conceptual model of the 3D viewing process.

6.1 PROJECTIONS

In general, projections transform points in a coordinate system of dimension n into points in a coordinate system of dimension less than n. In fact, computer graphics has long been used for studying n-dimensional objects by projecting them into 2D for viewing [NOLL67]. Here, we shall limit ourselves to the projection from 3D to 2D. The projection of a 3D object is defined by straight projection rays (called *projectors*) emanating from a *center of projection*, passing through each point of the object, and intersecting a *projection plane* to form the projection. Figure 6.2 shows two different projections of the same line. Fortunately, the projection of a line is itself a line, so only line endpoints need actually to be projected.

The class of projections we deal with here is known as *planar geometric projections* because the projection is onto a plane rather than some curved surface and uses straight rather than curved projectors. Many cartographic projections are either nonplanar or nongeometric. Similarly, the Omnimax film format requires a nongeometric projection [MAX82].

Planar geometric projections, hereafter referred to simply as *projections*, can be divided into two basic classes: *perspective* and *parallel*. The distinction is in the relation of the center of projection to the projection plane. If the distance from the one to the other is finite, then the projection is perspective: if the distance is infinite, the projection is parallel.
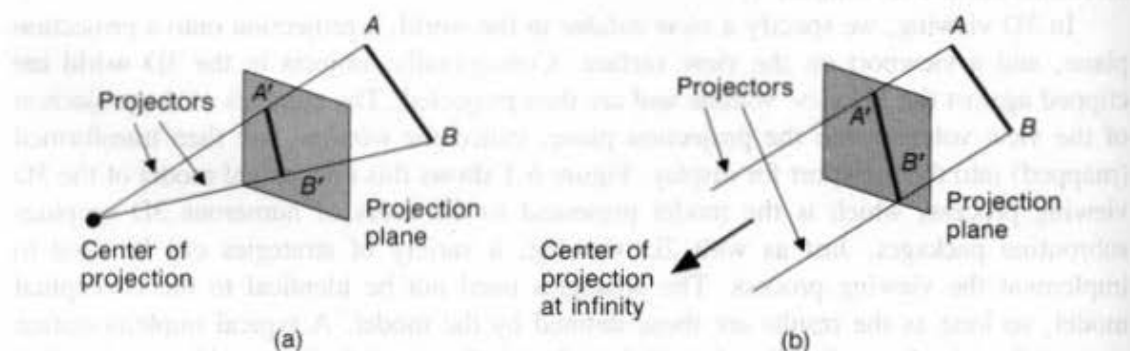
Fig. 6.2 (a) Line *AB* and its perspective projection *A'B'*. (b) Line *AB* and its parallel projection *A'B'*. Projectors *AA'* and *BB'* are parallel.

Figure 6.2 illustrates these two cases. The parallel projection is so named because, with the center of projection infinitely distant, the projectors are parallel. When defining a perspective projection, we explicitly specify its *center of projection*; for a parallel projection, we give its *direction of projection*. The center of projection, being a point, has homogeneous coordinates of the form $(x, y, z, 1)$. Since the direction of projection is a vector (i.e., a difference between points), it can be computed by subtracting two points $d = (x, y, z, 1) - (x', y', z', 1) = (a, b, c, 0)$. Thus, *directions* and *points at infinity* correspond in a natural way. A perspective projection whose center is a point at infinity becomes a parallel projection.

The visual effect of a perspective projection is similar to that of photographic systems and of the human visual system, and is known as *perspective foreshortening*: The size of the perspective projection of an object varies inversely with the distance of that object from the center of projection. Thus, although the perspective projection of objects tend to look realistic, it is not particularly useful for recording the exact shape and measurements of the objects; distances cannot be taken from the projection, angles are preserved only on those faces of the object parallel to the projection plane, and parallel lines do not in general project as parallel lines.

The parallel projection is a less realistic view because perspective foreshortening is lacking, although there can be different constant foreshortenings along each axis. The projection can be used for exact measurements and parallel lines do remain parallel. As with the perspective projection, angles are preserved only on faces of the object parallel to the projection plane.

The different types of perspective and parallel projections are discussed and illustrated at length in the comprehensive paper by Carlbom and Paciorek [CARL78]. In the following two subsections, we summarize the basic definitions and characteristics of the more commonly used projections; we then move on in Section 6.2 to understand how the projections are actually specified to PHIGS.

6.1.1 Perspective Projections

The perspective projections of any set of parallel lines that are not parallel to the projection plane converge to a *vanishing point*. In 3D, the parallel lines meet only at infinity, so the vanishing point can be thought of as the projection of a point at infinity. There is of course an infinity of vanishing points, one for each of the infinity of directions in which a line can be oriented.

If the set of lines is parallel to one of the three principal axes, the vanishing point is called an *axis vanishing point*. There are at most three such points, corresponding to the number of principal axes cut by the projection plane. For example, if the projection plane cuts only the z axis (and is therefore normal to it), only the z axis has a principal vanishing point, because lines parallel to either the y or x axes are also parallel to the projection plane and have no vanishing point.

Perspective projections are categorized by their number of principal vanishing points and therefore by the number of axes the projection plane cuts. Figure 6.3 shows two different one-point perspective projections of a cube. It is clear that they are one-point
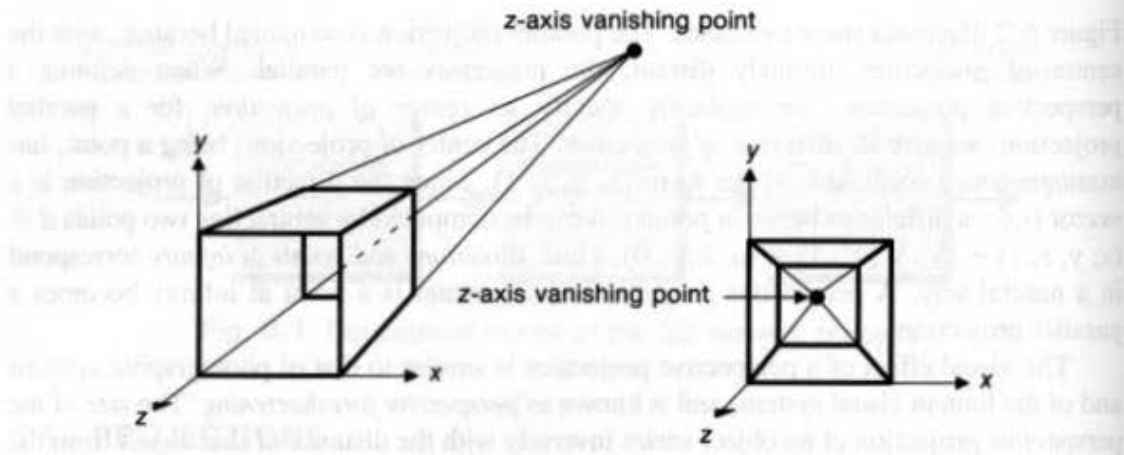
z-axis vanishing point

z-axis vanishing point

Fig. 6.3 One-point perspective projections of a cube onto a plane cutting the z axis, showing vanishing point of lines perpendicular to projection plane.

projections because lines parallel to the x and y axes do not converge; only lines parallel to the z axis do so. Figure 6.4 shows the construction of a one-point perspective with some of the projectors and with the projection plane cutting only the z axis.

Figure 6.5 shows the construction of a two-point perspective. Notice that lines parallel to the y axis do not converge in the projection. Two-point perspective is commonly used in architectural, engineering, industrial design, and in advertising drawings. Three-point perspectives are used less frequently, since they add little realism beyond that afforded by the two-point perspective.

6.1.2 Parallel Projections

Parallel projections are categorized into two types, depending on the relation between the direction of projection and the normal to the projection plane. In *orthographic* parallel

Projection plane

Center of projection

Projection plane normal
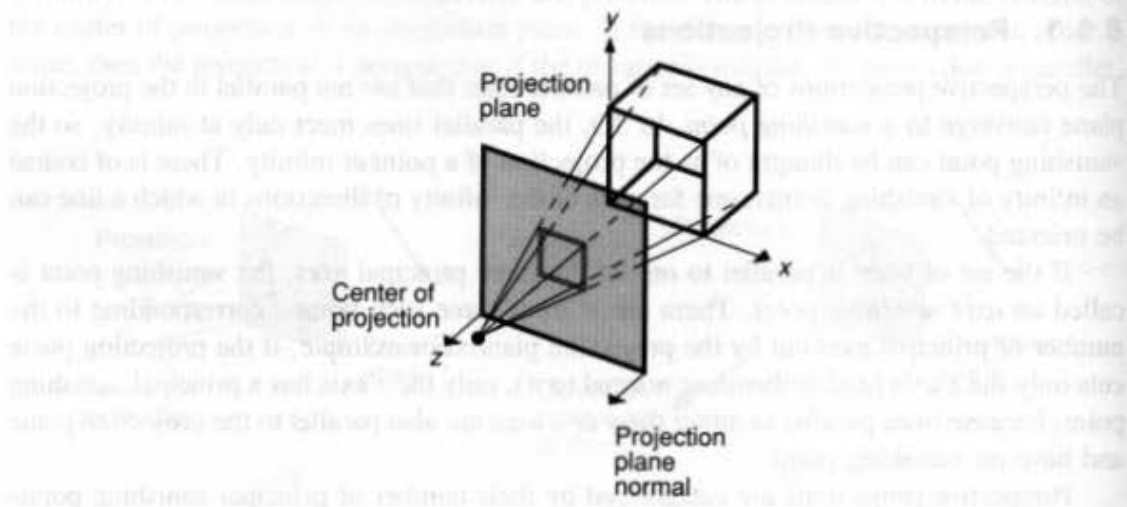
Fig. 6.4 Construction of one-point perspective projection of cube onto plane cutting the z axis. Projection-plane normal is parallel to z axis. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)
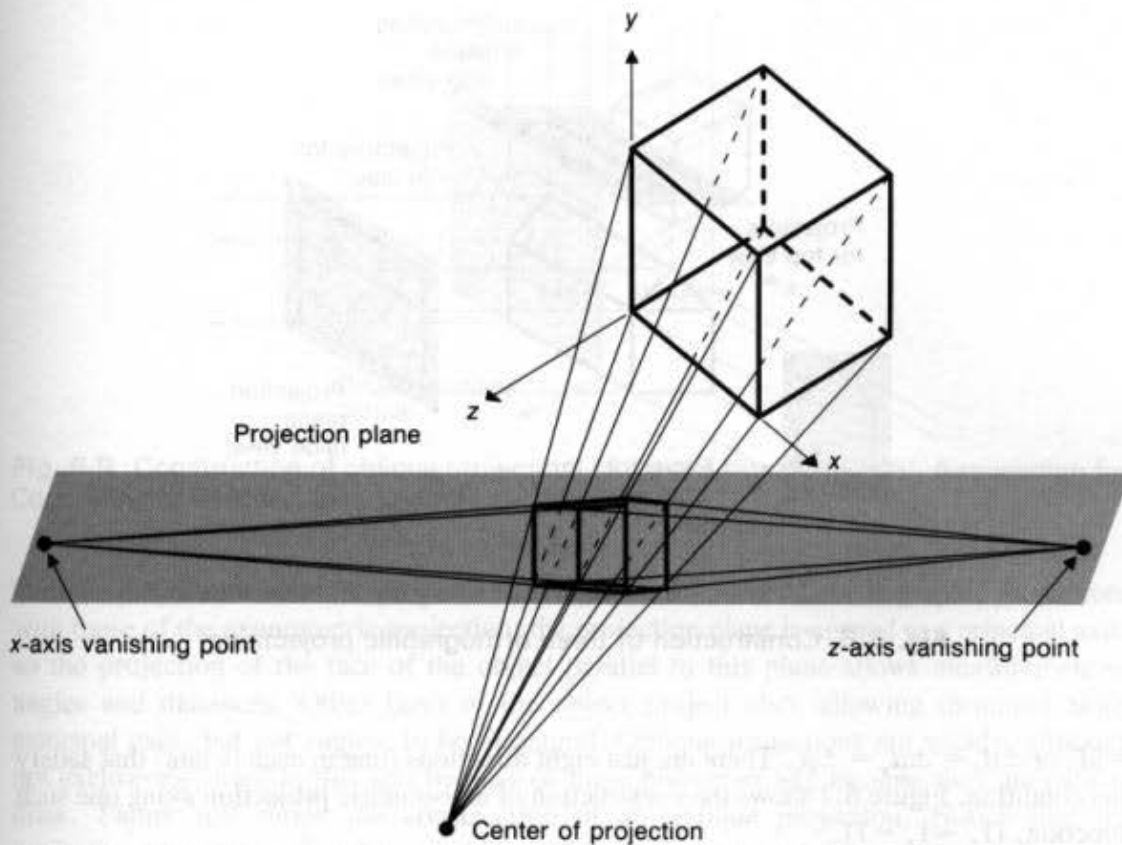
Fig. 6.5 Two-point perspective projection of a cube. The projection plane cuts the x and z axes.

projections, these directions are the same (or the reverse of each other), so the direction of projection is normal to the projection plane. For the *oblique* parallel projection, they are not.

The most common types of orthographic projections are the *front-elevation*, *top-elevation* (also called *plan-elevation*), and *side-elevation* projections. In all these, the projection plane is perpendicular to a principal axis, which is therefore the direction of projection. Figure 6.6 shows the construction of these three projections; they are often used in engineering drawings to depict machine parts, assemblies, and buildings, because distances and angles can be measured from them. Since each depicts only one face of an object, however, the 3D nature of the projected object can be difficult to deduce, even if several projections of the same object are studied simultaneously.

Axonometric orthographic projections use projection planes that are not normal to a principal axis and therefore show several faces of an object at once. They resemble the perspective projection in this way, but differ in that the foreshortening is uniform rather than being related to the distance from the center of projection. Parallelism of lines is preserved but angles are not, while distances can be measured along each principal axis (in general, with different scale factors).

The *isometric projection* is a commonly used axonometric projection. The projection-plane normal (and therefore the direction of projection) makes equal angles with each principal axis. If the projection-plane normal is (d_x, d_y, d_z), then we require that $|d_x| = |d_y|$
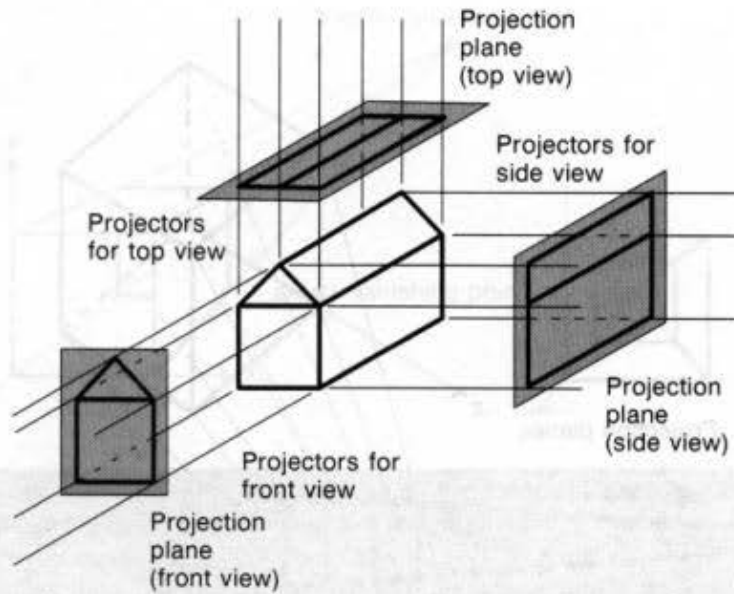
Fig. 6.6 Construction of three orthographic projections.

$= |d_z|$ or $\pm d_x = \pm d_y = \pm d_z$. There are just eight directions (one in each octant) that satisfy this condition. Figure 6.7 shows the construction of an isometric projection along one such direction, $(1, -1, -1)$.

The isometric projection has the useful property that all three principal axes are equally foreshortened, allowing measurements along the axes to be made to the same scale (hence the name: *iso* for equal, *metric* for measure). In addition, the principal axes project so as to make equal angles one with another, as shown in Fig. 6.8.

Oblique projections, the second class of parallel projections, differ from orthographic projections in that the projection-plane normal and the direction of projection differ.
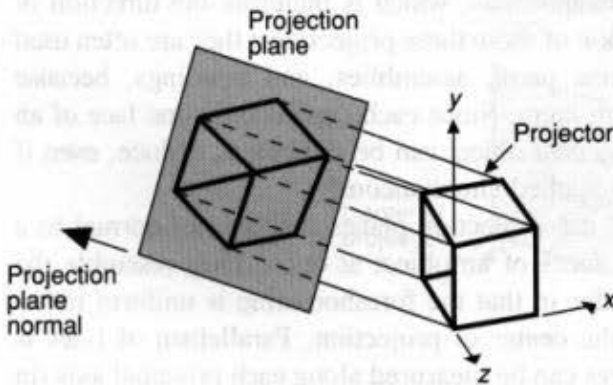
Fig. 6.7 Construction of an isometric projection of a unit cube. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)
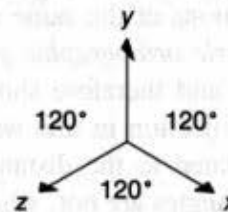
Fig. 6.8 Isometric projection of unit vectors, with direction of projection (1, 1, 1).
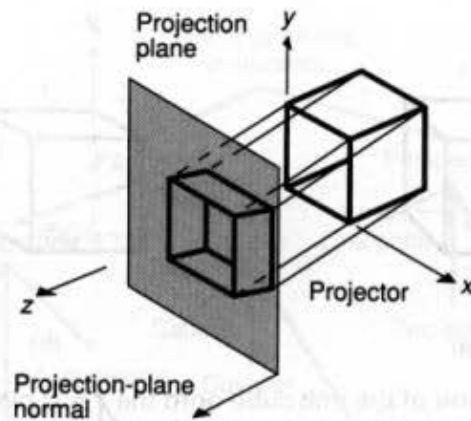
Fig. 6.9 Construction of oblique projection. (Adapted from [CARL78], Association for Computing Machinery, Inc.; used by permission.)

Oblique projections combine properties of the front, top, and side orthographic projections with those of the axonometric projection: the projection plane is normal to a principal axis, so the projection of the face of the object parallel to this plane allows measurement of angles and distances. Other faces of the object project also, allowing distances along principal axes, but not angles, to be measured. Oblique projections are widely, although not exclusively, used in this text because of these properties and because they are easy to draw. Figure 6.9 shows the construction of an oblique projection. Notice that the projection-plane normal and the direction of projection are not the same.

Two frequently used oblique projections are the *cavalier* and the *cabinet*. For the cavalier projection, the direction of projection makes a 45° angle with the projection plane. As a result, the projection of a line perpendicular to the projection plane has the same length as the line itself; that is, there is no foreshortening. Figure 6.10 shows several cavalier projections of the unit cube onto the (x, y) plane; the receding lines are the projections of the cube edges that are perpendicular to the (x, y) plane, and they form an angle α to the horizontal. This angle is typically 30° or 45°.
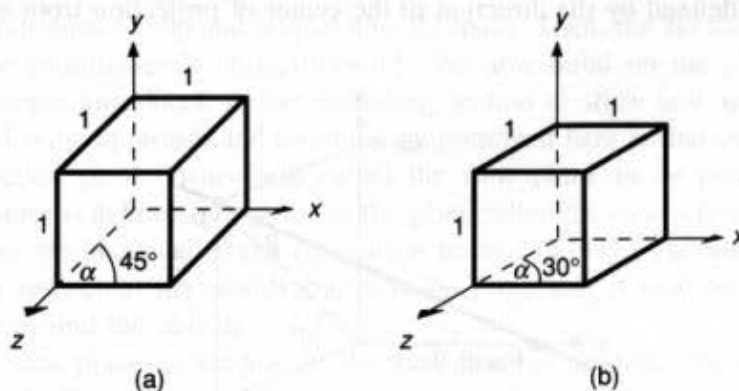

Fig. 6.10 Cavalier projection of the unit cube onto the $z = 0$ plane. All edges project at unit length. In (a), the direction of projection is $(\sqrt{2}/2, \sqrt{2}/2, -1)$; in (b), it is $(\sqrt{3}/2, 1/2, -1)$.
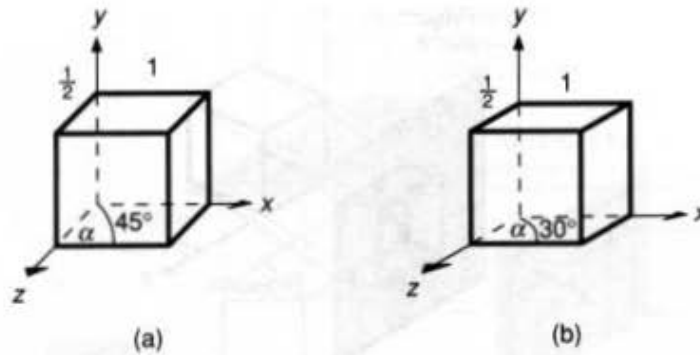
Fig. 6.11 Cabinet projection of the unit cube onto the $z = 0$ plane. Edges parallel to the x and y axes project at unit length. In (a), the direction of projection is $(\sqrt{2}/4, \sqrt{2}/4, -1)$; in (b), it is $(\sqrt{3}/4, 1/4, -1)$.

Cabinet projections, such as those in Fig. 6.11, have a direction of projection that makes an angle of $\arctan(2) = 63.4°$ with the projection plane, so lines perpendicular to the projection plane project at one-half their actual length. Cabinet projections are a bit more realistic than cavalier ones are, since the foreshortening by one-half is more in keeping with our other visual experiences.

Figure 6.12 helps to explain the angles made by projectors with the projection plane for cabinet and cavalier projections. The (x, y) plane is the projection plane and the point P' is the projection of $(0, 0, 1)$ onto the projection plane. The angle α and length l are the same as are used in Figs. 6.10 and 6.11, and we can control them by varying the direction of projection (l is the length at which the z-axis unit vector projects onto the (x, y) plane; α is the angle the projection makes with the x axis). Designating the direction of projection as $(dx, dy, -1)$, we see from Fig. 6.12 that $dx = l \cos\alpha$ and $dy = l \sin\alpha$. Given a desired l and α, the direction of projection is $(l \cos\alpha, l \sin\alpha, -1)$.

Figure 6.13 shows the logical relationships among the various types of projections. The common thread uniting them all is that they involve a projection plane and either a center of projection for the perspective projection, or a direction of projection for the parallel projection. We can unify the parallel and perspective cases further by thinking of the center of projection as defined by the direction to the center of projection from some reference
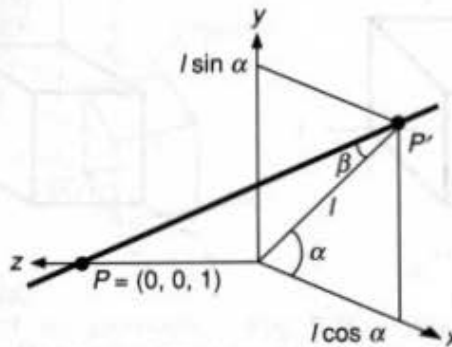

Fig. 6.12 Oblique parallel projection of $P = (0, 0, 1)$ onto $P' = (l \cos \alpha, l \sin \beta, 0)$. The direction of projection is $P' - P = (l \cos \alpha, l \sin \beta, -1)$.
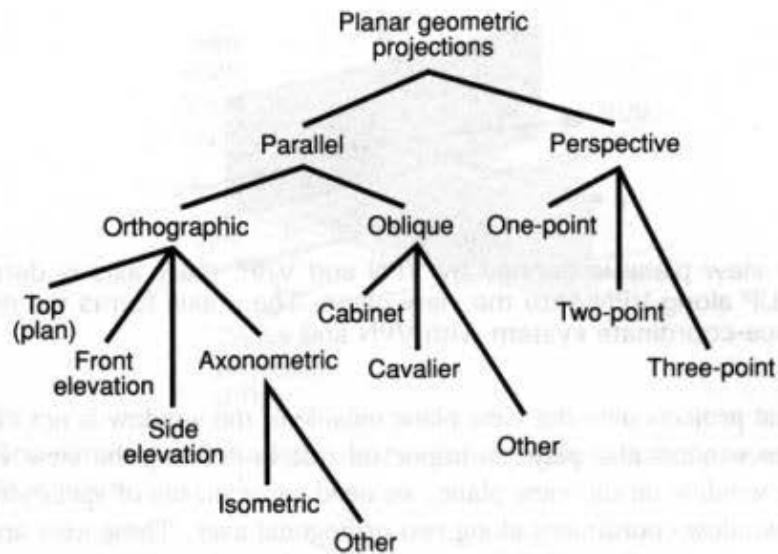
Fig. 6.13 The subclasses of planar geometric projections. *Plan view* is another term for a top view. *Front* and *side* are often used without the term *elevation*.

point, plus the distance to the reference point. When this distance increases to infinity, the projection becomes a parallel projection. Hence, we can also say that the common thread uniting these projections is that they involve a projection plane, a direction to the center of projection, and a distance to the center of projection.

In the next section, we consider how to integrate these various types of projections into the 3D viewing process.

6.2 SPECIFYING AN ARBITRARY 3D VIEW

As suggested by Fig. 6.1, 3D viewing involves not just a projection but also a view volume against which the 3D world is clipped. The projection and view volume together provide all the information needed to clip and project into 2D space. Then, the 2D transformation into physical device coordinates is straightforward. We now build on the planar-geometric-projection concepts introduced in the preceding section to show how to specify a view volume. The viewing approach and terminology presented here is that used in PHIGS.

The projection plane, henceforth called the *view plane* to be consistent with the graphics literature, is defined by a point on the plane called the *view reference point* (VRP) and a normal to the plane called the *view-plane normal* (VPN).[1] The view plane may be anywhere with respect to the world objects to be projected: it may be in front of, cut through, or be behind the objects.

Given the view plane, a window on the view plane is needed. The window's role is similar to that of a 2D window: its contents are mapped into the viewport, and any part of

------

[1]PHIGS has an additional variable, the view-plane distance (VPD): the view plane can be a distance VPD from the VRP. VPD is positive in the direction of VPN. See Exercise 6.22.
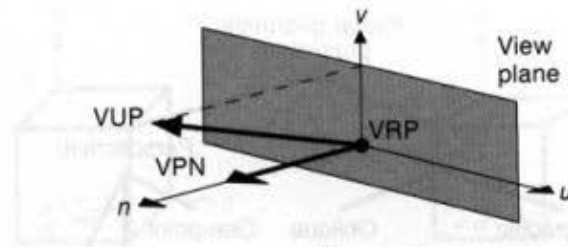
Fig. 6.14 The view plane is defined by VPN and VRP; the v axis is defined by the projection of VUP along VPN onto the view plane. The u axis forms the right-handed viewing reference-coordinate system with VPN and v.

the 3D world that projects onto the view plane outside of the window is not displayed. We shall see that the window also plays an important role in defining the view volume.

To define a window on the view plane, we need some means of specifying minimum and maximum window coordinates along two orthogonal axes. These axes are part of the 3D *viewing-reference coordinate* (VRC) system. The origin of the VRC system is the VRP. One axis of the VRC is VPN; this axis is called the *n* axis. A second axis of the VRC is found from the *view up vector* (VUP), which determines the *v*-axis direction on the view plane. The *v* axis is defined such that the projection of VUP parallel to VPN onto the view plane is coincident with the *v* axis. The *u*-axis direction is defined such that *u*, *v*, and *n* form a right-handed coordinate system, as in Fig. 6.14. The VRP and the two direction vectors VPN and VUP are specified in the right-handed world-coordinate system. (Some graphics packages use the *y* axis as VUP, but this is too restrictive and fails if VPN is parallel to the *y* axis, in which case VUP would be undefined.)

With the VRC system defined, the window's minimum and maximum *u* and *v* values can be defined, as in Fig. 6.15. This figure illustrates that the window need not be symmetrical about the view reference point, and explicitly shows the center of the window, CW.

The center of projection and direction of projection (DOP) are defined by a *projection reference point* (PRP) plus an indicator of the projection type. If the projection type is perspective, then PRP is the center of projection. If the projection type is parallel, then the DOP is from the PRP to CW. The CW is in general *not* the VRP, which need not even be within the window bounds.

The PRP is specified in the VRC system, not in the world-coordinate system; thus, the position of the PRP relative to the VRP does not change as VUP or VRP are moved. The
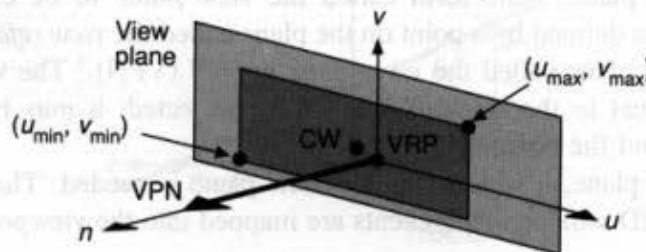

Fig. 6.15 The view reference-coordinate system (VRC) is a right-handed system made up of the *u, v,* and *n* axes. The *n* axis is always the VPN. CW is the center of the window.
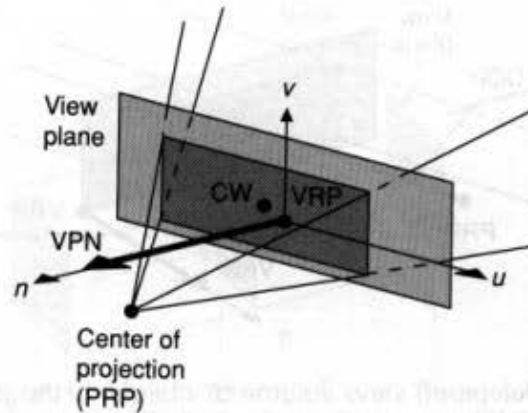
Fig. 6.16 The semi-infinite pyramid view volume for perspective projection. CW is the center of the window.

advantage of this is that the programmer can specify the direction of projection required, for example, by a cavalier projection, and then change VPN and VUP (hence changing VRC), without having to recalculate the PRP needed to maintain the desired projection. On the other hand, moving the PRP about to get different views of an object may be more difficult.

The *view volume* bounds that portion of the world that is to be clipped out and projected onto the view plane. For a perspective projection, the view volume is the semi-infinite pyramid with apex at the PRP and edges passing through the corners of the window. Figure 6.16 shows a perspective-projection view volume. Positions behind the center of projection are not included in the view volume and thus are not projected. In reality, of course, our eyes see an irregularly shaped conelike view volume. However, a pyramidal view volume is mathematically more tractable, and is consistent with the concept of a rectangular viewport.

For parallel projections, the view volume is an infinite parallelepiped with sides parallel to the direction of projection, which is the direction from the PRP to the center of the window. Figures 6.17 and 6.18 show parallel-projection view volumes and their
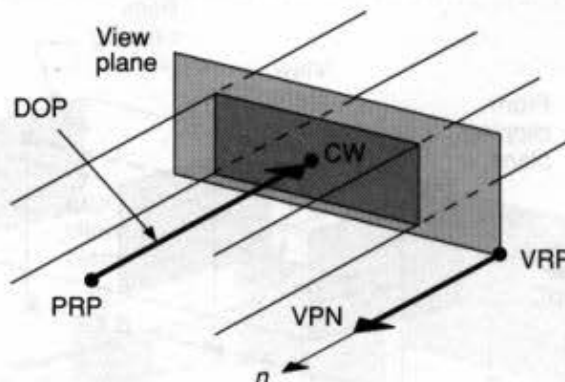

Fig. 6.17 Infinite parallelepiped view volume of parallel orthographic projection. The VPN and direction of projection (DOP) are parallel. DOP is the vector from PRP to CW, and is parallel to the VPN.
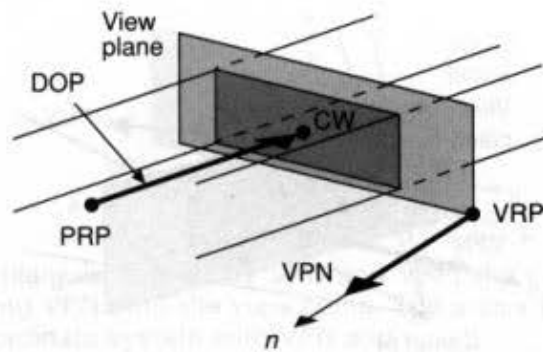
Fig. 6.18 Infinite parallelepiped view volume of oblique orthographic projection. The direction of projection (DOP) is not parallel to the VPN.

relation to the view plane, window, and PRP. For orthographic parallel projections, but not for oblique parallel projections, the sides of the view volume are normal to the view plane.

At times, we might want the view volume to be finite, in order to limit the number of output primitives projected onto the view plane. Figures 6.19, 6.20, and 6.21 show how the view volume is made finite with a *front clipping plane* and *back clipping plane*.These planes, sometimes called the *hither* and *yon* planes, are parallel to the view plane; their normal is the VPN. The planes are specified by the signed quantities *front distance* (*F*) and *back distance* (*B*) relative to the view reference point and along the VPN, with positive distances in the direction of the VPN. For the view volume to be positive, the front distance must be algebraically greater than the back distance.

