CHAPTER 1

Introduction

Computer graphics is generally regarded as a branch of computer science that deals with the theory and technology for computerized image synthesis. A computer-generated image can depict a scene as simple as the outline of a triangle on a uniform background and as complex as a magnificent dinosaur in a tropical forest. But how do these things become part of the picture? What makes drawing on a computer different from sketching with a pen or photographing with a camera? In this chapter we will introduce some important concepts and outline the relationship among these concepts. The goal of such a mini-survey of the field of computer graphics is to enable us to appreciate the various answers to these questions that we will detail in the rest of the book not only in their own right but also in the context of the overall framework.

1.1 A MINI-SURVEY

First let's consider drawing the outline of a triangle (see Fig. 1-1). In real life this would begin with a decision in our mind regarding such geometric characteristics as the type and size of the triangle, followed by our action to move a pen across a piece of paper. In computer graphics terminology, what we have envisioned is called the object definition, which defines the triangle in an abstract space of our choosing. This space is continuous and is called the *object space*. Our action to draw maps the imaginary object into a triangle on paper, which constitutes a continuous display surface in another space called the *image space*. This mapping action is further influenced by our choice regarding such factors as the location and orientation of the triangle. In other words, we may place the triangle in the middle of the paper, or we may draw it near the upper left corner. We may have the sharp corner of the triangle pointing to the right, or we may have it pointing to the left.

A comparable process takes place when a computer is used to produce the picture. The major computational steps involved in the process give rise to several important areas of computer graphics. The area that attends to the need to define objects, such as the triangle, in an efficient and effective manner is called geometric representation. In our example we can place a two-dimensional Cartesian coordinate system into the object space. The triangle can then be represented by the x and y coordinates of its three vertices, with the understanding that the computer system will connect the first and second vertices with a line segment, the second and third vertices with another line segment, and the third and first with yet another line segment.

The next area of computer graphics that deals with the placement of the triangle is called transformation. Here we use matrices to realize the mapping of the triangle to its final destination in the image space. We can set up the transformation matrix to control the location and orientation of the displayed triangle. We can even enlarge or reduce its size. Furthermore, by using multiple settings for the

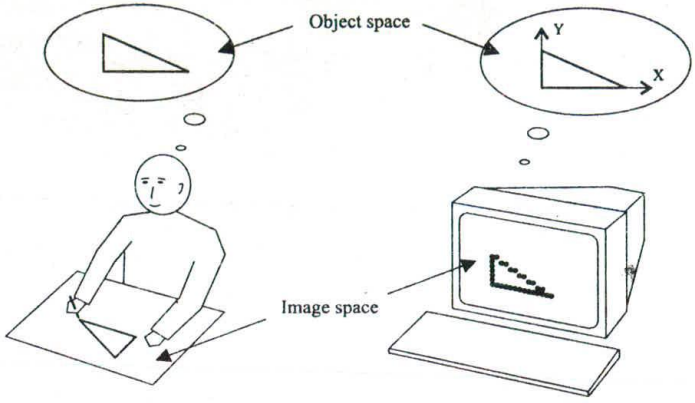


Fig. 1-1 Drawing a triangle.

transformation matrix, we can instruct the computer to display several triangles of varying size and orientation at different locations, all from the same model in the object space.

At this point most readers may have already been wondering about the crucial difference between the triangle drawn on paper and the triangle displayed on the computer monitor (an exaggerated version of what you would see on a real monitor). The former has its vertices connected by smooth edges, whereas the latter is not exactly a line-drawing. The fundamental reason here is that the image space in computer graphics is, generally speaking, not continuous. It consists of a set of discrete pixels, i.e., picture elements, that are arranged in a row-and-column fashion. Hence a horizontal or vertical line segment becomes a group of adjacent pixels in a row or column, respectively, and a slanted line segment becomes something that resembles a staircase. The area of computer graphics that is responsible for converting a continuous figure, such as a line segment, into its discrete approximation is called scan conversion.

The distortion introduced by the conversion from continuous space to discrete space is referred to as the aliasing effect of the conversion. While reducing the size of individual pixels should make the distortion less noticeable, we do so at a significant cost in terms of computational resources. For instance, if we cut each pixel by half in both the horizontal and the vertical direction we would need four times the number of pixels in order to keep the physical dimension of the picture constant. This would translate into, among other things, four times the memory requirement for storing the image. Exploring other ways to alleviate the negative impact of the aliasing effect is the focus of another area of computer graphics called anti-aliasing.

Putting together what we have so far leads to a simplified graphics pipeline (see Fig. 1-2), which exemplifies the architecture of a typical graphics system. At the start of the pipeline, we have primitive objects represented in some application-dependent data structures. For example, the coordinates of the vertices of a triangle, viz., (x_1, y_1), (x_2, y_2), and (x_3, y_3), can be easily stored in a 3×2 array. The graphics system first performs transformation on the original data according to user-specified parameters, and then carries out scan conversion with or without anti-aliasing to put the picture on the screen. The coordinate system in the middle box in Fig. 1-2 serves as an intermediary between the object coordinate system on the

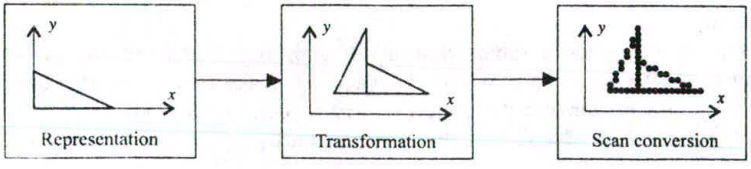


Fig. 1-2 A simple graphics pipeline.

left and the image or device coordinate system on the right. It is called the world coordinate system, representing where we place transformed objects to compose the picture we want to draw. The example in the box shows two triangles: the one on the right is a scaled copy of the original that is moved up and to the right, the one on the left is another scaled copy of the original that is rotated 90° counterclockwise around the origin of the coordinate system and then moved up and to the right in the same way.

In a typical implementation of the graphics pipeline we would write our application program in a host programming language and call library subroutines to perform graphics operations. Some subroutines are used to prescribe, among other things, transformation parameters. Others are used to draw, i.e., to feed original data into the pipeline so current system settings are automatically applied to shape the end product coming out of the pipeline, which is the picture on the screen.

Having looked at the key ingredients of what is called two-dimensional graphics, we now turn our attention to three-dimensional graphics. With the addition of a third dimension one should notice the profound distinction between an object and its picture. Figure 1-3 shows several possible ways to draw a cubic object, but none of the drawings even come close to being the object itself. The drawings simply represent projections of the three-dimensional object onto a two-dimensional display surface. This means that besides three-dimensional representation and transformation, we have an additional area of computer graphics that covers projection methods.

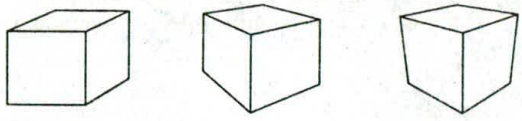


Fig. 1-3 Several ways to depict a cube.

Did you notice that each drawing in Fig. 1-3 shows only three sides of the cubic object? Being a solid three-dimensional object the cube has six plane surfaces. However, we depict it as if we were looking at it in real life. We only draw the surfaces that are visible to us. Surfaces that are obscured from our eyesight are not shown. The area of computer graphics that deals with this computational task is called hidden surface removal. Adding projection and hidden surface removal to our simple graphics pipeline, right after transformation but before scan conversion, results in a prototype for three-dimensional graphics.

Now let's follow up on the idea that we want to produce a picture of an object in real-life fashion. This presents a great challenge for computer graphics, since there is an extremely effective way to produce such a picture: photography. In order to generate a picture that is photo-realistic, i.e., that looks as good as a photograph, we need to explore how a camera and nature work together to produce a snapshot.

When a camera is used to photograph a real-life object illuminated by a light source, light energy coming out of the light source gets reflected from the object surface through the camera lens onto the negative, forming an image of the object. Generally, the part of the object that is closer to the light source should appear brighter in the picture than the part that is further away, and the part of the object that is facing away from the light source should appear relatively dark. Figure 1-4 shows a computer-generated

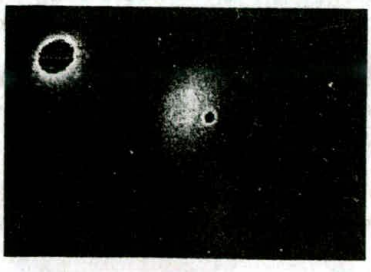


Fig. 1-4 Two shaded spheres.

image that depicts two spherical objects illuminated by a light source that is located somewhere between the spheres and the "camera" at about the ten to eleven o'clock position. Although both spheres have gradual shadings, the bright spot on the large sphere looks like a reflection of the light source and hence suggests a difference in their reflectance property (the large sphere being shinier than the small one). The mathematical formulae that mimic this type of optical phenomenon are referred to as local illumination models, for the energy coming directly from the light source to a particular object surface is not a full account of the energy arriving at that surface. Light energy is also reflected from one object surface to another, and it can go through a transparent or translucent object and continue on to other places. Computational methods that strive to provide a more accurate account of light transport than local illumination models are referred to as global illumination models.

Now take a closer look at Fig. 1-4. The two objects seem to have super-smooth surfaces. What are they made of? How can they be so perfect? Do you see many physical objects around you that exhibit such surface characteristics? Furthermore, it looks like the small sphere is positioned between the light source and the large sphere. Shouldn't we see its shadow on the large sphere? In computer graphics the surface shading variations that distinguish a wood surface from a marble surface or other types of surface are referred to as surface textures. There are various techniques to add surface textures to objects to make them look more realistic. On the other hand, the computational task to include shadows in a picture is called shadow generation.

Before moving on to prepare for a closer look at each of the subject areas we have introduced in this mini-survey, we want to briefly discuss a couple of allied fields of computer science that also deal with graphical information.

Image Processing

The key element that distinguishes image processing (or digital image processing) from computer graphics is that image processing generally begins with images in the image space and performs pixel-based operations on them to produce new images that exhibit certain desired features. For example, we may reset each pixel in the image displayed on the monitor screen in Fig. 1-1 to its complementary color (e.g., black to white and white to black), turning a dark triangle on a white background to a white triangle on a dark background, or vice versa. While each of these two fields has its own focus and strength, they also overlap and complement each other. In fact, stunning visual effects are often achieved by using a combination of computer graphics and image processing techniques.

Computer–Human Interaction

While the main focus of computer graphics is the production of images, the field of computer–human interaction promotes effective communication between man and machine. The two fields join forces when it comes to such areas as graphical user interfaces. There are many kinds of physical devices that can be attached to a computer for the purpose of interaction, starting with the keyboard and the mouse. Each physical device can often be programmed to deliver the function of various logical devices (e.g., Locator, Choice—see below). For example, a mouse can be used to specify locations in the image space (acting as a Locator device). In this case a cursor is often displayed as visual feedback to allow the user see the locations being specified. A mouse can also be used to select an item in a pull-down or pop-up manual (acting as a Choice device). In this case it is the identification of the selected manual item that counts and the item is often highlighted as a whole (the absolute location of the cursor is essentially irrelevant). From these we can see that a physical device may be used in different ways and information can be conveyed to the user in different graphical forms. The key challenge is to design interactive protocols that make effective use of devices and graphics in a way that is user-friendly—easy, intuitive, efficient, etc.

1.2 WHAT'S AHEAD

We hope that our brief flight over the landscape of the graphics kingdom has given you a good impression of some of the important landmarks and made you eager to further your exploration. The following chapters are dedicated to the various subject areas of computer graphics. Each chapter begins with the necessary background information (e.g., context and terminology) and a summary account of the material to be discussed in subsequent sections.

We strive to provide clear explanation and inter-subject continuity in our presentation. Illustrative examples are used freely to substantiate discussion on abstract concepts. While the primary mission of this book is to offer a relatively well-focused introduction to the fundamental theory and underlying technology, significant variations in such matters as basic definitions and implementation protocols are presented in order to have a reasonably broad coverage of the field. In addition, interesting applications are introduced as early as possible to highlight the usefulness of the graphics technology and to encourage those who are eager to engage in hands-on practice.

Algorithms and programming examples are given in pseudo-code that resembles the C programming language, which shares similar syntax and basic constructs with other widely used languages such as C++ and Java. We hope that the relative simplicity of the C-style code presents little grammatical difficulty and hence makes it easy for you to focus your attention on the technical substance of the code.

There are numerous solved problems at the end of each chapter to help reinforce the theoretical discussion. Some of the problems represent computation steps that are omitted in the text and are particularly valuable for those looking for further details and additional explanation. Other problems may provide new information that supplements the main discussion in the text.

CHAPTER 2

Image Representation

A digital image, or image for short, is composed of discrete pixels or picture elements. These pixels are arranged in a row-and-column fashion to form a rectangular picture area, sometimes referred to as a raster. Clearly the total number of pixels in an image is a function of the size of the image and the number of pixels per unit length (e.g. inch) in the horizontal as well as the vertical direction. This number of pixels per unit length is referred to as the resolution of the image. Thus a 3×2 inch image at a resolution of 300 pixels per inch would have a total of 540,000 pixels.

Frequently image size is given as the total number of pixels in the horizontal direction times the total number of pixels in the vertical direction (e.g., 512×512, 640×480, or 1024×768). Although this convention makes it relatively straightforward to gauge the total number of pixels in an image, it does not specify the size of the image or its resolution, as defined in the paragraph above. A 640×480 image would measure $6\frac{2}{3}$ inches by 5 inches when presented (e.g., displayed or printed) at 96 pixels per inch. On the other hand, it would measure 1.6 inches by 1.2 inches at 400 pixels per inch.

The ratio of an image's width to its height, measured in unit length or number of pixels, is referred to as its aspect ratio. Both a 2×2 inch image and a 512×512 image have an aspect ratio of 1/1, whereas both a $6 \times 4\frac{1}{2}$ inch image and a 1024×768 image have an aspect ratio of 4/3.

Individual pixels in an image can be referenced by their coordinates. Typically the pixel at the lower left corner of an image is considered to be at the origin $(0, 0)$ of a pixel coordinate system. Thus the pixel at the lower right corner of a 640×480 image would have coordinates $(639, 0)$, whereas the pixel at the upper right corner would have coordinates $(639, 479)$.

The task of composing an image on a computer is essentially a matter of setting pixel values. The collective effects of the pixels taking on different color attributes give us what we see as a picture. In this chapter we first introduce the basics of the most prevailing color specification method in computer graphics (Sect. 2.1). We then discuss the representation of images using direct coding of pixel colors (Sect. 2.2) versus using the lookup-table approach (Sect. 2.3). Following a discussion of the working principles of two representative image presentation devices, the display monitor (Sect. 2.4) and the printer (Sect. 2.5), we examine image files as the primary means of image storage and transmission (Sect. 2.6). We then take a look at some of the most primitive graphics operations, which primarily deal with setting the color attributes of pixels (Sect. 2.7). Finally, to illustrate the construction of beautiful images directly in the discrete image space, we introduce the mathematical background and detail the algorithmic aspects of visualizing the Mandelbrot set (Sect. 2.8).

2.1 THE RGB COLOR MODEL

Color is a complex, interdisciplinary subject spanning from physics to psychology. In this section we only introduce the basics of the most widely used color representation method in computer graphics. We will have additional discussion later in another chapter.

Figure 2-1 shows a color coordinate system with three primary colors: R (red), G (green), and B (blue). Each primary color can take on an intensity value ranging from 0 (off—lowest) to 1 (on—highest). Mixing these three primary colors at different intensity levels produces a variety of colors. The collection of all the colors obtainable by such a linear combination of red, green, and blue forms the cube-shaped RGB color space. The corner of the RGB color cube that is at the origin of the coordinate system corresponds to black, whereas the corner of the cube that is diagonally opposite to the origin represents white. The diagonal line connecting black and white corresponds to all the gray colors between black and white. It is called the gray axis.

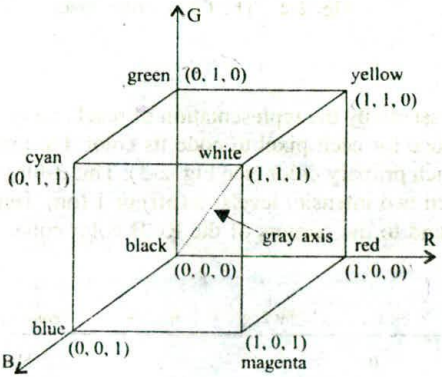


Fig. 2-1 The RGB color space.

Given this RGB color model an arbitrary color within the cubic color space can be specified by its color coordinates: (r, g, b). For example, we have $(0, 0, 0)$ for black, $(1, 1, 1)$ for white, $(1, 1, 0)$ for yellow, etc. A gray color at $(0.7, 0.7, 0.7)$ has an intensity halfway between one at $(0.9, 0.9, 0.9)$ and one at $(0.5, 0.5, 0.5)$.