Limiting the view volume in this way can be useful in order to eliminate extraneous objects and to allow the user to concentrate on a particular portion of the world. Dynamic modification of either the front or rear distances can give the viewer a good sense of the spatial relationships between different parts of the object as these parts appear and disappear from view (see Chapter 14). For perspective projections, there is an additional motivation. An object very distant from the center of projection projects onto the view surface as a
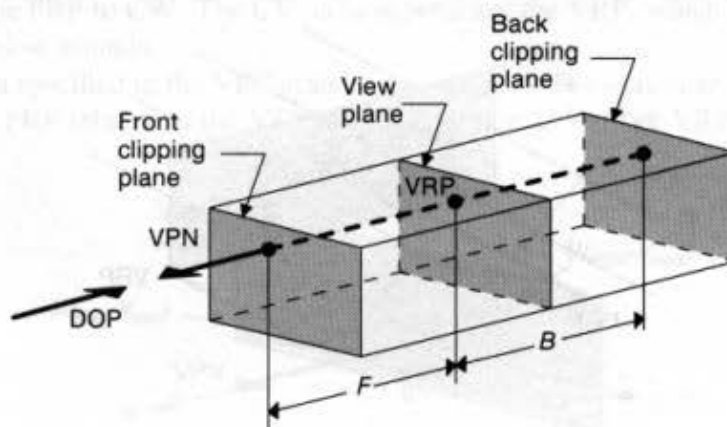

Fig. 6.19 Truncated view volume for an orthographic parallel projection. DOP is the direction of projection.
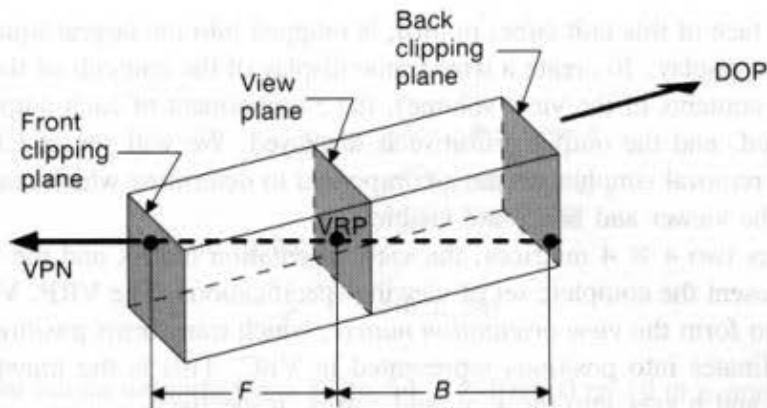
Fig. 6.20 Truncated view volume for oblique parallel projection showing VPN oblique to direction of projection (DOP); VPN is also normal to the front and back clipping planes.

"blob" of no distinguishable form. In displaying such an object on a plotter, the pen can wear through the paper; on a vector display, the CRT phosphor can be burned by the electron beam; and on a film recorder, the high concentration of light causes a fuzzy white area to appear. Also, an object very near the center of projection may extend across the window like so many disconnected pick-up sticks, with no discernible structure. Specifying the view volume appropriately can eliminate such problems.

How are the contents of the view volume mapped onto the display surface? First, consider the unit cube extending from 0 to 1 in each of the three dimensions of *normalized projection coordinates* (NPC). The view volume is transformed into the rectangular solid of NPC, which extends from x_{min} to x_{max} along the x axis, from y_{min} to y_{max} along the y axis, and from z_{min} to z_{max} along the z axis. The front clipping plane becomes the z_{max} plane and the back clipping plane becomes the z_{min} plane. Similarly, the u_{min} side of the view volume becomes the x_{min} plane and the u_{max} side becomes the x_{max} plane. Finally, the v_{min} side of the view volume becomes the y_{min} plane and the v_{max} side becomes the y_{max} plane. This rectangular solid portion of NPC, called a *3D viewport*, is within the unit cube.
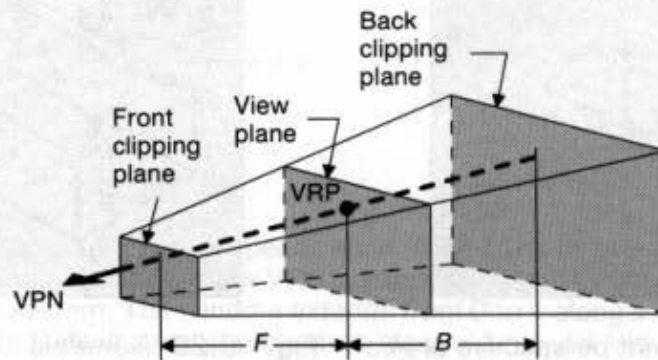

Fig. 6.21 Truncated view volume for perspective projection.

The $z = 1$ face of this unit cube, in turn, is mapped into the largest square that can be inscribed on the display. To create a wire-frame display of the contents of the 3D viewport (which are the contents of the view volume), the z-component of each output primitive is simply discarded, and the output primitive is displayed. We will see in Chapter 15 that hidden surface removal simply uses the z-component to determine which output primitives are closest to the viewer and hence are visible.

PHIGS uses two 4×4 matrices, the view orientation matrix and the view mapping matrix, to represent the complete set of viewing specifications. The VRP, VPN, and VUP are combined to form the *view orientation matrix*, which transforms positions represented in world coordinates into positions represented in VRC. This is the transformation that takes the u, v, and n axes into the x, y, and z axes, respectively.

The view volume specifications, given by PRP, u_{min}, u_{max}, v_{min}, v_{max}, F, and B, along with the 3D viewport specification, given by x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}, are combined to form the *view mapping matrix*, which transforms points in VRC to points in normalized projection coordinates. The subroutine calls that form the view orientation matrix and view mapping matrix are discussed in Section 7.3.4.

In the next section, we see how to obtain various views using the concepts introduced in this section. In Section 6.4 the basic mathematics of planar geometric projections is introduced, whereas in Section 6.5 the mathematics and algorithms needed for the entire viewing operation are developed.

6.3 EXAMPLES OF 3D VIEWING

In this section, we consider how the basic viewing concepts introduced in the previous section can be applied to create a variety of projections, such as those shown in Figs. 6.22 and 6.23. Because the house shown in these figures is used throughout this section, it will be helpful to remember its dimensions and position, which are indicated in Fig. 6.24. For each view discussed, we give a table showing the VRP, VPN, VUP, PRP, window, and
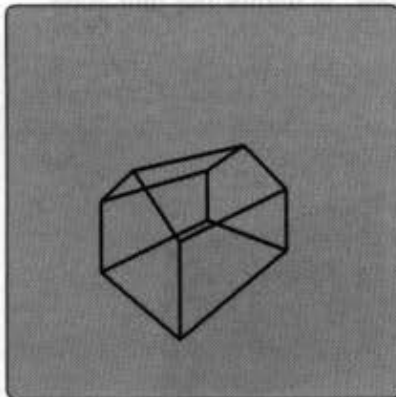

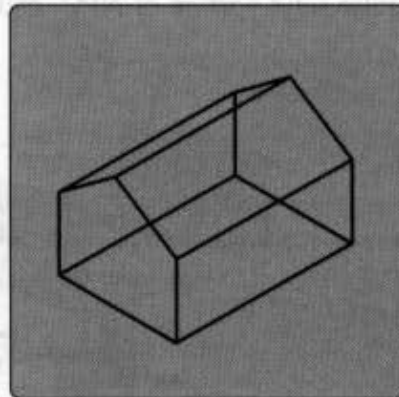**Fig. 6.22** Two-point perspective projection of a house.
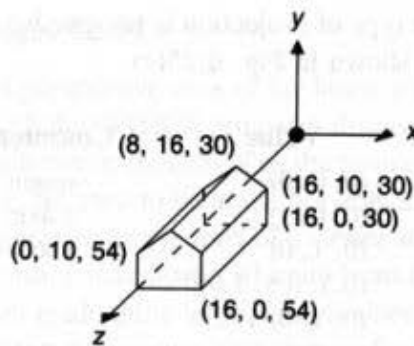

Fig. 6.23 Isometric projection of a house.

Fig. 6.24 The house extends from 30 to 54 in z, from 0 to 16 in x, and from 0 to 16 in y.

projection type (perspective or parallel). The 3D viewport default, which is the unit cube in NPC, is assumed throughout this section. The notation (WC) or (VRC) is added to the table as a reminder of the coordinate system in which the viewing parameter is given. The form of the table is illustrated here for the default viewing specification used by PHIGS. The defaults are shown in Fig. 6.25(a). The view volume corresponding to these defaults is
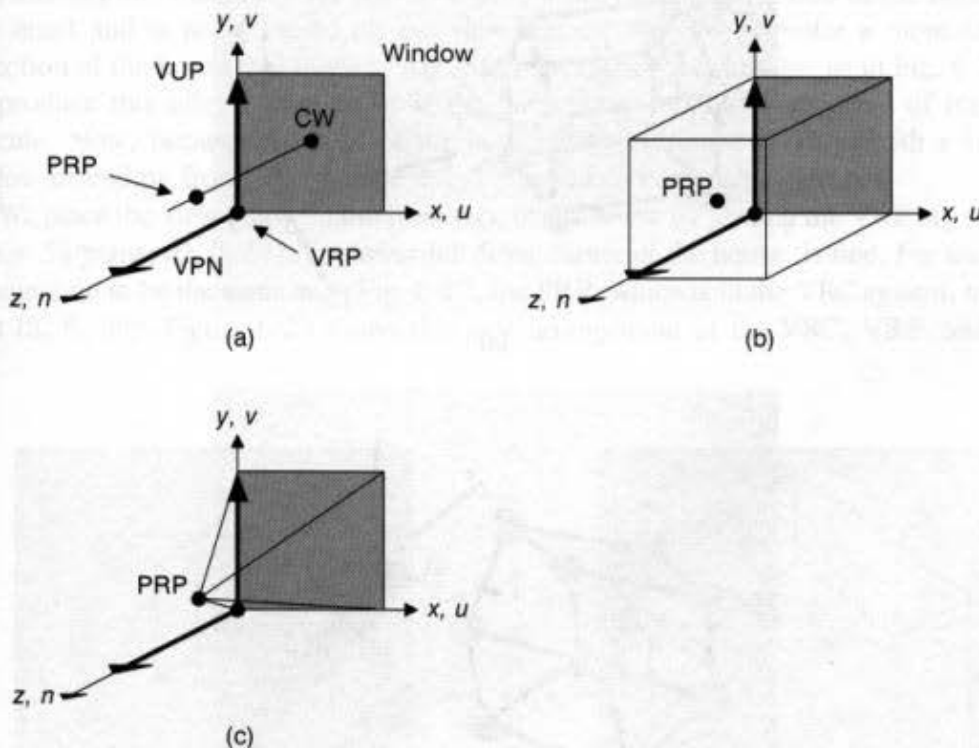

Fig. 6.25 (a) The default viewing specification: VRP is at the origin, VUP is the y axis, and VPN is the z axis. This makes the VRC system of u, v, and n coincide with the x, y, z world-coordinate system. The window extends from 0 to 1 along u and v, and PRP is at (0.5, 0.5, 1.0). (b) Default parallel projection view volume. (c) View volume if default projection were perspective.

shown in Fig. 6.25(b). If the type of projection is perspective rather than parallel, then the view volume is the pyramid shown in Fig. 6.25(c).

Viewing parameter	Value	Comments
VRP(WC)	(0, 0, 0)	origin
VPN(WC)	(0, 0, 1)	z axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	(0.5, 0.5, 1.0)	
window (VRC)	(0, 1, 0, 1)	
projection type	parallel	

Readers wanting to review how all these parameters interrelate are encouraged to construct a house, the world coordinate system, and the VRC system with Tinker Toys, as pictured in Fig. 6.26. The idea is to position the VRC system in world coordinates as in the viewing example and to imagine projectors from points on the house intersecting the view plane. In our experience, this is a useful way to understand (and teach) 3D viewing concepts.
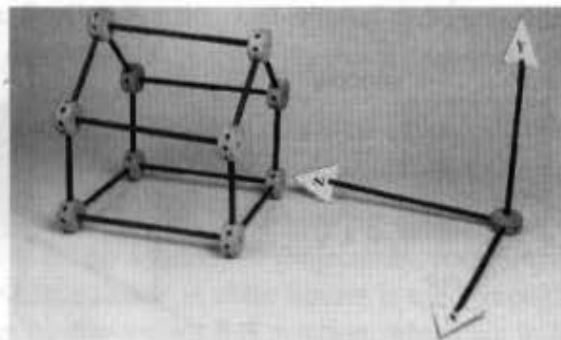
(a)
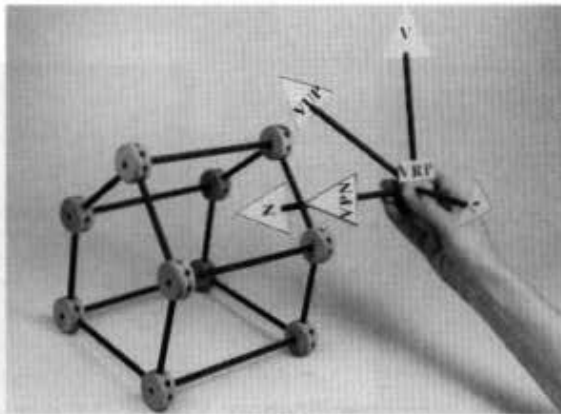
(b)

Fig. 6.26 Stick models useful for understanding 3D viewing. (a) House and world-coordinate system. (b) House and VRC system.

6.3.1 Perspective Projections

To obtain the front one-point perspective view of the house shown in Fig. 6.27 (this and all similar figures were made with the SPHIGS program, discussed in Chapter 7), we position the center of projection (which can be thought of as the position of the viewer) at $x = 8$, $y = 6$, and $z = 84$. The x value is selected to be at the horizontal center of the house and the y value to correspond to the approximate eye level of a viewer standing on the (x, z) plane; the z value is arbitrary. In this case, z is removed 30 units from the front of the house ($z = 54$ plane). The window has been made quite large, to guarantee that the house fits within the view volume. All other viewing parameters have their default values, so the overall set of viewing parameters is as follows:

VRP(WC)	(0, 0, 0)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(8, 6, 84)
window(VRC)	(−50, 50, −50, 50)
projection type	perspective

Although the image in Fig. 6.27 is indeed a perspective projection of the house, it is very small and is not centered on the view surface. We would prefer a more centered projection of the house that more nearly spans the entire view surface, as in Fig. 6.28. We can produce this effect more easily if the view plane and the front plane of the house coincide. Now, because the front of the house extends from 0 to 16 in both x and y, a window extending from −1 to 17 in x and y produces reasonable results.

We place the view plane on the front face of the house by placing the VRP anywhere in the $z = 54$ plane; (0, 0, 54), the lower-left front corner of the house, is fine. For the center of projection to be the same as in Fig. 6.27, the PRP, which is in the VRC system, needs to be at (8, 6, 30). Figure 6.29 shows this new arrangement of the VRC, VRP, and PRP,
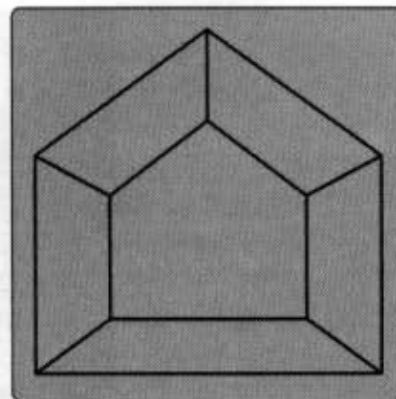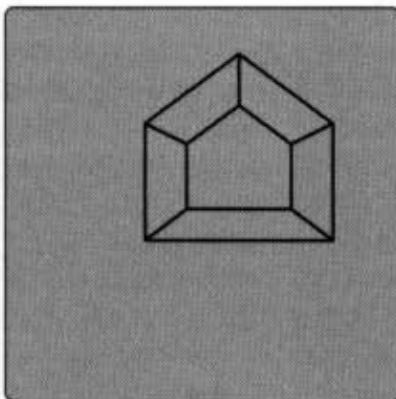

Fig. 6.27 One-point perspective projection of the house.

Fig. 6.28 Centered perspective projection of a house.
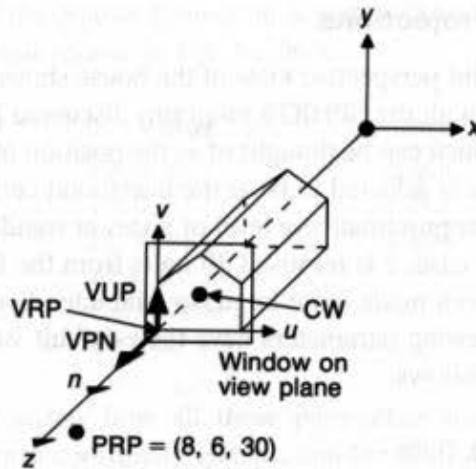
Fig. 6.29 The viewing situation for Fig. 6.28.

which corresponds to the following set of viewing parameters:

VRP(WC)	(0, 0, 54)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(8, 6, 30)
window(VRC)	(−1, 17, −1, 17)
projection type	perspective

This same result can be obtained in many other ways. For instance, with the VRP at (8, 6, 54), as in Fig. 6.30, the center of projection, given by the PRP, becomes (0, 0, 30). The window must also be changed, because its definition is based on the VRC system, the origin of which is the VRP. The appropriate window extends from −9 to 9 in u and from −7 to 11 in v. With respect to the house, this is the same window as that used in the previous

Fig. 6.30 An alternative viewing situation for Fig. 6.28.

example, but it is now specified in a different VRC system. Because the view-up direction is the y axis, the u axis and x axis are pa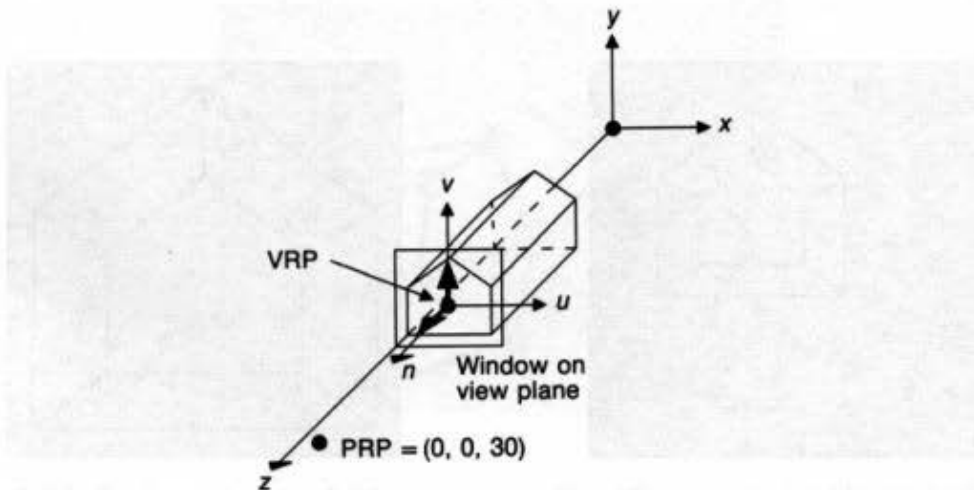rallel, as are the v and y axes. In summary, the following viewing parameters, shown in Fig. 6.30, also produce Fig. 6.28:

VRP(WC)	(8, 6, 54)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(0, 0, 30)
window(VRC)	(−9, 9, −7, 11)
projection type	perspective

Next, let us try to obtain the two-point perspective projection shown in Fig. 6.22. The center of projection is analogous to the position of a camera that takes snapshots of world-coordinate objects. With this analogy in mind, the center of projection in Fig. 6.22 seems to be somewhat above and to the right of the house, as viewed from the positive z axis. The exact center of projection is (36, 25, 74). Now, if the corner of the house at (16, 0, 54) is chosen as the VRP, then this center of projection is at (20, 25, 20) relative to it. With the view plane coincident with the front of the house (the $z = 54$ plane), a window ranging from −20 to 20 in u and from −5 to 35 in v is certainly large enough to contain the projection. Hence, we can specify the view of Fig. 6.31 with the viewing parameters:

VRP(WC)	(16, 0, 54)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(20, 25, 20)
window(VRC)	(−20, 20, −5, 35)
projection type	perspective

This view is similar to, but clearly is not the same as, that in Fig. 6.22. For one thing, Fig. 6.22 is a two-point perspective projection, whereas Fig. 6.31 is a one-point perspective. It is apparent that simply moving the center of projection is not sufficient to produce Fig. 6.22. In fact, we need to reorient the view plane such that it cuts both the x and
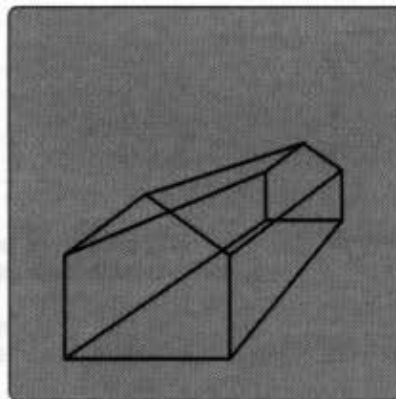
Fig. 6.31 Perspective projection of a house from (36, 25, 74) with VPN parallel to the z axis.
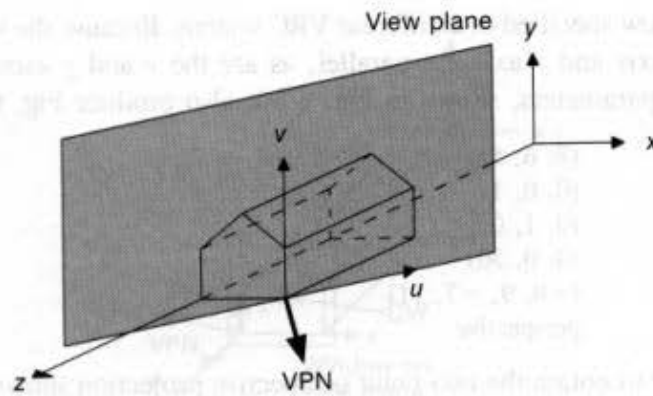
Fig. 6.32 The view plane and VRC system corresponding to Fig. 6.22.

z axes, by setting VPN to (1, 0, 1). Thus, the viewing parameters for Fig. 6.22 are as follows:

VRP(WC)	(16, 0, 54)
VPN(WC)	(1, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(0, 25, 20$\sqrt{2}$)
window(VRC)	(−20, 20, −5, 35)
projection type	perspective

Figure 6.32 shows the view plane established with this *VPN*. The 20$\sqrt{2}$ n component of the *PRP* is used so that the center of projection is a distance 20$\sqrt{2}$ away from the VRP in the (*x*, *y*) plane, as shown in Fig. 6.33.

There are two ways to choose a window that completely surrounds the projection, as does the window in Fig. 6.22. One can estimate the size of the projection of the house onto the view plane using a sketch, such as Fig. 6.33, to calculate the intersections of projectors

Fig. 6.33 Top (plan) view of a house for determining an appropriate window size.

Fig. 6.34 Projection of house produced by rotating VUP.

with the view plane. A better alternative, however, is to allow the window bounds to be variables in a program that are determined interactively via a valuator or locator device.

Figure 6.34 is obtained from the same projection as is Fig. 6.22, but the window has a different orientation. In all previous examples, the v axis of the VRC 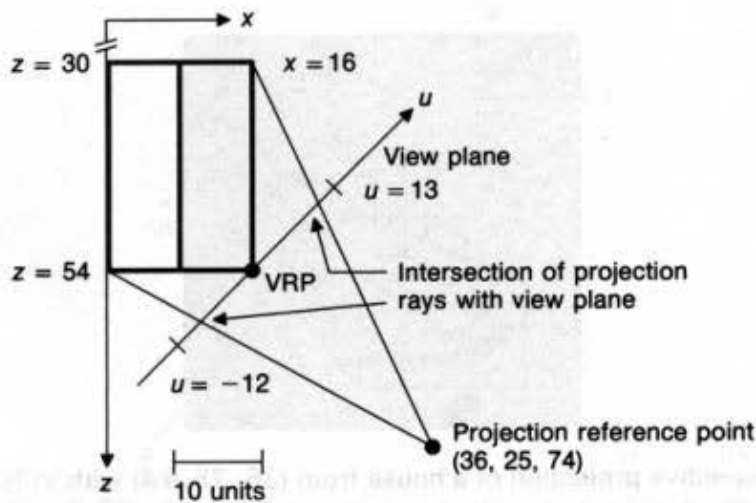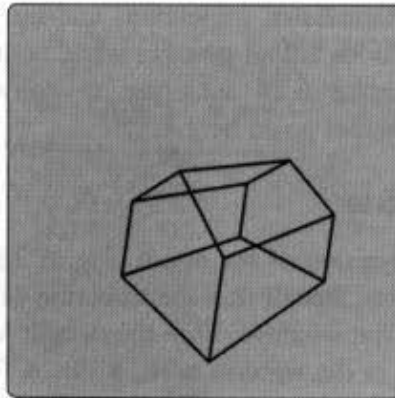system was parallel to the y axis of the world-coordinate system; thus, the window (two of whose sides are parallel to the v axis) was nicely aligned with the vertical sides of the house. Figure 6.34 has exactly the same viewing parameters as does Fig. 6.22, except that VUP has been rotated away from the y axis by about 10°.

Another way to specify viewing parameters for perspective projections is suggested in Fig. 6.35. This figure is modeled after the way a photographer might think about positioning a camera. Six parameters are needed: the center of projection, which is the camera position; the center of attention, which is a point at which the camera is aimed (the VPN is the vector from the center of attention to the center of projection); VUP, the up vector; D, the distance from the center of projection to the projection plane; W, the width of the window on the projection plane, and H, the height of the window on the projection plane. The center of attention need not be on the view plane. In this model, the VPN is
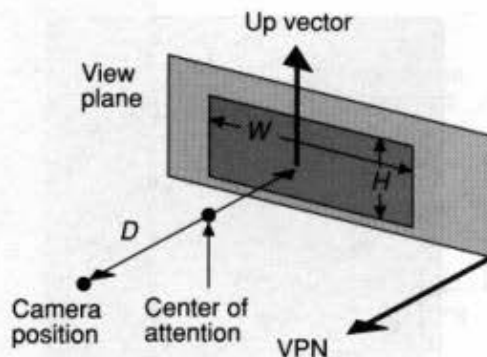

Fig. 6.35 Another way to specify a view, with a camera position (the center of projection), center of attention, up vector, distance from the center of projection to the projection plane, and the height and width of the window on the projection plane. VPN is parallel to the direction from the center of attention to the camera position.

always pointed directly at the center of projection, and the view volume is symmetrical about its center line. The positions are all given in world coordinates—there is no concept of viewing coordinates. Exercise 6.24 asks you to convert from these six viewing parameters into the viewing model given here.

6.3.3 Parallel Projections

We create a front parallel projection of the house (Fig. 6.36) by making the direction of projection parallel to the z axis. Recall that the direction of projection is determined by the PRP and by the center of the window. With the default VRC system and a window of $(-1, 17, -1, 17)$, the center of the window is $(8, 8, 0)$. A PRP of $(8, 8, 100)$ provides a direction of projection parallel to the z axis. Figure 6.37 shows the viewing situation that creates Fig. 6.36. The viewing parameters are as follows:

VRP(WC)	(0, 0, 0)
VPN(WC)	(0, 0, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(8, 8, 100)
window(VRC)	(−1, 17, −1, 17)
projection type	parallel

To create the side view (Fig. 6.38), we require the viewing situation of Fig. 6.39, with the (y, z) plane (or any plane parallel to it) as the view plane. This situation corresponds to the following viewing parameters:

VRP(WC)	(0, 0, 54)	
VPN(WC)	(1, 0, 0)	x axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	(12, 8, 16)	
window(VRC)	(−1, 25, −5, 21)	
projection type	parallel	
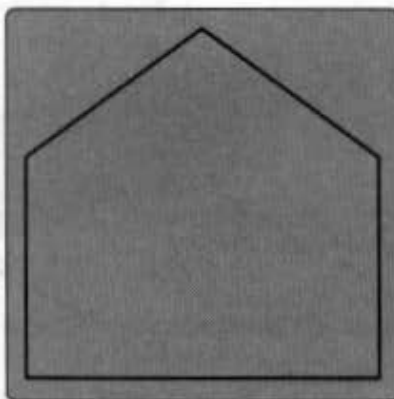
Fig. 6.36 Front parallel projection of the house.
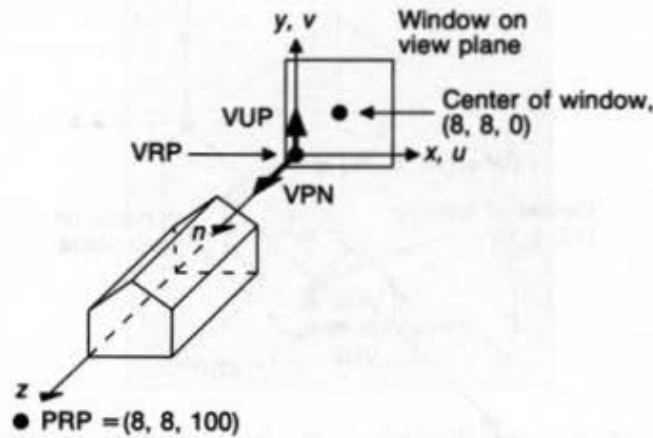
Fig. 6.37 Viewing situation that creates Fig. 6.36, a front view of the house. The PRP could be any point with $x = 8$ and $y = 8$.

The center of the window is at (12, 8, 0) in VRC; hence, the PRP has these same u and v coordinates.

We create a top view of the house by using the (x, z) plane as the view plane and VPN as the y axis. The default view-up direction of $+y$ must be changed; we use the negative x axis. With VRP again specified as a corner of the house, we have the viewing situation in Fig. 6.40, defined by the following viewing parameters:

|		
VRP(WC)	(16, 0, 54)	
VPN(WC)	(0, 1, 0)	y axis
VUP(WC)	(−1, 0, 0)	negative x axis
PRP(VRC)	(12, 8, 30)	
window(VRC)	(−1, 25, −5, 21)	
projection type	parallel	
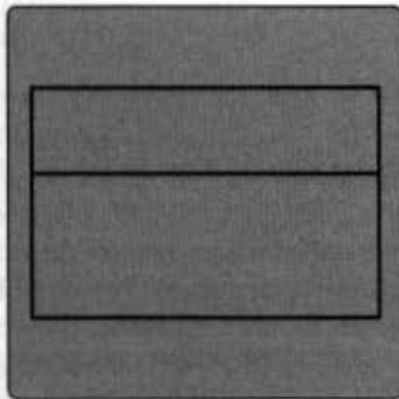
Fig. 6.38 Parallel projection from the side of the house.
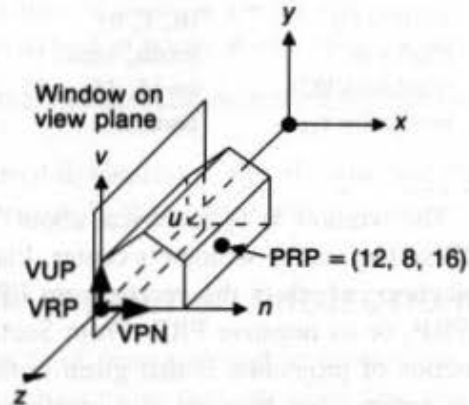
Fig. 6.39 The viewing situation for Fig. 6.38.
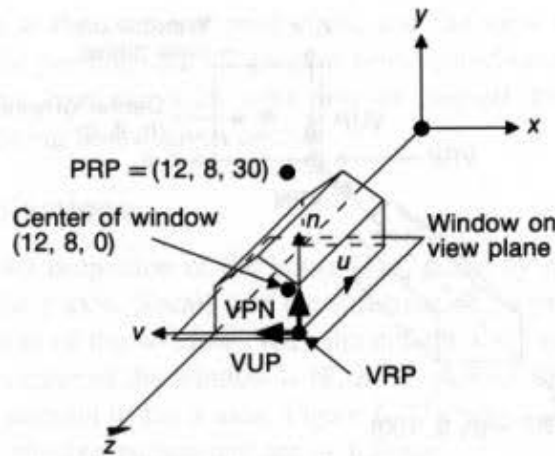
Fig. 6.40 The viewing situation for a top view of the house.

Figure 6.23 is an isometric (parallel orthographic) projection in the direction $(-1, -1, -1)$, one of the eight possible directions for an isometric (see Section 6.1.2). The following viewing parameters create such an isometric projection:

VRP(WC)	(8, 8, 42)
VPN(WC)	(1, 1, 1)
VUP(WC)	(0, 1, 0)
PRP(VRC)	(0, 0, 10)
window(VRC)	$(-20, 20, -20, 20)$
projection type	parallel

A cavalier projection with angle α (see Fig. 6.10) is specified with

VRP(WC)	(8, 8, 54)	middle front of house
VPN(WC)	(0, 0, 1)	z axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	$(\cos\alpha, \sin\alpha, 1)$	
window(VRC)	$(-15, 15, -15, 15)$	
projection type	parallel	

The window is symmetrical about VRP, which implies that VRP, the origin of the VRC system, is the window's center. Placing PRP as specified means that the direction of projection, which is the vector from PRP to the window center, is $(0, 0, 0) -$ PRP $=$ $-$ PRP, or its negative PRP. From Section 6.1.2, we know that the cavalier projection's direction of projection is that given in the preceding table: $-$ PRP $= -(\cos\alpha, \sin\alpha, 1)$. Now notice what happens if a cavalier projection onto a plane of constant x instead of constant z is desired: only VRP, VPN, and VUP need to be changed to establish a new VRC system. The PRP, window, and projection type can remain fixed.
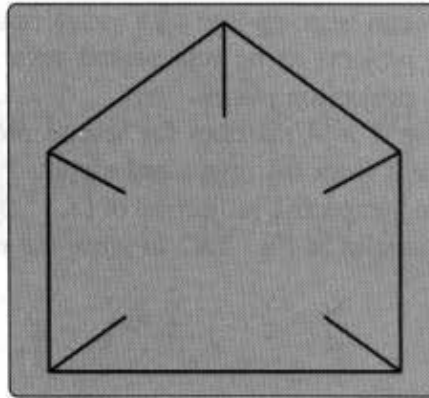
Fig. 6.41 Perspective projection of the house with back clipping plane at $z = 31$.

6.3.4 Finite View Volumes

In all the examples so far, the view volume has been assumed to be infinite. The front and back clipping planes, described Section 6.2, help to determine a *finite view volume*. These planes, both of which are parallel to the view plane, are at distances F and B respectively from the view reference point, measured from VRP along VPN. To avoid a negative view volume, we must ensure that F is algebraically greater than B.

A front perspective view of the house with the rear wall clipped away (Fig. 6.41) results from the following viewing specification, in which F and B have been added. If a distance is given, then clipping against the corresponding plane is assumed; otherwise, it is not. The viewing specification is as follows:

VRP(WC)	(0, 0, 54)	lower-left front of house
VPN(WC)	(0, 0, 1)	z axis
VUP(WC)	(0, 1, 0)	y axis
PRP(VRC)	(8, 6, 30)	
window(VRC)	(−1, 17, −1, 17)	
projection type	perspective	
F(VRC)	+1	one unit in front of house, at $z = 54 + 1 = 55$
B(VRC)	−23	one unit from back of house, at $z = 54 − 23 = 31$

The viewing situation for this case is the same as that in Fig. 6.29, except for the addition of the clipping planes.

If the front and back clipping planes are moved dynamically, the 3D structure of the object being viewed can often be discerned more readily than it can with a static view.

6.4 THE MATHEMATICS OF PLANAR GEOMETRIC PROJECTIONS

In this section, we introduce the basic mathematics of planar geometric projections. For simplicity, we start by assuming that, in the perspective projection, the projection plane is normal to the z axis at $z = d$, and that, in the parallel projection, the projection plane is the $z = 0$ plane. Each of the projections can be defined by a 4×4 matrix. This is convenient,

because the projection matrix can be composed with transformation matrices, allowing two operations (transform, then project) to be represented as a single matrix. In the next section, we discuss arbitrary projection planes.

In this section, we derive 4 × 4 matrices for several projections, beginning with a projection plane at a distance d from the origin and a point P to be projected onto it. To calculate $P_p = (x_p, y_p, z_p)$, the perspective projection of (x, y, z) onto the projection plane at $z = d$, we use the similar triangles in Fig. 6.42 to write the ratios

$$\frac{x_p}{d} = \frac{x}{z}; \qquad \frac{y_p}{d} = \frac{y}{z}. \tag{6.1}$$

Multiplying each side by d yields

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d}, \qquad y_p = \frac{d \cdot y}{z} = \frac{y}{z/d}. \tag{6.2}$$

The distance d is just a scale factor applied to x_p and y_p. The division by z causes the perspective projection of more distant objects to be smaller than that of closer objects. All values of z are allowable except $z = 0$. Points can be behind the center of projection on the negative z axis or between the center of projection and the projection plane.

The transformation of Eq. (6.2) can be expressed as a 4 × 4 matrix:

$$M_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}. \tag{6.3}$$
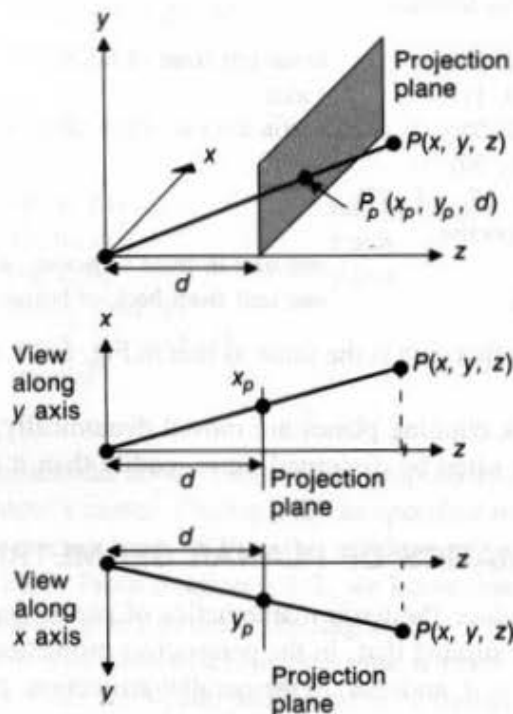
Fig. 6.42 Perspective projection.

Multiplying the point $P = [x \ y \ z \ 1]^T$ by the matrix M_{per} yields the general homogeneous point $[X \ Y \ Z \ W]^T$:

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = M_{per} \cdot P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.4}$$

or

$$[X \ Y \ Z \ W]^T = \left[x \ y \ z \ \frac{z}{d} \right]^T \tag{6.5}$$

Now, dividing by W (which is z/d) and dropping the fourth coordinate to come back to 3D, we have

$$\left(\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W} \right) = (x_p, y_p, z_p) = \left(\frac{x}{z/d}, \frac{y}{z/d}, d \right); \tag{6.6}$$

these equations are the correct results of Eq. (6.1), plus the transformed z coordinate of d, which is the position of the projection plane along the z axis.

An alternative formulation for the perspective projection places the projection plane at $z = 0$ and the center of projection at $z = -d$, as in Fig. 6.43. Similarity of the triangles now gives

$$\frac{x_p}{d} = \frac{x}{z + d}, \qquad \frac{y_p}{d} = \frac{y}{z + d}. \tag{6.7}$$

Multiplying by d, we get

$$x_p = \frac{d \cdot x}{z + d} = \frac{x}{(z/d) + 1}, \qquad y_p = \frac{d \cdot y}{z + d} = \frac{y}{(z/d) + 1}. \tag{6.8}$$
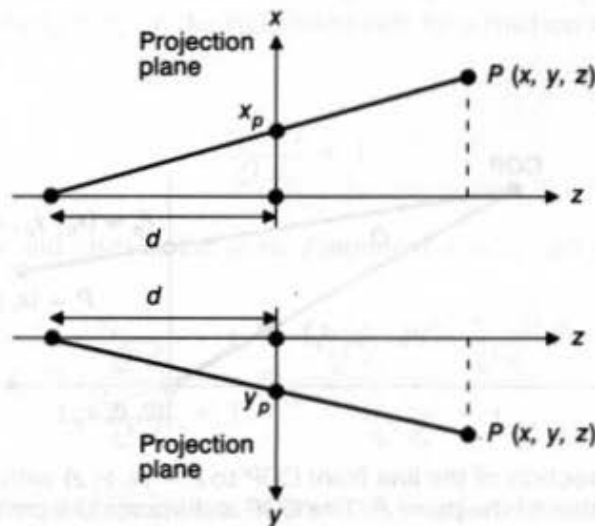
Fig. 6.43 Alternative perspective projection.

The matrix is

$$M'_{per} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}. \tag{6.9}$$

This formulation allows d, the distance to the center of projection, to tend to infinity.

The orthographic projection onto a projection plane at $z = 0$ is straightforward. The direction of projection is the same as the projection-plane normal—the z axis, in this case. Thus, point P projects as

$$x_p = x, \quad y_p = y, \quad z_p = 0. \tag{6.10}$$

This projection is expressed by the matrix

$$M_{ort} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.11}$$

Notice that as d in Eq. (6.9) tends to infinity, Eq. (6.9) becomes Eq. (6.11).

M_{per} applies only in the special case in which the center of projection is at the origin; M_{ort} applies only when the direction of projection is parallel to the z axis. A more robust formulation, based on a concept developed in [WEIN87], not only removes these restrictions but also integrates parallel and perspective projections into a single formulation. In Fig. 6.44, the projection of the general point $P = (x, y, z)$ onto the projection plane is $P_p = (x_p, y_p, z_p)$. The projection plane is perpendicular to the z axis at a distance z_p from the origin, and the center of projection (COP) is a distance Q from the point $(0, 0, z_p)$. The direction from $(0, 0, z_p)$ to COP is given by the normalized direction vector (d_x, d_y, d_z). P_p is
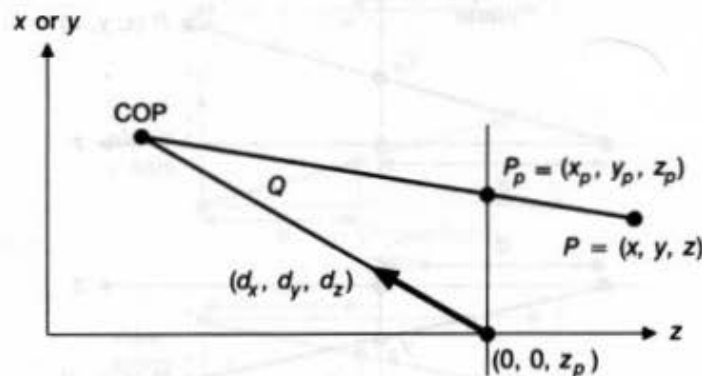
Fig. 6.44 The intersection of the line from COP to $P = (x, y, z)$ with the projection plane at $z = z_p$ is the projection of the point P. The COP is distance Q from the point $(0, 0, z_p)$, in direction (d_x, d_y, d_z).

on the line between COP and P, which can be specified parametrically as

$$\text{COP} + t(P - \text{COP}), \qquad 0 \le t \le 1. \tag{6.12}$$

Rewriting Eq. (6.12) as separate equations for the arbitrary point $P' = (x', y', z')$ on the line, with $\text{COP} = (0, 0, z_p) + Q(d_x, d_y, d_z)$, yields

$$x' = Q\,d_x + (x - Q\,d_x)t, \tag{6.13}$$
$$y' = Q\,d_y + (y - Q\,d_y)t, \tag{6.14}$$
$$z' = (z_p + Q\,d_z) + (z - (z_p + Q\,d_z))t. \tag{6.15}$$

We find the projection P_p of the point P, at the intersection of the line between COP and P with the projection plane, by substituting $z' = z_p$ into Eq. (6.15) and solving for t:

$$t = \frac{z_p - (z_p + Q\,d_z)}{z - (z_p + Q\,d_z)}. \tag{6.16}$$

Substituting this value of t into Eq. (6.13) and Eq. (6.14) to find $x' = x_p$ and $y' = y_p$ yields

$$x_p = \frac{x - z\dfrac{d_x}{d_z} + z_p\dfrac{d_x}{d_z}}{\dfrac{z_p - z}{Q\,d_z} + 1}, \tag{6.17}$$

$$y_p = \frac{y - z\dfrac{d_y}{d_z} + z_p\dfrac{d_y}{d_z}}{\dfrac{z_p - z}{Q\,d_z} + 1}. \tag{6.18}$$

Multiplying the identity $z_p = z_p$ on the right-hand side by a fraction whose numerator and denominator are both

$$\frac{z_p - z}{Q\,d_z} + 1 \tag{6.19}$$

maintains the identity and gives z_p the same denominator as x_p and y_p:

$$z_p = z_p \frac{\dfrac{z_p - z}{Q\,d_z} + 1}{\dfrac{z_p - z}{Q\,d_z} + 1} = \frac{-z\dfrac{z_p}{Q\,d_z} + \dfrac{z_p^2 + z_p Q\,d_z}{Q\,d_z}}{\dfrac{z_p - z}{Q\,d_z} + 1}. \tag{6.20}$$

Now Eqs. (6.17), (6.18), and (6.20) can be rewritten as a 4×4 matrix M_{general} arranged so that the last row of M_{general} multiplied by $[x \ \ y \ \ z \ \ 1]^T$ produces their common

denominator, which is the homogeneous coordinate W and is hence the divisor of X, Y, and Z:

$$M_{general} = \begin{bmatrix} 1 & 0 & -\dfrac{d_x}{d_z} & z_p\dfrac{d_x}{d_z} \\ 0 & 1 & -\dfrac{d_y}{d_z} & z_p\dfrac{d_y}{d_z} \\ 0 & 0 & -\dfrac{z_p}{Q\,d_z} & \dfrac{z_p^2}{Q\,d_z} + z_p \\ 0 & 0 & -\dfrac{1}{Q\,d_z} & \dfrac{z_p}{Q\,d_z} + 1 \end{bmatrix}. \tag{6.21}$$

$M_{general}$ specializes to all three of the previously derived projection matrixes M_{per}, M'_{per}, and M_{ort}, given the following values:

	z_p	Q	$[d_x \quad d_y \quad d_z]$
M_{ort}	0	∞	$[0 \quad 0 \quad -1]$
M_{per}	d	d	$[0 \quad 0 \quad -1]$
M'_{per}	0	d	$[0 \quad 0 \quad -1]$

When $Q \neq \infty$, $M_{general}$ defines a one-point perspective projection. The vanishing point of a perspective projection is calculated by multiplying the point at infinity on the z axis, represented in homogeneous coordinates as $[0 \quad 0 \quad 1 \quad 0]^T$, by $M_{general}$. Taking this product and dividing by W gives

$$x = Q\,d_x, \quad y = Q\,d_y, \quad z = z_p.$$

Given a desired vanishing point (x, y) and a known distance Q to the center of projection, these equations uniquely define $[d_x \quad d_y \quad d_z]$, because $\sqrt{d_x^2 + d_y^2 + d_z^2} = 1$.

Similarly, it is easy to show that, for cavalier and cabinet projections onto the (x, y) plane, with α the angle shown in Figs. 6.10 and 6.11,

	z_p	Q	$[d_x \qquad d_y \qquad d_z]$
Cavalier	0	∞	$[\cos\alpha \quad \sin\alpha \quad -1]$
Cabinet	0	∞	$\left[\dfrac{\cos\alpha}{2} \quad \dfrac{\sin\alpha}{2} \quad -1\right]$

In this section, we have seen how to formulate M_{per}, M'_{per}, and M_{ort}, all of which are special cases of the more general $M_{general}$. In all these cases, however, the projection plane is perpendicular to the z axis. In the next section, we remove this restriction and consider the clipping implied by finite view volumes.

6.5 IMPLEMENTING PLANAR GEOMETRIC PROJECTIONS

Given a view volume and a projection, let us consider how the *viewing operation* of clipping and projecting is actually applied. As suggested by the conceptual model for viewing (Fig. 6.1), we could clip lines against the view volume by calculating their intersections with each

of the six planes that define the view volume. Lines remaining after clipping would be projected onto the view plane, by solution of simultaneous equations for the intersection of the projectors with the view plane. The coordinates would then be transformed from 3D world coordinates to 2D device coordinates. However, the large number of calculations required for this process, repeated for many lines, involves considerable computing. Happily, there is a more efficient procedure, based on the divide-and-conquer strategy of breaking down a difficult problem into a series of simpler ones.

Certain view volumes are easier to clip against than is the general one (clipping algorithms are discussed in Section 6.5.3). For instance, it is simple to calculate the intersections of a line with each of the planes of a parallel-projection view volume defined by the six planes

$$x = -1, \quad x = 1, \quad y = -1, \quad y = 1, \quad z = 0, \quad z = -1. \tag{6.22}$$

This is also true of the perspective-projection view volume defined by the planes

$$x = z, \quad x = -z, \quad y = z, \quad y = -z, \quad z = -z_{min}, \quad z = -1. \tag{6.23}$$

These *canonical view volumes* are shown in Fig. 6.45.

Our strategy is to find the *normalizing transformations* N_{par} and N_{per} that transform an arbitrary parallel- or perspective-projection view volume into the parallel and perspective canonical view volumes, respectively. Then clipping is performed, followed by projection into 2D, via the matrices in Section 6.4. This strategy risks investing effort in transforming points that are subsequently discarded by the clip operation, but at least the clipping is easy to do.

Figure 6.46 shows the sequence of processes involved here. We can reduce it to a transform–clip–transform sequence by composing steps 3 and 4 into a single transformation matrix. With perspective projections, a division is also needed to map from homogeneous coordinates back to 3D coordinates. This division is done following the second transformation of the combined sequence. An alternative strategy, clipping in homogeneous coordinates, is discussed in Section 6.5.4.
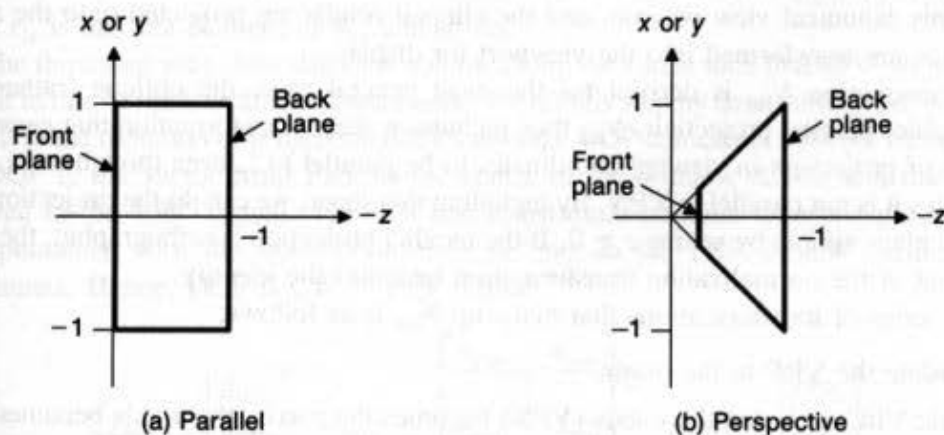

Fig. 6.45 The two canonical view volumes, for the (a) parallel and (b) perspective projections. Note that $-z$ is to the right.

3D world-
coordinate
output
primitives

2D device
coordinates
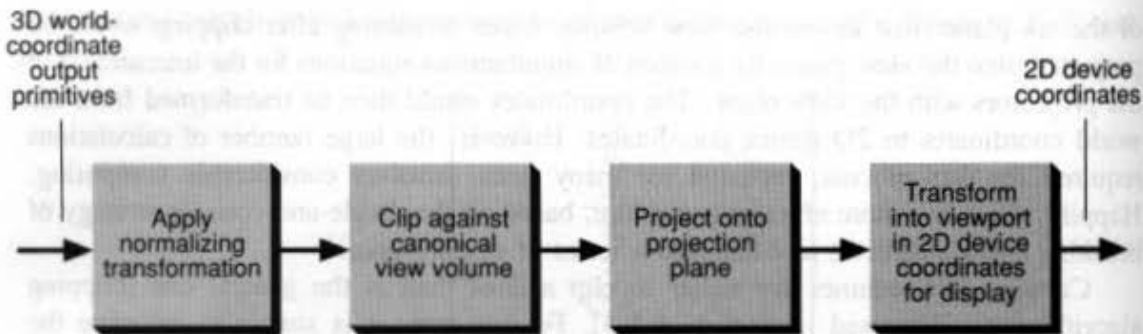
Fig. 6.46 Implementation of 3D viewing.

Readers familiar with PHIGS will notice that the canonical view volumes of Eqs. (6.22) and (6.23) are different than the *default view volumes* of PHIGS: the unit cube from 0 to 1 in x, y, and z for parallel projection, and the pyramid with apex at $(0.5, 0.5, 1.0)$ and sides passing through the unit square from 0 to 1 in x and y on the $z = 0$ plane for perspective projection. The canonical view volumes are defined to simplify the clipping equations and to provide the consistency between parallel and perspective projections discussed in Section 6.5.4. On the other hand, the PHIGS default view volumes are defined to make 2D viewing be a special case of 3D viewing. Exercise 6.26 concerns the PHIGS defaults.

In the next two sections, we derive the normalizing transformations for perspective and parallel projections, which are used as the first step in the transform–clip–transform sequence.

6.5.1   Parallel Projection

In this section, we derive the normalizing transformation N_{par} for parallel projections in order to transform world-coordinate positions such that the view volume is transformed into the canonical view volume defined by Eq. (6.22). The transformed coordinates are clipped against this canonical view volume, and the clipped results are projected onto the $z = 0$ plane, then are transformed into the viewport for display.

Transformation N_{par} is derived for the most general case, the oblique (rather than orthographic) parallel projection. N_{par} thus includes a shear transformation that causes the direction of projection in viewing coordinates to be parallel to z, even though in (u, v, n) coordinates it is not parallel to VPN. By including this shear, we can do the projection onto the $z = 0$ plane simply by setting $z = 0$. If the parallel projection is orthographic, the shear component of the normalization transformation becomes the identity.

The series of transformations that make up N_{par} is as follows:

1. Translate the VRP to the origin

2. Rotate VRC such that the n axis (VPN) becomes the z axis, the u axis becomes the x axis, and the v axis becomes the y axis

3. Shear such that the direction of projection becomes parallel to the z axis

4. Translate and scale into the parallel-projection canonical view volume of Eq. (6.22).

In PHIGS, steps 1 and 2 define the *view-orientation matrix*, and steps 3 and 4 define the *view-mapping matrix*.

Figure 6.47 shows this sequence of transformations as applied to a parallel-projection view volume and to an outline of a house; Fig. 6.48 shows the parallel projection that results.

Step 1 is just the translation $T(-\text{VRP})$. For step 2, we use the properties of orthonormal matrices discussed in Section 5.5 and illustrated in the derivation of Eqs. (5.66) and (5.67). The row vectors of the rotation matrix to perform step 2 are the unit vectors that are rotated by R into the x, y, and z axes. VPN is rotated into the z axis, so

$$R_z = \frac{\text{VPN}}{\|\text{VPN}\|}. \tag{6.24}$$

The u axis, which is perpendicular to VUP and to VPN and is hence the cross-product of the unit vector along VUP and R_z (which is in the same direction as VPN), is rotated into the x axis, so

$$R_x = \frac{\text{VUP} \times R_z}{\|\text{VUP} \times R_z\|}. \tag{6.25}$$

Similarly, the v axis, which is perpendicular to R_z and R_x, is rotated into the y axis, so

$$R_y = R_z \times R_x. \tag{6.26}$$

Hence, the rotation in step 2 is given by the matrix

$$R = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} & 0 \\ r_{1y} & r_{2y} & r_{3y} & 0 \\ r_{1z} & r_{2z} & r_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6.27}$$

where r_{1x} is the first element of R_x, and so on.

The third step is to shear the view volume along the z axis such that all of its planes are normal to one of the coordinate system axes. We do this step by determining the shear to be applied to the direction of projection (DOP) to make DOP coincident with the z axis. Recall that DOP is the vector from PRP to the center of the window (CW), and that PRP is specified in the VRC system. The first two transformation steps have brought VRC into correspondence with the world-coordinate system, so the PRP is now itself in world coordinates. Hence, DOP is CW − PRP. Given

$$\text{DOP} = \begin{bmatrix} dop_x \\ dop_y \\ dop_z \\ 0 \end{bmatrix}, \quad \text{CW} = \begin{bmatrix} \dfrac{u_{max} + u_{min}}{2} \\ \dfrac{v_{max} + v_{min}}{2} \\ 0 \\ 1 \end{bmatrix}, \quad \text{PRP} = \begin{bmatrix} prp_u \\ prp_v \\ prp_n \\ 1 \end{bmatrix}, \tag{6.28}$$

(a)

(b)

(c)

(d)

(e)
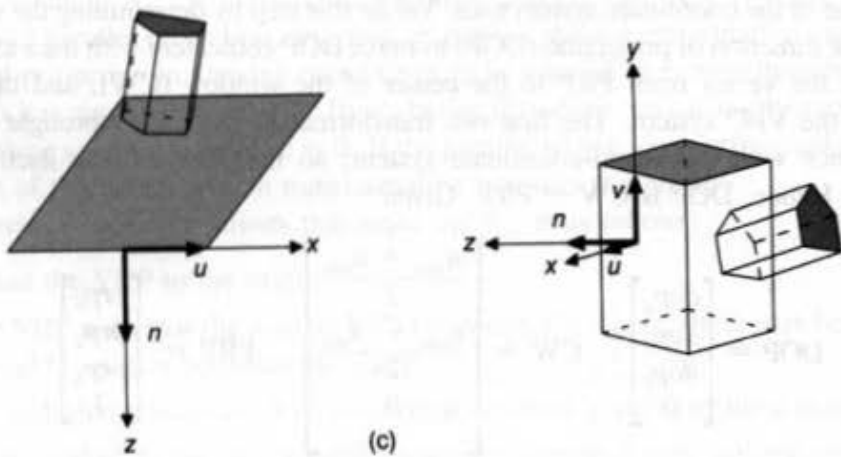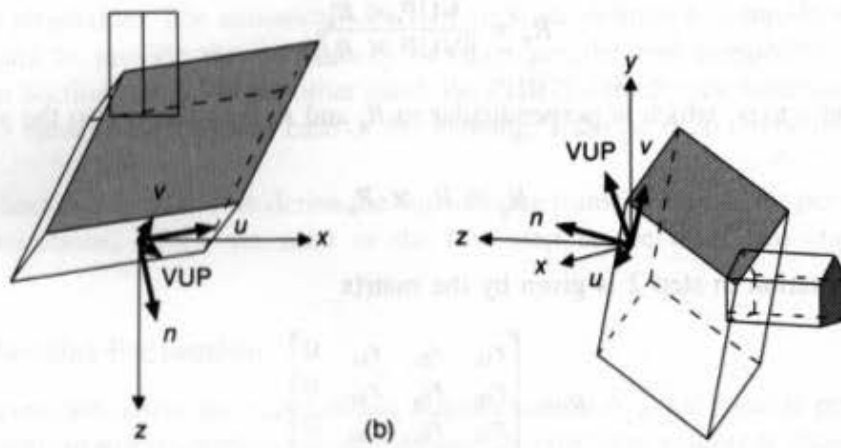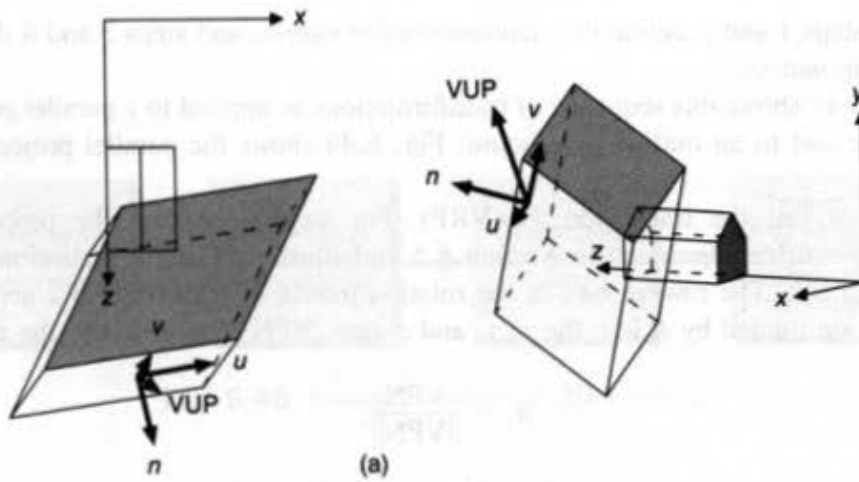
Fig. 6.47 Results at various stages in the parallel-projection viewing pipeline. A top and off-axis parallel projection are shown in each case. (a) The original viewing situation. (b) The VRP has been translated to the origin. (c) The (u, v, n) coordinate system has been rotated to be aligned with the (x, y, z) system. (d) The view volume has been sheared such that the direction of projection (DOP) is parallel to the z axis. (e) The view volume has been translated and scaled into the canonical parallel-projection view volume. The viewing parameters are VRP = (0.325, 0.8, 4.15), VPN = (.227, .267, 1.0), VUP = (.293, 1.0, .227), PRP = (0.6, 0.0, −1.0), Window = (−1.425, 1.0, −1.0, 1.0), $F = 0.0$, $B = −1.75$. (Figures made with program written by Mr. L. Lu, The George Washington University.)
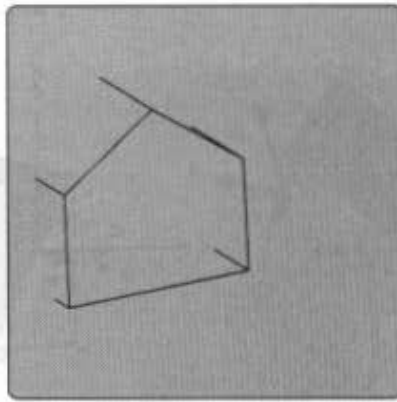
Fig. 6.48 Final parallel projection of the clipped house.

then

$$DOP = CW - PRP$$

$$= \left[\frac{u_{max} + u_{min}}{2} \quad \frac{v_{max} + v_{min}}{2} \quad 0 \quad 1 \right]^T - [prp_u \quad prp_v \quad prp_n \quad 1].^T \qquad (6.29)$$

Figure 6.49 shows the DOP so specified and the desired transformed DOP'.

The shear can be accomplished with the (x, y) shear matrix from Section 5.6 Eq. (5.47). With coefficients shx_{par} and shy_{par}, the matrix is

$$SH_{par} = SH_{xy}(shx_{par}, shy_{par}) = \begin{bmatrix} 1 & 0 & shx_{par} & 0 \\ 0 & 1 & shy_{par} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (6.30)$$

As described in Section 5.6, SH_{xy} leaves z unaffected, while adding to x and y the terms $z \cdot shx_{par}$ and $z \cdot shy_{par}$. We want to find shx_{par} and shy_{par} such that

$$DOP' = [0 \quad 0 \quad dop_z \quad 0]^T = SH_{par} \cdot DOP. \qquad (6.31)$$
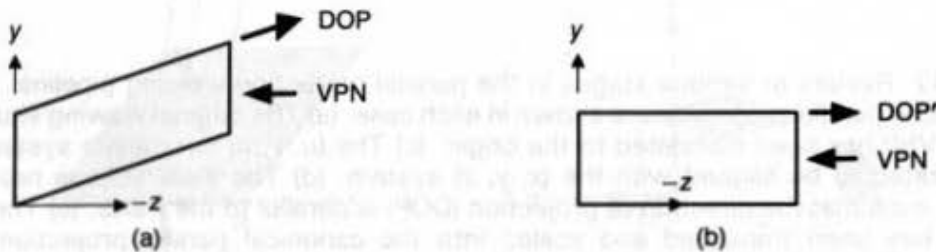
Fig. 6.49 Illustration of shearing using side view of view volume as example. The parallelogram in (a) is sheared into the rectangle in (b); VPN is unchanged because it is parallel to the z axis.

Performing the multiplication of Eq. (6.31) followed by algebraic manipulation shows that the equality occurs if

$$ shx_{par} = -\frac{dop_x}{dop_z}, \quad shy_{par} = -\frac{dop_y}{dop_z}. \tag{6.32} $$

Notice that, for an orthographic projection, $dop_x = dop_y = 0$, so $shx_{par} = shy_{par} = 0$, and the shear matrix reduces to the identity.

Figure 6.50 shows the view volume after these three transformation steps have been applied. The bounds of the volume are

$$ u_{min} \leq x \leq u_{max}, \quad v_{min} \leq y \leq v_{max}, \quad B \leq z \leq F; \tag{6.33} $$

here F and B are the distances from VRP along the VPN to the front and back clipping planes, respectively.

The fourth and last step in the process is transforming the sheared view volume into the canonical view volume. We accomplish this step by translating the front center of the view volume of Eq. (6.33) to the origin, then scaling to the $2 \times 2 \times 1$ size of the final canonical view volume of Eq. (6.22). The transformations are

$$ T_{par} = T\left(-\frac{u_{max} + u_{min}}{2}, -\frac{v_{max} + v_{min}}{2}, -F\right), \tag{6.34} $$

$$ S_{par} = S\left(\frac{2}{u_{max} - u_{min}}, \frac{2}{v_{max} - v_{min}}, \frac{1}{F - B}\right). \tag{6.35} $$

If F and B have not been specified (because front- and back-plane clipping are off), then any values that satisfy $B \leq F$ may be used. Values of 0 and 1 are satisfactory.