Color specification using the RGB model is an additive process. We begin with black and add on the appropriate primary components to yield a desired color. This closely matches the working principles of the display monitor (see Sect. 2.4). On the other hand, there is a complementary color model, called the CMY color model, that defines colors using a subtractive process, which closely matches the working principles of the printer (see Sect. 2.5).

In the CMY model we begin with white and take away the appropriate primary components to yield a desired color. For example, if we subtract red from white, what remains consists of green and blue, which is cyan. Looking at this from another perspective, we can use the amount of cyan, the complementary color of red, to control the amount of red, which is equal to one minus the amount of cyan. Figure 2-2 shows a coordinate system using the three primaries' complementary colors: C (cyan), M (magenta), and Y (yellow). The corner of the CMY color cube that is at $(0, 0, 0)$ corresponds to white, whereas the corner of the cube that is at $(1, 1, 1)$ represents black (no red, no green, no blue). The following formulas summarize the conversion between the two color models:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix} \qquad \begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

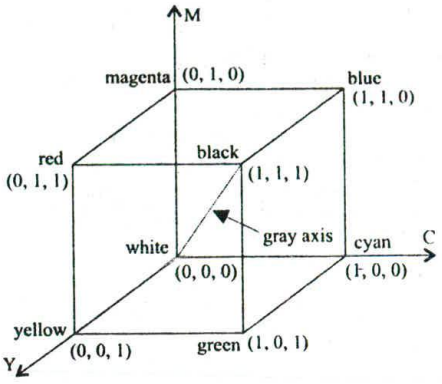


Fig. 2-2 The CMY color space.

2.2 DIRECT CODING

Image representation is essentially the representation of pixel colors. Using direct coding we allocate a certain amount of storage space for each pixel to code its color. For example, we may allocate 3 bits for each pixel, with one bit for each primary color (see Fig. 2-3). This 3-bit representation allows each primary to vary independently between two intensity levels: 0 (off) or 1 (on). Hence each pixel can take on one of the eight colors that correspond to the corners of the RGB color cube.

bit 1: r	bit 2: g	bit 3: b	color name
0	0	0	black
0	0	1	blue
0	1	0	green
0	1	1	cyan
1	0	0	red
1	0	1	magenta
1	1	0	yellow
1	1	1	white

Fig. 2-3 Direct coding of colors using 3 bits.

A widely accepted industry standard uses 3 bytes, or 24 bits, per pixel, with one byte for each primary color. This way we allow each primary color to have 256 different intensity levels, corresponding to binary values from 00000000 to 11111111. Thus a pixel can take on a color from 256 × 256 × 256 or 16.7 million possible choices. This 24-bit format is commonly referred to as the true color representation, for the difference between two colors that differ by one intensity level in one or more of the primaries is virtually undetectable under normal viewing conditions. Hence a more precise representation involving more bits is of little use in terms of perceived color accuracy.

A notable special case of direct coding is the representation of black-and-white (bilevel) and gray-scale images, where the three primaries always have the same value and hence need not be coded separately. A black-and-white image requires only one bit per pixel, with bit value 0 representing black and 1 representing white. A gray-scale image is typically coded with 8 bits per pixel to allow a total of 256 intensity or gray levels.

Although this direct coding method features simplicity and has supported a variety of applications, we can see a relatively high demand for storage space when it comes to the 24-bit standard. For example, a 1000 × 1000 true color image would take up three million bytes. Furthermore, even if every pixel in that

image had a different color, there would only be one million colors in the image. In many applications the number of colors that appear in any one particular image is much less. Therefore the 24-bit representation's ability to have 16.7 million different colors appear simultaneously in a single image seems to be somewhat overkill.

2.3 LOOKUP TABLE

Image representation using a lookup table can be viewed as a compromise between our desire to have a lower storage requirement and our need to support a reasonably sufficient number of simultaneous colors. In this approach pixel values do not code colors directly. Instead, they are addresses or indices into a table of color values. The color of a particular pixel is determined by the color value in the table entry that the value of the pixel references.

Figure 2-4 shows a lookup table with 256 entries. The entries have addresses 0 through 255. Each entry contains a 24-bit RGB color value. Pixel values are now 1-byte, or 8-bit, quantities. The color of a pixel whose value is i, where $0 \leq i \leq 255$, is determined by the color value in the table entry whose address is i. This 24-bit 256-entry lookup table representation is often referred to as the 8-bit format. It reduces the storage requirement of a 1000×1000 image to one million bytes plus 768 bytes for the color values in the lookup table. It allows 256 simultaneous colors that are chosen from 16.7 million possible colors.

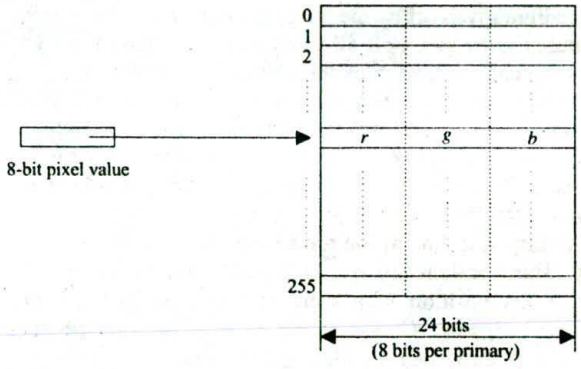


Fig. 2-4 A 24-bit 256-entry lookup table.

It is important to remember that, using the lookup table representation, an image is defined not only by its pixel values but also by the color values in the corresponding lookup table. Those color values form a *color map* for the image.

2.4 DISPLAY MONITOR

Among the numerous types of image presentation or output devices that convert digitally represented images into visually perceivable pictures is the display or video monitor.

We first take a look at the working principle of a monochromatic display monitor, which consists mainly of a cathode ray tube (CRT) along with related control circuits. The CRT is a vacuum glass tube with the display screen at one end and connectors to the control circuits at the other (see Fig. 2-5). Coated on the inside of the display screen is a special material, called phosphor, which emits light for a period of time when hit by a beam of electrons. The color of the light and the time period vary from one type of

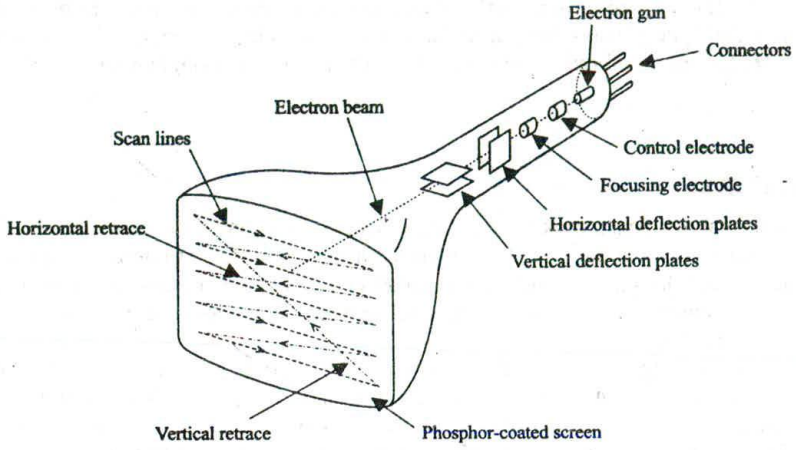


Fig. 2-5 Anatomy of a monochromatic CRT.

phosphor to another. The light given off by the phosphor during exposure to the electron beam is known as fluorescence, the continuing glow given off after the beam is removed is known as phosphorescence, and the duration of phosphorescence is known as the phosphor's persistence.

Opposite to the phosphor-coated screen is an electron gun that is heated to send out electrons. The electrons are regulated by the control electrode and forced by the focusing electrode into a narrow beam striking the phosphor coating at small spots. When this electron beam passes through the horizontal and vertical deflection plates, it is bent or deflected by the electric fields between the plates. The horizontal plates control the beam to scan from left to right and retrace from right to left. The vertical plates control the beam to go from the first scan line at the top to the last scan line at the bottom and retrace from the bottom back to the top. These actions are synchronized by the control circuits so that the electron beam strikes each and every pixel position in a scan line by scan line fashion. As an alternative to this electrostatic deflection method, some CRTs use magnetic deflection coils mounted on the outside of the glass envelope to bend the electron beam with magnetic fields.

The intensity of the light emitted by the phosphor coating is a function of the intensity of the electron beam. The control circuits shut off the electron beam during horizontal and vertical retraces. The intensity of the beam at a particular pixel position is determined by the intensity value of the corresponding pixel in the image being displayed.

The image being displayed is stored in a dedicated system memory area that is often referred to as the frame buffer or refresh buffer. The control circuits associated with the frame buffer generate proper video signals for the display monitor. The frequency at which the content of the frame buffer is sent to the display monitor is called the refreshing rate, which is typically 60 times or frames per second (60 Hz) or higher. A determining factor here is the need to avoid flicker, which occurs at lower refreshing rates when our visual system is unable to integrate the light impulses from the phosphor dots into a steady picture. The persistence of the monitor's phosphor, on the other hand, needs to be long enough for a frame to remain visible but short enough for it to fade before the next frame is displayed.

Some monitors use a technique called interlacing to "double" their refreshing rate. In this case only half of the scan lines in a frame is refreshed at a time, first the odd numbered lines, then the even numbered lines. Thus the screen is refreshed from top to bottom in half the time it would have taken to sweep across all the scan lines. Although this approach does not really increase the rate at which the entire screen is refreshed, it is quite effective in reducing flicker.

Color Display

Moving on to color displays there are now three electron guns instead of one inside the CRT (see Fig. 2-6), with one electron gun for each primary color. The phosphor coating on the inside of the display screen consists of dot patterns of three different types of phosphors. These phosphors are capable of emitting red, green, and blue light, respectively. The distance between the center of the dot patterns is called the pitch of the color CRT. It places an upper limit on the number of addressable positions on the display area. A thin metal screen called a shadow mask is placed between the phosphor coating and the electron guns. The tiny holes on the shadow mask constrain each electron beam to hit its corresponding phosphor dots. When viewed at a certain distance, light emitted by the three types of phosphors blends together to give us a broad range of colors.

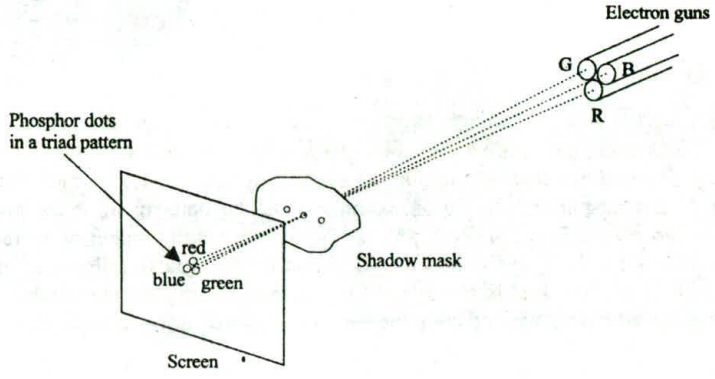


Fig. 2-6 Color CRT using a shadow mask.

2.5 PRINTER

Another typical image presentation device is the printer. A printer deposits color pigments onto a print media, changing the light reflected from its surface and making it possible for us to see the print result.

Given the fact that the most commonly used print media is a piece of white paper, we can in principle utilize three types of pigments (cyan, magenta, and yellow) to regulate the amount of red, green, and blue light reflected to yield all RGB colors (see Sect. 2.1). However, in practice, an additional black pigment is often used due to the relatively high cost of color pigments and the technical difficulty associated with producing high-quality black from several color pigments.

While some printing methods allow color pigments to blend together, in many cases the various color pigments remain separate in the form of tiny dots on the print media. Furthermore, the pigments are often deposited with a limited number of intensity levels. There are various techniques to achieve the effect of multiple intensity levels beyond what the pigment deposits can offer. Most of these techniques can also be adapted by the display devices that we have just discussed in the previous section.

Halftoning

Let's first take a look at a traditional technique called halftoning from the printing industry for bilevel devices. This technique uses variably sized pigment dots that, when viewed from a certain distance, blend with the white background to give us the sensation of varying intensity levels. These dots are arranged in a pattern that forms a 45° screen angle with the horizon (see Fig. 2-7 where the dots are enlarged for illustration). The size of the dots is inversely proportional to the intended intensity level. When viewed at a

far enough distance, the stripe in Fig. 2-7 exhibits a gradual shading from white (high intensity) on the left to black (low intensity) on the right. An image produced using this technique is called a halftone. In practice, newspaper halftones use 60 to 80 dots per inch, whereas book and magazine halftones use 120 to 200 dots per inch.

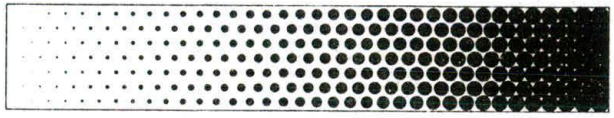


Fig. 2-7 A halftone stripe.

Halftone Approximation

Instead of changing dot size we can approximate the halftone technique using pixel-grid patterns. For example, with a 2×2 bilevel pixel grid we can construct five grid patterns to produce five overall intensity levels (see Fig. 2-8). We can increase the number of overall intensity levels by increasing the size of the pixel grid (see the following paragraphs for an example). On the other hand, if the pixels can be set to multiple intensity levels, even a 2×2 grid can produce a relatively high number of overall intensity levels. For example, if the pixels can be intensified to four different levels, we can follow the pattern sequence in Fig. 2-8 to bring each pixel from one intensity level to the next to approximate a total of thirteen overall intensity levels (one for all pixels off and four for each of the three non-zero intensities, see Fig. 2-9).

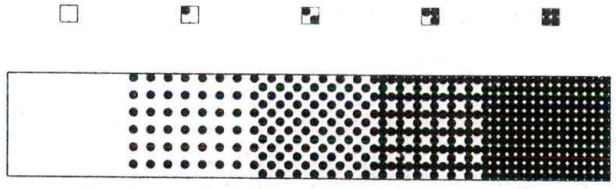


Fig. 2-8 Halftone approximation.

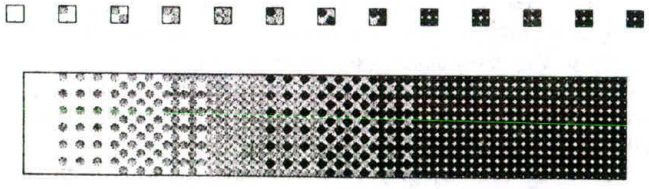


Fig. 2-9 Halftone approximation with 13 intensity levels.

These halftone grid patterns are sometimes referred to as dither patterns. There are several considerations in the design of dither patterns. First, the pixels should be intensified in a growth-from-the-grid-center fashion in order to mimic the growth of dot size. Second, a pixel that is intensified to a certain level to approximate a particular overall intensity should remain at least at that level for all subsequent overall intensity levels. In other words, the patterns should evolve from one to the next in order to minimize the differences in the patterns for successive overall intensity levels. Third, symmetry should be avoided in order to minimize visual artifacts such as streaks that would show up in image areas of

uniform intensity. Fourth, isolated "on" pixels should be avoided since they are sometimes hard to reproduce.

We can use a dither matrix to represent a series of dither patterns. For example, the following 3×3 matrix:

$$\begin{pmatrix} 5 & 2 & 7 \\ 1 & 0 & 3 \\ 6 & 8 & 4 \end{pmatrix}$$

represents the order in which pixels in a 3×3 grid are to be intensified. For bilevel reproduction, this gives us ten intensity levels from level 0 to level 9, and intensity level I is achieved by turning on all pixels that correspond to values in the dither matrix that are less than I. If each pixel can be intensified to three different levels, we follow the order defined by the matrix to set the pixels to their middle intensity level and then to their high intensity level to approximate a total of nineteen overall intensity levels.

This halftone approximation technique is readily applicable to the reproduction of color images. All we need is to replace the dot at each pixel position with an RGB or CMY dot pattern (e.g., the triad pattern shown in Fig. 2-6). If we use a 2×2 pixel grid and each primary or its complement can take on two intensity levels, we achieve a total of $5 \times 5 \times 5 = 125$ color combinations.

At this point we can turn to the fact that halftone approximation is a technique that trades spatial resolution for more colors/intensity levels. For a device that is capable of producing images at a resolution of 400×400 pixels per inch, halftone approximation using 2×2 dither patterns would mean lowering its resolution effectively to 200×200 pixels per inch.

Dithering

A technique called dithering can be used to approximate halftones without reducing spatial resolution. In this approach the dither matrix is treated very much like a floor tile that can be repeatedly positioned one copy next to another to cover the entire floor, i.e., the image. A pixel at (x, y) is intensified if the intensity level of the image at that position is greater than the corresponding value in the dither matrix. Mathematically, if D_n stands for an $n \times n$ dither matrix, the element $D_n(i, j)$ that corresponds to pixel position (x, y) can be found by $i = x \bmod n$ and $j = y \bmod n$. For example, if we use the 3×3 matrix given earlier for a bilevel reproduction and the pixel of the image at position (2, 19) has intensity level 5, then the corresponding matrix element is $D_3(2, 1) = 3$, and hence a dot should be printed or displayed at that location.

It should be noted that, for image areas that have constant intensity, the results of dithering are exactly the same as the results of halftone approximation. Reproduction differences between these two methods occur only when intensity varies.

Error Diffusion

Another technique for continuous-tone reproduction without sacrificing spatial resolution is called the Floyd–Steinberg error diffusion. Here a pixel is printed using the closest intensity the device can deliver. The error term, i.e., the difference between the exact pixel value and the approximated value in the reproduction, is then propagated to several yet-to-be-processed neighboring pixels for compensation. More specifically, let S be the source image that is processed in a left-to-right and top-to-bottom pixel order, $S(x, y)$ be the pixel value at location (x, y), and e be $S(x, y)$ minus the approximated value. We update the value of the pixel's four neighbors (one to its right and three in the next scan line) as follows:

$$\begin{aligned} S(x+1, y) &= S(x+1, y) + ae \\ S(x-1, y-1) &= S(x-1, y-1) + be \\ S(x, y-1) &= S(x, y-1) + ce \\ S(x+1, y-1) &= S(x+1, y-1) + de \end{aligned}$$

where parameters a through d often take values $\frac{7}{16}$, $\frac{3}{16}$, $\frac{5}{16}$, and $\frac{1}{16}$, respectively. These modifications are for the purpose of using the neighboring pixels to offset the reproduction error at the current pixel location. They are not permanent changes made to the original image.

Consider, for example, the reproduction of a gray scale image (0: black, 255: white) on a bilevel device (level 0: black, level 1: white), if a pixel whose current value is 96 has just been mapped to level 0, we have $e = 96$ for this pixel location. The value of the pixel to its right is now increased by $96 \times \frac{7}{16} = 42$ in order to determine the appropriate reproduction level. This increment tends to cause such neighboring pixel to be reproduced at a higher intensity level, partially compensating the discrepancy brought on by mapping value 96 to level 0 (which is lower than the actual pixel value) at the current location. The other three neighboring pixels (one below and to the left, one immediately below, and one below and to the right) receive 18, 30, and 6 as their share of the reproduction error at the current location, respectively.

Results produced by this error diffusion algorithm are generally satisfactory, with occasional introduction of slight echoing of certain image parts. Improved performance can sometimes be obtained by alternating scanning direction between left-to-right and right-to-left (minor modifications need to be made to the above formulas).

2.6 IMAGE FILES

A digital image is often encoded in the form of a binary file for the purpose of storage and transmission. Among the numerous encoding formats are BMP (Windows Bitmap), JPEG (Joint Photographic Experts Group File Interchange Format), and TIFF (Tagged Image File Format). Although these formats differ in technical details, they share structural similarities.

Figure 2-10 shows the typical organization of information encoded in an image file. The file consists largely of two parts: header and image data. In the beginning of the file header a binary code or ASCII string identifies the format being used, possibly along with the version number. The width and height of the image are given in numbers of pixels. Common image types include black and white (1 bit per pixel), 8-bit gray scale (256 levels along the gray axis), 8-bit color (lookup table), and 24-bit color. Image data format specifies the order in which pixel values are stored in the image data section. A commonly used order is left to right and top to bottom. Another possible order is left to right and bottom to top. Image data format also specifies if the RGB values in the color map or in the image are interlaced. When the values are given in an interlaced fashion, the three primary color components for a particular lookup table entry or a particular pixel stay together consecutively, followed by the three color components for the next entry or pixel. Thus the color values in the image data section are a sequence of red, green, blue, red, green, blue, etc. When the values are given in a non-interlaced fashion, the values of one primary for all table entries or pixels appear

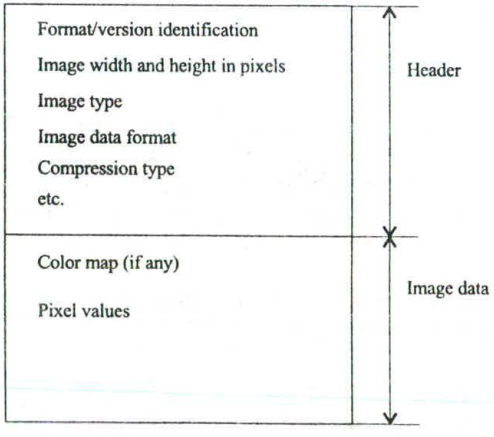


Fig. 2-10 Typical image file format.

first, then the values of another primary, followed by the values of the third primary. Thus the image data are in the form of red, red, . . . , green, green, . . . , blue, blue, ...

The values in the image data section may be compressed, using such compression algorithms as run-length encoding (RLE). The basic idea behind RLE can be illustrated with a character string "xxxxxxyyzzzz", which takes 12 bytes of storage. Now if we scan the string from left to right for segments of repeating characters and replace each segment by a 1-byte repeat count followed by the character being repeated, we convert or compress the given string to "6x2y4z", which takes only 6 bytes. This compressed version can be expanded or decompressed by repeating the character following each repeat count to recover the original string.

The length of the file header is often fixed, for otherwise it would be necessary to include length information in the header to indicate where image data starts (some formats include header length anyway). The length of each individual component in the image data section is, on the other hand, dependent on such factors as image type and compression type. Such information, along with additional format-specific information, can also be found in the header.

2.7 SETTING THE COLOR ATTRIBUTES OF PIXELS

Setting the color attributes of individual pixels is arguably the most primitive graphics operation. It is typically done by making system library calls to write the respective values into the frame buffer. An aggregate data structure, such as a three-element array, is often used to represent the three primary color components. Regardless of image type (direct coding versus lookup table), there are two possible protocols for the specification of pixel coordinates and color values.

In one protocol the application provides both coordinate information and color information simultaneously. Thus a call to set the pixel at location (x, y) in a 24-bit image to color (r, g, b) would look like

setPixel(x, y, rgb)

where *rgb* is a three-element array with $rgb[0] = r$, $rgb[1] = g$, and $rgb[2] = b$. On the other hand, if the image uses a lookup table then, assuming that the color is defined in the table, the call would look like

setPixel(x, y, i)

where i is the address of the entry containing (r, g, b).

Another protocol is based on the existence of a current color, which is maintained by the system and can be set by calls that look like

setColor(*rgb*)

for direct coding, or

setColor(*i*)

for the lookup table representation. Calls to set pixels now need only to provide coordinate information and would look like

setPixel(x, y)

for both image types. The graphics system will automatically use the most recently specified current color to carry out the operation.

Lookup table entries can be set from the application by a call that looks like

setEntry(*i*, *rgb*)

which puts color (r, g, b) in the entry whose address is i. Conversely, values in the lookup table can be read back to the application with a call that looks like

getEntry(*i*, *rgb*)

which returns the color value in entry i through array parameter *rgb*.

There are sometimes two versions of the calls that specify RGB values. One takes RGB values as floating point numbers in the range of [0.0, 1.0], whereas the other takes them as integers in the range of [0, 255]. Although the floating point version is handy when the color values come from some continuous formula, the floating point values are mapped by the graphics system into integer values before being written into the frame buffer.

In order to provide basic support for pixel-based image-processing operations there are calls that look like

getPixel(x, y, rgb)

for direct coding or

getPixel(x, y, i)

for the lookup table representation to return the color or index value of the pixel at (x, y) back to the application.

There are also calls that read and write rectangular blocks of pixels. A useful example would be a call to set all pixels to a certain background color. Assuming that the system uses a current color we would first set the current color to be the desired background color, and then make a call that looks like:

clear()

to achieve the goal.

2.8 EXAMPLE: VISUALIZING THE MANDELBROT SET

An elegant and illustrative example showing the construction of beautiful images by setting the color attributes of individual pixels directly from the application is the visualization of the Mandelbrot set. This remarkable set is based on the following transformation:

$$x_{n+1} = x_n^2 + z$$

where both x and z represent complex numbers. For readers who are unfamiliar with complex numbers it suffices to know that a complex number is defined in the form of $a + bi$. Here both a and b are real numbers; a is called the real part of the complex number and b the imaginary part (identified by the special symbol i). The magnitude of $a + bi$, denoted by $|a + bi|$, is equal to the square root of $a^2 + b^2$. The sum of two complex numbers $a + bi$ and $c + di$ is defined to be $(a + c) + (b + d)i$. The product of $a + bi$ and $c + di$ is defined to be $(ac - bd) + (ad + bc)i$. Thus the square of $a + bi$ is equal to $(a^2 - b^2) + 2abi$. For example, the sum of $0.5 + 2.0i$ and $1.0 - 1.0i$ is $1.5 + 1.0i$. The product of the two is $2.5 + 1.5i$. The square of $0.5 + 2.0i$ is $-3.75 + 2.0i$ and the square of $1.0 - 1.0i$ is $0.0 - 2.0i$.

The Mandelbrot set is the set of complex numbers z that do not diverge under the above transformation with $x_0 = 0$ (both the real and imaginary parts of x_0 are 0). In other words, to determine if a particular complex number z is a member of the set, we begin with $x_0 = 0$, followed by $x_1 = x_0^2 + z$, $x_2 = x_1^2 + z, \ldots, x_{n+1} = x_n^2 + z, \ldots$. If $|x|$ goes towards infinity when n increases, then z is not a member. Otherwise, z belongs to the Mandelbrot set.

Figure 2-11 shows how to produce a discrete snapshot of the Mandelbrot set. On the left hand side is the complex plane where the horizontal axis Re measures the real part of complex numbers and the vertical axis Im measures the imaginary part. Hence an arbitrary complex number z corresponds to a point in the complex plane. Our goal is to produce an image of width by height (in numbers of pixels) that depicts the z values in a rectangular area defined by (Re_min, Im_min) and (Re_max, Im_max). This rectangular area has the same aspect ratio as the image so as not to introduce geometric distortion. We subdivide the area to match the pixel grid in the image. The color of a pixel, shown as a little square in the pixel grid, is determined by the complex number z that corresponds to the lower left corner of the little square. Although only width × height points in the complex plane are used to compute the image, this relatively

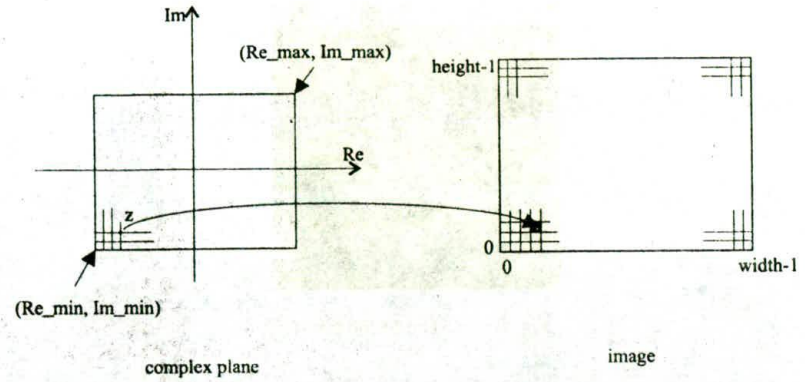


Fig. 2-11 Visualizing the Mandelbrot set.

straightforward approach to discrete sampling produces reasonably good approximations for the purpose of visualizing the set.

There are many ways to decide the color of a pixel based on the corresponding complex number z. What we do here is to produce a gray scale image where the gray level of a non-black pixel represents proportionally the number of iterations it takes for $|x|$ to be greater than 2. We use 2 as a threshold for divergence because x diverges quickly under the given transformation once $|x|$ becomes greater than 2. If $|x|$ remains less than or equal to 2 after a preset maximum number of iterations, we simply set the pixel value to 0 (black).

The following pseudo-code implements what we have discussed in the above paragraphs. We use N to represent the maximum number of iterations, z.real the real part of z, and z.imag the imaginary part of z. We also assume a 256-entry gray scale lookup table where the color value in entry i is (i, i, i). The formula in the second call to setColor is to obtain a proportional mapping from $[0, N]$ to $[1, 255]$:

```
int i, j, count;
float delta = (Re_max − Re_min)/width;

for (i = 0, z.real = Re_min; i < width; i ++ , z.real+ = delta)
  for (j = 0, z.imag = Im_min; j < height; j++, z.imag+ = delta) {
    count = 0;
    complex number x = 0;
    while (|x| <= 2.0 && count < N) {
      compute x = x^2 + z;
      count++;
    }
    if (|x| <= 2.0) setColor(0);
    else setColor(1 + 254*count/N);
    setPixel(i, j);
  }
```

The image in Fig. 2-12 shows what is nicknamed the Mandelbrot bug. It visualizes an area where $-2.0 \le z.\text{real} \le 0.5$ and $-1.25 \le z.\text{imag} \le 1.25$ with $N = 64$. Most z values that are outside the area lead x to diverge quickly, whereas the z values in the black region belong to the Mandelbrot set. It is along the contour of the bug-like figure that we see the most dynamic alterations between divergence and non-divergence, together with the most significant variations in the number of iterations used in the divergence test. The brighter a pixel, the longer it took to conclude divergence for the corresponding z. In principle the rectangular area can be reduced indefinitely to zoom in on any active region to show more intricate details.

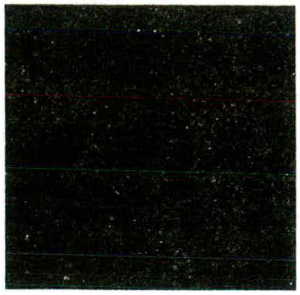


Fig. 2-12 The Mandelbrot set

Julia Sets

Now if we set z to some fixed non-zero value and vary x_0 across the complex plane, we obtain a set of non-divergence numbers (values of x_0 that do not diverge under the given transformation) that form a Julia set. Different z values lead to different Julia sets. The image in Fig. 2-13 is produced by making slight modifications to the pseudo-code for the Mandelbrot set. It shows the Julia set defined by $z = -0.74543 + 0.11301i$ with $-1.2 \leq x_0.\text{real} \leq 1.2$, $-1.2 \leq x_0.\text{imag} \leq 1.2$, and $N = 128$.

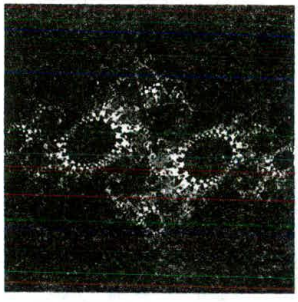


Fig. 2-13 A Julia set.

Solved Problems

2.1 What is the resolution of an image?

SOLUTION

The number of pixels (i.e., picture elements) per unit length (e.g., inch) in the horizontal as well as vertical direction.

2.2 Compute the size of a 640×480 image at 240 pixels per inch.

SOLUTION

640/240 by 480/240 or $2\frac{2}{3}$ by 2 inches.

2.3 Compute the resolution of a 2×2 inch image that has 512×512 pixels.

SOLUTION

512/2 or 256 pixels per inch.

2.4 What is an image's aspect ratio?

SOLUTION

The ratio of its width to its height, measured in unit length or number of pixels.

2.5 If an image has a height of 2 inches and an aspect ratio of 1.5, what is its width?

SOLUTION

width $= 1.5 \times$ height $= 1.5 \times 2 = 3$ inches.

2.6 If we want to resize a 1024×768 image to one that is 640 pixels wide with the same aspect ratio, what would be the height of the resized image?

SOLUTION

height $= 640 \times 768/1024 = 480$.

2.7 If we want to cut a 512×512 sub-image out from the center of an 800×600 image, what are the coordinates of the pixel in the large image that is at the lower left corner of the small image?

SOLUTION

$[(800 - 512)/2, (600 - 512)/2] = (144, 44)$.

2.8 Sometimes the pixel at the upper left corner of an image is considered to be at the origin of the pixel coordinate system (a left-handed system). How to convert the coordinates of a pixel at (x, y) in this coordinate system into its coordinates (x', y') in the lower-left-corner-as-origin coordinate system (a right-handed system)?

SOLUTION

$(x', y') = (x, m - y - 1)$ where m is the number of pixels in the vertical direction.

2.9 Find the CMY coordinates of a color at $(0.2, 1, 0.5)$ in the RGB space.

SOLUTION

$(1 - 0.2, 1 - 1, 1 - 0.5) = (0.8, 0, 0.5)$.

2.10 Find the RGB coordinates of a color at $(0.15, 0.75, 0)$ in the CMY space.

SOLUTION

$(1 - 0.15, 1 - 0.75, 1 - 0) = (0.85, 0.25, 1)$.

2.11 If we use direct coding of RGB values with 2 bits per primary color, how many possible colors do we have for each pixel?

SOLUTION

$2^2 \times 2^2 \times 2^2 = 4 \times 4 \times 4 = 64$.

2.12 If we use direct coding of RGB values with 10 bits per primary color, how many possible colors do we have for each pixel?