In summary, we have:

$$ N_{par} = S_{par} \cdot T_{par} \cdot SH_{par} \cdot R \cdot T(-VRP). \tag{6.36} $$

N_{par} transforms an arbitrary parallel-projection view volume into the parallel-projection canonical view volume, and hence permits output primitives to be clipped against the parallel-projection canonical view volume.
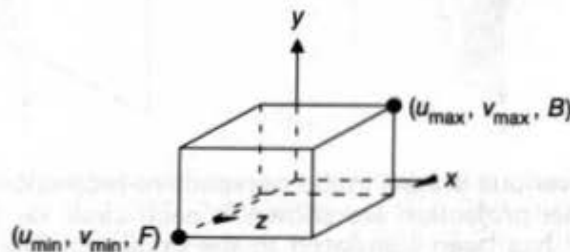
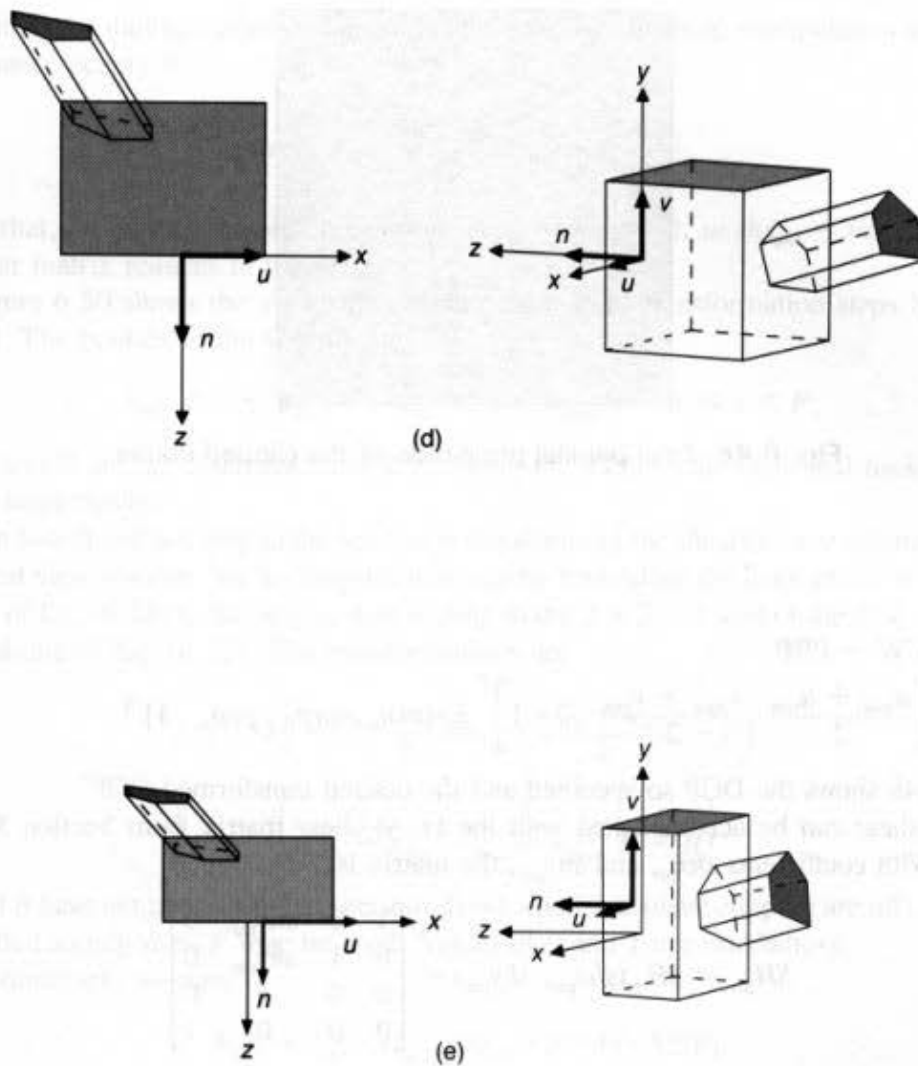**Fig. 6.50** View volume after transformation steps 1 to 3.
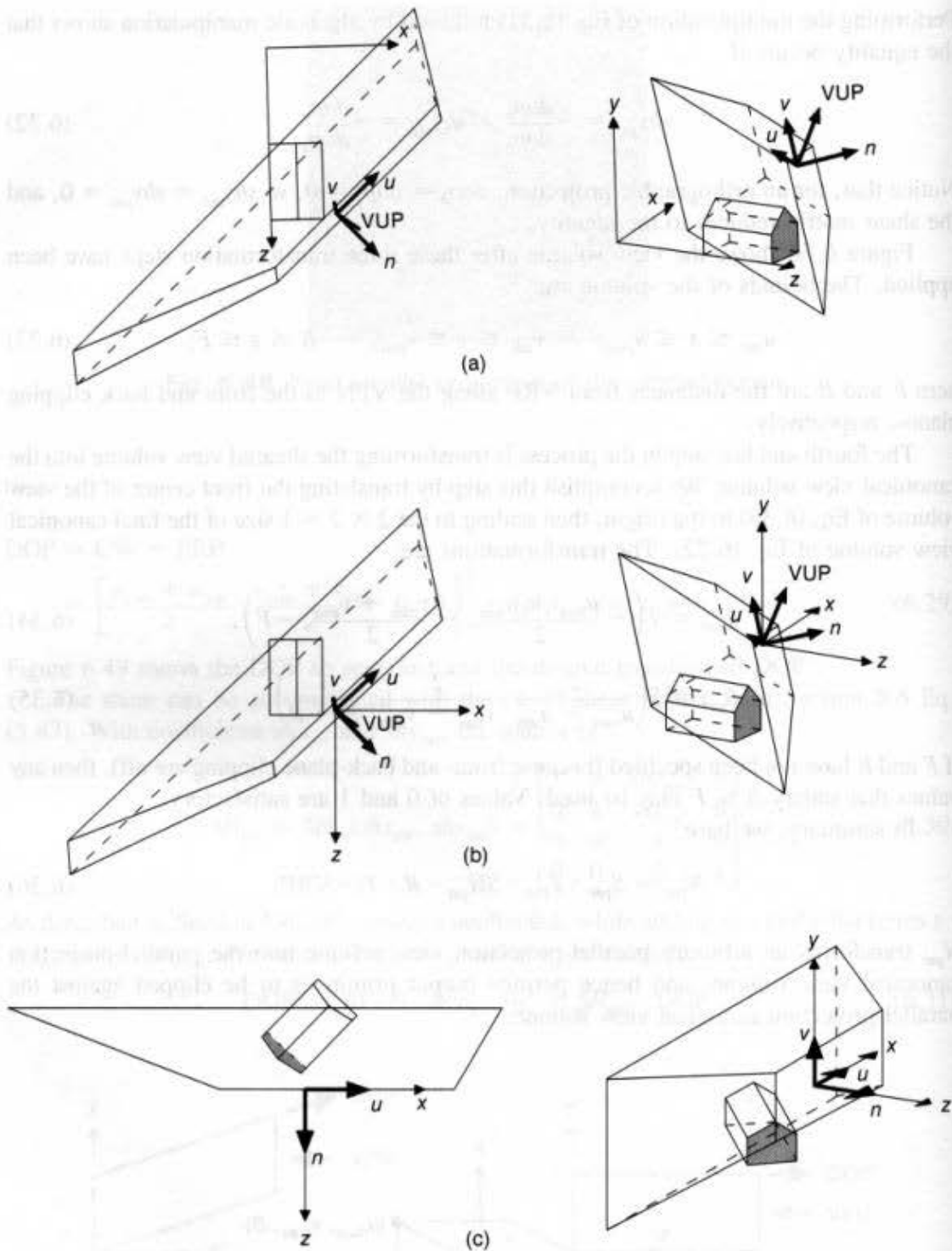
Fig. 6.51 Results at various stages in the perspective-projection viewing pipeline. A top and off-axis parallel projection are shown in each case. (a) The original viewing situation. (b) The VRP has been translated to the origin. (c) The (*u*, *v*, *n*) coordinate system has been rotated to be aligned with the (*x*, *y*, *z*) system. (d) The center of projection (COP) has been translated to the origin. (e) The view volume has been sheared, so the direction of projection (DOP) is parallel to the *z* axis. (f) The view volume

(d)

(e)

(f)
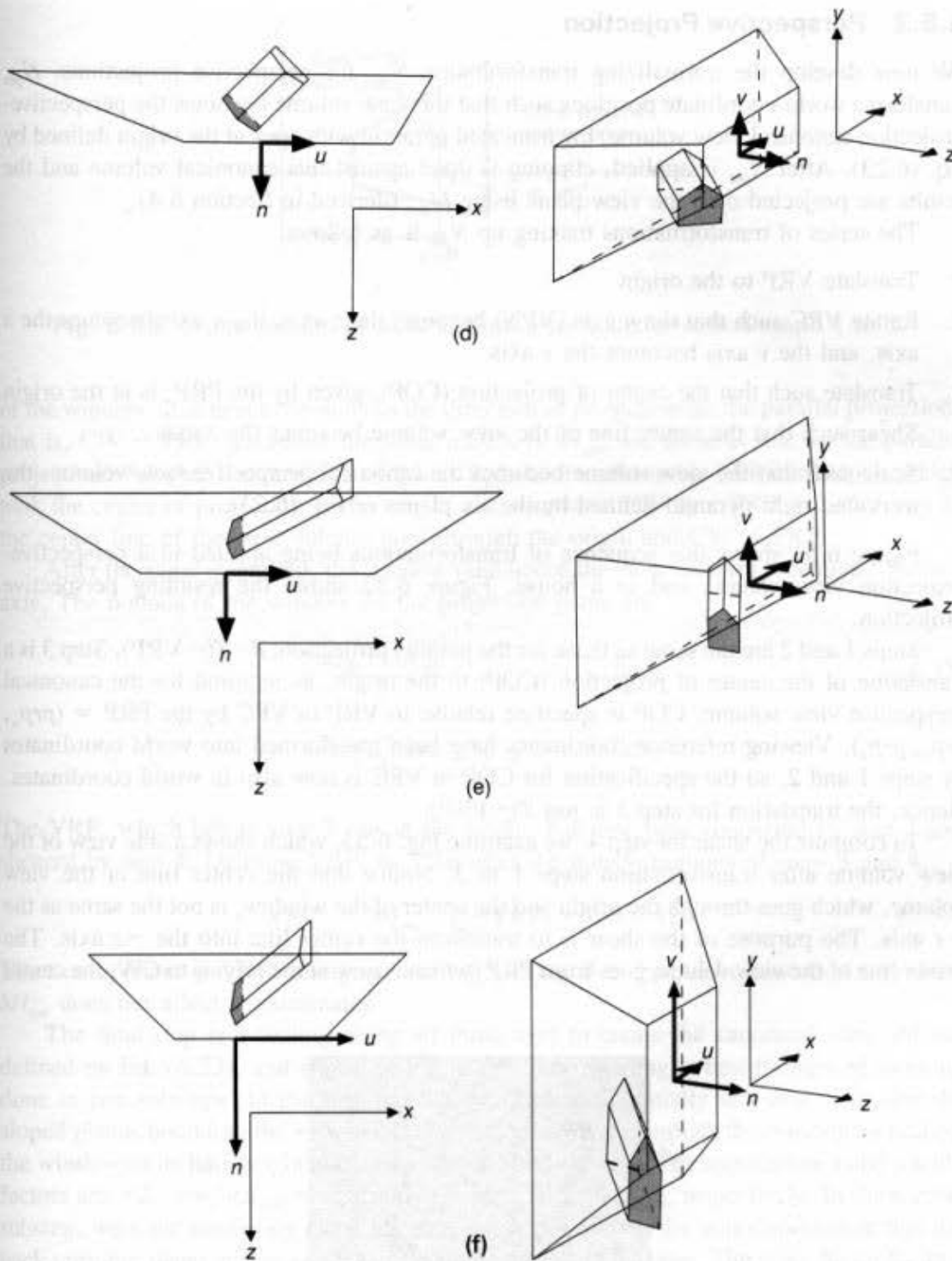
has been scaled into the canonical perspective-projection view volume. The viewing parameters are VRP = (1.0, 1.275, 2.6), VPN = (1.0, 0.253, 1.0), VUP = (0.414, 1.0, 0.253), PRP = (1.6, 0.0, 1.075), Window = (−1.325, 2.25, −0.575, 0.575), F = 0, B = −1.2. (Figures made with program written by Mr. L. Lu, The George Washington University.)

6.5.2 Perspective Projection

We now develop the normalizing transformation N_{per} for perspective projections. N_{per} transforms world-coordinate positions such that the view volume becomes the perspective-projection canonical view volume, the truncated pyramid with apex at the origin defined by Eq. (6.23). After N_{per} is applied, clipping is done against this canonical volume and the results are projected onto the view plane using M_{per} (derived in Section 6.4).

The series of transformations making up N_{per} is as follows:

1. Translate VRP to the origin

2. Rotate VRC such that the n axis (VPN) becomes the z axis, the u axis becomes the x axis, and the v axis becomes the y axis

3. Translate such that the center of projection (COP), given by the PRP, is at the origin

4. Shear such that the center line of the view volume becomes the z axis

5. Scale such that the view volume becomes the canonical perspective view volume, the truncated right pyramid defined by the six planes of Eq. (6.23).

Figure 6.51 shows this sequence of transformations being applied to a perspective-projection view volume and to a house. Figure 6.52 shows the resulting perspective projection.

Steps 1 and 2 are the same as those for the parallel projection: $R \cdot T(-VRP)$. Step 3 is a translation of the center of projection (COP) to the origin, as required for the canonical perspective view volume. COP is specified relative to VRP in VRC by the PRP = (prp_u, prp_v, prp_n). Viewing-reference coordinates have been transformed into world coordinates by steps 1 and 2, so the specification for COP in VRC is now also in world coordinates. Hence, the translation for step 3 is just $T(-PRP)$.

To compute the shear for step 4, we examine Fig. 6.53, which shows a side view of the view volume after transformation steps 1 to 3. Notice that the center line of the view volume, which goes through the origin and the center of the window, is not the same as the $-z$ axis. The purpose of the shear is to transform the center line into the $-z$ axis. The center line of the view volume goes from PRP (which is now at the origin) to CW, the center
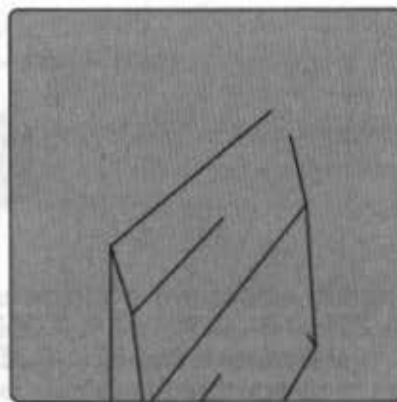
Fig. 6.52 Final perspective projection of the clipped house.
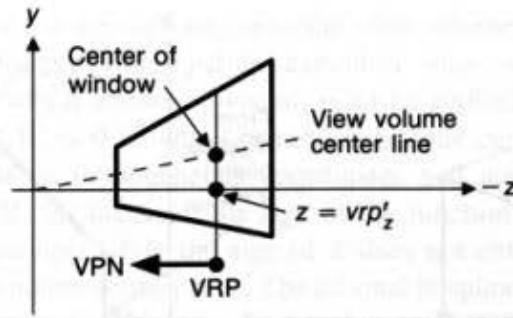
Fig. 6.53 Cross-section of view volume after transformation steps 1 to 3.

of the window. It is hence the same as the direction of projection for the parallel projection, that is, CW − PRP. Therefore the shear matrix is SH_{par}, the same as that for the parallel projection! Another way to think of this is that the translation by −PRP in step 3, which took the center of projection to the origin, also translated CW by −PRP, so, after step 3, the center line of the view volume goes through the origin and CW − PRP.

After the shear is applied, the window (and hence the view volume) is centered on the z axis. The bounds of the window on the projection plane are

$$-\frac{u_{\max} - u_{\min}}{2} \le x \le \frac{u_{\max} - u_{\min}}{2}, \tag{6.37}$$

$$-\frac{v_{\max} - v_{\min}}{2} \le y \le \frac{v_{\max} - v_{\min}}{2}.$$

The VRP, which before step 3 was at the origin, has now been translated by step 3 and sheared by step 4. Defining VRP′ as VRP after the transformations of steps 3 and 4,

$$\text{VRP}' = SH_{\text{par}} \cdot T(-\text{PRP}) \cdot [0\ 0\ 0\ 1]^T. \tag{6.38}$$

The z component of VRP′, designated as vrp_z', is equal to $-prp_n$, because the (x, y) shear SH_{par} does not affect z coordinates.

The final step is a scaling along all three axes to create the canonical view volume defined by Eq. (6.23), and shown in Fig. 6.54. Thus, scaling is best thought of as being done in two substeps. In the first substep, we scale differentially in x and y, to give the sloped planes bounding the view-volume unit slope. We accomplish this substep by scaling the window so its half-height and half-width are both $-vrp_z'$. The appropriate x and y scale factors are $-2 \cdot vrp_z'/(u_{\max} - u_{\min})$ and $-2 \cdot vrp_z'/(v_{\max} - v_{\min})$, respectively. In the second substep, we scale uniformly along all three axes (to maintain the unit slopes) such that the back clipping plane at $z = vrp_z' + B$ becomes the $z = -1$ plane. The scale factor for this substep is $-1/(vrp_z' + B)$. The scale factor has a negative sign so that the scale factor will be positive, since $vrp_z' + B$ is itself negative.

Bringing these two substeps together, we get the perspective scale transformation:

$$S_{\text{per}} = S\left(\frac{2\ vrp_z'}{(u_{\max} - u_{\min})(vrp_z' + B)}, \frac{2\ vrp_z'}{(v_{\max} - v_{\min})(vrp_z' + B)}, \frac{-1}{vrp_z' + B}\right). \tag{6.39}$$

(a) Before scaling   (b) After scaling
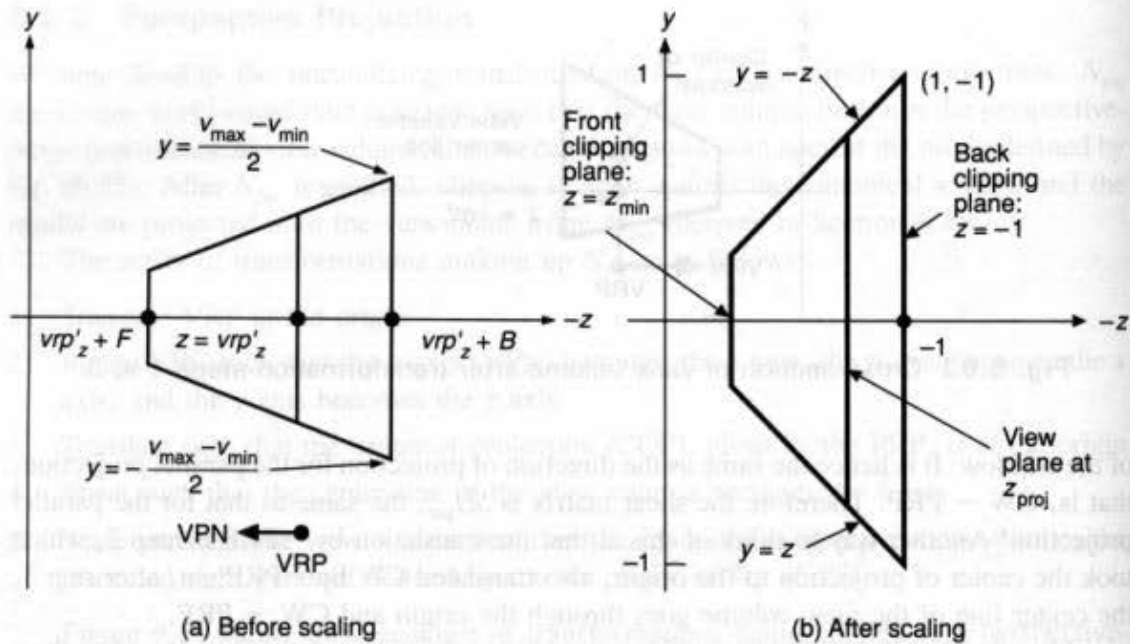
Fig. 6.54 Cross-section of view volume before and after final scaling steps. In this example, F and B have opposite signs, so the front and back planes are on opposite sides of VRP.

Applying the scale to z changes the positions of the projection plane and clipping planes to the new positions:

$$z_{proj} = -\frac{vrp_z'}{vrp_z' + B}, \quad z_{min} = -\frac{vrp_z' + F}{vrp_z' + B}, \quad z_{max} = -\frac{vrp_z' + B}{vrp_z' + B} = -1. \quad (6.40)$$

In summary, the normalizing viewing transformation that takes the perspective-projection view volume into the perspective-projection canonical view volume is

$$N_{per} = S_{per} \cdot SH_{par} \cdot T(-PRP) \cdot R \cdot T(-VRP). \quad (6.41)$$

Similarly, recall the normalizing viewing transformation that takes the parallel-projection view volume into the parallel-projection canonical view volume:

$$N_{par} = S_{par} \cdot T_{par} \cdot SH_{par} \cdot R \cdot T(-VRP). \quad (6.36)$$

These transformations occur in homogeneous space. Under what conditions can we now come back to 3D to clip? So long as we know that $W > 0$. This condition is easy to understand. A negative W implies that, when we divide by W, the signs of Z and z will be opposite. Points with negative Z will have positive z and might be displayed even though they should have been clipped.

When can we be sure that we will have $W > 0$? Rotations, translations, scales, and shears (as defined in Chapter 5) applied to points, lines, and planes will keep $W > 0$; in fact, they will keep $W = 1$. Hence, neither N_{per} nor N_{par} affects the homogeneous coordinate of transformed points, so division by W will not normally be necessary to map back into

3D, and clipping against the appropriate canonical view volume can be performed. After clipping against the perspective-projection canonical view volume, the perspective-projection matrix M_{per}, which involves division, must be applied.

It is possible to get $W < 0$ if output primitives include curves and surfaces that are represented as functions in homogeneous coordinates and are displayed as connected straight-line segments. If, for instance, the sign of the function for W changes from one point on the curve to the next while the sign of X does not change, then X/W will have different signs at the two points on the curve. The rational B-splines discussed in Chapter 11 are an example of such behavior. Negative W can also result from using some transformations other than those discussed in Chapter 5, such as with "fake" shadows [BLIN88].

In the next section, several algorithms for clipping in 3D are discussed. Then, in Section 6.5.4, we discuss how to clip when we cannot ensure that $W > 0$.

6.5.3   Clipping Against a Canonical View Volume in 3D

The canonical view volumes are the unit cube for parallel projections and the truncated right regular pyramid for perspective projections. Both the Cohen–Sutherland and Cyrus–Beck clipping algorithms discussed in Chapter 3 readily extend to 3D.

The extension of the 2D Cohen–Sutherland algorithm for the canonical parallel view volume uses an outcode of 6 bits; a bit is true (1) when the appropriate condition is satisfied:

bit 1—point is above view volume	$y > 1$
bit 2—point is below view volume	$y < -1$
bit 3—point is right of view volume	$x > 1$
bit 4—point is left of view volume	$x < -1$
bit 5—point is behind view volume	$z < -1$
bit 6—point is in front of view volume	$z > 0$

As in 2D, a line is trivially accepted if both endpoints have a code of all zeros, and is trivially rejected if the bit-by-bit logical **and** of the codes is not all zeros. Otherwise, the process of line subdivision begins. Up to six intersections may have to be calculated, one for each side of the view volume.

The intersection calculations use the parametric representation of a line from $P_0(x_0, y_0, z_0)$ to $P_1(x_1, y_1, z_1)$:

$$x = x_0 + t(x_1 - x_0), \tag{6.42}$$

$$y = y_0 + t(y_1 - y_0), \tag{6.43}$$

$$z = z_0 + t(z_1 - z_0) \quad 0 \le t \le 1. \tag{6.44}$$

As t varies from 0 to 1, the three equations give the coordinates of all points on the line, from P_0 to P_1.

To calculate the intersection of a line with the $y = 1$ plane of the view volume, we replace the variable y of Eq. (6.43) with 1 and solve for t to find $t = (1 - y_0)/(y_1 - y_0)$. If t is outside the 0 to 1 interval, the intersection is on the infinite line through points P_0 and P_1 but is not on the portion of the line between P_0 and P_1 and hence is not of interest. If t is in

[0, 1], then its value is substituted into the equations for x and z to find the intersection's coordinates:

$$x = x_0 + \frac{(1 - y_0)(x_1 - x_0)}{y_1 - y_0}, \quad z = z_0 + \frac{(1 - y_0)(z_1 - z_0)}{y_1 - y_0}. \tag{6.45}$$

The algorithm uses outcodes to make the t in [0, 1] test unnecessary.

The outcode bits for clipping against the canonical perspective view volume are as follows:

bit 1—point is above view volume $y > -z$
bit 2—point is below view volume $y < z$
bit 3—point is right of view volume $x > -z$
bit 4—point is left of view volume $x < z$
bit 5—point is behind view volume $z < -1$
bit 6—point is in front of view volume $z > z_{min}$

Calculating the intersections of lines with the sloping planes is simple. On the $y = z$ plane, for which Eq. (6.43) must be equal to Eq. (6.44), $y_0 + t(y_1 - y_0) = z_0 + t(z_1 - z_0)$. Then,

$$t = \frac{z_0 - y_0}{(y_1 - y_0) - (z_1 - z_0)}. \tag{6.46}$$

Substituting t into Eqs. (6.42) and (6.43) for x and y gives

$$x = x_0 + \frac{(x_1 - x_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}, \quad y = y_0 + \frac{(y_1 - y_0)(z_0 - y_0)}{(y_1 - y_0) - (z_1 - z_0)}. \tag{6.47}$$

We know that $z = y$. The reason for choosing this canonical view volume is now clear: The unit slopes of the planes make the intersection computations simpler than would arbitrary slopes.

The Cyrus–Beck clipping algorithm was formulated for clipping a line against a general convex 3D polyhedron, and specialized to the 3D viewing pyramid [CYRU78]. Liang and Barsky later independently developed a more efficient and more specialized version for upright 2D and 3D clip regions [LIAN84]. In 2D, at most four values of t are computed, one for each of the four window edges; in 3D, at most six values. The values are kept or discarded on the basis of exactly the same criteria as those for 2D until exactly two values of t remain (the values will be in the interval [0, 1]). For any value of t that is neither 0 nor 1, Eqs. (6.42), (6.43), and (6.44) are used to compute the values of x, y, and z. In the case of the parallel-projection canonical view volume, we do the extension to 3D by working with the code given in Section 3.12.4. In the case of the perspective-projection canonical view volume, new values of N_i, P_{Ei}, $P_0 - P_{Ei}$, and t need to be found for the six planes; they are given in Table 6.1. The actual code developed by Liang and Barsky but modified to be consistent with our variable names and to use outward rather than inward normals is given in Fig. 6.55. The correspondence between the numerator and denominator of the fractions in the last column of Table 6.1 and the terms in the code can be seen by inspection. The sign of $-N_i \cdot D$, which is the denominator of t, is used by procedure CLIPt in Fig. 6.55 to determine whether the intersection is potentially entering or potentially leaving. Hence, in

TABLE 6.1 KEY VARIABLES AND EQUATIONS FOR CYRUS–BECK 3D CLIPPING AGAINST THE CANONICAL PERSPECTIVE PROJECTION VIEW VOLUME*

Clip edge	Outward normal N_i	Point on edge, P_{Ei}	$P_0 - P_{Ei}$	$t = \dfrac{N_i \cdot (P_0 - P_{Ei})}{-N_i \cdot D}$	
right: $x = -z$	$(1, 0, 1)$	$(x, y, -x)$	$(x_0 - x,\ y_0 - y,\ z_0 + x)$	$\dfrac{(x_0 - x) + (z_0 + x)}{-(dx + dz)}$	$= \dfrac{x_0 + z_0}{-dx - dz}$
left: $x = z$	$(-1, 0, 1)$	(x, y, x)	$(x_0 - x,\ y_0 - y,\ z_0 - x)$	$\dfrac{-(x_0 - x) + (z_0 - x)}{dx - dz}$	$= \dfrac{-x_0 + z_0}{dx - dz}$
bottom: $y = z$	$(0, -1, 1)$	(x, y, y)	$(x_0 - x,\ y_0 - y,\ z_0 - y)$	$\dfrac{-(y_0 - y) + (z_0 - y)}{dy - dz}$	$= \dfrac{-y_0 + z_0}{dy - dz}$
top: $y = -z$	$(0, 1, 1)$	$(x, y, -y)$	$(x_0 - x,\ y_0 - y,\ z_0 + y)$	$\dfrac{(y_0 - y) + (z_0 + y)}{-dy - dz}$	$= \dfrac{y_0 + z_0}{-dy - dz}$
front: $z = z_{min}$	$(0, 0, 1)$	(x, y, z_{min})	$(x_0 - x,\ y_0 - y,\ z_0 - z_{min})$	$\dfrac{(z_0 - z_{min})}{-dz}$	$= \dfrac{z_0 - z_{min}}{-dz}$
back: $z = -1$	$(0, 0, -1)$	$(x, y, -1)$	$(x_0 - x,\ y_0 - y,\ z_0 + 1)$	$\dfrac{-(z_0 + 1)}{dz}$	$= \dfrac{-z_0 - 1}{dz}$

*The variable D, which is $P_1 - P_0$, is represented as (dx, dy, dz). The exact coordinates of the point P_{Ei} on each edge are irrelevant to the computation, so they have been denoted by variables x, y, and z. For a point on the right edge, $z = -x$, as indicated in the first row, third entry.

```
void Clip3D (double *x0, double *y0, double *z0, double *x1, double *y1, double *z1,
    double *zmin, boolean *accept)
{
    double tmin = 0.0, tmax = 1.0;
    double dx = *x1 − *x0, dz = *z1 − *z0;
    *accept = FALSE;        /* Assume initially that none of the line is visible */
    if (CLIPt (−dx − dz, *x0 + *z0, &tmin, &tmax))              /* Right side */
        if (CLIPt (dx − dz, −*x0 + *z0, &tmin, &tmax)) {        /* Left side */
            /* If get this far, part of line is in −z ≤ x ≤ z */
            double dy = *y1 − *y0;
            if (CLIPt (dy − dz, −*y0 + *z0, &tmin, &tmax))      /* Bottom */
                if (CLIPt (−dy − dz, *y0 + *z0, &tmin, &tmax))  /* Top */
                    /* If get this far, part of line is in −z ≤ x ≤ z, −z ≤ y ≤ z */
                    if (CLIPt (−dz, *z0 − *zmin, &tmin, &tmax))     /* Front */
                        if (CLIPt (dz, −*z0 − 1, &tmin, &tmax)) {   /* Back */
                            /* If get here, part of line is visible in −z ≤ x ≤ z, */
                            /* −z ≤ y ≤ z, −1 ≤ z ≤ zmin */
                            *accept = TRUE;              /* Part of line is visible */
                            /* If endpoint 1 (t = 1) is not in the region, compute intersection */
                            if (tmax < 1.0) {                /* Eqs. 6.37 to 6.39 */
                                *x1 = *x0 + tmax * dx;
                                *y1 = *y0 + tmax * dy;
                                *z1 = *z0 + tmax * dz;
                            }
                            /* If endpoint 0 (t = 0) is not in the region, compute intersection */
                            if (tmin > 0.0) {                /* Eqs. 6.37 to 6.39 */
                                *x0 += tmin * dx;
                                *y0 += tmin * dy;
                                *z0 += tmin * dz;
                            }
                        } /* Calculating intersection */
        }
} /* Clip3D */
```

Fig. 6.55 The Liang–Barsky 2D clipping algorithm, extended to the 3D canonical perspective-projection view volume. The code is from [LIAN84]. The function CLIPt is in Chapter 3, Fig. 3.45.

the algebraic simplifications done in the table, the sign of the denominator of t is always maintained.

The Sutherland–Hodgman polygon-clipping algorithm can be readily adapted to 3D. We use six clipping planes rather than four, by making six calls to S_H_CLIP (Chapter 3) instead of four.

Once the clipping is done, the surviving output primitives are projected onto the projection plane with either M_{ort} or M_{per}, and are transformed for display into the physical-device-coordinate viewport.

6.5.4 Clipping in Homogeneous Coordinates

There are two reasons to clip in homogeneous coordinates. The first has to do with efficiency: It is possible to transform the perspective-projection canonical view volume into the parallel-projection canonical view volume, so a single clip procedure, optimized for the parallel-projection canonical view volume, can always be used. However, the clipping must be done in homogeneous coordinates to ensure correct results. A single clip procedure is typically provided in hardware implementations of the viewing operation (Chapter 18). The second reason is that points that can occur as a result of unusual homogeneous transformations and from use of rational parametric splines (Chapter 11) can have negative W and can be clipped properly in homogeneous coordinates but not in 3D.

With regard to clipping, it can be shown that the transformation from the perspective-projection canonical view volume to the parallel-projection canonical view volume is

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{1}{1+z_{min}} & \dfrac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{bmatrix}, \quad z_{min} \neq -1. \tag{6.48}$$

Recall from Eq. (6.40) that $z_{min} = -(\text{vrp}'_z + F)/(\text{vrp}'_z + B)$, and from Eq. (6.38) that $\text{VRP}' = SH_{par} \cdot T(-\text{PRP}) \cdot [0\ 0\ 0\ 1]^T$. Figure 6.56 shows the results of applying M to the perspective-projection canonical view volume.

The matrix M is integrated with the perspective-projection normalizing transformation N_{per}:

$$N'_{per} = M \cdot N_{per} = M \cdot S_{per} \cdot SH_{par} \cdot T(-\text{PRP}) \cdot R \cdot T(-\text{VRP}). \tag{6.49}$$

By using N'_{per} instead of N_{per} for perspective projections, and by continuing to use N_{par} for parallel projections, we can clip against the parallel-projection canonical view volume rather than against the perspective-projection canonical view volume.
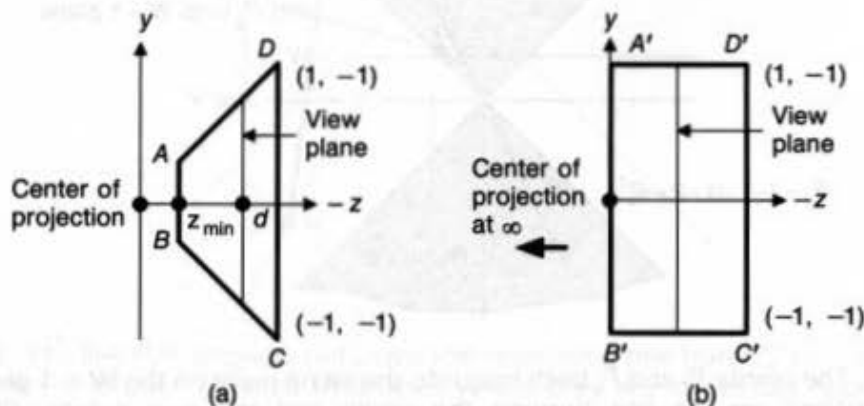
Fig 6.56 Side views of normalized perspective view volume before (a) and after (b) application of matrix M.