SOLUTION

$2^{10} \times 2^{10} \times 2^{10} = 1024^3 = 1073{,}741{,}824 > 1$ billion.

2.13 The direct coding method is flexible in that it allows the allocation of a different number of bits to each primary color. If we use 5 bits each for red and blue and 6 bits for green for a total of 16 bits per pixel, how many possible simultaneous colors do we have?

SOLUTION

$2^5 \times 2^5 \times 2^6 = 2^{16} = 65{,}536$.

2.14 If we use 12-bit pixel values in a lookup table representation, how many entries does the lookup table have?

SOLUTION

$2^{12} = 4096$.

2.15 If we use 2-byte pixel values in a 24-bit lookup table representation, how many bytes does the lookup table occupy?

SOLUTION

$2^{16} \times 24/8 = 65{,}536 \times 3 = 196{,}608$.

2.16 True or false: fluorescence is the term used to describe the light given off by a phosphor after it has been exposed to an electron beam. Explain your answer.

SOLUTION

False. Phosphorescence is the correct term. Fluorescence refers to the light given off by a phosphor while it is being exposed to an electron beam.

2.17 What is persistence?

SOLUTION

The duration of phosphorescence exhibited by a phosphor.

2.18 What is the function of the control electrode in a CRT?

SOLUTION

Regulate the intensity of the electron beam.

2.19 Name the two methods by which an electron beam can be bent?

SOLUTION

Electrostatic deflection and magnetic deflection.

2.20 What do you call the path the electron beam takes when returning to the left side of the CRT screen?

SOLUTION

Horizontal retrace.

2.21 What do you call the path the electron beam takes at the end of each refresh cycle?

SOLUTION

Vertical retrace.

2.22 What is the pitch of a color CRT?

SOLUTION

The distance between the center of the phosphor dot patterns on the inside of the display screen.

2.23 Why do many color printers use black pigment?

SOLUTION

Color pigments (cyan, magenta, and yellow) are relatively more expensive and it is technically difficult to produce high-quality black using several color pigments.

2.24 Show that with an $n \times n$ pixel grid, where each pixel can take on m intensity levels, we can approximate $n \times n \times (m-1)+1$ overall intensity levels.

SOLUTION

Since the $n \times n$ pixels can be set to a non-zero intensity value one after another to produce $n \times n$ overall intensity levels, and there are $m-1$ non-zero intensity levels for the individual pixels, we can approximate a total of $n \times n \times (m-1)$ non-zero overall intensity levels. Finally we need to add one more overall intensity level that corresponds to zero intensity (all pixels off).

2.25 Represent the grid patterns in Fig. 2-8 with a dither matrix.

SOLUTION

$$\begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}$$

2.26 What are the error propagation formulas for a top-to-bottom and right-to-left scanning order in the Floyd–Steinberg error diffusion algorithm?

SOLUTION

$$\begin{aligned} S(x-1,y) &= S(x-1,y)+ae \\ S(x+1,y-1) &= S(x+1,y-1)+be \\ S(x,y-1) &= S(x,y-1)+ce \\ S(x-1,y-1) &= S(x-1,y-1)+de \end{aligned}$$

2.27 What is RLE?

SOLUTION

RLE stands for run-length encoding, a technique for image data compression.

2.28 Follow the illustrative example in the text to reconstruct the string that has been compressed to "981435" using RLE.

SOLUTION

"8888888884555"

2.29 If an 8-bit gray scale image is stored uncompressed in sequential memory or in an image file in left-to-right and bottom-to-top pixel order, what is the offset or displacement of the byte for the pixel at (x, y) from the beginning of the memory segment or the file's image data section?

SOLUTION

offset $= y \times n + x$ where n is the number of pixels in the horizontal direction.

2.30 What if the image in Prob. 2.29 is stored in left-to-right and top-to-bottom order?

SOLUTION

offset $= (m - y - 1)n + x$ where n and m are the number of pixels in the horizontal and vertical direction, respectively.

2.31 Develop a pseudo-code segment to initialize a 24-bit 256-entry lookup table with gray-scale values.

SOLUTION

```
int i, rgb[3];
for (i = 0; i < 256; i++) {
  rgb[0] = rgb[1] = rgb[2] = i;
  setEntry(i, rgb);
}
```

2.32 Develop a pseudo-code segment to swap the red and green components of all colors in a 256-entry lookup table.

SOLUTION

```
int i, x, rgb[3];
for (i = 0; i < 256; i++) {
  getEntry(i, rgb);
  x = rgb[0];
  rgb[0] = rgb[1];
  rgb[1] = x;
  setEntry(i, rgb);
}
```

2.33 Develop a pseudo-code segment to draw a rectangular area of $w \times h$ (in number of pixels) that starts at (x, y) using color *rgb*.

SOLUTION

```
int i, j;
setColor(rgb);
for (j = y; j < y + h; j++)
  for (i = x; i < x + w; i++) setPixel(i, j);
```

2.34 Develop a pseudo-code segment to draw a triangular area with the three vertices at (x, y), $(x, y + t)$, and $(x + t, y)$, where integer $t \geq 0$, using color *rgb*.

SOLUTION

```
int i, j;
setColor(rgb);
for (j = y; j <= y + t; j++)
  for (i = x; i <= x + y + t - j; i++) setPixel(i, j);
```

2.35 Develop a pseudo-code segment to reset every pixel in an image that is in the 24-bit 256-entry lookup table representation to its complementary color.

SOLUTION

```
int i, rgb[3];
for (i = 0; i < 256; i++) {
  getEntry(i, rgb);
    rgb[0] = 255 - rgb[0];
    rgb[1] = 255 - rgb[1];
    rgb[2] = 255 - rgb[2];
    setEntry(i, rgb);
  }
```

2.36 What if the image in Prob. 2.35 is in the 24-bit true color representation?

SOLUTION

```
int i, j, rgb[3];
for (j = 0; j < height; j++)
  for (i = 0; i < width; i++) {
  getPixel(i, j, rgb);
    rgb[0] = 255 - rgb[0];
    rgb[1] = 255 - rgb[1];
    rgb[2] = 255 - rgb[2];
    setPixel(i, j, rgb);
  }
```

2.37 Calculate the sum and product of $0.5 + 2.0i$ and $1.0 - 1.0i$.

SOLUTION

$$(0.5 + 1.0) + (2.0 + (-1.0))i = 1.5 + 1.0i$$
$$(0.5 \times 1.0 - 2.0 \times (-1.0)) + (0.5 \times (-1.0) + 2.0 \times 1.0)i = 2.5 + 1.5i$$

2.38 Calculate the square of the two complex numbers in Prob. 2.37.

SOLUTION

$$(0.5^2 - 2.0^2) + 2 \times 0.5 \times 2.0i = -3.75 + 2.0i$$
$$(1.0^2 - (-1.0)^2) + 2 \times 1.0 \times (-1.0)i = 0.0 - 2.0i$$

2.39 Show that $1 + 254 \times \text{count}/N$ provides a proportional mapping from count in $[0, N]$ to c in $[1, 255]$.

SOLUTION

Proportional mapping means that we want

$$(c - 1)/(255 - 1) = (\text{count} - 0)/(N - 0)$$

Hence $c = 1 + 254 \times \text{count}/N$.

2.40 Modify the pseudo code for visualizing the Mandelbrot set to visualize the Julia sets.

SOLUTION

```
int i, j, count;
float delta = (Re_max – Re_min)/width;
for (i = 0, x.real = Re_min; i < width; i++, x.real+ = delta)
  for ( j = 0, x.imag = Im_min; j < height; j++, x.imag+ = delta) {
    count = 0;
    while (|x| ≤ 2.0 && count < N) {
      compute x = x^2 + z;
      count++;
    }
    if (|x| ≤ 2.0) setColor(0);
    else setColor(1 + 254*count/N);
    setPixel(i, j);
  }
```

2.41 How to avoid the calculation of square root in an actual implementation of the algorithms for visualizing the Mandelbrot and Julia sets?

SOLUTION

Test for $|x|^2 \leq 4.0$ instead of $|x| \leq 2.0$.

Supplementary Problems

2.42 Can a 5 by $3\frac{1}{2}$ inch image be presented at 6 by 4 inch without introducing geometric distortion?

2.43 Refering to Prob. 2.42, what if the original is $5\frac{1}{4}$ by $3\frac{1}{2}$ inch?

2.44 Given the portrait image of a person, describe a simple way to make the person look more slender.

2.45 An RGB color image can be converted to a gray-scale image using the formula $0.299R + 0.587G + 0.114B$ for gray levels (see Chap. 11, Sec. 11.1 under "The NTSC YIQ Color Model"). Assuming that getPixel(x, y, rgb) now reads pixel values from a 24-bit input image and setPixel(x, y, i) assigns pixel values to an output image that uses a gray-scale lookup table, develop a pseudo-code segment to convert the input image to a gray-scale output image.

CHAPTER 3

Scan Conversion

Many pictures, from 2D drawings to projected views of 3D objects, consist of graphical primitives such as points, lines, circles, and filled polygons. These picture components are often defined in a continuous space at a higher level of abstraction than individual pixels in the discrete image space. For instance, a line is defined by its two endpoints and the line equation, whereas a circle is defined by its radius, center position, and the circle equation. It is the responsibility of the graphics system or the application program to convert each primitive from its geometric definition into a set of pixels that make up the primitive in the image space. This conversion task is generally referred to as scan conversion or rasterization.

The focus of this chapter is on the mathematical and algorithmic aspects of scan conversion. We discuss ways to handle several commonly encountered primitives including points, lines, circles, ellipses, characters, and filled regions in an efficient and effective manner. We also discuss techniques that help to "smooth out" the discrepancies between the original element and its discrete approximation. The implementation of these algorithms and mathematical solutions (and many others in subsequent chapters) varies from one system to another and can be in the form of various combinations of hardware, firmware, and software.

3.1 SCAN-CONVERTING A POINT

A mathematical point (x, y) where x and y are real numbers within an image area, needs to be scan-converted to a pixel at location (x', y'). This may be done by making x' to be the integer part of x, and y' the integer part of y. In other words, $x' = \text{Floor}(x)$ and $y' = \text{Floor}(y)$, where function Floor returns the largest integer that is less than or equal to the argument. Doing so in essence places the origin of a continuous coordinate system for (x, y) at the lower left corner of the pixel grid in the image space [see Fig. 3-1(a)]. All points that satisfy $x' \leq x < x' + 1$ and $y' \leq y < y' + 1$ are mapped to pixel (x', y'). For example, point P_1 (1.7, 0.8) is represented by pixel (1, 0). Points P_2 (2.2, 1.3) and P_3 (2.8, 1.9) are both represented by pixel (2, 1).

Another approach is to align the integer values in the coordinate system for (x, y) with the pixel coordinates [see Fig. 3-1(b)]. Here we scan convert (x, y) by making $x' = \text{Floor}(x + 0.5)$ and $y' = \text{Floor}(y + 0.5)$. This essentially places the origin of the coordinate system for (x, y) at the center of pixel (0, 0). All points that satisfy $x' - 0.5 \leq x < x' + 0.5$ and $y' - 0.5 \leq y < y' + 0.5$ are mapped to pixel (x', y'). This means that points P_1 and P_2 are now both represented by pixel (2, 1), whereas point P_3 is represented by pixel (3, 2).

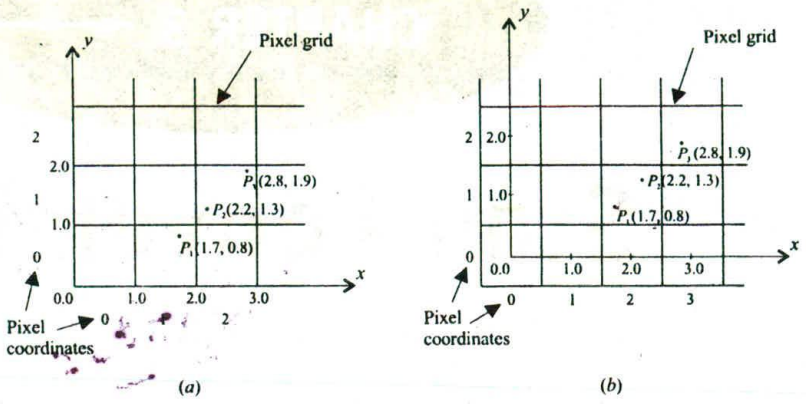


Fig. 3-1 Scan-converting points.

We will assume, in the following sections, that this second approach to coordinate system alignment is used. Thus all pixels are centered at the integer values of a continuous coordinate system where abstract graphical primitives are defined.

3.2 SCAN-CONVERTING A LINE

A line in computer graphics typically refers to a line segment, which is a portion of a straight line that extends indefinitely in opposite directions. It is defined by its two endpoints and the line equation $y = mx + b$, where m is called the slope and b the y intercept of the line. In Fig. 3-2 the two endpoints are described by $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$. The line equation describes the coordinates of all the points that lie between the two endpoints.

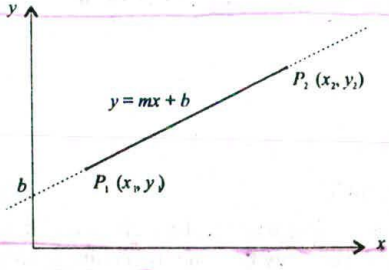


Fig. 3-2 Defining a line.

A note of caution: this slope–intercept equation is not suitable for vertical lines. Horizontal, vertical, and diagonal ($|m| = 1$) lines can, and often should, be handled as special cases without going through the following scan-conversion algorithms. These commonly used lines can be mapped to the image space in a straightforward fashion for high execution efficiency.

Direct Use of the Line Equation

A simple approach to scan-converting a line is to first scan-convert P_1 and P_2 to pixel coordinates (x_1', y_1') and (x_2', y_2'), respectively; then set $m = (y_2' - y_1')/(x_2' - x_1')$ and $b = y_1' - mx_1'$. If $|m| \leq 1$, then for

every integer value of x between and excluding x'_1 and x'_2, calculate the corresponding value of y using the equation and scan-convert (x, y). If $|m| > 1$, then for every integer value of y between and excluding y'_1 and y'_2, calculate the corresponding value of x using the equation and scan-convert (x, y).

While this approach is mathematically sound, it involves floating-point computation (multiplication and addition) in every step that uses the line equation since m and b are generally real numbers. The challenge is to find a way to achieve the same goal as quickly as possible.

DDA Algorithm

The digital differential analyzer (DDA) algorithm is an incremental scan-conversion method. Such an approach is characterized by performing calculations at each step using results from the preceding step. Suppose at step i we have calculated (x_i, y_i) to be a point on the line. Since the next point (x_{i+1}, y_{i+1}) should satisfy $\Delta y/\Delta x = m$ where $\Delta y = y_{i+1} - y_i$ and $\Delta x = x_{i+1} - x_i$, we have

$$y_{i+1} = y_i + m\Delta x$$

or

$$x_{i+1} = x_i + \Delta y/m$$

These formulas are used in the DDA algorithm as follows. When $|m| \leq 1$, we start with $x = x'_1$ (assuming that $x'_1 < x'_2$) and $y = y'_1$, and set $\Delta x = 1$ (i.e., unit increment in the x direction). The y coordinate of each successive point on the line is calculated using $y_{i+1} = y_i + m$. When $|m| > 1$, we start with $x = x'_1$ and $y = y'_1$ (assuming that $y'_1 < y'_2$), and set $\Delta y = 1$ (i.e., unit increment in the y direction). The x coordinate of each successive point on the line is calculated using $x_{i+1} = x_i + 1/m$. This process continues until x reaches x'_2 (for the $|m| \leq 1$ case) or y reaches y'_2 (for the $|m| > 1$ case) and all points found are scan-converted to pixel coordinates.

The DDA algorithm is faster than the direct use of the line equation since it calculates points on the line without any floating-point multiplication. However, a floating-point addition is still needed in determining each successive point. Furthermore, cumulative error due to limited precision in the floating-point representation may cause calculated points to drift away from their true position when the line is relatively long.

Bresenham's Line Algorithm

Bresenham's line algorithm is a highly efficient incremental method for scan-converting lines. It produces mathematically accurate results using only integer addition, subtraction, and multiplication by 2, which can be accomplished by a simple arithmetic shift operation.

The method works as follows. Assume that we want to scan-convert the line shown in Fig. 3-3 where $0 < m < 1$. We start with pixel $P'_1(x'_1, y'_1)$, then select subsequent pixels as we work our way to the right, one pixel position at a time in the horizontal direction towards $P'_2(x'_2, y'_2)$. Once a pixel is chosen at any step, the next pixel is either the one to its right (which constitutes a lower bound for the line) or the one to its right and up (which constitutes an upper bound for the line) due to the limit on m. The line is best approximated by those pixels that fall the least distance from its true path between P'_1 and P'_2.