The 3D parallel-projection view volume is defined by $-1 \leq x \leq 1$, $-1 \leq y \leq 1$, $-1 \leq z \leq 0$. We find the corresponding inequalities in homogeneous coordinates by replacing x by X/W, y by Y/W, and z by Z/W, which results in

$$-1 \leq X/W \leq 1, \quad -1 \leq Y/W \leq 1, \quad -1 \leq Z/W \leq 0. \tag{6.50}$$

The corresponding plane equations are

$$X = -W, X = W, Y = -W, Y = W, Z = -W, Z = 0. \tag{6.51}$$

To understand how to use these limits and planes, we must consider separately the cases of $W > 0$ and $W < 0$. In the first case, we can multiply the inequalities of Eq. (6.50) by W without changing the sense of the inequalities. In the second case, the multiplication changes the sense. This result can be expressed as

$$W > 0: -W \leq X \leq W, \quad -W \leq Y \leq W, \quad -W \leq Z \leq 0, \tag{6.52}$$

$$W < 0: -W \geq X \geq W, \quad -W \geq Y \geq W, \quad -W \geq Z \geq 0. \tag{6.53}$$

In the case at hand—that of clipping ordinary lines and points—only the region given by Eq. (6.52) needs to be used, because prior to application of M, all visible points have $W > 0$ (normally $W = 1$).

As we shall see in Chapter 11, however, it is sometimes desirable to represent points directly in homogeneous coordinates with arbitrary W coordinates. Hence, we might have a $W < 0$, meaning that clipping must be done against the regions given by Eqs. (6.52) and (6.53). Figure 6.57 shows these as region A and region B, and also shows why both regions must be used.
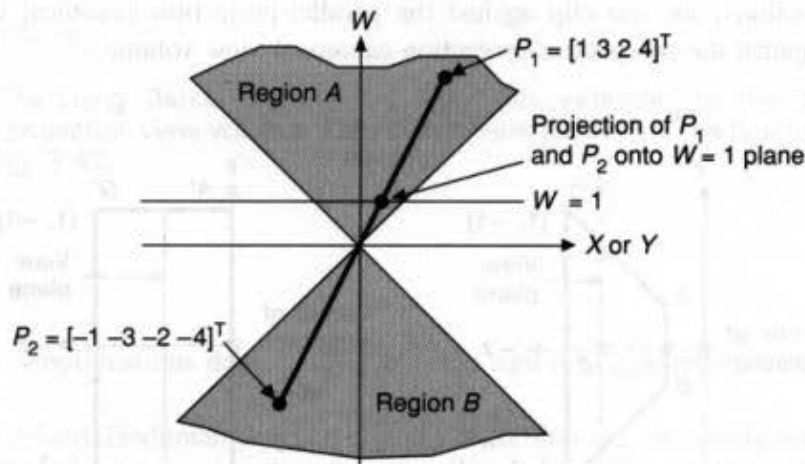
Fig. 6.57 The points P_1 and P_2 both map into the same point on the $W = 1$ plane, as do all other points on the line through the origin and the two points. Clipping in homogeneous coordinates against just region A will incorrectly reject P_2.

The point $P_1 = [1 \quad 3 \quad 2 \quad 4]^T$ in region A transforms into the 3D point $(\frac{1}{4}, \frac{3}{4}, \frac{2}{4})$, which is in the canonical view volume $-1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 0$. The point $P_2 = -P_1 = [-1 \quad -3 \quad -2 \quad -4]^T$, which is *not* in region A but *is* in region B, transforms into the same 3D point as P_1; namely, $(\frac{1}{4}, \frac{3}{4}, \frac{2}{4})$. If clipping were only to region A, then P_2 would be discarded incorrectly. This possibility arises because the homogeneous coordinate points P_1 and P_2 differ by a constant multiplier (-1), and we know that such homogeneous points correspond to the same 3D point (on the $W = 1$ plane of homogeneous space).

There are two solutions to this problem of points in region B. One is to clip all points twice, once against each region. But doing two clips is expensive. A better solution is to negate points, such as P_2, with negative W, and then to clip them. Similarly, we can clip properly a line whose endpoints are both in region B of Fig. 6.57 by multiplying both endpoints by -1, to place the points in region A.

Another problem arises with lines such as P_1P_2, shown in Fig. 6.58, whose endpoints have opposite values of W. The projection of the line onto the $W = 1$ plane is two segments, one of which goes to $+\infty$, the other to $-\infty$. The solution now is to clip twice, once against each region, with the possibility that each clip will return a visible line segment. A simple way to do this is to clip the line against region A, to negate both endpoints of the line, and to clip again against region A. This approach preserves one of the original purposes of clipping in homogeneous coordinates: using a single clip region. Interested readers are referred to [BLIN78a] for further discussion.
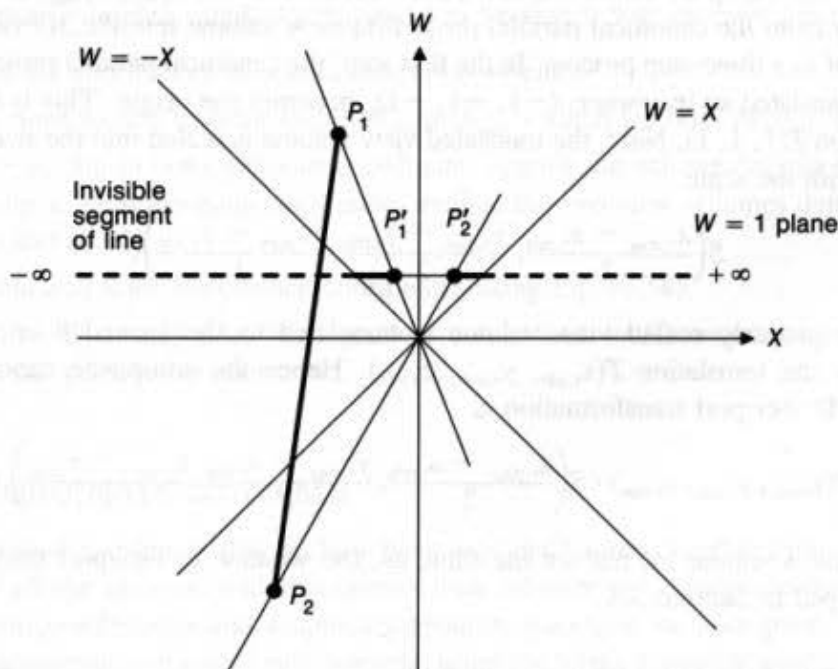
Fig. 6.58 The line P_1P_2 projects onto two line segments, one from P_2' to $+\infty$, the other from P_1' to $-\infty$ (shown as solid thick lines where they are in the clip region, and as dashed thick lines where they are outside the clip region). The line must be clipped twice, once against each region.

Given Eq. (6.51), the Cohen–Sutherland or Cyrus–Beck algorithm can be used for the actual clipping. ([LIAN84] gives code for the Cyrus–Beck approach; see also Exercise 6.25.) The only difference is that the clipping is in 4D, as opposed to 3D.

6.5.5 Mapping into a Viewport

Output primitives are clipped in the normalized projection coordinate system, which is also called the 3D screen coordinate system. We will assume for this discussion that the canonical parallel projection view volume has been used for clipping (the perspective projection M transforms the perspective projection view volume into the parallel projection view volume if this assumption is incorrect). Hence the coordinates of all output primitives that remain are in the view volume $-1 \leq x \leq 1, -1 \leq y \leq 1, -1 \leq z \leq 0$.

The PHIGS programmer specifies a 3D viewport into which the contents of this view volume are mapped. The 3D viewport is contained in the unit cube $0 \leq x \leq 1, 0 \leq y \leq 1, 0 \leq z \leq 1$. The $z = 1$ front face of the unit cube is mapped into the largest square that can be inscribed on the display screen. We assume that the lower left corner of the square is at $(0, 0)$. For example, on a display device with a horizontal resolution of 1024 and a vertical resolution of 800, the square is the region $0 \leq x \leq 799, 0 \leq y \leq 799$. Points in the unit cube are displayed by discarding their z coordinate. Hence the point $(0.5, 0.75, 0.46)$ would be displayed at the device coordinates $(400, 599)$. In the case of visible-surface determination (Chapter 15), the z coordinate of each output primitive is used to determine which primitives are visible and which are obscured by other primitives with larger z.

Given a 3D viewport within the unit cube with coordinates $x_{v.min}$, $x_{v.max}$, and so forth, the mapping from the canonical parallel projection view volume into the 3D viewport can be thought of as a three-step process. In the first step, the canonical parallel projection view volume is translated so its corner, $(-1, -1, -1)$, becomes the origin. This is effected by the translation $T(1, 1, 1)$. Next, the translated view volume is scaled into the size of the 3D viewport, with the scale

$$S\left(\frac{x_{v.max} - x_{v.min}}{2}, \frac{y_{v.max} - y_{v.min}}{2}, \frac{z_{v.max} - z_{v.min}}{1}\right).$$

Finally, the properly scaled view volume is translated to the lower-left corner of the viewport by the translation $T(x_{v.min}, y_{v.min}, z_{z.min})$. Hence the composite canonical view volume to 3D viewport transformation is

$$M_{VV3DV} = T(x_{v.min}, y_{v.min}, z_{v.min}) \cdot S\left(\frac{x_{v.max} - x_{v.min}}{2}, \frac{y_{v.max} - y_{v.min}}{2}, \frac{z_{v.max} - z_{v.min}}{1}\right) \cdot T(1, 1, 1)$$

$$(6.54)$$

Note that this is similar to, but not the same as, the window to viewport transformation M_{WV} developed in Section 5.4.

6.5.6 Implementation Summary

There are two generally used implementations of the overall viewing transformation. The first, depicted in Fig. 6.46 and discussed in Sections 6.5.1 through 6.5.3, is appropriate when

output primitives are defined in 3D and the transformations applied to the output primitives never create a negative W. Its steps are as follows:

1. Extend 3D coordinates to homogeneous coordinates
2. Apply normalizing transformation N_{par} or N_{per}
3. Divide by W to map back to 3D (in some cases, it is known that $W = 1$, so the division is not needed)
4. Clip in 3D against the parallel-projection or perspective-projection canonical view volume, whichever is appropriate
5. Extend 3D coordinates to homogeneous coordinates
6. Perform parallel projection using M_{ort}, Eq. (6.11), or perform perspective projection, using M_{per}, Eq. (6.3) with $d = -1$ (because the canonical view volume lies along the $-z$ axis)
7. Translate and scale into device coordinates using Eq. (6.54)
8. Divide by W to map from homogeneous to 2D coordinates; the division effects the perspective projection.

Steps 6 and 7 are performed by a single matrix multiplication, and correspond to stages 3 and 4 in Fig. 6.46.

The second way to implement the viewing operation is required whenever output primitives are defined in homogeneous coordinates and might have $W < 0$, when the transformations applied to the output primitives might create a negative W, or when a single clip algorithm is implemented. As discussed in Section 6.5.4, its steps are as follows:

1. Extend 3D coordinates to homogeneous coordinates
2. Apply normalizing transformation N_{par} or N_{per}' (which includes M, Eq. (6.48))
3. If $W > 0$, clip in homogeneous coordinates against the volume defined by Eq. 6.52; else, clip in homogeneous coordinates against the two view volumes defined by Eqs. (6.52) and (6.53)
4. Translate and scale into device coordinates using Eq. (6.54)
5. Divide by W to map from homogeneous coordinates to 2D coordinates; the division effects the perspective projection.

6.6 COORDINATE SYSTEMS

Several different coordinate systems have been used in Chapters 5 and 6. In this section, we summarize all the systems, and also discuss their relationships to one another. Synonyms used in various references and graphics subroutine packages are also given. Figure 6.59 shows the progression of coordinate systems, using the terms generally used in this text; in any particular graphics subroutine package, only some of the coordinate systems are actually used. We have chosen names for the various coordinate systems to reflect common usage; some of the names therefore are not logically consistent with one another. Note that the term *space* is sometimes used as a synonym for *system*.
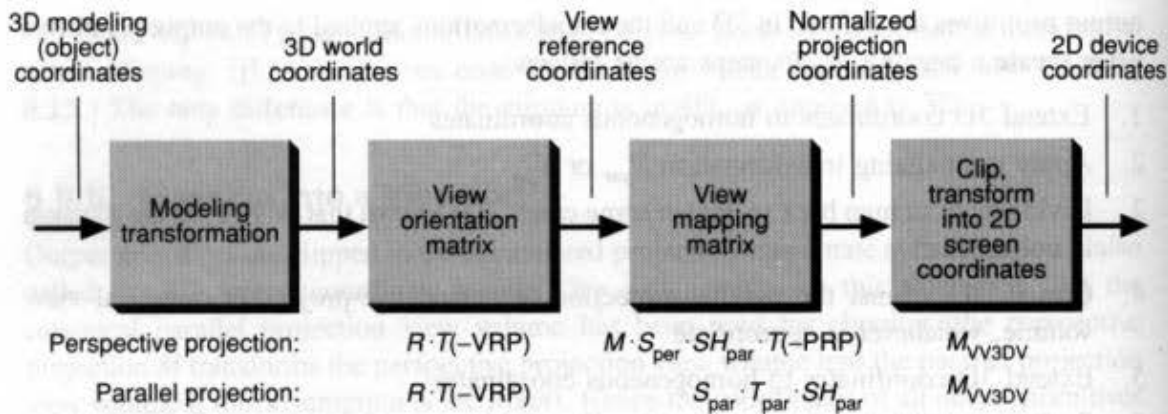
Fig. 6.59 Coordinate systems and how they relate to one another. The matrices underneath each stage effect the transformation applied at that stage for the perspective and parallel projections.

Starting with the coordinate system that is furthest removed from the actual display device, on the left of Fig. 6.59, individual objects are defined in an *object-coordinate system*. PHIGS calls this the *modeling-coordinate system*; the term *local coordinate system* is also commonly used. As we shall discuss further in Chapter 7, there is often a hierarchy of modeling coordinate systems.

Objects are transformed into the *world-coordinate system*, the system in which a scene or complete object is represented in the computer, by the *modeling transformation*. This system is sometimes called the *problem-coordinate system* or *application-coordinate system*.

The *view-reference coordinate system* is used by PHIGS as a coordinate system to define a view volume. It is also called the (u, v, n) system, or the (u, v, VPN) system. The Core system [GSPC79] used a similar, but unnamed, left-handed system. The left-handed system is used so that, with the eye or camera at the origin looking toward $+z$, increasing values of z are farther away from the eye, x is to the right, and y is up.

Other packages, such as Pixar's RenderMan [PIXA88], place constraints on the view-reference coordinate system, requiring that the origin be at the center of projection and that the view plane normal be the z axis. We call this the *eye-coordinate system*; Renderman and some other systems use the term *camera-coordinate system*. Referring back to Section 6.5, the first three steps of the perspective-projection normalizing transformation convert from the world-coordinate system into the eye-coordinate system. The eye-coordinate system is sometimes left-handed.

From eye coordinates, we next go to the *normalized-projection coordinate system*, or *3D screen coordinates*, the coordinate system of the parallel-projection canonical view volume (and of the perspective-projection canonical view volume after the perspective transformation). The Core system calls this system *3D normalized device coordinates*. Sometimes, the system is called *3D logical device coordinates*. The term *normalized* generally means that all the coordinate values are in either the interval [0, 1] or [−1, 1], whereas the term *logical* generally means that coordinate values are in some other

prespecified range, such as [0, 1023], which is typically defined to correspond to some widely available device's coordinate system. In some cases, this system is not normalized.

Projecting from 3D into 2D creates what we call the *2D device coordinate system*, also called the *normalized device-coordinate system*, the *image-coordinate system* by [SUTH74], or the *screen-coordinate system* by RenderMan. Other terms used include *screen coordinates*, *device coordinates*, *2D device coordinates*, *physical device coordinates* (in contrast to the logical device coordinates mentioned previously). RenderMan calls the physical form of the space *raster coordinates*.

Unfortunately, there is no single standard usage for many of these terms. For example, the term *screen-coordinate system* is used by different authors to refer to the last three systems discussed, covering both 2D and 3D coordinates, and both logical and physical coordinates.

EXERCISES

6.1 Write a program that accepts a viewing specification, calculates either N_{par} or N_{per}, and displays the house.

6.2 Program 3D clipping algorithms for parallel and perspective projections.

6.3 Show that, for a parallel projection with $F = -\infty$ and $B = +\infty$, the result of clipping in 3D and then projecting to 2D is the same as the result of projecting to 2D and then clipping in 2D.

6.4 Show that, if all objects are in front of the center of projection and if $F = -\infty$ and $B = +\infty$, then the result of clipping in 3D against the perspective-projection canonical view volume followed by perspective projection is the same as first doing a perspective projection into 2D and then clipping in 2D.

6.5 Verify that S_{per} (Section 6.5.2) transforms the view volume of Fig. 6.54(a) into that of Fig. 6.54(b).

6.6 Write the code for 3D clipping against the unit cube. Generalize the code to clip against any rectangular solid with faces normal to the principal axes. Is the generalized code more or less efficient than that for the unit-cube case?

6.7 Write the code for 3D clipping against the perspective-projection canonical view volume. Now generalize to the view volume defined by

$$-a \cdot z_v \leq x_v \leq b \cdot z_v, \quad -c \cdot z_v \leq y_v \leq d \cdot z_v, \quad z_{min} \leq z_v \leq z_{max}.$$

This is the general form of the view volume after steps 1 to 4 of the perspective normalizing transformation. Which case is more efficient?

6.8 Write the code for 3D clipping against a general six-faced polyhedral view volume whose faces are defined by

$$A_i x + B_i y + C_i z + D_i = 0, 1 \leq i \leq 6.$$

Compare the computational effort needed with that required for each of the following:

    a. Clipping against either of the canonical view volumes
    b. Applying N_{par} and then clipping against the unit cube.

6.9 Consider a line in 3D going from the world-coordinate points P_1 (6, 10, 3) to P_2 (−3, −5, 2) and a semi-infinite viewing pyramid in the region $-z \leq x \leq z$, $-z \leq y \leq z$, which is bounded by the planes $z = +x$, $z = -x$, $z = +y$, $z = -y$. The projection plane is at $z = 1$.

    a. Clip the line in 3D (using parametric line equations), then project it onto the projection plane. What are the clipped endpoints on the plane?

    b. Project the line onto the plane, then clip the lines using 2D computations. What are the clipped endpoints on the plane?

(Hint: If your answers to (a) and (b) are not identical, try again!)

6.10 Show what happens when an object "behind" the center of projection is projected by M_{per} and then clipped. Your answer should demonstrate why, in general, one cannot project and then clip.

6.11 Consider the 2D viewing operation, with a rotated window. Devise a normalized transformation to transform the window into the unit square. The window is specified by u_{min}, v_{min}, u_{max}, v_{max} in the VRC coordinate system, as in Fig. 6.60. Show that this transformation is the same as that for the general 3D N_{par}, when the projection plane is the (x, y) plane and VUP has an x component of $-\sin\theta$ and a y component of $\cos\theta$ (i.e., the parallel projection of VUP onto the view plane is the v axis).

6.12 The matrix M_{per} in Section 6.4 defines a one-point perspective projection. What is the form of the 4×4 matrix that defines a two-point perspective? What is the form of the matrix that defines a three-point perspective? (Hint: Try multiplying M_{per} by various rotation matrices.)

6.13 What is the effect of applying M_{per} to points whose z coordinate is less than zero?

6.14 Devise a clipping algorithm for a cone-shaped (having a circular cross-section) view volume. The cone's apex is at the origin, its axis is the positive z axis, and it has a 90° interior angle. Consider using a spherical coordinate system.

6.15 Design and implement a set of utility subroutines to generate a 4×4 transformation matrix from an arbitrary sequence of R, S, and T primitive transformations.

6.16 Draw a decision tree to be used when determining the type of a projection used in creating an image. Apply this decision tree to the figures in this chapter that are projections from 3D.

6.17 Evaluate the speed tradeoffs of performing 3D clipping by using the Cohen–Sutherland algorithm versus using the 3D Cyrus–Beck algorithm. First, write an assembly-language version of the program. Count operations executed for each fundamentally different case treated by the algorithms, and weight the cases equally. (For example, in the Cohen–Sutherland algorithm, trivial accept, trivial reject, and calculating 1, 2, 3, or 4 interesections are the fundamentally different cases.) Ignore the time taken by subroutine calls (assume that in-line code is used). Differentiate instructions by the number of cycles they require.
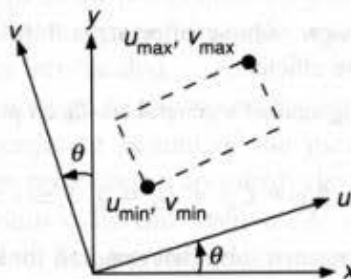
Fig. 6.60 A rotated window.
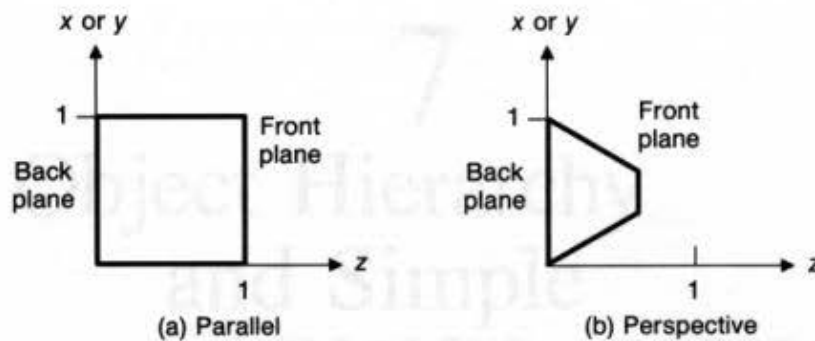
Fig. 6.61 The default view volumes used by PHIGS. The view volumes are slightly different from those used in this chapter, as given in Fig. 6.45.

6.18 In Exercise 6.17, we made the assumption that all cases were equally likely. Is this a good assumption? Explain your answer. Apply either of the algorithms to a variety of real objects, using various view volumes, to get a sense of how the mix of cases varies in a realistic setting.

6.19 The canonical view volume for the parallel projection was taken to be the $2 \times 2 \times 1$ rectangular parallelepiped. Suppose the unit cube in the positive octant, with one corner at the origin, is used instead.

    a. Find the normalization N_{par}' for this view volume.

    b. Find the corresponding homogeneous-coordinate view volume.

6.20 Give the viewing parameters for top, front, and side views of the house with the VRP in the middle of the window. Must the PRP be different for each of the views? Why or why not?

6.21 Verify that Eq. (6.48) does indeed transform the canonical perspective view volume of Fig. 6.56(a) into the canonical parallel view volume of Fig. 6.56(b).

6.22 In PHIGS, the viewing specification allows the view plane to be at a distance VPD from the VRP. Redevelop the canonical viewing transformations N_{par} and N_{per} to include VPD. Set VPD = 0 to make sure that your results specialize to those in the text.

6.23 Reformulate the derivations of N_{per} and N_{par} to incorporate the matrix $M_{general}$.

6.24 Show how to convert from the six viewing parameters of Fig. 6.35 into the viewing parameters discussed in this chapter. Write and test a utility subroutine to effect the conversion.

6.25 For the homogeneous clip planes $X = -W$, $X = W$, $Y = -W$, $Y = W$, $Z = 0$, $Z = -W$, complete a table like Table 6.1 for using the Cyrus–Beck clipping approach for the W > 0 region given by $W > 0$, $-W \le X \le W$, $-W \le Y \le W$, $-W \le Z \le 0$.

6.26 PHIGS uses the default view volumes shown in Fig. 6.61. These are slightly different from the canonical view volumes used in this chapter, as shown in Fig. 6.45. Modify the N_{par} and N_{per} transformations to map into the PHIGS default.

6.27 Stereo pairs are two views of the same scene made from slightly different projection reference points, but with the same view reference point. Let d be the stereo separation; that is, the distance between the two reference points. If we think of the reference points as our eyes, then d is the distance between our eyes. Let P be the point midway between our eyes. Given P, d, VRP, VPN, and VUP, derive expressions for the two projection reference points.

7
Object Hierarchy and Simple PHIGS (SPHIGS)

Andries van Dam
and David Sklar

A graphics package is an intermediary between an application program and the graphics hardware. The output primitives and interaction devices that a graphics package supports can range from rudimentary to extremely rich. In Chapter 2, we described the fairly simple and low-level SRGP package, and we noted some of its limitations. In this chapter, we describe a package based on a considerably richer but more complex standard graphics package, PHIGS (Programmer's Hierarchical Interactive Graphics System[1]). A *standard graphics package* such as PHIGS or GKS (Graphical Kernel System) implements a specification designated as standard by an official national or international standards body; GKS and PHIGS have been so designated by ANSI (American National Standards Institute) and ISO (International Standards Organization). The main purpose of such standards is to promote portability of application programs and of programmers. Nonofficial standards are also developed, promoted, and licensed by individual companies or by consortia of companies and universities; Adobe's PostScript and MIT's X Window System are two of the current industry standards.

The package described here is called SPHIGS (for Simple PHIGS; pronounced "ess-figs") because it is essentially a subset of PHIGS. It preserves most of PHIGS's capabilities and power, but simplifies or modifies various features to suit straightforward applications. SPHIGS also includes several enhancements adapted from PHIGS+ extensions. Our aim in designing SPHIGS has been to introduce concepts in the simplest possible

---

[1]The term "PHIGS" in this chapter also includes a set of extensions to PHIGS, called PHIGS+, that supports advanced geometric primitives such as polyhedra, curves, and surfaces, as well as rendering techniques that use lighting, shading, and depth-cueing, discussed in Chapters 14–16.

way, not to provide a package that is strictly upward-compatible with PHIGS. However, an SPHIGS application can easily be adapted to run with PHIGS. Footnotes present some of the important differences between SPHIGS and PHIGS; in general, an SPHIGS feature is also present in PHIGS unless otherwise noted.

There are three major differences between SPHIGS and integer raster packages, such as SRGP or the Xlib package of the X Window System. First, to suit engineering and scientific applications, SPHIGS uses a 3D, floating-point coordinate system, and implements the 3D viewing pipeline discussed in Chapter 6.

The second, farther-reaching difference is that SPHIGS maintains a database of structures. A *structure* is a logical grouping of primitives, attributes, and other information. The programmer can modify structures in the database with a few editing commands; SPHIGS ensures that the screen's image is an accurate representation of the contents of the stored database. Structures contain not only specifications of primitives and attributes, but also invocations of subordinate structures. They thus exhibit some of the properties of procedures in programming languages. In particular, just as procedure hierarchy is induced by procedures invoking subprocedures, structure hierarchy is induced by structures invoking substructures. Such hierarchical composition is especially powerful when one can control the geometry (position, orientation, size) and appearance (color, style, thickness, etc.) of any invocation of a substructure.

The third difference is that SPHIGS operates in an abstract, 3D world-coordinate system, not in 2D screen space, and therefore does not support direct pixel manipulation. Because of these differences, SPHIGS and SRGP address different sets of needs and applications; as we pointed out in Chapter 2, each has its place—no single graphics package meets all needs.

Because of its ability to support structure hierarchy, SPHIGS is well suited to applications based on models with component–subcomponent hierarchy; indeed, the SPHIGS structure hierarchy can be viewed as a special-purpose modeling hierarchy. We therefore look at modeling in general in Section 7.1, before discussing the specifics of geometric modeling with SPHIGS. Sections 7.2 through 7.9 show how to create, display, and edit the SPHIGS structure database. Section 7.10 discusses interaction, particularly pick correlation. The remainder of the chapter presents PHIGS features not found in SPHIGS, discusses implementation issues, and closes with evaluations of SPHIGS and of alternative methods for encoding hierarchy.

7.1 GEOMETRIC MODELING

7.1.1 What Is a Model?

You have encountered many examples of models in courses in the physical and social sciences. For example, you are probably familiar with the Bohr model of the atom, in which spheres representing electrons orbit a spherical nucleus containing neutron and proton spheres. Other examples are the exponential unconstrained growth model in biology, and macro- or microeconometric models that purport to describe some aspect of an economy. A model is a representation of some (not necessarily all) features of a concrete or abstract entity. The purpose of a model of an entity is to allow people to visualize and understand

the structure or behavior of the entity, and to provide a convenient vehicle for "experimentation" with and prediction of the effects of inputs or changes to the model. Quantitative models common in physical and social sciences and engineering are usually expressed as systems of equations, and the modeler will experiment by varying the values of independent variables, coefficients, and exponents. Often, models simplify the actual structure or behavior of the modeled entity to make the model easier to visualize or, for those models represented by systems of equations, to make the model computationally tractable.

We restrict ourselves in this book to the discussion of computer-based models—in particular, to those that lend themselves to graphic interpretation. Graphics can be used to create and edit the model, to obtain values for its parameters, and to visualize its behavior and structure. The model and the graphical means for creating and visualizing it are distinct; models such as population models need not have any inherent graphical aspects. Among common types of models for which computer graphics is used are these:

- *Organizational models* are hierarchies representing institutional bureaucracies and taxonomies, such as library classification schemes and biological taxonomies. These models have various directed-graph representations, such as the organization chart.

- *Quantitative models* are equations describing econometric, financial, sociological, demographic, climatic, chemical, physical, and mathematical systems. These are often depicted by graphs or statistical plots.

- *Geometric models* are collections of components with well-defined geometry and, often, interconnections between components, including engineering and architectural structures, molecules and other chemical structures, geographic structures, and vehicles. These models are usually depicted by block diagrams or by pseudorealistic "synthetic photographs."

Computer-assisted modeling allows pharmaceutical drug designers to model the chemical behavior of new compounds that may be effective against particular diseases, aeronautical engineers to predict wing deformation during supersonic flight, pilots to learn to fly, nuclear-reactor experts to predict the effects of various plant malfunctions and to develop the appropriate remedies, and automobile designers to test the integrity of the passenger compartment during crashes. In these and many other instances, it is far easier, cheaper, and safer to experiment with a model than with a real entity. In fact, in many situations, such as training of space-shuttle pilots and studies of nuclear-reactor safety, modeling and simulation are the only feasible method for learning about the system. For these reasons, computer modeling is replacing more traditional techniques, such as wind-tunnel tests. Engineers and scientists now can perform many of their experiments with digital wind tunnels, microscopes, telescopes, and so on. Such numerically based simulation and animation of models is rapidly becoming a new paradigm in science, taking its place beside the traditional branches of theory and physical experimentation. Needless to say, the modeling and simulation are only as good as the model and its inputs—the caution "garbage in, garbage out" pertains especially to modeling.

Models need not necessarily contain intrinsically geometric data; abstractions such as organizational models are not spatially oriented. Nonetheless, most such models can be

represented geometrically; for example, an organizational model may be represented by an organization chart, or the results of a clinical drug evaluation may be represented by a histogram. Even when a model represents an intrinsically geometric object, no fixed graphical representation in the model or view of that model is dictated. For example, we can choose whether to represent a robot as a collection of polyhedra or of curved surfaces, and we can specify how the robot is to be "photographed"—from which viewpoint, with which type of geometric projection, and with what degree of realism. Also, we may choose to show either the structure or the behavior of a model pictorially; for instance, we may want to see both a VLSI circuit's physical layout on the chip and its electrical and logical behaviors as functions of inputs and time.

7.1.2 Geometric Models

Geometric or graphical models describe components with inherent geometrical properties and thus lend themselves naturally to graphical representation. Among the ingredients a geometric model may represent are the following:

- Spatial layout and shape of components (i.e., the *geometry* of the entity), and other attributes affecting the appearance of components, such as color

- Connectivity of components (i.e., the structure or *topology* of the entity); note that connectivity information may be specified abstractly (say, in an adjacency matrix for networks or in a tree structure for a hierarchy), or may have its own intrinsic geometry (the dimensions of channels in an integrated circuit)

- Application-specific data values and properties associated with components, such as electrical characteristics or descriptive text.

Associated with the model may be processing algorithms, such as linear-circuit analysis for discrete circuit models, finite-element analysis for mechanical structures, and energy minimization for molecular models.

There is a tradeoff between what is stored explicitly in the model and what must be computed prior to analysis or display—a classical space–time tradeoff. For example, a model of a computer network could store the connecting lines explicitly or could recompute them from a connectivity matrix with a simple graph-layout algorithm each time a new view is requested. Enough information must be kept with the model to allow analysis and display, but the exact format and the choices of encoding techniques depend on the application and on space-time tradeoffs.

7.1.3 Hierarchy in Geometric Models

Geometric models often have a hierarchical structure induced by a bottom-up construction process: Components are used as building blocks to create higher-level entities, which in turn serve as building blocks for yet higher-level entities, and so on. Like large programming systems, hierarchies are seldom constructed strictly bottom-up or top-down; what matters is the final hierarchy, not the exact construction process. Object hierarchies are common because few entities are monolithic (indivisible); once we decompose an entity into a collection of parts, we have created at least a two-level hierarchy. In the uncommon
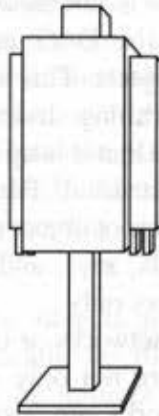
Fig. 7.1 Perspective view of simplified android robot.

case that each object is included only once in a higher-level object, the hierarchy can be symbolized as a tree, with objects as nodes and inclusion relations between objects as edges. In the more common case of objects included multiple times, the hierarchy is symbolized by a *directed acyclic graph* (DAG). As a simple example of object hierarchy, Fig. 7.1 shows a perspective view of a rudimentary "android" robot; Fig. 7.2(a) shows the robot's structure as a DAG. Note that we can duplicate the multiply included objects to convert the DAG to a tree (Fig. 7.2b). By convention, the arrows are left off as redundant because the ordering relationship between nodes is indicated by the nodes' relative positions in the tree—if node A is above node B, then A includes B.