Using the notation of Fig. 3-3, the coordinates of the last chosen pixel upon entering step i are (x_i, y_i). Our task is to choose the next one between the bottom pixel S and the top pixel T. If S is chosen, we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$. If T is chosen, we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i + 1$. The actual y coordinate of the line at $x = x_{i+1}$ is $y = mx_{i+1} + b = m(x_i + 1) + b$. The distance from S to the actual line in the y direction is $s = y - y_i$. The distance from T to the actual line in the y direction is $t = (y_i + 1) - y$.

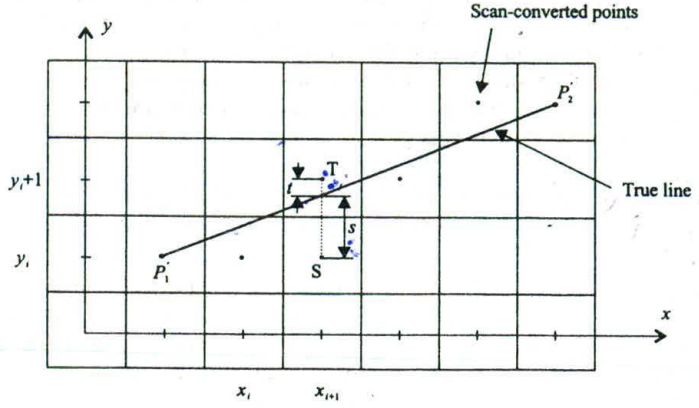


Fig. 3-3 Scan-converting a line.

Now consider the difference between these two distance values: $s - t$. When $s - t$ is less than zero, we have $s < t$ and the closest pixel is S. Conversely, when $s - t$ is greater than zero, we have $s > t$ and the closest pixel is T. We also choose T when $s - t$ is equal to zero. This difference is

$$\begin{aligned} s - t &= (y - y_i) - [(y_i + 1) - y] \\ &= 2y - 2y_i - 1 = 2m(x_i + 1) + 2b - 2y_i - 1 \end{aligned}$$

Substituting m by $\Delta y/\Delta x$ and introducing a decision variable $d_i = \Delta x(s - t)$, which has the same sign as $(s - t)$ since Δx is positive in our case, we have

$$d_i = 2\Delta y^* x_i - 2\Delta x^* y_i + C \qquad \text{where } C = 2\Delta y + \Delta x(2b - 1)$$

Similarly, we can write the decision variable d_{i+1} for the next step as

$$d_{i+1} = 2\Delta y^* x_{i+1} - 2\Delta x^* y_{i+1} + C$$

Then

$$d_{i+1} - d_i = 2\Delta y(x_{i+1} - x_i) - 2\Delta x(y_{i+1} - y_i)$$

Since $x_{i+1} = x_i + 1$, we have

$$d_{i+1} = d_i + 2\Delta y - 2\Delta x(y_{i+1} - y_i)$$

If the chosen pixel is the top pixel T (meaning that $d_i \geq 0$) then $y_{i+1} = y_i + 1$ and so

$$d_{i+1} = d_i + 2(\Delta y - \Delta x)$$

On the other hand, if the chosen pixel is the bottom pixel S (meaning that $d_i < 0$) then $y_{i+1} = y_i$ and so

$$d_{i+1} = d_i + 2\Delta y$$

Hence we have

$$d_{i+1} = \begin{cases} d_i + 2(\Delta y - \Delta x) & \text{if } d_i \geq 0 \\ d_i + 2\Delta y & \text{if } d_i < 0 \end{cases}$$

Finally, we calculate d_1, the base case value for this recursive formula, from the original definition of the decision variable d_i:

$$\begin{aligned} d_1 &= \Delta x[2m(x_1 + 1) + 2b - 2y_1 - 1] \\ &= \Delta x[2(mx_1 + b - y_1) + 2m - 1] \end{aligned}$$

Since $mx_1 + b - y_1 = 0$, we have

$$d_1 = 2\Delta y - \Delta x$$

In summary, Bresenham's algorithm for scan-converting a line from $P_1'(x_1', y_1')$ to $P_2'(x_2', y_2')$ with $x_1' < x_2'$ and $0 < m < 1$ can be stated as follows:

```
int x = x1', y = y1';
int dx = x2' - x1', dy = y2' - y1', dT = 2(dy - dx), dS = 2dy;
int d = 2dy - dx;
setPixel(x, y);
while (x < x2') {
  x++;
  if (d < 0)
    d = d + dS;
  else {
    y++;
    d = d + dT;
  }
  setPixel(x, y);
}
```

Here we first initialize decision variable d and set pixel P_1'. During each iteration of the while loop, we increment x to the next horizontal position, then use the current value of d to select the bottom or top (increment y) pixel and update d, and at the end set the chosen pixel.

As for lines that have other m values we can make use of the fact that they can be mirrored either horizontally, vertically, or diagonally into this 0° to 45° angle range. For example, a line from (x_1', y_1') to (x_2', y_2') with $-1 < m < 0$ has a horizontally mirrored counterpart from $(x_1', -y_1')$ to $(x_2', -y_2')$ with $0 < m < 1$. We can simply use the algorithm to scan-convert this counterpart, but negate the y coordinate at the end of each iteration to set the right pixel for the line. For a line whose slope is in the 45° to 90° range, we can obtain its mirrored counterpart by exchanging the x and y coordinates of its endpoints. We can then scan-convert this counterpart but we must exchange x and y in the call to setPixel.

3.3 SCAN-CONVERTING A CIRCLE

A circle is a symmetrical figure. Any circle-generating algorithm can take advantage of the circle's symmetry to plot eight points for each value that the algorithm calculates. Eight-way symmetry is used by reflecting each calculated point around each 45° axis. For example, if point 1 in Fig. 3-4 were calculated with a circle algorithm, seven more points could be found by reflection. The reflection is accomplished by reversing the x, y coordinates as in point 2, reversing the x, y coordinates and reflecting about the y axis as in point 3, reflecting about the y axis as in point 4, switching the signs of x and y as in point 5, reversing the x, y coordinates, reflecting about the y axis and reflecting about the x axis as in point 6, reversing the x, y coordinates and reflecting about the y axis as in point 7, and reflecting about the x axis as in point 8.

To summarize:

$$\begin{aligned} P_1 &= (x, y) & P_5 &= (-x, -y) \\ P_2 &= (y, x) & P_6 &= (-y, -x) \\ P_3 &= (-y, x) & P_7 &= (y, -x) \\ P_4 &= (-x, y) & P_8 &= (x, -y) \end{aligned}$$

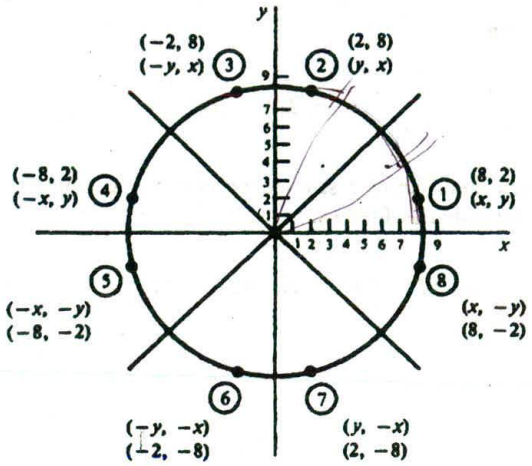


Fig. 3-4 Eight-way symmetry of a circle.

Defining a Circle

There are two standard methods of mathematically defining a circle centered at the origin. The first method defines a circle with the second-order polynomial equation (see Fig. 3-5)

$$y^2 = r^2 - x^3$$

where x = the x coordinate

y = the y coordinate

r = the circle radius

With this method, each x coordinate in the sector, from 90° to 45°, is found by stepping x from 0 to $r/\sqrt{2}$, and each y coordinate is found by evaluating $\sqrt{r^2 - x^2}$ for each step of x. This is a very inefficient method, however, because for each point both x and r must be squared and subtracted from each other; then the square root of the result must be found.

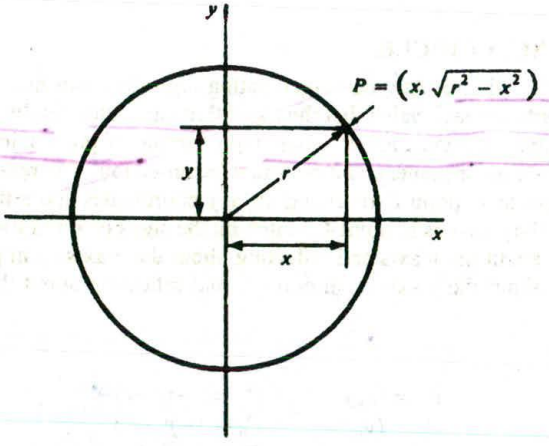


Fig. 3-5 Circle defined with a second-degree polynomial equation.

The second method of defining a circle makes use of trigonometric functios (see Fig. 3-6):

$$x = r\cos\theta \qquad y = r\sin\theta$$

where θ = current angle
r = circle radius
$x = x$ coordinate
$y = y$ coordinate

By this method, θ is stepped from θ to $\pi/4$, and each value of x and y is calculated. However, computation of the values of $\sin\theta$ and $\cos\theta$ is even more time-consuming than the calculations required by the first method.

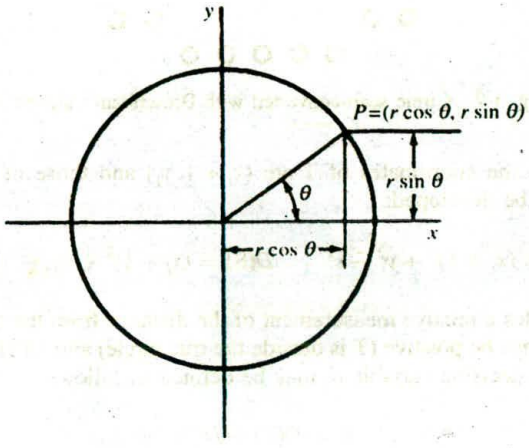


Fig. 3-6 Circle defined with trigonometric functions.

Bresenham's Circle Algorithm

If a circle is to be plotted efficiently, the use of trigonometric and power functions must be avoided. And, as with the generation of a straight line, it is also desirable to perform the calculations necessary to find the scan-converted points with only integer addition, subtraction, and multiplication by powers of 2. *Bresenham's circle algorithm* allows these goals to be met.

Scan-converting a circle using Bresenham's algorithm works as follows. If the eight-way symmetry of a circle is used to generate a circle, points will only have to be generated through a 45° angle. And, if points are generated from 90° to 45°, moves will be made only in the $+x$ and $-y$ directions (see Fig. 3-7).

The best approximation of the true circle will be described by those pixels in the raster that fall the least distance from the true circle. Examine Fig. 3-8. Notice that, if points are generated from 90° and 45°, each new point closest to the true circle can be found by taking either of two actions: (1) move in the x direction one unit or (2) move in the x direction one unit and move in the negative y direction one unit. Therefore, a method of selecting between these two choices is all that is necessary to find the points closest to the true circle.

Assume that (x_i, y_i) are the coordinates of the last scan-converted pixel upon entering step i (see Fig. 3-8). Let the distance from the origin to pixel T squared minus the distance to the true circle squared = $D(\mathrm{T})$. Then let the distance from the origin to pixel S squared minus the distance to the true

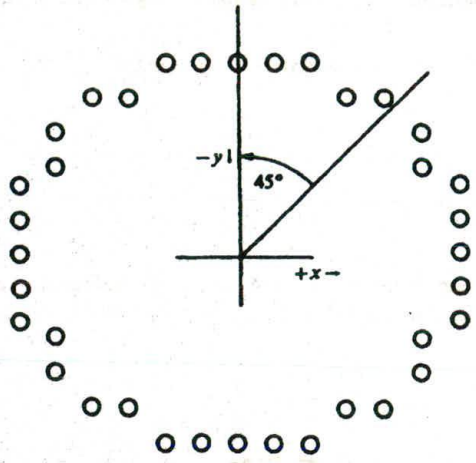


Fig. 3-7 Circle scan-converted with Bresenham's algorithm.

circle squared = $D(S)$. As the coordinates of T are $(x_i + 1, y_i)$ and those of S are $(x_i + 1, y_i - 1)$, the following expressions can be developed:

$$D(T) = (x_i + 1)^2 + y_i^2 - r^2 \qquad D(S) = (x_i + 1)^2 + (y_i - 1)^2 - r^2$$

This function D provides a relative measurement of the distance from the center of a pixel to the true circle. Since $D(T)$ will always be positive (T is outside the true circle) and $D(S)$ will always be negative (S is inside the true circle), a decision variable d_i may be defined as follows:

$$d_i = D(T) + D(S)$$

Therefore

$$d_i = 2(x_i + 1)^2 + y_i^2 + (y_i - 1)^2 - 2r^2$$

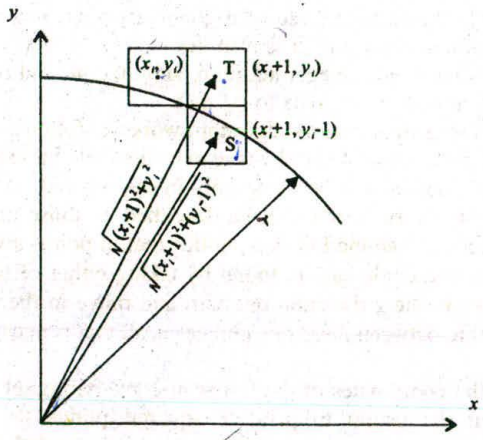


Fig. 3-8 Choosing pixels in Bresenham's circle algorithm.

When $d_i < 0$, we have $|D(T)| < |D(S)|$ and pixel T is chosen. When $d_i \geq 0$, we have $|D(T)| \geq |D(S)|$ and pixel S is selected. We can also write the decision variable d_{i+1} for the next step:

$$d_{i+1} = 2(x_{i+1} + 1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2r^2$$

Hence

$$d_{i+1} - d_i = 2(x_{i+1} + 1)^2 + y_{i+1}^2 + (y_{i+1} - 1)^2 - 2(x_i + 1)^2 - y_i^2 - (y_i - 1)^2$$

Since $x_{i+1} = x_i + 1$, we have

$$d_{i+1} = d_i + 4x_i + 2(y_{i+1}^2 - y_i^2) - 2(y_{i+1} - y_i) + 6$$

If T is the chosen pixel (meaning that $d_i < 0$) then $y_{i+1} = y_i$ and so

$$d_{i+1} = d_i + 4x_i + 6$$

On the other hand, if S is the chosen pixel (meaning that $d_i \geq 0$) then $y_{i+1} = y_i - 1$ and so

$$d_{i+1} = d_i + 4(x_i - y_i) + 10$$

Hence we have

$$d_{i+1} = \begin{cases} d_i + 4x_i + 6 & \text{if } d_i < 0 \\ d_i + 4(x_i - y_i) + 10 & \text{if } d_i \geq 0 \end{cases}$$

Finally, we set $(0, r)$ to be the starting pixel coordinates and compute the base case value d_1 for this recursive formula from the original definition of d_i:

$$d_1 = 2(0 + 1)^2 + r^2 + (r - 1)^2 - 2r^2 = 3 - 2r$$

We can now summarize the algorithm for generating all the pixel coordinates in the 90° to 45° octant that are needed when scan-converting a circle of radius r:

```
int x = 0, y = r, d = 3 - 2r;
while (x <= y) {
  setPixel(x, y);
  if (d < 0)
    d = d + 4x + 6;
  else {
    d = d + 4(x - y) + 10;
    y--;
  }
  x++;
}
```

Note that during each iteration of the while loop we first set a pixel whose position has already been determined, starting with $(0, r)$. We then test the current value of decision variable d in order to update d and determine the proper y coordinate of the next pixel. Finally we increment x.