The robot is composed of an upper body swiveling on a base. The upper body is composed of a head that rotates relative to a trunk; the trunk also has attached to it two identical arms that may rotate independently through a horizontal axis "at the shoulder." The arm is composed of a fixed part, "the hand," and a thumb that slides parallel to the hand to form a primitive gripper. Thus, the thumb object is invoked once in the arm, and the arm object is invoked twice in the upper body. We discuss the creation of this robot throughout this chapter; its shape is also presented in three orthographic projections in windows 2–4 of the screen shown in Fig. 7.7(b).
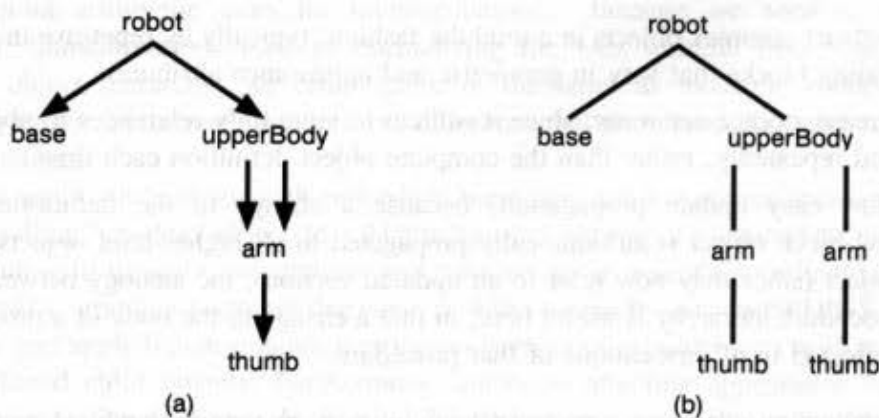
Fig. 7.2 Hierarchy of robot components. (a) Directed acyclic graph (DAG). (b) Tree.

Although an object in a hierarchy is composed of geometric primitives as well as inclusions of lower-level subobjects, the DAG and the tree representing the robot's hierarchy show only references to subobjects. This is analogous to the procedure-hierarchy diagram commonly used to show the calling structure of a program in a high-level procedural language. It is important to note that it is up to the designer to decide exactly how a composite object is hierarchically constructed. For example, the robot could have been modeled as a two-level hierarchy, with a root object consisting of a base, a head and a trunk as geometric primitives (parallelepipeds, say), and two references to an ''atomic'' arm object composed of geometric primitives only.

Many systems, such as computer networks or chemical plants, can be represented by network diagrams, in which objects are not only included multiple times, but are also interconnected arbitrarily. Such networks are also modeled as graphs, possibly even containing cycles, but they can still exhibit properties of object-inclusion hierarchy when subnetworks occur multiple times.

To simplify the task of building complex objects (and their models), we commonly use application-specific atomic components as the basic building blocks. In 2D, these components are often drawn by using plastic or computer-drawn templates of standard symbolic shapes (also called symbols or stencils). In drawing programs, these shapes, in turn, are composed of geometric primitives, such as lines, rectangles, polygons, ellipse arcs, and so on. In 3D, shapes such as cylinders, parallelepipeds, spheres, pyramids, and surfaces of revolution are used as basic building blocks. These 3D shapes may be defined in terms of lower-level geometric primitives, such as 3D polygons; in this case, smoothly curved surfaces must be approximated by polygonal ones, with attendant loss of resolution. Alternatively, in advanced modeling systems that deal directly with free-form surfaces or volumes, shapes such as parametric polynomial surfaces, and solids such as cylinders, spheres, and cones, are themselves geometric primitives and are defined analytically, without loss of resolution—see Chapters 11 and 12. We use the term *object* in this chapter for those 2D or 3D components that are defined in their own modeling coordinate systems in terms of geometric primitives and lower-level objects, and that usually have not only geometrical data but also associated application data. An object is thus a (composite) shape and all its data.

A hierarchy, then, is created for a variety of purposes:

- To construct complex objects in a modular fashion, typically by repetitive invocations of building blocks that vary in geometric and appearance attributes

- To increase storage economy, since it suffices to store only references to objects that are used repeatedly, rather than the complete object definition each time

- To allow easy update propagation, because a change in the definition of one building-block object is automatically propagated to all higher-level objects that use that object (since they now refer to an updated version); the analogy between object and procedure hierarchy is useful here, in that a change to the body of a procedure is also reflected in all invocations of that procedure.

The application can use a variety of techniques to encode hierarchical models. For example, a network or relational database can be used to store information on objects and

on relationships between objects. Alternatively, a more efficient, customized linked-list structure can be maintained by the application program, with records for objects and pointers for relationships. In some models, connections between objects are objects in their own right; they must also be represented with data records in the model. Yet another alternative is to use an object-oriented database [ZDON90]. Object-oriented programming environments such as SmallTalk [GOLD83], MacApp [SCHM86] and ET++ [WEIN88] are increasingly being used to store modeling information for the geometric objects in graphics application programs.

Interconnections. In most networks, objects are placed in specified locations (either interactively by the user or automatically by the application program) and then are interconnected. Interconnections may be abstract and thus of arbitrary shape (e.g., in hierarchy or network diagrams, such as organization charts or project-scheduling charts), or they may have significant geometry of their own (e.g., a VLSI chip). If connections are abstract, we can use various standard conventions for laying out hierarchical or network diagrams, and we can employ attributes such as line style, line width, or color to denote various types of relationships (e.g., "dotted-line responsibility" in an organization chart). Connections whose shape matters, such as the channels connecting transistors and other components of a VLSI circuit, are essentially objects in their own right. Both abstract and nonabstract connections are often *constrained* to have horizontal or vertical orientations (sometimes called the *Manhattan* layout scheme) to simplify visualization and physical construction.

Parameter passing in object hierarchy. Objects invoked as building blocks must be positioned in exactly the right place in their parent objects, and, in order to fit, often must be resized and reoriented as well. Homogeneous coordinate matrices were used to transform primitives in Chapter 5 and to normalize the view volume in Chapter 6, and it should come as no surprise that, in a hierarchical model, one frequently applies scaling, rotation, and translation matrices to subobjects. Sutherland first used this capability for graphical modeling in Sketchpad [SUTH63], coining the terms *master* for the definition of an object and *instance* for a geometrically transformed invocation. As discussed in Section 4.3.3, graphics systems using hierarchical display lists (also called structured display files) implement master–instance hierarchy in hardware, using subroutine jumps and high-speed floating-point arithmetic units for transformations. Because we want to distinguish geometric transformations used in normalizing the view volume from those used in building object hierarchy, we often speak of the latter as *modeling transformations*. Mathematically, of course, there is no difference between modeling and normalizing transformations.

Once again, in analogy with procedure hierarchy, we sometimes speak of a parent object "calling" a child object in a hierarchy, and passing it "geometric parameters" corresponding to its scale, orientation, and position in the parent's coordinate system. As we see shortly, graphics packages that support object hierarchy, such as SPHIGS, can store, compose, and apply transformation matrices to vertices of primitives, as well as to vertices of instantiated child objects. Furthermore, attributes affecting appearance can also be passed to instantiated objects. In Section 7.5.3, however, we shall see that the SPHIGS parameter-passing mechanism is not as general as is that of a procedural language.

7.1.4 Relationship between Model, Application Program, and Graphics System

So far, we have looked at models in general, and geometric models with hierarchy and modeling transformations in particular. Before looking at SPHIGS, let us briefly review the conceptual model of graphics first shown in Fig. 1.5 and elaborated in Fig. 3.2, to show the interrelationship between the model, the application program, and the graphics system. In the diagram in Fig. 7.3, application programs are divided into five subsystems, labeled (a) through (e):

a. Build, modify, and maintain the model by adding, deleting, and replacing information in it

b. Traverse (scan) the model to extract information for display

c. Traverse the model to extract information used in the analysis of the model's behavior/performance

d. Display both information (e.g., rendering of a geometric model, output of an analysis) and user-interface "tools" (e.g., menus, dialog boxes)

e. Perform miscellaneous application tasks not directly involving the model or display (e.g., housekeeping).

The term *subsystem* does not imply major modules of code—a few calls or a short procedure may be sufficient to implement a given subsystem. Furthermore, a subsystem may be distributed throughout the application program, rather than being gathered in a separate program module. Thus, Fig. 7.3 simply shows logical components, not necessarily program structure components; moreover, while it does differentiate the procedures that
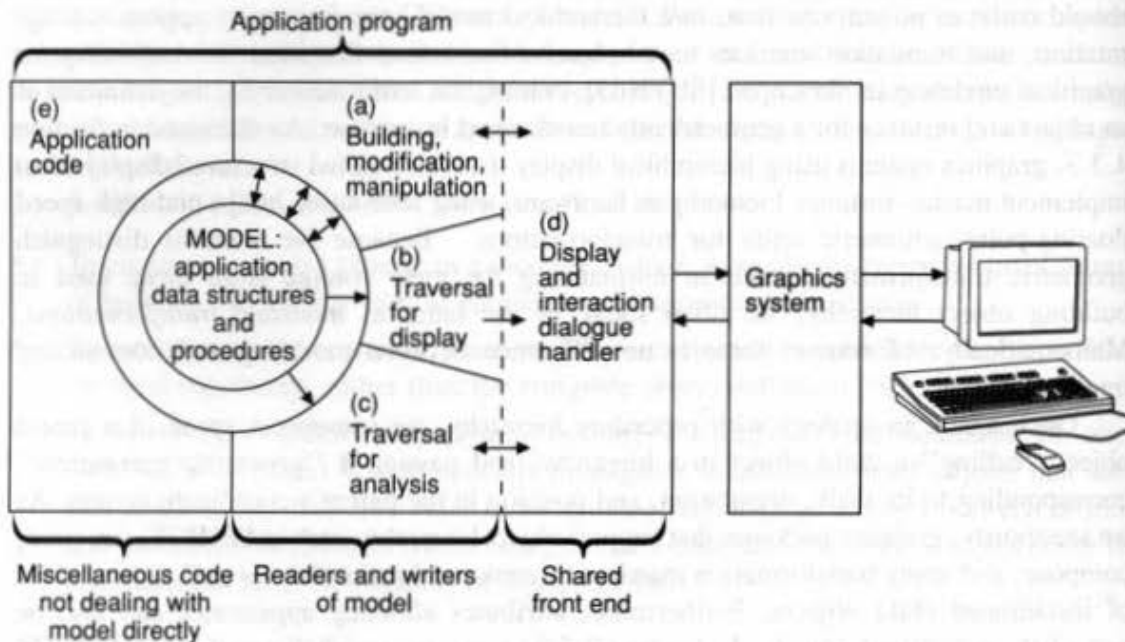

Fig. 7.3 The application model and its readers and writers.

build, modify, analyze, or display the model, it is not always clear whether to call a particular module part of the model or part of the model-maintenance code. It could be argued, for example, that a circuit-analysis module is really part of the model's definition because it describes how the circuit behaves. For a programmer using a traditional procedural language such as Pascal or C, Fig. 7.3 works best if one thinks of the model as primarily containing data. People familiar with object-oriented programming languages will find the mixture of data and procedures a natural one.

In many application programs, especially industrial ones, an "80/20" rule holds: the major portion of the program (80 percent) deals with modeling of entities, and only a minor portion (20 percent) deals with making pictures of them. In other words, in many applications such as CAD, pictorial representation of models is a means to an end, the end being analysis, physical construction, numerically controlled machining, or some other type of postprocessing. Naturally, there are also many applications for which "the picture is the thing"—for example, painting, drafting, film and video production, and animation of scenes for flight simulators. Of these, all but painting also require a model from which the images are rendered. In short, most graphics involves significant modeling (and simulation) and there is considerable support for the saying, "graphics *is* modeling"; Chapters 11, 12, and 20 are devoted to that important subject.

7.2 CHARACTERISTICS OF RETAINED-MODE GRAPHICS PACKAGES

In discussing the roles of the application program, the application model, and the graphics package, we glossed over exactly what capabilities the graphics package has and what happens when the model is modified. As noted in Chapter 2, SRGP operates in *immediate mode* and keeps no record of the primitives and attributes passed to it. Thus, deletions of and changes to application objects necessitate the removal of information on the screen and therefore either selective modification or complete regeneration of the screen; either of these requires the application to respecify primitives from its model. PHIGS, on the other hand, operates in *retained mode*: It keeps a record of all primitives and other related information to allow subsequent editing and automatic updating of the display, thereby offloading the application program.

7.2.1 Central Structure Storage and its Advantages

PHIGS stores information in a special-purpose database called the *central structure storage* (CSS). A *structure* in PHIGS is a sequence of *elements*—primitives, appearance attributes, transformation matrices, and invocations of subordinate structures—whose purpose is to define a coherent geometric object. Thus, PHIGS effectively stores a special-purpose modeling hierarchy, complete with modeling transformations and other attributes passed as "parameters" to subordinate structures. Notice the similarities between the CSS modeling hierarchy and a hardware hierarchical display list that stores a master–instance hierarchy. In effect, PHIGS may be viewed as the specification of a device-independent hierarchical display-list package; a given implementation is, of course, optimized for a particular display device, but the application programmer need not be concerned with that. Whereas many

PHIGS implementations are purely software, there is increasing use of hardware to implement part or all of the package.

As does a display list, the CSS duplicates geometric information stored in the application's more general-purpose model/database to facilitate rapid *display traversal*— that is, the traversal used to compute a new view of the model. The primary advantage of the CSS, therefore, is rapid automatic screen regeneration whenever the application updates the CSS. This feature alone may be worth the duplication of geometric data in the application database and the CSS, especially when the PHIGS implementation uses a separate processor as a "traversal engine" to offload display traversal from the CPU running the application (see Chapter 18). Small edits, such as changing a transformation matrix, are also done efficiently in PHIGS.

A second advantage of the CSS is automatic pick correlation: The package determines the identity and place within the hierarchy of the primitive picked by the user (see Sections 7.10.2 and 7.12.2). The pick-correlation facility exemplifies a common technique of moving frequently needed functionality into the underlying graphics package.

A third advantage of the CSS is that its editing facilities, in combination with the features of hierarchical modeling, make it easy to create various dynamic effects—for example, motion dynamics—in which time-varying transformations are used to scale, rotate, and position subobjects within parent objects. For example, we can model our simple robot so that each joint is represented by a rotation applied to a substructure (e.g., the arm is a rotated subordinate of the upper body), and dynamically rotate the arm by editing a single rotation matrix.

7.2.2   Limitations of Retained-Mode Packages

Although the CSS (as a special-purpose entity built primarily for display and fast incremental updating) facilitates certain common modeling operations, it is neither necessary nor sufficient for all modeling purposes. It is not necessary because an application can do its own screen regeneration when the model is changed, can do its own pick correlation (albeit with considerable work), and can implement its own object hierarchy via procedures defining objects and accepting transformations and other parameters. The CSS is generally not sufficient because, in most applications, a separately built and updated application data structure is still necessary to record all appropriate data for each application object. Thus, there is duplication of all geometric data, and the two representations must be synchronized properly. For all these reasons, some graphics packages support floating-point coordinates and generalized 2D and 3D viewing facilities without any type of structure storage. The rationale for such immediate-mode packages is that maintaining the CSS is often not worth the overhead, since the application typically maintains an application model sufficient for regenerating the screen image.

For applications in which there is significant structural change between successive images, using a retained-mode package does not pay. For example, in a "digital–wind-tunnel" analysis of an airplane wing, where the surface is represented by a mesh of triangles, most of the vertices shift slightly in position as the wing is subjected to aerodynamic forces. Editing a structure database for such a case makes no sense, since most of the data are replaced for each new image. Indeed, editing the PHIGS structure database is not advised unless the number of elements to be edited is small relative to the size of the

networks being displayed. The editing tools provided by PHIGS are rudimentary; for example, it is easy to change a modeling transformation, but to change a vertex of a polygon requires deleting the polygon and respecifying the changed version. Typically, implementations are likely to be optimized for display traversal, since that is the most common operation, rather than for massive editing. Furthermore, the application model must be updated in any case, and it is easier and faster to update just one database than to update two of them.

Because of these limitations, some implementations of PHIGS offer an immediate-mode output facility, although for technical reasons that facility is not part of the official PHIGS specification.

7.3   DEFINING AND DISPLAYING STRUCTURES

The previous section has discussed general properties of PHIGS and SPHIGS. With this section, we begin describing the SPHIGS package in detail; unless otherwise noted, the discussion is generally also applicable to PHIGS. The manipulations permitted on SPHIGS structures include the following:

- Opening (to initiate editing) and closing (to conclude editing)

- Deleting

- Inserting *structure elements* (the three primary types of structure elements are primitives, attributes, including those that specify modeling transformations, and elements that invoke substructures). An element is a data record that is created and inserted into the currently open structure whenever an *element-generator* procedure is called and that stores that procedure's parameters.

- Deleting structure elements

- *Posting* for display (by analogy to posting a snapshot on a bulletin board), subject to a *viewing operation* that specifies how to map the floating-point coordinates to the screen's coordinate system.

7.3.1   Opening and Closing Structures

To create a structure—for example, the collection of primitives and attributes forming the arm component of the robot in Fig. 7.2—we bracket calls to the element-generator procedures with calls to

     void SPH_openStructure (**int** *structureID*);
     void SPH_closeStructure (**void**);

These procedures do for structures essentially what the standard open- and close-file commands do for disk files. Unlike disk files, however, only one structure may be open at any time, and all elements specified while it is open are stored in it. Once closed, structures may be reopened for editing (see Section 7.9).

We note two additional properties of structures. First, primitives and attributes can be specified only as elements of a structure (much as all statements in a C program must be specified in some procedure or function). There are no rules about how many elements may be stored in a structure; a structure can be empty, or can contain an arbitrarily large

number of elements, limited only by memory space. Of course, the elements forming a structure should be, in general, a logically cohesive set defining a single object.

Second, structure IDs are integers. Since they are normally used only by the application program, not by the interactive user, they do not need to have the more general form of character strings, although the application programmer is free to define symbolic constants for structure IDs. Integer IDs also allow a convenient mapping between objects in the application data structure and the objects' corresponding structure IDs.

7.3.2 Specifying Output Primitives and Their Attributes

The procedures that generate output-primitive elements look like their SRGP counterparts, but there are important differences. First, points are specified with three double-precision coordinates (x, y, and z). Moreover, these procedures place elements in the currently open structure in the CSS rather than directly altering the screen image—displaying structures is a separate operation described in Section 7.3.3. In this chapter, the term *primitive* is used as shorthand for three related entities: an element-generation procedure, such as SPH_poly-Line; the structure element generated by that procedure (for example, the polyLine element); and the displayed image created when a primitive element is executed during display traversal of central structure storage. SPHIGS *executes* a primitive element by transforming the primitive's coordinates by modeling transformations and a viewing operation, including clipping it to the view volume, and then *rasterizing* it (i.e., converting it to pixels). Attributes are more specialized than in SRGP, in that each type of primitive has its own attributes. Thus, attributes such as color and line style are in effect "typed," so that the programmer can, for example, reset the current color for lines while preserving the current colors for polyhedra and text.

Primitives. SPHIGS has fewer output primitives than SRGP does, because the 3D "solid" equivalents of some of SRGP's primitives (e.g., an ellipsoid) are computationally expensive to implement, especially with respect to transformations, clipping, and scan conversion.

Most of the SPHIGS primitives are identical to their SRGP counterparts in their specification methods (except that the points are 3D):

```
void SPH_polyLine (int vertexCount, point *vertices);
void SPH_polyMarker (int vertexCount, point *vertices);
void SPH_fillArea (int vertexCount, point *vertices);    /* Like SRGP_polygon */
void SPH_text (point origin, char *str);                 /* Not fully 3D; see Section 7.7.2 */
```

Note that SPHIGS does not verify that fill areas (or facets, described next) are planar, and the results are undefined if they are not.

Now, consider the definition of a simple house, shown in Fig. 7.4. We can describe this house to SPHIGS by specifying each face (also called a *facet*) as a fill area, at the cost of unnecessary duplication in the specification and storage (in CSS) of each shared vertex. This duplication also slows down display generation, since the viewing-operation calculation would be performed on a single vertex more than once. It is far more efficient in storage
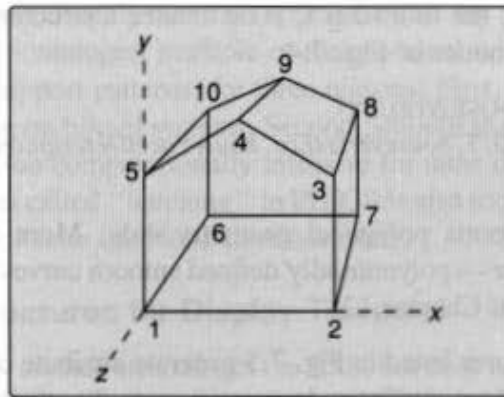
Fig. 7.4 A simple house defined as a set of vertices and facets.

and processing time to specify the facets using indirect references to the shared vertices. We thus think of a polyhedron as a collection of facets, each facet being a list of vertex indices and each index being a "pointer" into a list of vertices. We can describe a polyhedron's specification using the following notation:

Polyhedron = {VertexList, FacetList}
VertexList = {V1, V2, V3, V4, V5, V6, V7, V8, V9, V10}
$V1 = (x_1, y_1, z_1)$, $V2 = (x_2, y_2, z_2) \ldots$
FacetList = {front = {1, 2, 3, 4, 5}, right = {2, 7, 8, 3}, ... bottom = {. .}}

SPHIGS offers this efficient form of specification with its *polyhedron* primitive. In SPHIGS terminology, a polyhedron is a collection of facets that may or may not enclose a volume. In a closed polyhedron such as our house, vertices are typically shared by at least three facets, so the efficiency of the indirect method of specification is especially high. The appearance of a polyhedron is affected by the same attributes that apply to fill areas.

The list of facets is presented to the polyhedron element generator in the form of a single array of integers (SPHIGS type "vertexIndex*") storing a concatenated set of facet descriptions. Each facet description is a sequence of $(V+1)$ integers, where V is the number of vertices in the facet. The first V integers are indices into the vertex list; the last integer is (-1) and acts as a sentinel ending the facet specification. Thus, we would specify the facets of the house (via the fourth parameter of the procedure, described next) by sending the array: 1, 2, 3, 4, 5, −1, 2, 7, 8, 3, −1, ...

> **void** SPH_polyhedron (**int** *vertexCount*, **int** *facetCount*, point *vertices*, vertexIndex *facets*);

Note that the SPHIGS rendering algorithms require that a facet's two sides be distinguishable (external versus internal). Thus, the vertices must be specified in counterclockwise (right-hand rule) order, as one examines the external side of the facet.[2]

---

[2] SPHIGS requires that one side of each facet be deemed "external," even if the polyhedron's facets do not form a closed object. Furthermore, the "internal" side of a polyhedron facet is never visible.

As a simple example, the following C code creates a structure consisting of a single polyhedron modeling the house of Fig. 7.4:

```
SPH_openStructure (HOUSE_STRUCT);
    SPH_polyhedron (10, 7, houseVertexList, houseFacetDescription);
SPH_closeStructure();
```

In essence, SPHIGS supports polygonal geometry only. More advanced 3D modeling primitives are covered later —polynomially defined smooth curves and surfaces in Chapter 11, and solid primitives in Chapter 12.

Attributes.   The procedures listed in Fig. 7.5 generate attribute elements. During display traversal, the execution of an attribute element changes the attribute's value in a modal fashion: The new value stays in effect until it is changed explicitly. Attributes are bound to primitives during display traversal, as discussed in the next section and in Section 7.7.

The attributes of fill areas are different from those in SRGP. Both fill-area and polyhedron primitives have interiors and edges whose attributes are specified separately. The interior has only the color attribute, whereas the edge has style, width, and color attributes. Moreover, the visibility of edges can be turned off via the edge-flag attribute, which is useful for various rendering modes, as discussed in Section 7.8.

Line/edge width and marker size are specified in a *nongeometric* manner: They are not defined using world-coordinate–system units and therefore are not subject to geometric transformations. Thus, modeling transformations and the viewing operation may change a line's apparent length, but not its width. Similarly, the length of dashes in a noncontinuous line style is independent of the transformations applied to the line. However, unlike in SRGP, pixels are not used as the units of measurement, because their sizes are device-dependent. Rather, a nominal width/size has been preset for each device, so that a

polyLine:
    void SPH_setLineStyle (CONTINUOUS / DASHED / DOTTED / DOT_DASHED lineStyle);
    void SPH_setLineWidthScaleFactor (**double** scaleFactor);
    void SPH_setLineColor (**int** colorIndex);

fill area and polyhedron:
    void SPH_setInteriorColor (**int** colorIndex);
    void SPH_setEdgeFlag (EDGE_VISIBLE / EDGE_INVISIBLE flag);
    void SPH_setEdgeStyle (CONTINUOUS / DASHED / DOTTED / DOT_DASHED lineStyle);
    void SPH_setEdgeWidthScaleFactor (**double** scaleFactor);
    void SPH_setEdgeColor (**int** colorIndex);

polyMarker:
    void SPH_setMarkerStyle (MARKER_CIRCLE / MARKER_SQUARE/... markerStyle);
    void SPH_setMarkerSizeScaleFactor (**double** scaleFactor);
    void SPH_setMarkerColor (**int** colorIndex);

text:
    void SPH_setTextFont (**int** fontIndex);
    void SPH_setTextColor (**int** colorIndex);

Fig. 7.5 Procedures generating attribute elements.

unit of width/size will have roughly the same appearance on any output device; the SPHIGS application specifies a (noninteger) multiple of that nominal width/size.

SPHIGS does not support patterns, for three reasons. First, SPHIGS reserves patterns to simulate color shading on bilevel systems. Second, smooth shading of patterned areas on a color system is much too computationally intensive for most display systems. Third, the type of geometric pattern called "hatching" in PHIGS is also too time-consuming, even for display systems with real-time transformation hardware.

7.3.3   Posting Structures for Display Traversal

SPHIGS records a newly created structure in the CSS, but does not display it until the application *posts* the structure subject to a particular viewing specification.[3] SPHIGS then performs a *display traversal* of the structure's elements in the CSS, executing each element in order from the first to the last. Executing a primitive element contributes to the screen image (if a portion of the primitive is in view). Executing an attribute element (both geometric transformations and appearance attributes) changes the collection of attributes stored in a state vector (the *attribute-traversal state*) that is applied to subsequent primitives as they are encountered, in modal fashion. Thus, attributes are applied to primitives in display-traversal order.

The following procedure adds a structure to the list of posted structures maintained internally by SPHIGS:

    void SPH_postRoot (**int** *structureID*, **int** *viewIndex*);

The term *root* indicates that, in posting a structure S that invokes substructures, we are actually posting the hierarchical DAG, called the *structure network*, whose root is S. Even if a posted structure does not invoke substructures, it is called a root; all posted structures are roots.

The *viewIndex* parameter chooses an entry in the table of *views* (discussed in the next section); this entry specifies how the coordinates of the structure's primitives are to be mapped to the screen's integer coordinate space.

We can erase an object's image from the screen by deleting structures (or elements) from the CSS (see Section 7.9) or by using the less drastic SPH_unpostRoot procedure that removes the root from the list of posted roots, without deleting it from the CSS:

    void SPH_unpostRoot (**int** *structureID*, **int** *viewIndex*);

7.3.4   Viewing

The synthetic camera.   It is helpful to think of a 3D graphics package as a synthetic camera that takes "snapshots" of a 3D world inhabited by geometrically defined objects. Creating a structure is equivalent to positioning an object in a photography studio; posting a structure is analogous to activating an instant camera previously set up to point at the scene, and then having the snapshot of the scene posted on a bulletin board. As we see shortly,

---

[3]This way of specifying structure display is the most significant difference between PHIGS and SPHIGS. In PHIGS's more general mechanism, the view specification is a structure element; this allows the view to be changed during display traversal and to be edited like any other element. Many current PHIGS implementations also support the simpler SPHIGS-style posting mechanism.

each time anything changes in the scene, our synthetic camera automatically produces a new, updated image that is posted in place of the old one. To create animation, we show multiple static images in rapid succession, as a movie camera does.

Continuing the metaphor, let us consider how the synthetic picture is produced. First, the camera operator must position and orient the camera; then, he must decide how much of the scene should appear: For example, is the image to be a closeup of a portion of an object of interest, or a long-distance view of the entire scene? Subsequently, the photographer must decide how large a print to make for posting on the bulletin board: Is it to be a wallet-sized print or a poster? Finally, the place on the bulletin board where the photograph is to be posted must be determined. In SPHIGS, these criteria are represented in a *view* that includes a specification of a viewing operation; this operation's *viewport* specifies the size of the photograph and its position on the bulletin board. Not all objects in the structure database need be photographed with the same "camera setting." Indeed, multiple views may be specified for the bulletin board, as we shall see shortly.

The viewport. As discussed in the previous chapter, the viewport specifies a parallelepiped in the NPC system to which the contents of the view volume defined in VRC is mapped. Since the NPC system is mapped to the physical device's integer-coordinate system in a fixed manner, the viewport also specifies where the image is to appear on the screen. The 3D NPC system is mapped to the 2D screen coordinate system in this manner: The NPC unit cube having one corner at $(0, 0, 0)$ and the opposing corner at $(1, 1, 1)$ is mapped to the largest square that can be inscribed on the screen, with the z coordinate simply ignored. For example, on a display device having a resolution of 1024 horizontally and 800 vertically, a point $(0.5, 0.75, z)_{NPC}$ is mapped to $(512, 599)_{DC}$. For portability, an application should not use NPC space lying outside the unit cube; often, however, the benefits of taking full advantage of a nonsquare screen shape are worth the portability cost.

The view table. SPHIGS maintains a table of views that has an implementation-dependent number of entries. Each view consists of a specification of the view volume and viewport, called the *view representation*, and a list (initially empty) of the roots posted to it. Entry 0 in the table defines a *default view* having the volume described in Fig. 6.25 (b), with the front and back planes at $z = 0$ and $z = -\infty$, respectively. The viewport for this default view is the NPC unit cube.

The view representations for all entries in the table (except view 0) may be edited by the application via

```
void SPH_setViewRepresentation (
    int viewIndex, matrix_4x4 voMatrix, matrix_4x4 vmMatrix,
    double NPCviewport_minX, double NPCviewport_maxX,
    double NPCviewport_minY, double NPCviewport_maxY,
    double NPCviewport_minZ, double NPCviewport_maxZ);
```

The two 4×4 homogeneous-coordinate matrices are the view-orientation and view-mapping matrices described in Chapter 6. They are produced by the procedures shown in Fig. 7.6.

```
/* To set up UVN viewing reference coordinate system */
matrix_4x4 SPH_evaluateViewOrientationMatrix (
        point viewRefPoint,
        vector_3D viewPlaneNormal, vector_3D viewUpVector,
        matrix_4x4 voMatrix);

/* To set up view volume and to describe how it is to be mapped to NPC space */
matrix_4x4 SPH_evaluateViewMappingMatrix (
        /* First, we specify the view-volume in VRC */
        double umin, double umax, double vmin, double vmax,      /* View-plane boundaries */
        PARALLEL / PERSPECTIVE projectionType,
        point projectionReferencePoint;                          /* In VRC */
        double frontPlaneDistance, double backPlaneDistance,     /* Clipping planes */
        /* Then, we specify the NPC viewport. */
        double NPCvp_minX, double NPCvp_maxX,
        double NPCvp_minY, double NPCvp_maxY,
        double NPCvp_minZ, double NPCvp_maxZ,
        matrix_4x4 vmMatrix);
```

Fig. 7.6 Utilities for calculating viewing-transformation matrices.

Multiple views. The view index specified during posting refers to a specific NPC viewport describing where on the screen (bulletin board) the image of the structure (photograph) is to appear. Just as one can tack several photographic prints on a board, an application can divide the screen into a number of viewports.

The use of multiple views is powerful in many ways. We can display several different structures simultaneously in individual areas of the screen by posting them with different views. In Fig. 7.7(a), we present a schematic representation of the view table, showing only the pointers to the lists of structure networks posted to each view. We can see that there is one view showing a street scene; also, there are three separate views of a robot. The robot
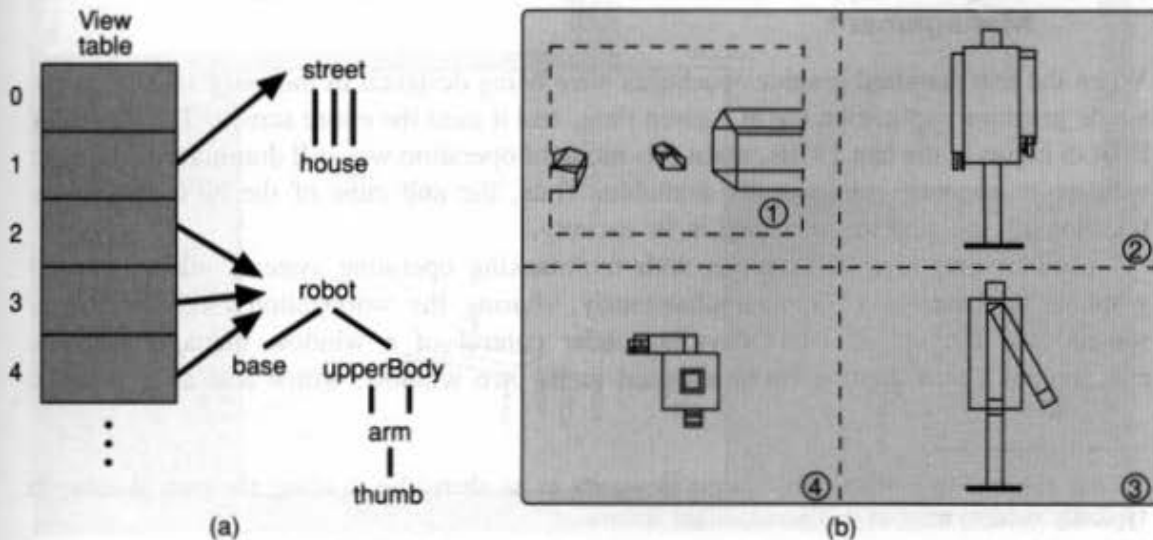

Fig. 7.7 Multiple views sharing screen space. (a) Schematic diagram of view table. Each view entry points to a list of the roots posted to that view. (b) Resulting image. Dashed viewport extents and circled numbers show the viewports and their associated view indices.

structure was posted three times, each time with a different view index. Figure 7.7(b) shows the resulting screen image. The multiple views of the robot vary not only in their viewport specifications, but also in their view-volume specifications.

The preceding scenario implies that each view has at most one posted structure; in fact, however, any number of roots can be posted to a single view. Thus, we can display different root structures in a single unified picture by posting them together to a view. In this case, our metaphorical camera would take a single composite snapshot of a scene that contains many different objects.

Another property of viewports is that, unlike real snapshots and window-manager windows, they are transparent.[4] In practice, many applications *tile* the viewports to avoid overlapping; however, we can also use overlapping to advantage. For example, we can "compose" two distinct images created by different viewing transformations or showing objects defined in different units of measurement; thus, in building a close-up diagram of an engine part, we could inset a small picture showing the entire engine (for context) overlapping the large close-up picture. (To avoid confusion, we would do this by selecting an area of the closeup that is just background.)

To regenerate the screen, SPHIGS displays the posted networks by traversing the roots posted to each view in the view table, in increasing order of view index, starting with view 0; thus, the images of objects posted to view N have *display priority* over the images of objects posted to a view with an index less than N, and therefore potentially overlap them. This ordering is significant, of course, only when viewports actually overlap.[5]

Note that an application can manufacture many independent WC spaces and can use any units of measurement desired. In Fig. 7.7, for example, the street structure is defined in a WC space in which each increment on an axis represents 10 yards, whereas the robot is defined in a wholly independent WC space measured in centimeters. Although each root structure is modeled in its own WC space, there is only one NPC space per display device, shared by all posted structures, since that space is an abstraction of the display device.

7.3.5 Graphics Applications Sharing a Screen via Window Management

When the first standard graphics packages were being designed in the early 1970s, only a single graphics application ran at a given time, and it used the entire screen. The design of PHIGS began in the late 1970s, when this mode of operation was still dominant and before window managers were generally available. Thus, the unit cube of the NPC space was traditionally mapped to the screen in its entirety.

Modern graphics workstations with multitasking operating systems allow multiple graphics applications to run simultaneously, sharing the workstation's resources, the screen, and the set of input devices, under control of a window manager. In this environment, each application is assigned to its own window, which acts as a "virtual

---

[4] Some versions of PHIGS offer opaque viewports as an alternative to tiling; the term *shielding* is typically used to refer to this nonstandard feature.

[5] This trivial view-priority system is less sophisticated than that of PHIGS, which allows explicit view priorities to be assigned by the application.

screen.'' The user can resize and move these windows by calling on the functionality of the window manager. The primary advantage is that each application can act as though it controls an entire screen; it does not need to know that its screen is only a portion of the actual display device's screen. An SPHIGS application therefore need not be modified for a window-manager environment; the package and the window manager cooperate to map NPC space to an assigned window rather than to the entire screen.[6] Figure 7.8 shows two SPHIGS applications and a terminal-emulator program running concurrently on a graphics workstation. Because SPHIGS maps the NPC space to the largest square that fits within the window-manager window, some portion of any nonsquare window is unavailable to the SPHIGS application, as illustrated by the SPHIGS window showing the table and chair scene.

---

[6] It is sometimes desirable to display only a portion of NPC space in a window-manager window. PHIGS supports the specification of an NPC *workstation window*, which is used to clip the image produced by display traversal; the clipped portion is then mapped to a *workstation viewport* of the same aspect ratio, specified in physical device coordinates. This workstation transformation can also be used to map a rectangular portion of NPC space to the physical display, allowing use of a nonsquare screen region.
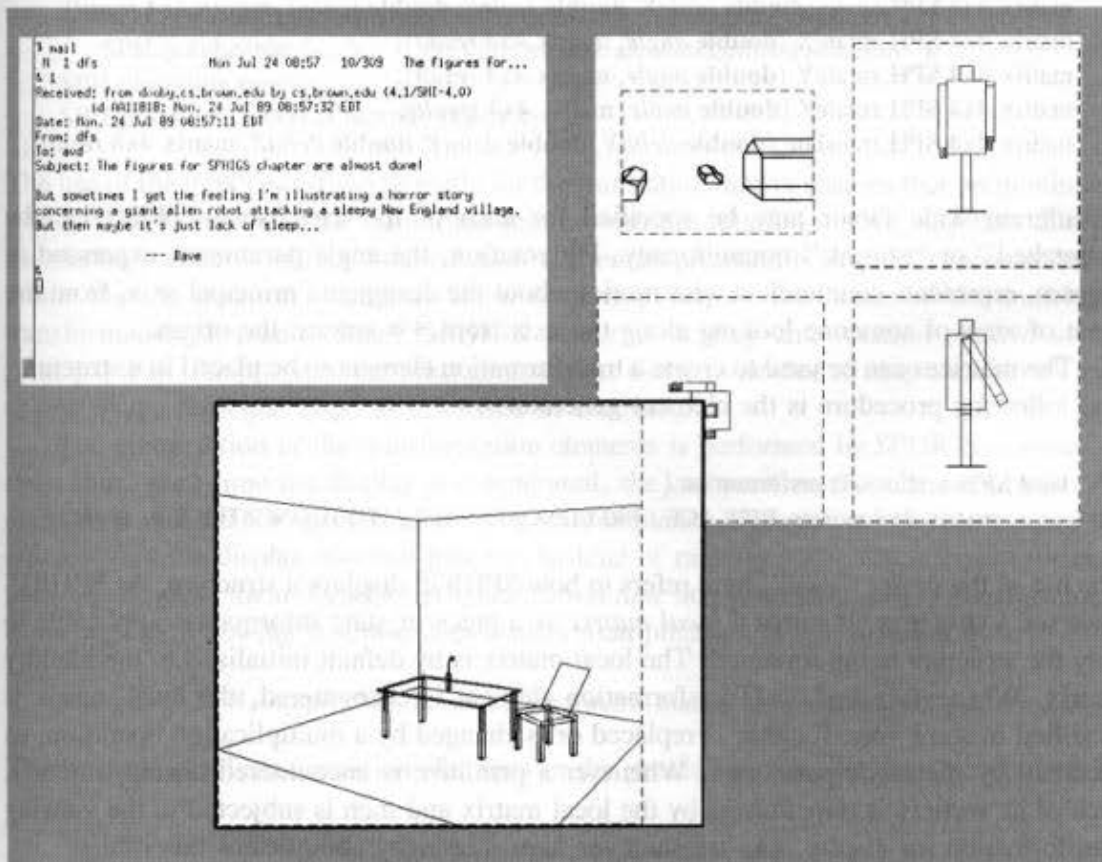

Fig. 7.8 Two SPHIGS applications running in window-manager windows.

7.4 MODELING TRANSFORMATIONS

Section 7.3.2 contained a Pascal code fragment that created a simple structure modeling a house. For simplicity's sake, we placed one of the house's corners at the origin, aligned the house's sides with the principal axes, and gave it dimensions that were whole units. We shall say that an object defined at the origin and (largely) aligned with the principal axes is *standardized*; not only is it easier to define (determine the vertex coordinates of) a standardized object than it is to define one arbitrarily positioned in space, but also it is easier to manipulate the geometry of a standardized object in order to resize, reorient, or reposition it.

Let us say we want our house to appear at a different location, not near the origin. We could certainly recompute the house's vertices ourselves, and create the house structure using the same Pascal code shown in Section 7.3.2 (changing only the vertex coordinates). Instead, however, let us examine the powerful technique of transforming a standardized building-block object in order to change its dimensions or placement.

As we saw in Chapter 5, we can transform a primitive such as a polygon by multiplying each vertex, represented as a column vector $[x, y, z, 1]^T$, by a 4 × 4 homogeneous-coordinate transformation matrix. The following utility functions create such matrices:

```
matrix_4x4 SPH_scale (double scaleX, double scaleY, double scaleZ, matrix_4x4 result);
matrix_4x4 SPH_rotateX (double angle, matrix_4x4 result);
matrix_4x4 SPH_rotateY (double angle, matrix_4x4 result);
matrix_4x4 SPH_rotateZ (double angle, matrix_4x4 result);
matrix_4x4 SPH_translate (double deltaX, double deltaY, double deltaZ, matrix_4x4 result);
```

A different scale factor may be specified for each of the axes, so an object can be "stretched" or "shrunk" nonuniformly. For rotation, the angle parameter, expressed in degrees, represents counterclockwise motion about the designated principal axis, from the point of view of someone looking along the axis from $+\infty$ toward the origin.

The matrices can be used to create a transformation element to be placed in a structure. The following procedure is the element generator:

```
void SPH_setLocalTransformation (
        matrix_4x4 matrix, REPLACE / PRECONCATENATE / POSTCONCATENATE mode);
```

The use of the prefix "local" here refers to how SPHIGS displays a structure. As SPHIGS traverses a structure, it stores a *local matrix* as a piece of state information applicable to only the structure being traversed. The local matrix is by default initialized to the identity matrix. Whenever a setLocalTransformation element is encountered, the local matrix is modified in some way: It either is replaced or is changed by a multiplication operation, as specified by the *mode* parameter. Whenever a primitive is encountered during traversal, each of its vertices is transformed by the local matrix and then is subjected to the viewing transformation for display. (As we shall see later, hierarchy complicates this rule.)

The following code creates a structure containing our house at an arbitrary location,

and posts that structure for display using the default view. The house maintains its original standardized size and orientation.

```
SPH_openStructure (HOUSE_STRUCT);
    SPH_setLocalTransformation (SPH_translate (...), REPLACE);
    SPH_polyhedron (...);      /* Vertices here are standardized as before */
SPH_closeStructure ();
SPH_postRoot (HOUSE_STRUCT, 0);
```

Simple transformations like this one are uncommon. We typically wish not only to translate the object, but also to affect its size and orientation. When multiple transformations of primitives are desired, the application "builds" the local matrix by successively concatenating (i.e., composing) individual transformation matrices in the exact order in which they are to be applied. In general, standardized building-block objects are scaled, then rotated, and finally translated to their desired location; as we saw in Chapter 5, this order avoids unwanted translations or shearing.

The following code creates and posts a house structure that is moved away from the origin and is rotated to a position where we see its side instead of its front:

```
SPH_openStructure (MOVED_HOUSE_STRUCT);
    SPH_setLocalTransformation (SPH_rotateY (...), REPLACE);
    SPH_setLocalTransformation (SPH_translate (...), PRECONCATENATE);
    SPH_polyhedron (...);      /* Vertices here are standardized as before */
SPH_closeStructure ();
SPH_postRoot (MOVED_HOUSE_STRUCT, 0);
```

The use of the PRECONCATENATE mode for the translation matrix ensures that premultiplication is used to compose the translation matrix with the rotation matrix, and thus that the translation's effect "follows" the rotation's. Premultiplication is thus a far more common mode than is postmultiplication, since it corresponds to the order of the individual transformation elements. Since SPHIGS performs scaling and rotation relative to the principal axes, the programmer must generate the matrices needed to map an arbitrary axis to one of the principal axes, as discussed in Chapter 5.

The composition of the transformation elements is performed by SPHIGS at *traversal time*; thus, each time the display is regenerated, the composition must be performed. An alternative method for specifying a contiguous sequence of transformations increases the efficiency of the display-traversal process: Instead of making a structure element for each one, we compose them ourselves at *specification time* and generate a single transformation element. The following function does matrix multiplication at specification time:

```
matrix_4x4 SPH_composeMatrix (matrix_4x4 mat1, matrix_4x4 mat2, matrix_4x4 result);
```

The two setLocalTransformation elements in the preceding code can thus be replaced by

```
SPH_setLocalTransformation (
    SPH_composeMatrix (SPH_translate (...), SPH_rotateY (...), result), REPLACE);
```

The disadvantage of this method is that it is no longer possible to make a dynamic change to the size, orientation, or position of a primitive by selectively "editing" the desired member of the sequence of setLocalTransformation elements; rather, the entire composite must be recomputed and respecified. The rule of thumb for efficiency is thus to use composition at specification time unless the individual transformations are to be updated selectively, in which case they should be specified individually.

Let us create the street structure that contains three copies of our simple house, as first seen in Fig. 7.7. A perspective view of the "house" on the left, the "mansion" on the right, and the "cottage" in the middle is shown in Fig. 7.9(a). We have added dashed lines parallel to the x axis and tick marks for the x axis to indicate the relative positions of the houses, and have used a display mode of SPHIGS that shows the wireframe edges of polyhedra with hidden edges removed (see Section 7.8). The leftmost house in the figure is an untransformed instance of our standardized house, and the other two copies differ in size, orientation, and position.

The brute-force way to create this street structure is to specify the standardized house polyhedron three times, preceded by the desired transformation elements, as shown in the schematic representation of the structure of Fig. 7.9(b). We show a block of consecutive transformation elements as a single unit; the first element uses REPLACE mode and all others use PRECONCATENATION mode, with the multiplication symbol (·) separating them to indicate composition. The code generating the structure is shown in Fig. 7.10.

We can eliminate the redundant specifications of the standardized house polyhedron by defining a Pascal procedure to perform the call generating the house polyhedron, as shown in the pseudocode of Fig. 7.11. Because the house is defined by a single polyhedron call, the efficiency of this technique is not obvious in this example; however, if our house were
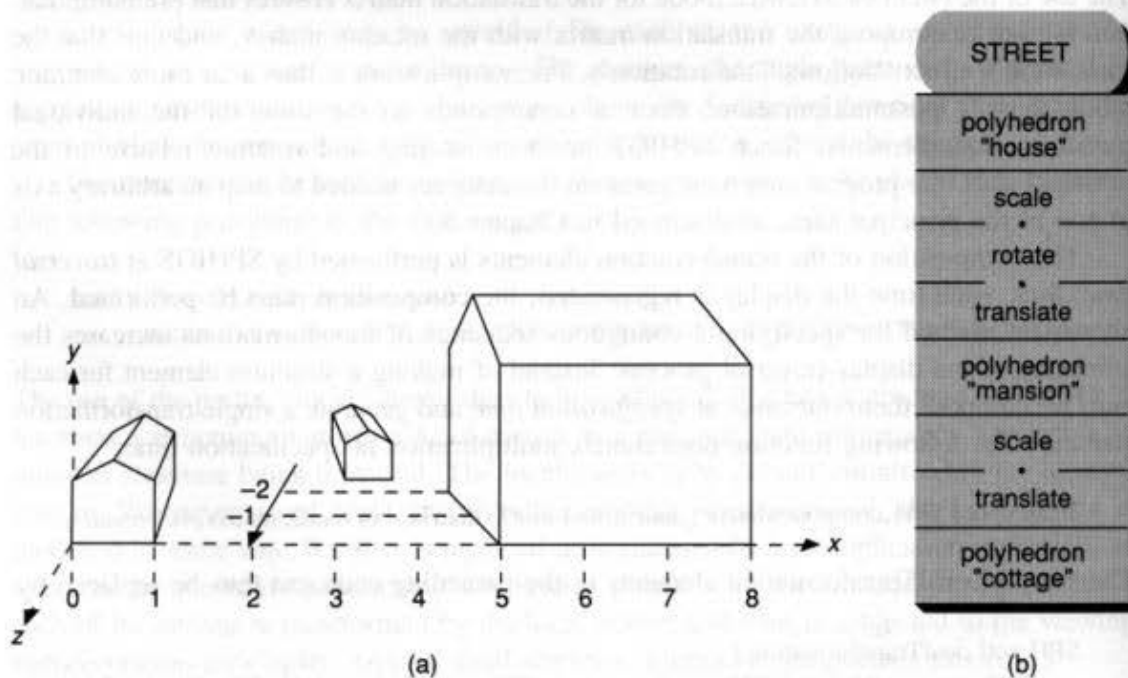

Fig. 7.9 Modeling a street with three houses. (a) Perspective view. (b) Structure.

```
SPH_openStructure (STREET_STRUCT);
    SPH_polyhedron (...);        /* Define first house, in its standardized form */

    /* Mansion is house scaled by 2 in x, 3 in y, 1 in z, rotated 90° about y, */
    /* then translated; note that its left side is subsequently front-facing */
    /* and lies in the (x, y) plane. */
    SPH_setLocalTransformation (SPH_scale (2.0, 3.0, 1.0, result), REPLACE);
    SPH_setLocalTransformation (SPH_rotateY (90.0, result), PRECONCATENATE);
    SPH_setLocalTransformation (SPH_translate (8.0, 0.0, 0.0, result), PRECONCATENATE);
    SPH_polyhedron (...);

    /* Cottage is house uniformly scaled by 0.75, unrotated, set back in z and over in x */
    SPH_setLocalTransformation (SPH_scale (0.75, 0.75, 0.75, result), REPLACE);
    SPH_setLocalTransformation (SPH_translate (3.5, 0.0, −2.5, result), PRECONCATENATE);
    SPH_polyhedron (...);
SPH_closeStructure ();
SPH_postRoot (STREET_STRUCT, 0);
```

Fig. 7.10 Code used to generate Fig. 7.9.

more complex and required a series of attribute and primitive specifications, this method clearly would require less C code. Moreover, the technique provides another benefit of modularization: We can change the house's shape or style by editing the House procedure without having to edit the code that creates the street.

We call a procedure such as House that generates a sequence of elements defining a standardized building block, and that is made to be used repetitively with arbitrary geometrical transformations, a *template procedure*. The template procedure is a convenience for the programmer and also represents good programming practice. Note, however, that although the House procedure adds a level of procedure hierarchy to the C program, no structure hierarchy is created—the model of the street is still "flat." Indeed, the structure network produced by the code of Fig. 7.11 is indistinguishable from that produced by the code of Fig. 7.10. There is no "savings" in terms of the number of elements produced for the structure.

```
void House (void)
{
    SPH_polyhedron (...);
}

/* Mainline */
SPH_openStructure (STREET_STRUCT);
    House ();                    /* First House */
    set local transformation matrix;
    House ();                    /* Mansion */
    set local transformation matrix;
    House ();                    /* Cottage */
SPH_closeStructure ();
SPH_postRoot (STREET_STRUCT, 0);
```

Fig. 7.11 Use of a template procedure to model the street.

One change we could make to our template procedure is to have it accept a transformation matrix as a parameter, which it would then use to specify a setLocalTransformation element.[7] Although in some cases passing transformation parameters would be convenient, this method lacks the generality inherent in our original method of being able to specify an arbitrary number of transformations before calling the template.

7.5 HIERARCHICAL STRUCTURE NETWORKS

7.5.1 Two-Level Hierarchy

So far, we have dealt with three types of structure elements: output primitives, appearance attributes, and transformations. Next, we show how the power of SPHIGS derives in large part from structure hierarchy, implemented via an element that "calls" a substructure when executed during traversal. Structure hierarchy should not be confused with the template-procedure hierarchy of the previous section. Template-procedure hierarchy is resolved at specification time, when the CSS is being edited, and produces in-line elements, not substructure invocations. By contrast, structure hierarchy induced by invocation of substructures is resolved when the CSS is traversed for display—the execution of an invocation element acts as a subroutine call. In Section 7.15, we take another look at the relative advantages and disadvantages of template-procedure hierarchy and structure hierarchy.

The *structure-execution element* that invokes a substructure is created by

void SPH_executeStructure (int *structureID*);

Let us replace the template procedure of the previous section by a procedure that builds a house structure in the CSS (see Fig. 7.12). This procedure is called exactly once by the mainline procedure, and the HOUSE_STRUCT is never posted; rather, its display results from its being invoked as a subobject of the street structure. Note that the only differences in the STREET_STRUCT specification are the addition of the call to the procedure that builds the house structure and the replacement of each template procedure call by the generation of an execute-structure element. Although the displayed image is the same as that of Fig. 7.9(a), the structure network is different, as shown in Fig. 7.13, in which the execute-structure element is depicted as an arrow.

Posting STREET_STRUCT tells SPHIGS to update the screen by traversing the STREET_STRUCT structure network; the traversal is in the form of a depth-first tree walk, just as a procedure/subroutine hierarchy is executed. In the preceding example, the traverser initializes the street structure's local matrix to the identity matrix, and then performs the first invocation of the house substructure, applying the street structure's local matrix to each of the house's vertices as though the house polyhedron were a primitive element in the street structure itself. When it returns from the first invocation, it sets the local matrix to a desired composite transformation, and then performs the second invocation, applying the

---

[7]Newman defines *display procedures* as template procedures that take scaling, rotation, and translation parameters [NEWM71].

```
void BuildStandardizedHouse (void)
{
    SPH_openStructure (HOUSE_STRUCT);
        SPH_polyhedron (...);
    SPH_closeStructure ();
}

/* Mainline */
BuildStandardizedHouse ();
SPH_openStructure (STREET_STRUCT);
    SPH_executeStructure (HOUSE_STRUCT);     /* First house */
    set local transformation matrix;
    SPH_executeStructure (HOUSE_STRUCT);     /* Mansion */
    set local transformation matrix;
    SPH_executeStructure (HOUSE_STRUCT);     /* Cottage */
SPH_closeStructure ();
SPH_postRoot (STREET_STRUCT, 0);
```

Fig. 7.12 Use of a subordinate structure to model the street.

new composite matrix to the vertices of the house to create the second instantiation of the house. When it returns, the local matrix is again changed; the new composite is applied to the house's vertices to produce the third house instance.

We think of a structure as an independent object, with its primitives defined in its own floating-point modeling-coordinate system (MCS); this way of thinking facilitates the building of low-level standardized building blocks. As we noted in Section 5.8, a transformation maps the vertices in one coordinate system into another; here, SPHIGS uses the local matrix of structure S to transform the primitives of substructures into S's own MCS.
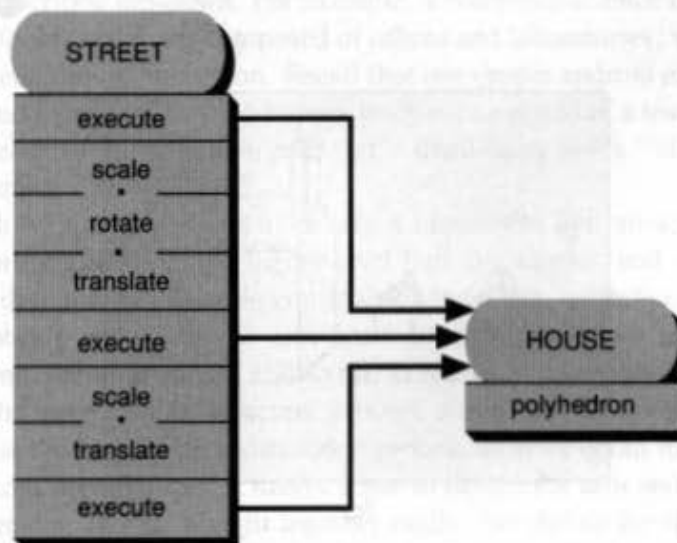

Fig. 7.13 Structure network showing invocation of subordinate structure.

7.5.2   Simple Three-Level Hierarchy

As a simple example of three-level hierarchy, we extend the house in our street example. The new house is composed of the original standardized house (renamed SIMPLE_HOUSE_STRUCT) and a chimney suitably scaled and translated to lie in the right place on top of the house. We could revise the house structure by adding the chimney's facets directly to the original polyhedron, or by adding a second polyhedron to the structure, but we choose here to induce three-level hierarchy by decomposing the house into two subobjects. An advantage of this modularization is that we can define the chimney in a standardized manner (at the origin, of unit size) in its own MCS (as shown in Fig. 7.14a),
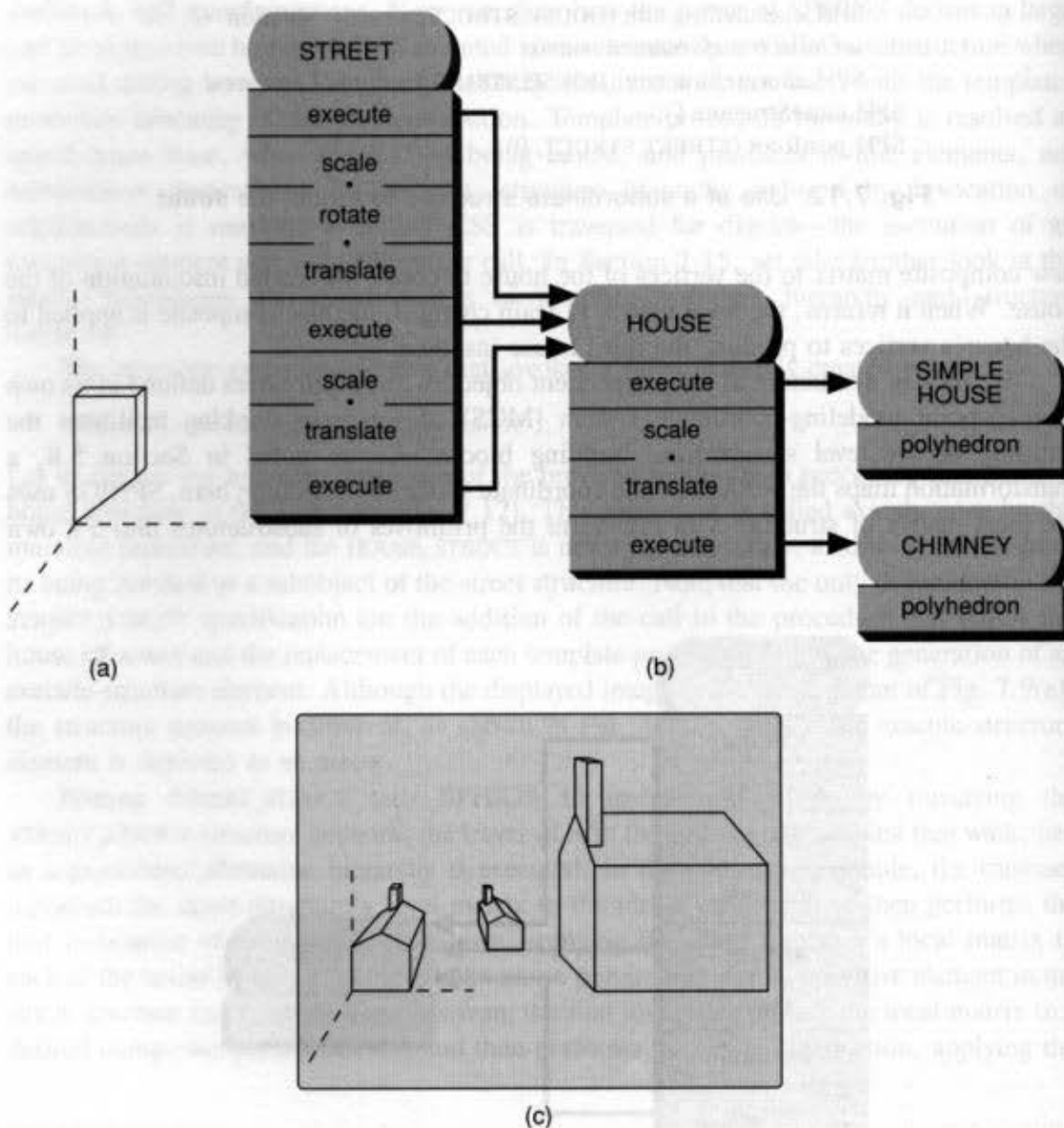

Fig. 7.14 Three-level hierarchy for street. (a) Standardized chimney. (b) Structure network. (c) Resulting image.

and then use scaling and translation to place it on the roof in the house's MCS. If we had to define the chimney to fit exactly on the roof and to map into the house's MCS without scaling, we would have to do messy calculations to compute the vertices explicitly. With modularity, however, we simply define the standardized chimney such that its bottom facet has the same slope as the roof itself; with that condition met, uniform scaling and arbitrary translation can be applied.

The revised house structure is built via

```
SPH_openStructure (HOUSE_STRUCT);
    SPH_executeStructure (SIMPLE_HOUSE_STRUCT);
    set local matrix to scale/translate standardized chimney onto roof of simple house;
    SPH_executeStructure (CHIMNEY_STRUCT);
SPH_closeStructure ();
```

What happens when this two-level house object is instantiated by the street structure with a transformation to yield the three-level hierarchy shown in Fig. 7.14(b)? Since SPHIGS transforms a parent by transforming the latter's component elements and substructures, we are assured that the two component primitives (the simple house and the chimney) are transformed together as a single unified object (Fig. 7.14c). The key point is that the street-structure specification did not need to be changed at all. Thus, the designer of the street structure does not need to be concerned with the internal details of how the house is constructed or subsequently edited—it is a black box.

7.5.3 Bottom-Up Construction of the Robot

Let us now look at a more interesting example, our simple robot, which combines the key ideas of modeling using structure hierarchy and of repeated editing of transformations to achieve motion dynamics. A complex object or system hierarchy is usually conceptualized and informally described top-down. For example, a computer-science department building is composed of floors, which are composed of offices and laboratories, which are composed of furniture and equipment, and so on. Recall that our simple android robot is composed of an upper body and a pedestal base; the upper body is composed of a trunk, a head, and two identical arms, each of which is composed of a fixed hand and a "sliding" (translating) thumb as gripper.

Even though we design top-down, we often implement bottom-up, defining building blocks for use in the definitions of higher-level building blocks, and so on, to create the hierarchy of building blocks. Thus, in constructing the robot, we define the thumb building block for the robot's arm, then the arm itself, and then join two instances of the arm building block onto the upper body, and so on, as shown in the symbolic parts hierarchy of Fig. 7.2 and in the more detailed structure network diagram of the upper body in Fig. 7.15.

Let us look at the bottom-up construction process in more detail to see what geometry and transformations are involved. It makes sense to design the arm and thumb in the same units of measurement, so that they fit together easily. We define the thumb structure in a standardized position in its own MCS in which it "hangs" along the y axis (Fig.7.16a). The arm structure is defined with the same unit of measurement as that used for the thumb;
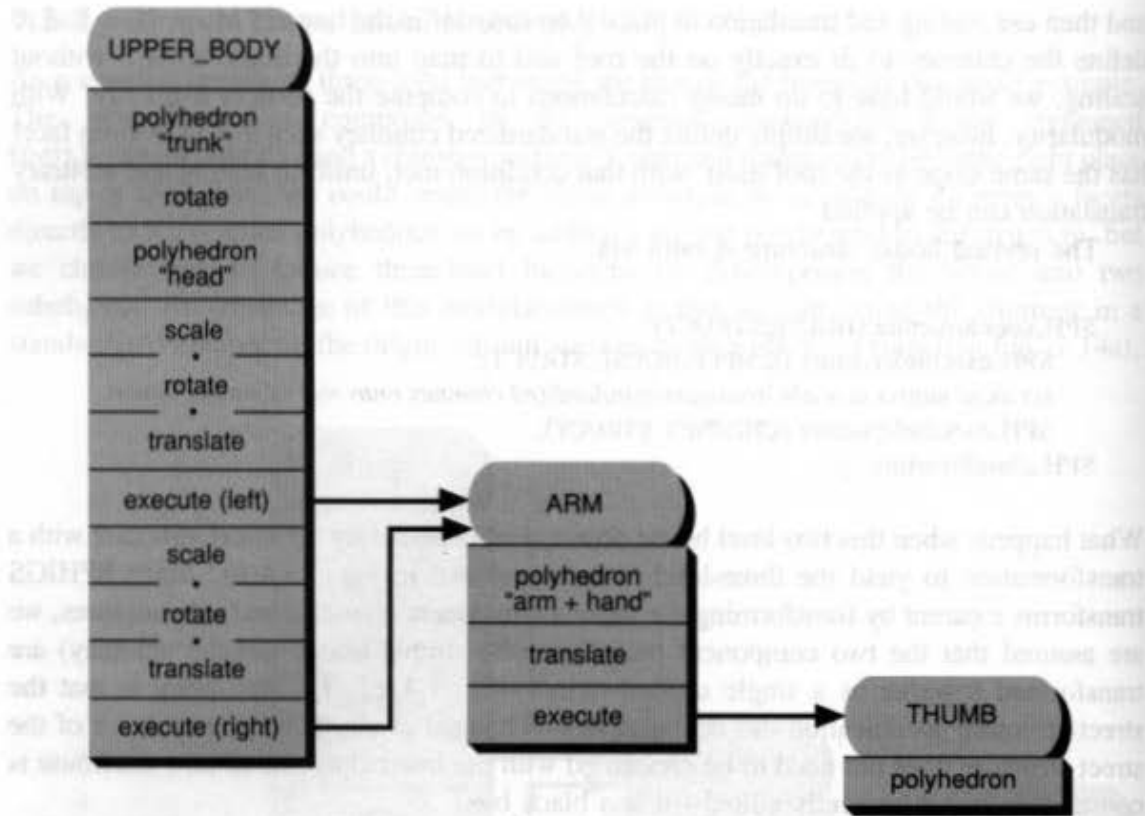
Fig. 7.15 Structure hierarchy for robot's upper body.

it consists of the arm+hand polyhedron (standardized, hanging down along the y axis, as shown in Fig. 7.16b) and a translated invocation of the thumb structure. The translation element preceding the invocation of the thumb is responsible for moving the thumb from its standardized position at the origin to its proper place on the wrist of the arm (c).