Midpoint Circle Algorithm

We present another incremental circle algorithm that is very similar to Bresenham's approach. It is based on the following function for testing the spatial relationship between an arbitrary point (x, y) and a circle of radius r centered at the origin:

$$f(x, y) = x^2 + y^2 - r^2 \begin{cases} < 0 & (x, y) \text{ inside the circle} \\ = 0 & (x, y) \text{ on the circle} \\ > 0 & (x, y) \text{ outside the circle} \end{cases}$$

Now consider the coordinates of the point halfway between pixel T and pixel S in Fig. 3-8: $(x_i + 1, y_i - \frac{1}{2})$. This is called the midpoint and we use it to define a decision parameter:

$$p_i = f(x_i + 1, y_i - \tfrac{1}{2}) = (x_i + 1)^2 + (y_i - \tfrac{1}{2})^2 - r^2$$

If p_i is negative, the midpoint is inside the circle, and we choose pixel T. On the other hand, if p_i is positive (or equal to zero), the midpoint is outside the circle (or on the circle), and we choose pixel S. Similarly, the decision parameter for the next step is

$$p_{i+1} = (x_{i+1} + 1)^2 + (y_{i+1} - \tfrac{1}{2})^2 - r^2$$

Since $x_{i+1} = x_i + 1$, we have

$$p_{i+1} - p_i = [(x_i + 1) + 1]^2 - (x_i + 1)^2 + (y_{i+1} - \tfrac{1}{2})^2 - (y_i - \tfrac{1}{2})^2$$

Hence

$$p_{i+1} = p_i + 2(x_i + 1) + 1 + (y_{i+1}^2 - y_i^2) - (y_{i+1} - y_i)$$

If pixel T is chosen (meaning $p_i < 0$), we have $y_{i+1} = y_i$. On the other hand, if pixel S is chosen (meaning $p_i \geq 0$), we have $y_{i+1} = y_i - 1$. Thus

$$p_{i+1} = \begin{cases} p_i + 2(x_i + 1) + 1 & \text{if } p_i < 0 \\ p_i + 2(x_i + 1) + 1 - 2(y_i - 1) & \text{if } p_i \geq 0 \end{cases}$$

We can continue to simplify this in terms of (x_i, y_i) and get

$$p_{i+1} = \begin{cases} p_i + 2x_i + 3 & \text{if } p_i < 0 \\ p_i + 2(x_i - y_i) + 5 & \text{if } p_i \geq 0 \end{cases}$$

Or we can write it in terms of (x_{i+1}, y_{i+1}) and have

$$p_{i+1} = \begin{cases} p_i + 2x_{i+1} + 1 & \text{if } p_i < 0 \\ p_i + 2(x_{i+1} - y_{i+1}) + 1 & \text{if } p_i \geq 0 \end{cases}$$

Finally, we compute the initial value for the decision parameter using the original definition of p_i and $(0, r)$:

$$p_i = (0 + 1)^2 + (r - \tfrac{1}{2})^2 - r^2 = \tfrac{5}{4} - r$$

One can see that this is not really integer computation. However, when r is an integer we can simply set $p_1 = 1 - r$. The error of being $\frac{1}{4}$ less than the precise value does not prevent p_1 from getting the appropriate sign. It does not affect the rest of the scan-conversion process either, because the decision variable is only updated with integer increments in subsequent steps.

The following is a description of this midpoint circle algorithm that generates the pixel coordinates in the 90° to 45° octant:

```
int x = 0, y = r, p = 1 − r;
while (x <= y) {
  setPixel(x, y);
  if (p < 0)
    p = p + 2x + 3;
  else {
    p = p + 2(x − y) + 5;
    y−−;
  }
  x++;
}
```

Arbitrarily Centered Circles

In the above discussion of the circle algorithms we have assumed that a circle is centered at the origin. To scan-convert a circle centered at (x_c, y_c), we can simply replace the setPixel(x, y) statement in the algorithm description with setPixel$(x + x_c, y + y_c)$. The reason for this to work is that a circle centered at (x_c, y_c) can be viewed as a circle centered at the origin that is moved by x_c and y_c in the x and y direction, respectively. We can achieve the effect of scan-converting this arbitrarily centered circle by relocating scan-converted pixels in the same way as moving the circle's center from the origin.

3.4 SCAN-CONVERTING AN ELLIPSE

The ellipse, like the circle, shows symmetry. In the case of an ellipse, however, symmetry is four- rather than eight-way. There are two methods of mathematically defining a ellipse.

Polynomial Method of Defining an Ellipse

The polynomial method of defining an ellipse (Fig. 3-9) is given by the expression

$$\frac{(x-h)^2}{a^2} + \frac{(y-k)^2}{b^2} = 1$$

where (h, k) = ellipse center
a = length of major axis
b = length of minor axis

When the polynomial method is used to define an ellipse, the value of x is incremented from h to a. For each step of x, each value of y is found by evaluating the expression

$$y = b\sqrt{1 - \frac{(x-h)^2}{a^2}} + k$$

This method is very inefficient, however, because the squares of a and $(x - h)$ must be found; then floating-point division of $(x - h)^2$ by a^2 and floating-point multiplication of the square root of $[1 - (x - h)^2/a^2]$ by b must be performed (see Prob. 3.20).

Routines have been found that will scan-convert general polynomial equations, including the ellipse. However, these routines are logic intensive and thus are very slow methods for scan-converting ellipses.

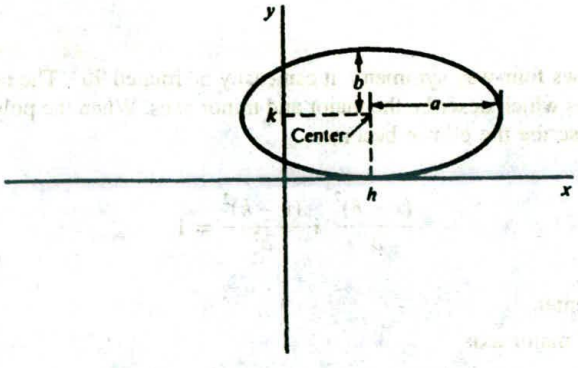


Fig. 3-9 Polynomial description of an ellipse.

Trigonometric Method of Defining an Ellipse

A second method of defining an ellipse makes use of trigonometric relationships (see Fig. 3-10). The following equations define an ellipse trigonometrically:

$$x = a\cos(\theta) + h \qquad \text{and} \qquad y = b\sin(\theta) + k$$

where (x, y) = the current coordinates
a = length of major axis
b = length of minor axis
θ = current angle
(h, k) = ellipse center

For generation of an ellipse using the trigonometric method, the value of θ is varied from 0 to $\pi/2$ radians (rad). The remaining points are found by symmetry. While this method is also inefficient and thus generally too slow for interactive applications, a lookup table containing the values for $\sin(\theta)$ and $\cos(\theta)$ with θ ranging from 0 to $\pi/2$ rad can be used. This method would have been considered unacceptable at one time because of the relatively high cost of the computer memory used to store the values of θ. However, because the cost of computer memory has plummeted in recent years, this method is now quite acceptable.

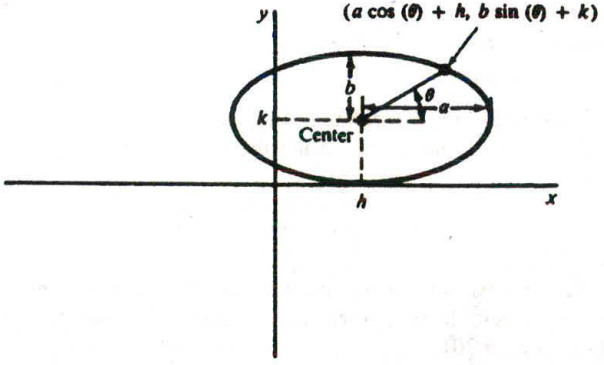


Fig. 3-10 Trigonometric description of an ellipse.

Ellipse Axis Rotation

Since the ellipse shows four-way symmetry, it can easily be rotated 90°. The new equation is found by trading a and b, the values which describe the major and minor axes. When the polynomial method is used, the equations used to describe the ellipse become

$$\frac{(x-h)^2}{b^2} + \frac{(y-k)^2}{a^2} = 1$$

where (h, k) = ellipse center
a = length of major axis
b = length of minor axis

When the trigonometric method is used, the equations used to describe the ellipse become

$$x = b\cos(\theta) + h \qquad \text{and} \qquad y = a\sin(\theta) + k$$

where (x, y) = current coordinates
a = length of major axis
b = length of minor axis
θ = current angle
(h, k) = ellipse center

Assume that you would like to rotate the ellipse through an angle other than 90°. It can be seen from Fig. 3-11 that rotation of the ellipse may be accomplished by rotating the x and y axis α degrees. When this is done, the equations describing the x, y coordinates of each scan-converted point become

$$x = a\cos(\theta) - b\sin(\theta + \alpha) + h \qquad y = b\sin(\theta) + a\cos(\theta + \alpha) + k$$

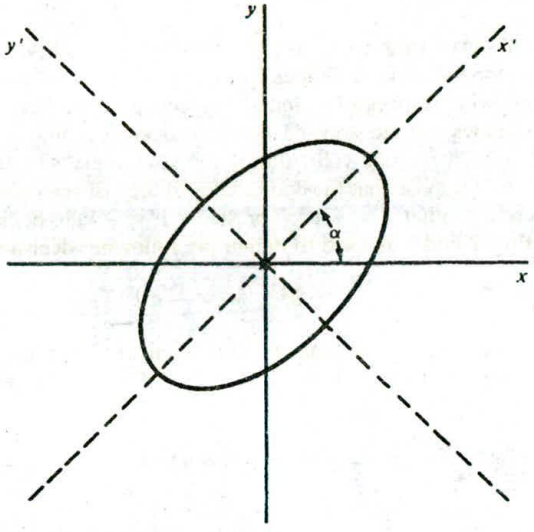


Fig. 3-11 Rotation of an ellipse.

Midpoint Ellipse Algorithm

This is an incremental method for scan-converting an ellipse that is centered at the origin in standard position (i.e., with its major and minor axes parallel to the coordinate system axes). It works in a way that is very similar to the midpoint circle algorithm. However, because of the four-way symmetry property we need to consider the entire elliptical curve in the first quadrant (see Fig. 3-12).

Let's first rewrite the ellipse equation and define function f that can be used to decide if the midpoint between two candidate pixels is inside or outside the ellipse:

$$f(x, y) = b^2x^2 + a^2y^2 - a^2b^2 \begin{cases} < 0 & (x, y) \text{ inside the ellipse} \\ = 0 & (x, y) \text{ on the ellipse} \\ > 0 & (x, y) \text{ outside the ellipse} \end{cases}$$

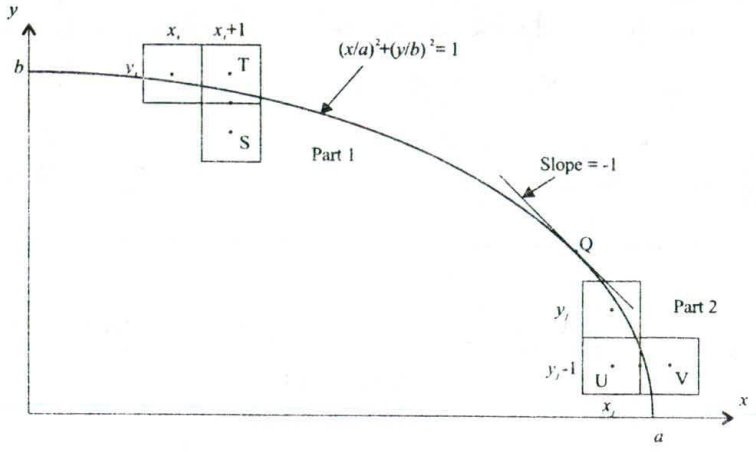


Fig. 3-12 Scan-converting an ellipse.

Now divide the elliptical curve from $(0, b)$ to $(a, 0)$ into two parts at point Q where the slope of the curve is -1. Recall that the slope of a curve defined by $f(x, y) = 0$ is $dy/dx = -fx/fy$, where fx and fy are partial derivatives of $f(x, y)$ with respect to x and y, respectively. We have $fx = 2b^2x$, $fy = 2a^2y$, and $dy/dx = -2b^2x/2a^2y$. This shows that the slope of the curve changes monotonically from one side of Q to the other. Hence we can monitor the slope value during the scan-conversion process to detect Q.

Our starting point is $(0, b)$. Suppose that the coordinates of the last scan-converted pixel upon entering step i are (x_i, y_i). We are to select either $T(x_i + 1, y_i)$ or $S(x_i + 1, y_i - 1)$ to be the next pixel. The midpoint of the vertical line connecting T and S is used to define the following decision parameter:

$$p_i = f(x_i + 1, y_i - \tfrac{1}{2}) = b^2(x_i + 1)^2 + a^2(y_i - \tfrac{1}{2})^2 - a^2b^2$$

If $p_i < 0$, the midpoint is inside the curve, and we choose pixel T. On the other hand, if $p_i \geq 0$, the midpoint is outside or on the curve, and we choose pixel S. Similarly, we can write the decision parameter for the next step:

$$p_{i+1} = f(x_{i+1} + 1, y_{i+1} - \tfrac{1}{2}) = b^2(x_{i+1} + 1)^2 + a^2(y_{i+1} - \tfrac{1}{2})^2 - a^2b^2$$

Since $x_{i+1} = x_i + 1$, we have

$$p_{i+1} - p_i = b^2[(x_{i+1} + 1)^2 - x_{i+1}^2] + a^2[(y_{i+1} - \tfrac{1}{2})^2 - (y_i - \tfrac{1}{2})^2]$$

Hence

$$p_{i+1} = p_i + 2b^2x_{i+1} + b^2 + a^2[(y_{i+1} - \tfrac{1}{2})^2 - (y_i - \tfrac{1}{2})^2]$$

If T is the chosen pixel (meaning $p_i < 0$), we have $y_{i+1} = y_i$. On the other hand, if pixel S is chosen (meaning $p_i \geq 0$), we have $y_{i+1} = y_i - 1$. Thus we can express p_{i+1} in terms of p_i and (x_{i+1}, y_{i+1}):

$$p_{i+1} = \begin{cases} p_i + 2b^2x_{i+1} + b^2 & \text{if } p_i < 0 \\ p_i + 2b^2x_{i+1} + b^2 - 2a^2y_{i+1} & \text{if } p_i \geq 0 \end{cases}$$

The initial value for this recursive expression can be obtained by evaluating the original definition of p_i with $(0, b)$:

$$p_1 = b^2 + a^2(b - \tfrac{1}{2})^2 - a^2b^2 = b^2 - a^2b + a^2/4$$

We now move on to derive a similar formula for part 2 of the curve. Suppose pixel (x_j, y_j) has just been scan-converted upon entering step j. The next pixel is either $U(x_j, y_j - 1)$ or $V(x_j + 1, y_j - 1)$. The midpoint of the horizontal line connecting U and V is used to define the decision parameter

$$q_j = f(x_j + \tfrac{1}{2}, y_j - 1) = b^2(x_j + \tfrac{1}{2})^2 + a^2(y_i - 1)^2 - a^2b^2$$

If $q_j < 0$, the midpoint is inside the curve and V is chosen. On the other hand, if $q_j \geq 0$, it is outside or on the curve and U is chosen. We also have

$$q_{j+1} = f(x_{j+1} + \tfrac{1}{2}, y_{j+1} - 1) = b^2(x_{j+1} + \tfrac{1}{2})^2 + a^2(y_{j+1} - 1)^2 - a^2b^2$$

Since $y_{j+1} = y_j - 1$, we have

$$q_{j+1} - q_j = b^2[(x_{j+1} + \tfrac{1}{2})^2 - (x_j + \tfrac{1}{2})^2] + a^2[(y_{j+1} - 1)^2 - y_{j+1}^2]$$

Hence

$$q_{j+1} = q_j + b^2[(x_{j+1} + \tfrac{1}{2})^2 - (x_j + \tfrac{1}{2})^2] - 2a^2y_{j+1} + a^2$$

If V is the chosen pixel (meaning $q_j < 0$), we have $x_{j+1} = x_j + 1$. On the other hand, if U is chosen (meaning $q_j \geq 0$), we have $x_{j+1} = x_j$. Thus we can express q_{j+1} in terms of q_j and (x_{j+1}, y_{j+1}):

$$q_{j+1} = \begin{cases} q_j + 2b^2x_{j+1} - 2a^2y_{j+1} + a^2 & \text{if } q_j < 0 \\ q_j - 2a^2y_{j+1} + a^2 & \text{if } q_j \geq 0 \end{cases}$$

The initial value for this recursive expression is computed using the original definition of q_j and the coordinates (x_k, y_k) of the last pixel chosen for part 1 of the curve:

$$q_1 = f(x_k + \tfrac{1}{2}, y_k - 1) = b^2(x_k + \tfrac{1}{2})^2 + a^2(y_k - 1)^2 - a^2b^2$$

We can now put together a pseudo-code description of the midpoint algorithm for scan-converting the elliptical curve in the first quadrant. There are a couple of technical details that are worth noting. First, we keep track of the slope of the curve by evaluating the partial derivatives *fx* and *fy* at each chosen pixel position. This means that $fx = 2b^2x_i$ and $fy = 2a^2y_i$ for position (x_i, y_i). Since we have $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i$ or $y_i - 1$ for part 1, and $x_{j+1} = x_j$ or $x_j + 1$ and $y_{j+1} = y_j - 1$ for part 2, the partial derivatives can be updated incrementally using $2b^2$ and/or $-2a^2$. For example, if $x_{i+1} = x_i + 1$ and $y_{i+1} = y_i - 1$, the partial derivatives for position (x_{i+1}, y_{i+1}) are $2b^2x_{i+1} = 2b^2x_i + 2b^2$ and $2a^2y_{i+1} = 2a^2y_i - 2a^2$. Second, since $2b^2x_{i+1}$ and $2a^2y_{i+1}$ appear in the recursive expression for the decision parameters, we can use them to efficiently compute p_{i+1} as well as q_{j+1}:

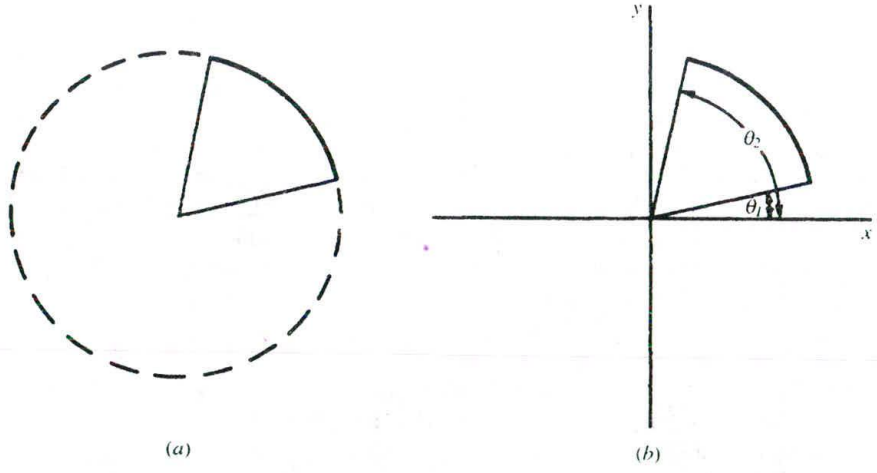
```
int x = 0, y = b;       /* starting point */
int aa = a*a, bb = b*b, aa2 = aa*2, bb2 = bb*2;
int fx = 0, fy = aa2*b;       /* initial partial derivatives */
int p = bb - aa*b + 0.25*aa;       /* compute and round off p1 */
while ( fx < fy) { /* |slope| < 1 */
  setPixel(x, y);
  x++;
  fx = fx + bb2;
  if (p < 0)
    p = p + fx + bb;
  else {
    y--;
    fy = fy - aa2;
    p = p + fx + bb - fy;
  }
}
setPixel(x, y);       /* set pixel at (xk, yk) */
```

```
p = bb(x + 0.5)(x + 0.5) + aa(y − 1)(y − 1) − aa*bb;      /* set q1 */
while (y > 0) {
  y−−;
  fy = fy − aa2;
  if (p >= 0)
    p = p − fy + aa;
  else {
    x++;
    fx = fx + bb2;
    p = p + fx − fy + aa;
  }
  setPixel(x, y);
}
```

3.5 SCAN-CONVERTING ARCS AND SECTORS

Arcs

An arc [Fig. 3-13(*a*)] may be generated by using either the polynomial or the trigonometric method. When the trigonometric method is used, the starting value is set equal to θ_1 and the ending value is set equal to θ_2 [see Figs. 3-13(*a*) and 3-13(*b*)]. The rest of the steps are similar to those used when scan-converting a circle, except that symmetry is not used.



(*a*) (*b*)

Fig. 3-13

When the polynomial method is used, the value of x is varied from x_1 to x_2 and the values of y are found by evaluating the expressio $\sqrt{r^2 - x^2}$ (Fig. 3-14).

From the graphics programmer's point of view, arcs would appear to be nothing more than portions of circles. However, problems occur if algorithms such as Bresenham's circle algorithm are used in drawing an arc. In the case of Bresenham's algorithm, the endpoints of an arc must be specified in terms of the x, y coordinates. The general formulation becomes inefficient when endpoints must be found (see Fig. 3-15). This occurs because the endpoints for each 45° increment of the arc must be found. Each of the eight points found by reflection must be tested to see if the point is between the specified endpoints of the arc. As a result, a routine to draw an arc based on Bresenham's algorithm must take the time to calculate and test

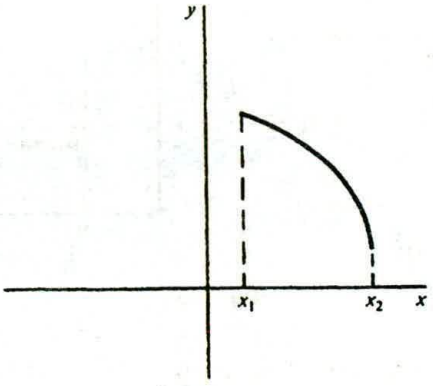


Fig. 3-14

every point on the circle's perimeter. There is always the danger that the endpoints will be missed when a method like this is used. If the endpoints are missed, the routine can become caught in an infinite loop.

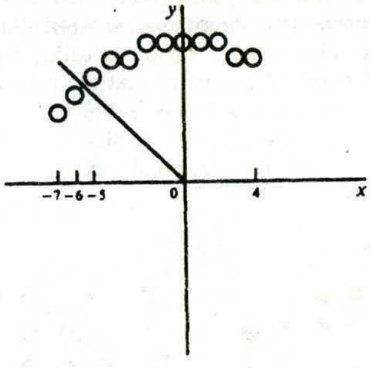


Fig. 3-15

Sectors

A sector is scan-converted by using any of the methods of scan-converting an arc and then scan-converting two lines from the center of the arc to the endpoints of the arc.

For example, assume that a sector whose center is at point (h, k) is to be scan-converted. First, scan-convert an arc from θ_1 to θ_2. Next, a line would be scan-converted from (h, k) to $(r\cos(\theta_1) + h, r\sin(\theta_1) + k)$. A second line would be scan-converted from (h, k) to $(r\cos(\theta_2) + h, r\sin(\theta_2) + k)$.

3.6 SCAN-CONVERTING A RECTANGLE

A rectangle whose sides are parallel to the coordinate axes may be constructed if the locations of two vertices are known [see Fig. 3-16(*a*)]. The remaining corner points are then derived [see Fig. 3-16(*b*)]. Once the vertices are known, the four sets of coordinates are sent to the line routine and the rectangle is

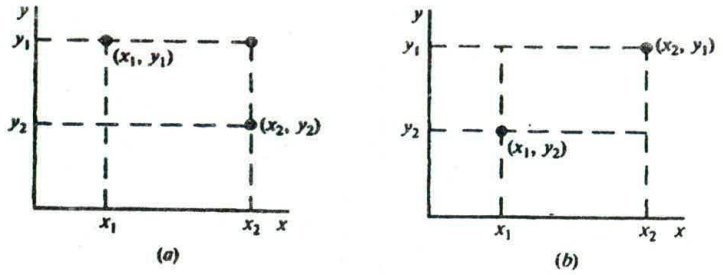


Fig. 3-16

scan-converted. In the case of the rectangle shown in Figs. 3-16(*a*) and 3-16(*b*), lines would be drawn as follows: line (x_1, y_1) to (x_1, y_2); line (x_1, y_2) to (x_2, y_2); line (x_2, y_2) to (x_2, y_1); and line (x_2, y_1) to (x_1, y_1).

3.7 REGION FILLING

Region filling is the process of "coloring in" a definite image area or region. Regions may be defined at the pixel or geometric level. At the pixel level, we describe a region either in terms of the bounding pixels that outline it or as the totality of pixels that comprise it (see Fig, 3-17). In the first case the region is called boundary-defined and the collection of algorithms used for filling such a region are collectively called boundary-fill algorithms. The other type of region is called an interior-defined region and the accompanying algorithms are called flood-fill algorithms. At the geometric level a region is defined or enclosed by such abstract contouring elements as connected lines and curves. For example, a polygonal region, or a filled polygon, is defined by a closed polyline, which is a polyline (i.e., a series of sequentially connected lines) that has the end of the last line connected to the beginning of the first line.

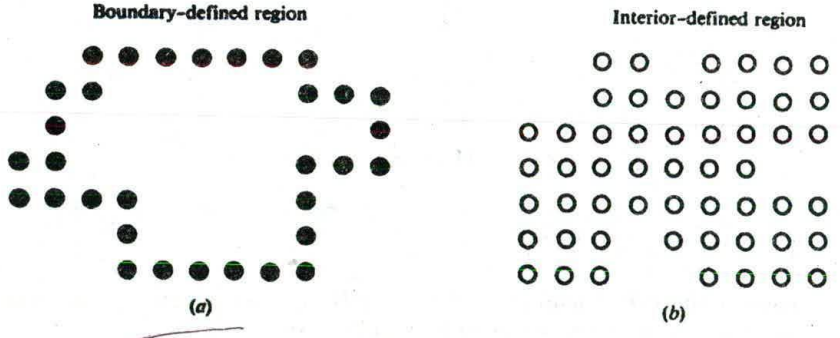


Fig. 3-17

4-Connected vs. 8-Connected

An interesting point here is that, while a geometrically defined contour clearly separates the interior of a region from the exterior, ambiguity may arise when an outline consists of discrete pixels in the image space. There are two ways in which pixels are considered connected to each other to form a "continuous" boundary. One method is called 4-connected, where a pixel may have up to four neighbors [see Fig. 3-18(*a*)]; the other is called 8-connected, where a pixel may have up to eight neighbors [see

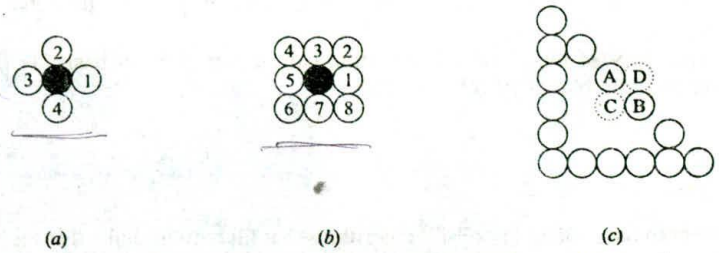


Fig. 3-18 4-connected vs. 8-connected pixels.

Fig. 3-18(*b*)]. Using the 4-connected approach, the pixels in Fig. 3-18(*c*) do not define a region since several pixels such as A and B are not connected. However using the 8-connected definition we identify a triangular region.

We can further apply the concept of connected pixels to decide if a region is connected to another region. For example, using the 8-connected approach, we do not have an enclosed region in Fig. 3-18(*c*) since "interior" pixel C is connected to "exterior" pixel D. On the other hand, if we use the 4-connected definition we have a triangular region since no interior pixel is connected to the outside.

Note that it is not a mere coincidence that the figure in Fig. 3-18(*c*) is a boundary-defined region when we use the 8-connected definition for the boundary pixels and the 4-connected definition for the interior pixels. In fact, using the same definition for both boundary and interior pixels would simply result in contradiction. For example, if we use the 8-connected approach we would have pixel A connected to pixel B (continuous boundary) and at the same time pixel C connected to pixel D (discontinuous boundary). On the other hand, if we use the 4-connectd definition we would have pixel A disconnected from pixel B (discontinuous boundary) and at the same time pixel C disconnected from pixel D (continuous boundary).

A Boundary-fill Algorithm

This is a recursive algorithm that begins with a starting pixel, called a seed, inside the region. The algorithm checks to see if this pixel is a boundary pixel or has already been filled. If the answer is no, it fills the pixel and makes a recursive call to itself using each and every neighboring pixel as a new seed. If the answer is yes, the algorithm simply returns to its caller.

This algorithm works elegantly on an arbitrarily shaped region by chasing and filling all non-boundary pixels that are connected to the seed, either directly or indirectly through a chain of neighboring relations. However, a straightforward implementation can take time and memory to execute due to the potentially high number of recursive calls, especially when the size of the region is relatively large.

Variations can be made to limit the number of recursive calls by structuring the order in which neighboring pixels are processed. For example, we can first fill pixels to the left and right of the seed on the same scan line until boundary pixels are hit (something that can be done using a loop control structure). We then inspect each pixel above and below the line just drawn (which can also be done with a loop) to see if it can be used as a new seed for the next horizontal line to fill. This way the number of recursive calls at any particular time is merely N when the current line is N scan lines away from the initial seed.

A Flood-fill Algorithm

This algorithm also begins with a seed (starting pixel) inside the region. It checks to see if the pixel has the region's original color. If the answer is yes, it fills the pixel with a new color and uses each of the pixel's neighbors as a new seed in a recursive call. If the answer is no, it returns to the caller.

This method shares great similarities in its operating principle with the boundary-fill algorithm. It is particularly useful when the region to be filled has no uniformly colored boundary. On the other hand, a

region that has a well-defined boundary but is itself multiply colored would be better handled by the boundary-fill method.

The execution efficiency of this flood-fill algorithm can be improved in basically the same way as discussed above regarding the boundary-fill algorithm.

A Scan-line Algorithm

In contrast to the boundary-fill and flood-fill algorithms that fill regions defined at the pixel level in the image space, this algorithm handles polygonal regions that are geometrically defined by the coordinates of their vertices (along with the edges that connect the vertices). Although such regions can be filled by first scan-converting the edges to get the boundary pixels and then applying a boundary-fill algorithm to finish the job, the following is a much more efficient approach that makes use of the information regarding edges that are available during scan conversion to facilitate the filling of interior pixels.

We represent a polygonal region in terms of a sequence of vertices $V_1, V_2, V_3, \ldots$, that are connected by edges $E_1, E_2, E_3, \ldots$, (see Fig. 3-19). We assume that each vertex V_i has already been scan-converted to integer coordinates (x_i, y_i).

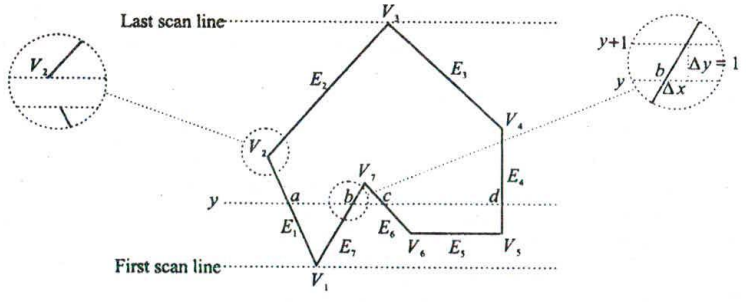


Fig. 3-19 Scan-converting a polygonal region.

The algorithm begins with the first scan line the polygon occupies and proceeds line by line towards the last scan line. For each scan line it finds all intersection points between the current scan line and the edges. For example, scan line y intersects edges E_1, E_7, E_6, and E_4 at points a, b, c, and b, respectively. The intersection points are sorted according to their x coordinates and grouped into pairs such as (a, b) and (c, d). A line is drawn from the first point to the second point in each pair.

Horizontal edges are ignored since the pixels that belong to them are automatically filled during scan conversion. For example, edge E_5 is drawn when the corresponding scan line is processed. The two intersection points between the scan line and edges E_6 and E_4 are connected by a line that equals exactly E_5.

Now we take a more careful look at cases when a scan line intersects a vertex. If the vertex is a local minimum or maximum such as V_1 and V_7, no special treatment is necessary. The two edges that join at the vertex will each yield an intersection point with the scan line. These two intersection points can be treated just like any other intersection points to produce pairs of points for the filling operation. As for vertices V_5 and V_6, they are simply local minimums, each with one joining edge that produces a single intersection point. On the other hand, if a scan line intersects a vertex (e.g., V_4) that is joined by two monotonically increasing or decreasing edges, getting two intersection points at that vertex location would lead to incorrect results (e.g., a total of three intersection points on the scan line that intersects V_4). The solution to this problem is to record only one intersection point at the vertex.

In order to support an efficient implementation of this scan line algorithm we create a data structure called an edge list (see Table 3-1). Each non-horizontal edge occupies one row/record. Information stored

Table 3-1 An edge list.

Edge	y_{min}	y_{max}	x coordinate of vertex with $y = y_{min}$	$1/m$
E_1	y_1	$y_2 - 1$	x_1	$1/m_1$
E_7	y_1	y_7	x_1	$1/m_7$
E_4	y_5	$y_4 - 1$	x_5	$1/m_4$
E_6	y_6	y_7	x_6	$1/m_6$
E_2	y_2	y_3	x_2	$1/m_2$
E_3	y_4	y_3	x_4	$1/m_3$

in each row includes the y coordinate of the edge's two endpoints, with y_{min} being the smaller value and y_{max} the larger value (may be decreased by 1 for reasons to be discussed below), the x coordinate of the endpoint whose y coordinate is y_{min}, and the inverse of the edge's slope m. The rows are sorted according to y_{min}. Going back to our example in Fig. 3-19, since edges E_1 and E_7 both originate from the lowest vertex V_1 at (x_1, y_1), they appear on top of the edge list, with m_1 and m_7 being their slope value, respectively.

Edges in the edge list become active when the y coordinate of the current scan line matches their y_{min} value. Only active edges are involved in the calculation of intersection points. The first intersection point between an active edge and a scan line is always the lower endpoint of the edge, whose coordinates are already stored in the edge's record. For example, when the algorithm begins at the first scan line, edges E_1 and E_7 become active. They intersect the scan line at (x_1, y_1).