The arm-invoking-thumb network is a two-level hierarchy similar to the street–house example. By editing the translation element in the arm structure, we can "slide" the thumb along the wrist of the arm (Fig. 7.16d).[8]

Next, we build the upper body. Since we want to be able to rotate the head, we first specify the trunk polyhedron, then a rotation, and next the head polyhedron (Fig. 7.16e). Our next step is to have the upper-body structure invoke the arm structure twice. What transformations should precede these invocations? If our sole concern is positioning (i.e., translating) each arm correctly in the upper-body MCS, we may produce a picture like Fig. 7.16(f), where the arm and upper body were clearly designed at different scales. This is easy to fix: We can add a scale transformation preceding the translation (Fig. 7.16g). However, a scale and a translation is not enough if we want arms that can swing on the axis

---

[8] The observant reader may have wondered how the thumb actually slides in this model, since it is not really attached in any way to the arm hand. In fact, none of the components of our robot model is attached to another via objects representing joints. Modeling of joints and the constraints that determine their modes of operation are beyond the scope of this book.
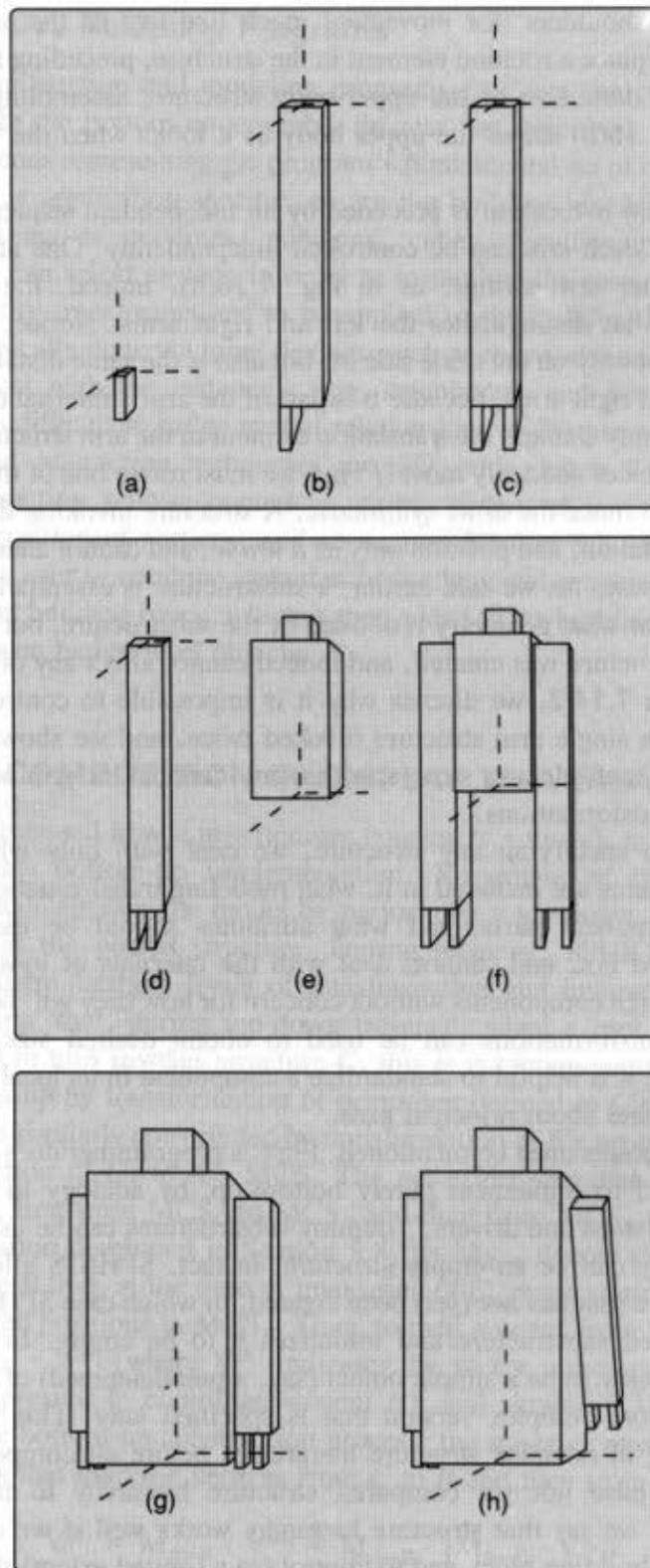
Fig. 7.16 Constructing the robot's upper body. (a) Thumb. (b) Fixed arm plus hand. (c) Completed arm. (d) Completed arm with translated thumb. (e) Trunk and head. (f) Upper body with outsized arms. (g) Corrected arms. (h) Left arm rotated.

connecting the two shoulders (for movement much like that of the arm of a marching soldier); for this, we place a rotation element in the structure, preceding the translation. We have completed our definition of the upper-body structure; assembling the full robot is Exercise 7.1. Fig. 7.16(h) shows the upper body as it looks when the left arm's rotation element is nonzero.

Because each arm invocation is preceded by an independent sequence of transformations, the motion of each arm can be controlled independently. One arm can be hanging down while the other arm swings, as in Fig. 7.16(h). Indeed, the difference in the transformations is what distinguishes the left and right arms. Notice, however, that the movable thumb is not only on the same side of, but also is the same distance from, the fixed hand on both left and right arms, because it is part of the arm's internal definition. (In fact, if the application simply changes the translation element in the arm structure, all the thumbs on all the robot instances suddenly move!) Thus we must rotate one of the arms 180° about the *y* axis in order to make the arms symmetric. A structure invoking the arm can control the arm's size, orientation, and position only *as a whole*, and cannot alter the arm's internal construction in any way. As we said earlier, a substructure is essentially a black box; the invoker needs to know what geometry is defined by the substructure, but it does not need to know how the substructure was created, and indeed cannot affect any of the substructure's internals. In Section 7.14.2, we discuss why it is impossible to control the two thumbs independently with a single arm structure invoked twice, and we show how to solve this problem by creating multiple arm structures that are identical in form but have potentially differing internal transformations.

In summary, in specifying any structure, we deal with only what primitives and lower-level substructures are included in it, what modeling transformations should be used to position its component parts, and what attributes should be used to affect their appearance. We need not, and cannot, deal with the internals of lower-level structures. Furthermore, we design components without concern for how they will be used by invoking structures, since transformations can be used to obtain desired size, orientation, and position. In practice, it is helpful to standardize a component in its local MCS, so that it is easy to scale and rotate about principal axes.

Two additional points must be mentioned. First, a programmer does not need to design purely top-down and to implement purely bottom-up; by analogy to the programming technique of using "stubs and drivers," dummy substructures can be used in the structure hierarchy. A dummy can be an empty structure; in fact, SPHIGS allows a structure to execute a substructure that has not (yet) been created, in which case SPHIGS automatically creates the referenced substructure and initializes it to be empty. In some cases, it is desirable for the dummy to be a simple object (say, a parallelepiped) of approximately the same size as the more complex version that is specified later. This technique permits top-down debugging of complex structure hierarchies before all components are defined fully. Second, we have not yet compared structure hierarchy to template-procedure hierarchy. For now, we say that structure hierarchy works well if we need to instantiate multiple copies of a building block and to control (to a limited extent) the appearance and placement of each copy but not its internal definition. We discuss the tradeoffs between template procedure and structure hierarchy more fully in Section 7.15.

7.5.4 Interactive Modeling Programs

Interactive 3D construction and modeling programs facilitate the construction of object hierarchies through the bottom-up assembly process just described. Most such programs offer a palette of icons representing the program's fundamental set of building blocks. If the drawing program is application-specific, so are the building blocks; otherwise, they are such common atoms as polylines, polygons, cubes, parallelepipeds, cylinders, and spheres. The user can select an icon in order to instantiate the associated building block, and can then specify transformations to be applied to the building-block instance. Such specification is typically done via input devices, such as the mouse or control dials, that let the user experiment with the instance's size, orientation, and position until it "looks right." Since it is difficult to judge spatial relationships in 2D projections of 3D scenes, more sophisticated interaction techniques use 3D grids, often with "gravity fields" surrounding intersection points, numerical scales, sliders of various types, numerical feedback on the position of vertices, and so on (see Section 8.2.6). Some construction programs allow the user to combine instances of fundamental graphical building blocks to create a higher-level building block, which is then added to the building-block palette, to be used in building even higher-level objects.

7.6 MATRIX COMPOSITION IN DISPLAY TRAVERSAL

So far, we have discussed how a programmer constructs a model, using top-down design and (more or less) bottom-up implementation. Regardless of how the model was constructed, SPHIGS displays the model by performing a top-down, depth-first search of the DAG rooted at the posted structure. During traversal, SPHIGS processes all the geometry specified by multiple levels of transformation and invocation. To see what is involved, we observe that, during top-down traversal, when a root structure A invokes structure B, which in turn invokes structure C, this is is tantamount to saying that B was constructed bottom-up by transformation of primitives defined in C's MCS to B's MCS, and that A was then similarly constructed by transformation of B's primitives (including any defined via invocation of C) to A's MCS. The net effect was that C's primitives were transformed twice, first from MCS_C to MCS_B and then from MCS_B to MCS_A.

Using the notation developed in Section 5.8, let $M_{B \leftarrow C}$ denote the value of the local matrix for structure B that, at the time of invocation of C, maps vertices in MCS_C to their properly transformed positions in MCS_B. Thus, to map a vertex from MCS_C to MCS_B, we write $V^{(B)} = M_{B \leftarrow C} \cdot V^{(C)}$ (where $V^{(H)}$ indicates the vector representing a vertex whose coordinates are expressed in coordinate-system H), and similarly, $V^{(A)} = M_{A \leftarrow B} \cdot V^{(B)}$. Thus, to mimic the bottom-up construction process, the traverser must successively apply the transformations that map the vertices from C to B and then from B to A:

$$V^{(A)} = M_{A \leftarrow B} \cdot V^{(B)} = M_{A \leftarrow B} \cdot (M_{B \leftarrow C} \cdot V^{(C)}) \tag{7.1}$$

By matrix associativity, $V^{(A)} = (M_{A \leftarrow B} \cdot M_{B \leftarrow C}) \cdot V^{(C)}$. Therefore, the traverser simply composes the two local matrices and applies the resulting matrix to each of C's vertices.

Using tree notation, let the root be at level 1 and the successive children be at levels 2, 3, 4, ... Then, by induction, for any structure at level j ($j > 4$), we can transform a vertex $V^{(j)}$ in that structure's MCS into the vertex $V^{(1)}$ in the root coordinate system via

$$V^{(1)} = (M_{1 \leftarrow 2} \cdot M_{2 \leftarrow 3} \cdot \ldots \cdot M_{(j-1) \leftarrow j}) \cdot V^{(j)} \qquad (7.2)$$

Since SPHIGS allows primitives to be transformed within the local MCS with the local matrix, a vertex $V^{(j)}$ is obtained by applying the local matrix to the coordinate values of the primitive:

$$V^{(j)} = M^{(j)} \cdot V^{(\text{prim})} \qquad (7.3)$$

We use $M^{(j)}$ to denote the local matrix while the structure is being traversed to show that the matrix is being used to transform primitives into the structure's own level-j MCS. If the structure subsequently invokes a subordinate, the matrix's use changes; it is then used to transform the invoked structure at level $j + 1$ into the level-j MCS, and we denote it with $M_{j \leftarrow (j+1)}$. This does not imply that the matrix's value changes—only its use does.

Combining Eqs. (7.2) and (7.3) using associativity, we get

$$V^{(1)} = (M_{1 \leftarrow 2} \cdot M_{2 \leftarrow 3} \cdot \ldots \cdot M_{(j-1) \leftarrow j} \cdot M^{(j)}) \cdot V^{(\text{prim})} \qquad (7.4)$$

Thus, to transform a primitive at level j in the hierarchy to the MCS of the root (which is the world-coordinate space), all we need to do is to apply the composition of the current values of the local matrix for each structure from the root down to the structure in which the primitive is defined. This composite of local matrices—the term in parentheses in Eq. (7.4)—is called the *composite modeling transformation matrix* (CMTM). When the state of a traversal is such that a level-j structure's elements are being executed, the CMTM is the composition of j matrices. Only the last of those matrices ($M^{(j)}$) may be changed by the structure, because a structure may modify only its local matrix.[9] Thus, while the structure is active, the first $j - 1$ matrices in the CMTM list are constant. The composite of these $j - 1$ matrices is the *global matrix* (GM)—the term in parentheses in Eq. (7.2)—for the level-j structure being executed. It is convenient for SPHIGS to maintain the GM during the traversal of a structure; when a setLocalTransformation element modifies the local matrix (LM), SPHIGS can easily compute the new CMTM by postconcatenating the new local matrix to the GM.

We can now summarize the traversal algorithm, to be elaborated in Section 7.12.1. SPHIGS does a depth-first traversal, saving the CMTM, GM, and LM just before any structure invocation; it then initializes the substructure's GM and CMTM to the inherited CMTM, and its LM to the identity matrix. The CMTM is applied to vertices and is updated by changes to the LM. Finally, when the traverser returns, it restores the CMTM, GM, and LM of the parent structure and continues. Because of the saving and restoring of the matrices, parents affect their children but not vice versa.

---

[9] We present the global matrix as a derived entity that cannot be modified by a structure. In true PHIGS, a structure can modify the GM active during its execution, but this power is still "localized" in the sense that it in no way affects the local matrices of its ancestors.

Let us watch as SPHIGS traverses the three-level upper-body–arm–thumb hierarchy of Fig. 7.15. We have posted the UPPER_BODY structure as a root. Figure 7.17(a) shows a sequence of snapshots of the traversal state; a snapshot has been created for each point marked with a number in the structure network diagram of Fig. 7.17(b).

The traversal state is maintained via a stack, shown in Fig. 7.17(a) growing downward, with the currently active structure in a solid rectangle and its ancestors in dashed ones. The values of the three state matrices for the currently active structure are shown to the right of the stack diagram. Arcs show the scope of a transformation: The GM arc illustrates that a structure's ancestors contribute to the GM, and the CMTM arc shows that the CMTM is the product of the GM and LM. Recall that in each group of transformations, the first is in REPLACE mode and the rest are PRECONCATENATEd. Thus the first ''rotate'' in the structure applies only to the head since it will be REPLACEd by the first ''scale'' that applies to the left arm.

At point 1 in Fig. 7.17(b), the traverser is about to execute the first element in the root. Because a root has no ancestors, its GM is identity; the LM is also identity, as is the case whenever a structure's execution begins. At point 2, the LM is set to the composite of the (Scale, Rotate, Translate) transformation triplet. Therefore, the CMTM is updated to the product of the identity GM and the SRT composite, and is then ready for use in transforming the arm subobject to the upper body MCS to become the left arm via $(SRT)_{ub \leftarrow la}$. Next, at point 3, the traverser is about to execute the first element of the arm structure. The GM for the arm execution is, as one would expect for a level-2 instantiation, its parent's LM at the point of invocation.

At point 4, the arm LM is set to position the thumb within the arm $(T_{arm \leftarrow th})$, and the CMTM is updated to the product of the GM and LM. This level-2 CMTM becomes the GM for the level-3 instantiation of the thumb (point 5). Since the LM of the thumb is identity, the CMTM of the thumb has the desired effect of transforming thumb coordinates first to arm coordinates, then to upper-body coordinates. At point 6, the traverser has returned from the thumb and arm invocations, back to the upper body. The matrices for the upper-body structure are as they were before the invocation, since its subordinates cannot change its local matrix. At point 7, the LM of the upper body is replaced with a new composite for the right arm. When we have descended into the thumb structure for the right arm (point 8), the CMTM is almost identical to that at point 5; the only difference is the level-2 matrix that moves the arm into position on the upper body.

To animate a composite object such as the robot, we need only to think about how each child structure is to be affected in its parent, and to define the appropriate transformation elements for each component that can be edited dynamically later. Thus, to make the robot spin about its axis, raise its arm, and open its hand, we change the rotation matrix in the robot structure to affect the upper body, the rotation matrix in the upper-body structure to affect the arm, and the translation matrix in the arm structure to affect the thumb. The transformations are done independently at each level of the hierarchy, but the net effect is cumulative. The difficult part of specifying an animation is working backward from a desired result, such as ''the robot moves to the northwest corner of the room and picks up a block from the table,'' to derive the sequence of transformations yielding that result. Chapter 21 briefly discusses the issues involved in such ''inverse kinematics'' problems.
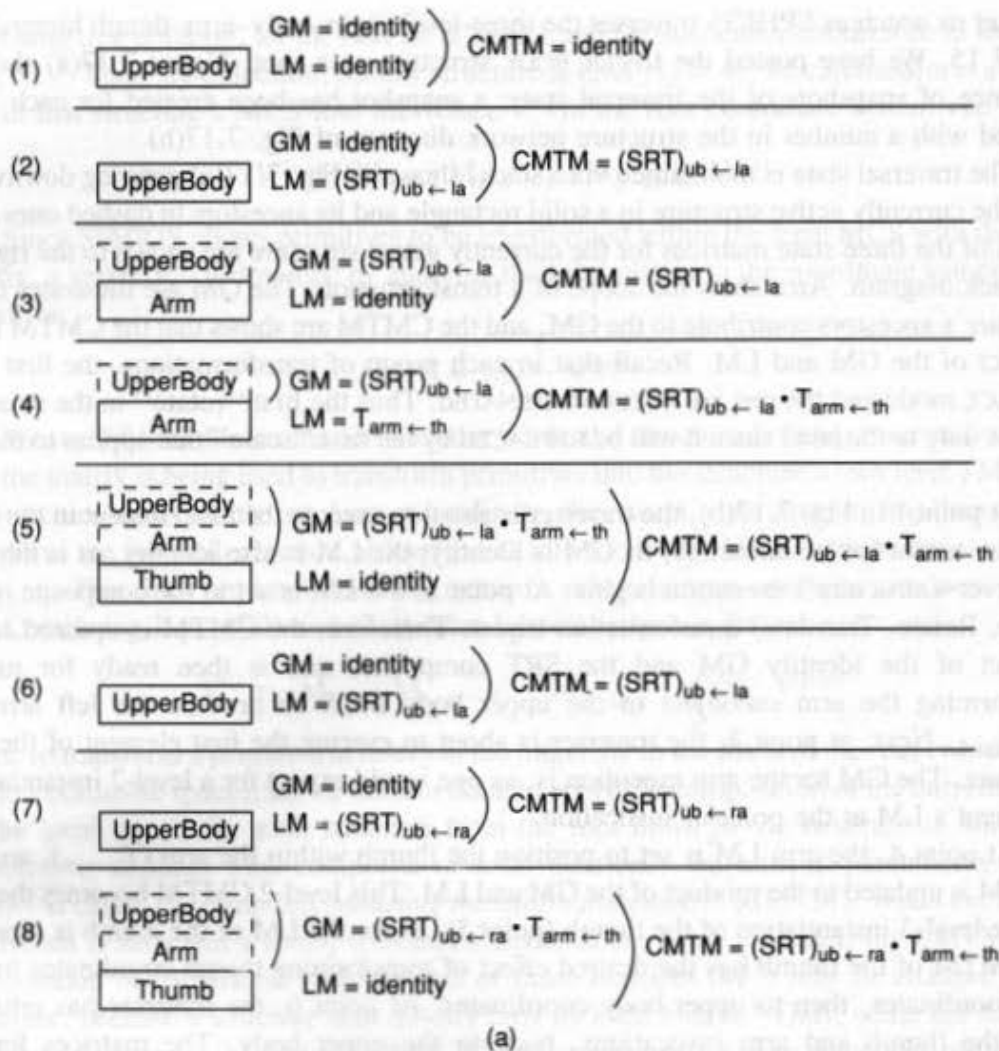
Fig. 7.17 Traversal of a three-level hierarchy. (a) Snapshots of traversal-state stack. (b) Annotated structure network.

7.7 APPEARANCE-ATTRIBUTE HANDLING IN HIERARCHY

7.7.1 Inheritance Rules

The attribute-traversal state is set by attribute elements during traversal, and, as in SRGP, is applied modally to all primitives encountered. We saw how parents affect their children via geometric transformations. What rules pertain to appearance attributes? In our street example, the houses all have the default color. To give an object a particular color (e.g., to make a house brown), we can specify that color as an initial element in the object structure itself, but that makes the object's color intrinsic and not changeable during traversal. We

Fig. 7.17 *(Cont'd.)*

(b)
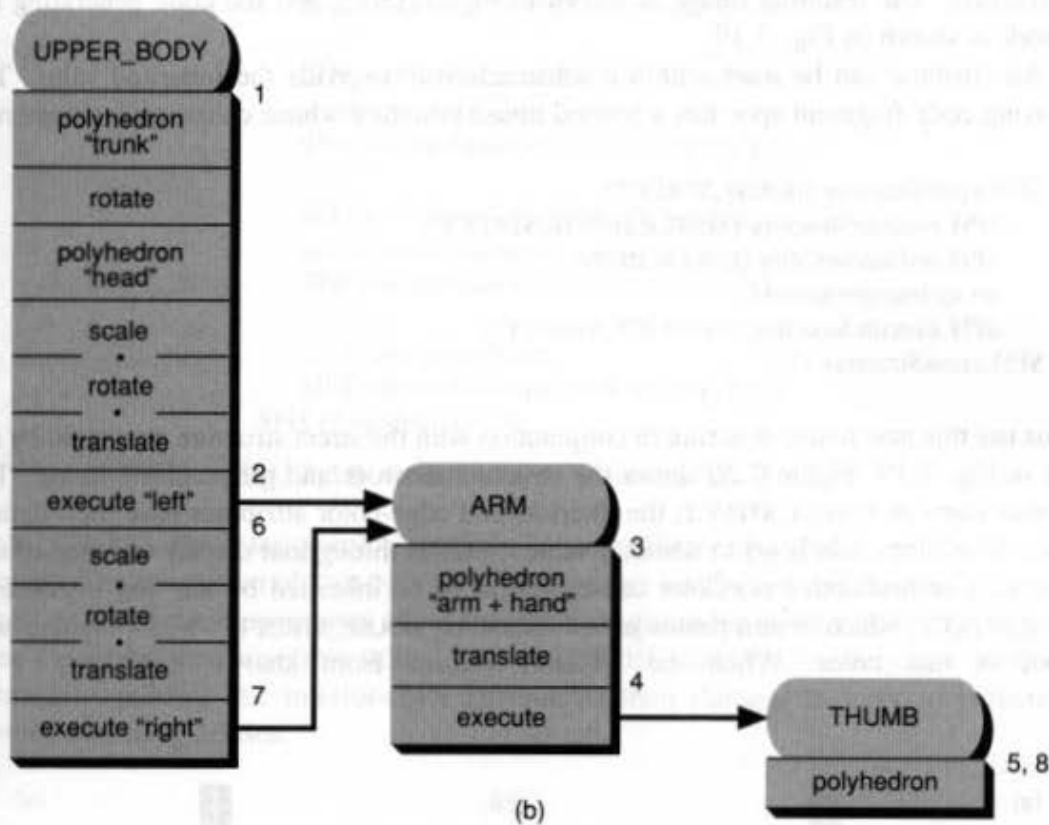
would prefer to "pass the color as a parameter," so that the child can inherit it the way a child inherits the CMTM as its GM.

Indeed, in SPHIGS, each substructure inherits the traversal state as the latter exists at the time of the invocation of the substructure, and can then modify that state at will without affecting its ancestors. In other words, attributes and transformations are bound dynamically at traversal time, rather than statically at specification time. This dynamic binding is one of the major features of SPHIGS, making it easy to customize instances of a substructure.

What substructures do with the inherited state depends on the type of data involved. We saw that, for geometric transformations, the substructure inherits the GM but cannot override its inheritance, since it can affect only its own local matrix. Attributes are simpler in that the substructure inherits the parent's attributes as the initial values of its local attribute state, but can change its local state subsequently. There is no need to distinguish between global and local attributes, since there is no notion of composing attributes. Note that this mechanism has the same problem we discovered with transformation inheritance—just as our robot's two arm instances cannot have differing thumb transformations, its two arm instances cannot have the same color for the fixed part but differing colors for the thumb.

In the structure network of Fig. 7.18(a), the street structure sets the colors for the house substructure. The resulting image is shown in Fig. 7.18(b), and the code generating the network is shown in Fig. 7.19.

An attribute can be reset within a substructure to override the inherited value. The following code fragment specifies a revised house structure whose chimney is always red.

```
SPH_openStructure (HOUSE_STRUCT);
    SPH_executeStructure (SIMPLE_HOUSE_STRUCT);
    SPH_setInteriorColor (COLOR_RED);
    set up transformation;
    SPH_executeStructure (CHIMNEY_STRUCT);
SPH_closeStructure ();
```

Let us use this new house structure in conjunction with the street structure generated by the code in Fig. 7.19. Figure 7.20 shows the structure network and the resulting image. The traverser starts at STREET_STRUCT; the interior- and edge-color attributes have their default values. The edge color is set to white, a value it retains throughout display traversal of this network. The first setInteriorColor causes yellow to be inherited by the first instance of HOUSE_STRUCT, which in turn passes yellow to SIMPLE_HOUSE_STRUCT, whose polyhedron is shown in that color. When the traverser returns from SIMPLE_HOUSE_STRUCT to
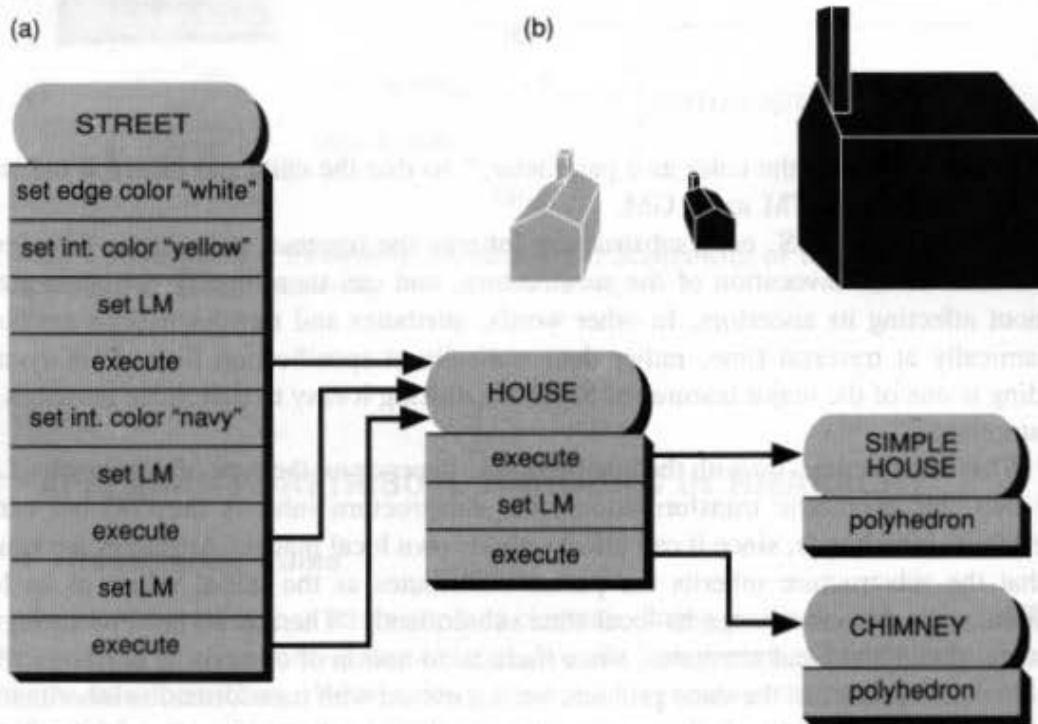

Fig. 7.18 Use of attribute inheritance to model street with colored houses. (a) Structure network. (b) Resulting image. (Interior colors are simulated by patterns.)

```
SPH_openStructure (STREET_STRUCT);
SPH_setEdgeColor (COLOR_WHITE);

SPH_setInteriorColor (COLOR_YELLOW);
set up transformation;
SPH_executeStructure (HOUSE_STRUCT);

SPH_setInteriorColor (COLOR_NAVY);
set up transformation;
SPH_executeStructure (HOUSE_STRUCT);

set up transformation;
SPH_executeStructure (HOUSE_STRUCT);
SPH_closeStructure ();
```

Fig. 7.19 Code used to generate Fig. 7.18.

HOUSE_STRUCT, the interior-color attribute is immediately changed to red by the next element. The invocation of CHIMNEY_STRUCT therefore results in a red chimney with white edges. None of these operations affect the attribute group for STREET_STRUCT, of course; when the traverser returns from HOUSE_STRUCT, STREET_STRUCT's interior-color attribute is restored to yellow. The interior-color attribute is then changed to navy to prepare for drawing two navy houses.
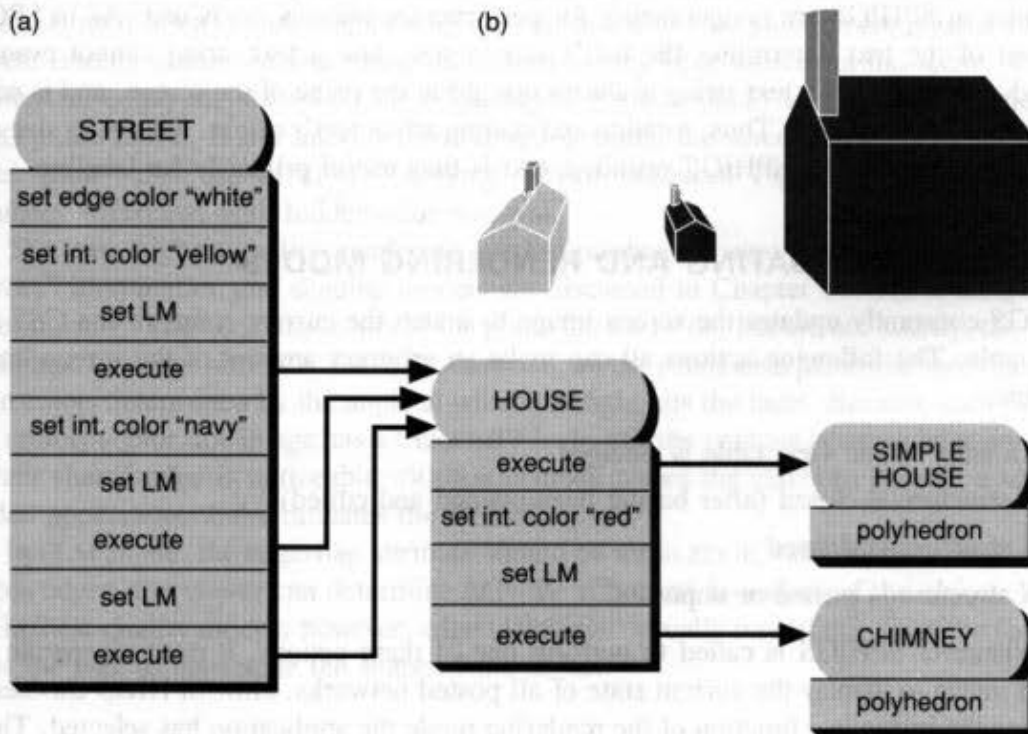

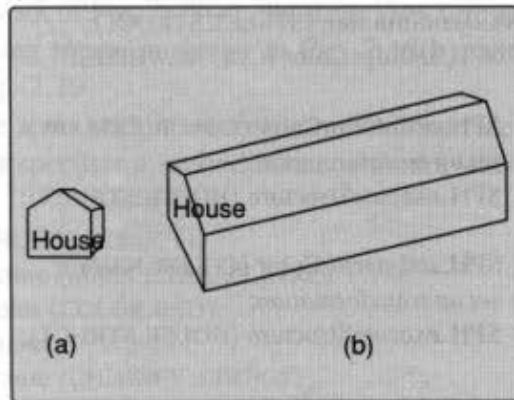**Fig. 7.20** Subordinate structure overriding an inherited attribute. (a) Structure network. (b) Resulting view.

Fig. 7.21 The nongeometric nature of text in SPHIGS. (a) Before transformation. (b) After transformation.

7.7.2 SPHIGS Attributes and Text Unaffected By Transformations

In true PHIGS implementations, text can be subjected to transformations like any other primitive. Thus, the text characters on the sides of a truck, displayed in perspective, are rotated and shown with appropriate perspective foreshortening, as though the letters were made out of individual polylines or fill areas. Similarly, dashes in dashed lines should be subject to geometric transformations and perspective foreshortening. However, just as attributes in SPHIGS are nongeometric for performance reasons, so is text. As in SRGP, the font of the text determines the text's screen size, and a text string cannot even be rotated—the image of a text string is always upright in the plane of the screen, and is never compressed or expanded. Thus, rotation and scaling affect text's origin, but not its size and orientation (Fig. 7.21). SPHIGS primitive text is thus useful primarily for labeling.

7.8 SCREEN UPDATING AND RENDERING MODES

SPHIGS constantly updates the screen image to match the current status of the CSS and view table. The following actions all can make an arbitrary amount of the screen image obsolete:

- An entry in the view table is changed
- A structure is closed (after having been opened and edited)
- A structure is deleted
- A structure is posted or unposted.

Whenever SPHIGS is called to perform one of these actions, it must regenerate the screen image to display the current state of all posted networks. How SPHIGS chooses to generate the image is a function of the rendering mode the application has selected. These modes present a choice between quality and speed of regeneration: The higher the quality, the longer it takes to render the image. The rendering mode is set by

procedure SPH_setRenderingMode (*mode* : WIREFRAME / FLAT / LIT_FLAT / GOURAUD);