Additional intersection points between an edge and successive scan lines can be calculated incrementally. If the edge intersects the current scan line at (x, y), it intersects the next scan line at $(x + 1/m, y + 1)$. For example, edge E_7 intersects scan line y at point b, and so the next intersection point on scan line $y + 1$ can be calculated by $\Delta x = 1/m_7$ since $\Delta y = 1$. This new x value can simply be kept in the x field of the edge's record.

An edge is deactivated or may even be removed from the edge list once the scan line whose y coordinate matches its y_{max} value has been processed, since all subsequent scan lines stay clear from it. The need that was mentioned early to give special treatment to a vertex where two monotonically increasing or decreasing edges meet is elegantly addressed by subtracting one from the y_{max} value of the lower edge. This means that the lower edge is deactivated one line before the scan line that intersects the vertex. Thus only the upper edge produces an intersection point with that scan line (see V_2 in Fig. 3-19). This explains why the initial y_{max} value for edges E_1 and E_4 has been decreased by one.

3.8 SCAN-CONVERTING A CHARACTER

Characters such as letters and digits are the building blocks of an image's textual contents. They can be presented in a variety of styles and sizes. The overall design style of a set of characters is referred to as its typeface or font. Commonly used fonts include Arial, Century Schoolbook, `Courier`, and Times New Roman. In addition, fonts can vary in appearance: **bold**, *italic*, and ***bold and italic***. Character size is typically measured in height in inches, points (approximately $\frac{1}{72}$ inch), and picas (12 points).

Bitmap Font

There are two basic approaches to character representation. The first is called a raster or bitmap font, where each character is represented by the on pixels in a bilevel pixel grid pattern called a bitmap (see Fig. 3-20). This approach is simple and effective since characters are defined in already-scan-converted form. Putting a character into an image basically entails a direct mapping or copying of its bitmap to a specific

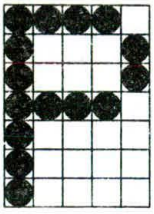


Fig. 3-20 Bitmap font.

location in the image space. On the other hand, a separate font consisting of scores of bitmaps for a set of characters is often needed for each combination of style, appearance, and size.

Although one might generate variations in appearance and size from one font, the overall results tends to be less than satisfactory. The example in Fig. 3-21 shows that we may overlay the bitmap in Fig. 3-20 onto itself with a horizontal offset of one pixel to produce (a) bold, and shift rows of pixels in Fig. 3-20 to produce (b) italic.

(*a*) Bold (*b*) Italic

Fig. 3-21 Generating variations in appearance.

Furthermore, the size of a bitmap font is dependent on image resolution. For example, a font using bitmaps that are 12 pixels high produces 12-point characters in an image with 72 pixels per inch. However, the same font will result in 9-point characters in an image with 96 pixels per inch, since 12 pixels now measure 0.125 inch, which is about 9 points. To get 12-point characters in the second image we would need a font with bitmaps that are 16 pixels high.

Outline Font

The second character representation method is called a vector or outline font, where graphical primitives such as lines and arcs are used to define the outline of each character (see Fig. 3-22). Although an outline definition tends to be less compact than a bitmap definition and requires relatively time-consuming scan-conversion operations, it can be used to produce characters of varying size, appearance, and even orientation. For example, the outline definition in Fig. 3-22 can be resized through a scaling transformation, made into italic through a shearing transformation, and turned around with respect to a reference point through a rotation transformation (see Chap. 4).

These transformed primitives can be scan-converted directly into characters in the form of filled regions in the target image area. Or they can be used to create the equivalent bitmaps that are then used to

Fig. 3-22 Outline font.

produce characters. This alternative is particularly effective in limiting scan-conversion time when the characters have repetitive occurrences in the image.

3.9 ANTI-ALIASING

Scan conversion is essentially a systematic approach to mapping objects that are defined in continuous space to their discrete approximation. The various forms of distortion that result from this operation are collectively referred to as the aliasing effects of scan conversion.

Staircase

A common example of aliasing effects is the staircase or jagged appearance we see when scan-converting a primitive such as a line or a circle. We also see the stair steps or "jaggies" along the border of a filled region.

Unequal Brightness

Another artifact that is less noticeable is the unequal brightness of lines of different orientation. A slanted line appears dimmer than a horizontal or vertical line, although all are presented at the same intensity level. The reason for this problem can be explained using Fig. 3-23, where the pixels on the horizontal line are placed one unit apart, whereas those on the diagonal line are approximately 1.414 units apart. This difference in density produces the perceived difference in brightness.

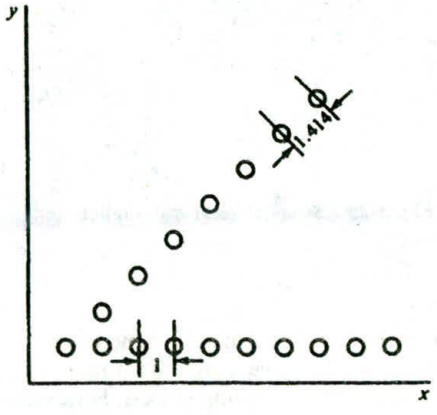
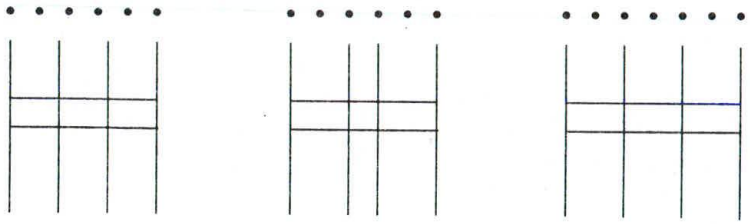


Fig. 3-23

The Picket Fence Problem

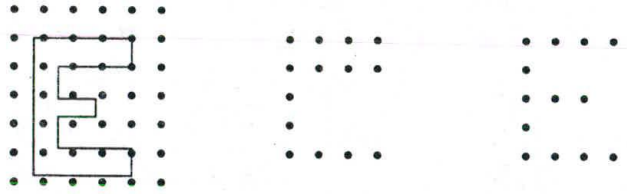
The picket fence problem occurs when an object is not aligned with, or does not fit into, the pixel grid properly. Figure 3-24(*a*) shows a picket fence where the distance between two adjacent pickets is not a multiple of the unit distance between pixels. Scan-converting it normally into the image space will result in uneven distances between pickets since the endpoints of the pickets will have to be snapped to pixel coordinates [see Fig. 3-24(*b*)]. This is sometimes called global aliasing, as the overall length of the picket fence is approximately correct. On the other hand, an attempt to maintain equal spacing will greatly distort the overall length of the fence [see Fig. 3-24(*c*)]. This is sometimes called local aliasing, as the distances between pickets are kept close to their true distances.



(*a*) (*b*) (*c*)

Fig. 3-24 The picket fence problem.

Another example of such a problem arises with the outline font. Suppose we want to scan-convert the uppercase character "E" in Fig. 3-25(*a*) from its outline description to a bitmap consisting of pixels inside the region defined by the outline. The result in Fig. 3-25(*b*) exhibits both asymmetry (the upper arm of the character is twice as thick as the other parts) and dropout (the middle arm is absent). A slight adjustment and/or realignment of the outline can lead to a reasonable outcome [see Fig. 3-25(*c*)].



(*a*) (*b*) (*c*)

Fig. 3-25 Scan-converting an outline font.

Anti-aliasing

Most aliasing artifacts, when appear in a static image at a moderate resolution, are often tolerable, and in many cases, negligible. However, they can have a significant impact on our viewing experience when left untreated in a series of images that animate moving objects. For example, a line being rotated around one of its endpoints becomes a rotating escalator with length-altering steps. A moving object with small parts or surface details may have some of those features intermittently change shape or even disappear.

Although increasing image resolution is a straightforward way to decrease the size of many aliasing artifacts and alleviate their negative impact, we pay a heavy price in terms of system resource (going from $W \times H$ to $2W \times 2H$ means quadrupling the number of pixels) and the results are not always satisfactory. On the other hand, there are techniques that can greatly reduce aliasing artifacts and improve the appearance of images without increasing their resolution. These techniques are collectively referred to as anti-aliasing techniques.

Some anti-aliasing techniques are designed to treat a particular type of artifact. For instance, an outline font can be associated with a set of rules or hints to guide the adjustment and realignment that is necessary for its conversion into bitmaps of relatively low resolution. An example of such approach is called the TrueType font.

Pre-filtering and Post-filtering

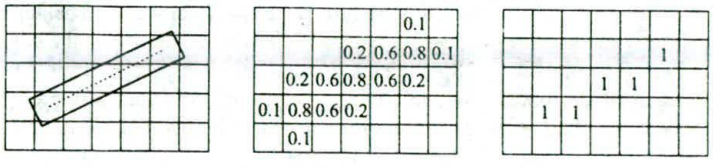
Pre-filtering and post-filtering are two types of general-purpose anti-aliasing techniques. The concept of filtering originates from the field of signal processing, where true intensity values are continuous signals that consists of elements of various frequencies. Constant intensity values that correspond to a uniform region are at the low end of the frequency range. Intensity values that change abruptly and correspond to a sharp edge are at the high end of the spectrum. In order to lessen the jagged appearance of lines and other contours in the image space, we seek to smooth out sudden intensity changes, or in signal-processing terms, to filter out the high frequency components. A pre-filtering technique works on the true signal in the continuous space to derive proper values for individual pixels (filtering before sampling), whereas a post-filtering technique takes discrete samples of the continuous signal and uses the samples to compute pixel values (sampling before filtering).

Area Sampling

Area sampling is a pre-filtering technique in which we superimpose a pixel grid pattern onto the continuous object definition. For each pixel area that intersects the object, we calculate the percentage of overlap by the object. This percentage determines the proportion of the overall intensity value of the corresponding pixel that is due to the object's contribution. In other words, the higher the percentage of overlap, the greater influence the object has on the pixel's overall intensity value.

In Fig. 3-26(a) a mathematical line shown in dotted form is represented by a rectangular region that is one pixel wide. The percentage of overlap between the rectangle and each intersecting pixel is calculated analytically. Assuming that the background is black and the line is white, the percentage values can be used directly to set the intensity of the pixels [see Fig. 3-26(b)]. On the other hand, had the background been gray (0.5, 0.5, 0.5) and the line green (0, 1, 0), each blank pixel in the grid would have had the background gray value and each pixel filled with a fractional number f would have been assigned a value of $[0.5(1-f), 0.5(1-f)+f, 0.5(1-f)]$—a proportional blending of the background and object colors.

Although the resultant discrete approximation of the line in Fig. 3-26(b) takes on a blurry appearance, it no longer exhibits the sudden transition from an on pixel to an off pixel and vice versa, which is what we



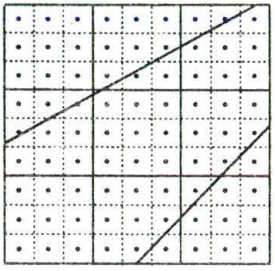
(a) (b) (c)

Fig. 3-26 Area sampling.

would get with an ordinary scan-conversion method [see Fig. 3-26(*c*)]. This tradeoff is characteristic of an anti-aliasing technique based on filtering.

Super Sampling

In this approach we subdivide each pixel into subpixels and check the position of each subpixel in relation to the object to be scan-converted. The object's contribution to a pixel's overall intensity value is proportional to the number of subpixels that are inside the area occupied by the object. Figure 3-27(*a*) shows an example where we have a white object that is bounded by two slanted lines on a black background. We subdivide each pixel into nine (3×3) subpixels. The scene is mapped to the pixel values in Fig. 3-27(*b*). The pixel at the upper right corner, for instance, is assigned $\frac{7}{9}$ since seven of its nine subpixels are inside the object area. Had the object been red $(1, 0, 0)$ and the background light yellow $(0.5, 0.5, 0)$, the pixel would have been assigned $(1 \times \frac{7}{9} + 0.5 \times \frac{2}{9}, 0.5 \times \frac{2}{9}, 0)$, which is $(\frac{8}{9}, \frac{1}{9}, 0)$.



(*a*)

0	2/9	7/9
6/9	1	8/9
1	8/9	1/9

(*b*)

Fig. 3-27 Super sampling.

Super sampling is often regarded as a post-filtering technique since discrete samples are first taken and then used to calculate pixel values. On the other hand, it can be viewed as an approximation to the area sampling method since we are simply using a finite number of values in each pixel area to approximate the accurate analytical result.

Lowpass Filtering

This is a post-filtering technique in which we reassign each pixel a new value that is a weighted average of its original value and the original values of its neighbors. A lowpass filter in the form of a $(2n + 1) \times (2n + 1)$ grid, where $n \geq 1$, holds the weights for the computation. All weight values in a filter should sum to one. An example of a 3×3 filter is given in Fig. 3-28(*a*).

To compute a new value for a pixel, we align the filter with the pixel grid and center it at the pixel. The weighted average is simply the sum of products of each weight in the filter times the corresponding pixel's original value. The filter shown in Fig. 3-28(*a*) means that half of each pixel's original value is retained in its new value, while each of the pixel's four immediate neighbors contributes one eighth of its original value to the pixel's new value. The result of applying this filter to the pixel values in Fig. 3-26(*c*) is shown in Fig. 3-28(*b*).

A lowpass filter with equal weights, sometimes referred to as a box filter, is said to be doing neighborhood averaging. On the other hand, a filter with its weight values conforming to a two-dimensional Gaussian distribution is called a Gaussian filter.

0	1/8	0
1/8	1/2	1/8
0	1/8	0

(a)

0	0	0	0	0	1/8	0
0	0	0	1/8	1/4	1/2	1/8
0	1/8	1/4	5/8	5/8	1/4	0
1/8	5/8	5/8	1/4	1/8	0	0
0	1/8	1/8	0	0	0	0

(b)

Fig. 3-28 Lowpass filtering.

Pixel Phasing

Pixel phasing is a hardware-based anti-aliasing technique. The graphics system in this case is capable of shifting individual pixels from their normal positions in the pixel grid by a fraction (typically $\frac{1}{4}$ and $\frac{1}{2}$) of the unit distance between pixels. By moving pixels closer to the true line or other contour, this technique is very effective in smoothing out the stair steps without reducing the sharpness of the edges.

3.10 EXAMPLE: RECURSIVELY DEFINED DRAWINGS

In this section we use two common graphical primitives to produce some interesting drawings. Each of these drawings is defined by applying a modification rule to a line or a filled triangle, breaking it into smaller pieces so the rule can be used to recursively modify each piece in the same manner. As the pieces become smaller and smaller, an intriguing picture emerges.

C Curve

A line by itself is a first-order C curve, denoted by C_0 (see Fig. 3-29). The modification rule for constructing successive generations of the C curve is to replace a line by two shorter, equal-length lines joining each other at a 90° angle, with the original line and the two new lines forming a right-angled triangle. See Fig. 3-29 for C_1, C_2, C_3, C_4, and C_5.

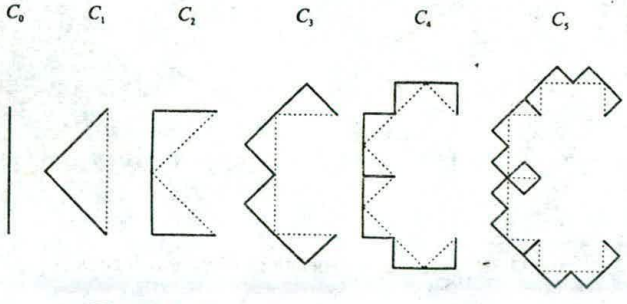


Fig. 3-29 Successive generations of the C curve.

Presume that the following call to the graphics library causes a line to be drawn from (x_1, y_1) to (x_2, y_2) using the system's current color:

line(x_1, y_1, x_2, y_2)

We can describe a pseudo-code procedure that generates C_n:

```
C-curve (float x, y, len, alpha; int n)
{
  if (n > 0) {
    len = len/sqrt(2.0);
    C-curve(x, y, len, alpha + 45, n − 1);
    x = x + len*cos(alpha + 45);
    y = y + len*sin(alpha + 45);
    C-curve(x, y, len, alpha − 45, n − 1);
  } else
    line(x, y, x + len*cos(alpha), y + len*sin(alpha));
}
```

where x and y are the coordinates of the starting point of a line, len the length of the line, alpha the angle (in degrees) between the line and the x axis, and n the number of recursive applications of the modification rule that is necessary to produce C_n. If $n = 0$, no modification is done and the line itself is drawn. Otherwise, two properly shortened lines, with one rotated counter-clockwise by 45° and the other clockwise by 45° from the current line position, are generated (representing one application of the modification rule), each of which is the basis of the remaining $n - 1$ steps of recursive construction.

The Koch Curve

As in the case of the C curve, a line by itself is a first-order Koch curve, denoted by K_0 (see Fig. 3-30). The modification rule for constructing successive generations of the Koch curve is to divide a line into three equal segments and replace the middle segment with two lines of the same length (the replaced segment and the two added lines form an equilateral triangle). See Fig. 3-30 for K_1, K_2, and K_3.

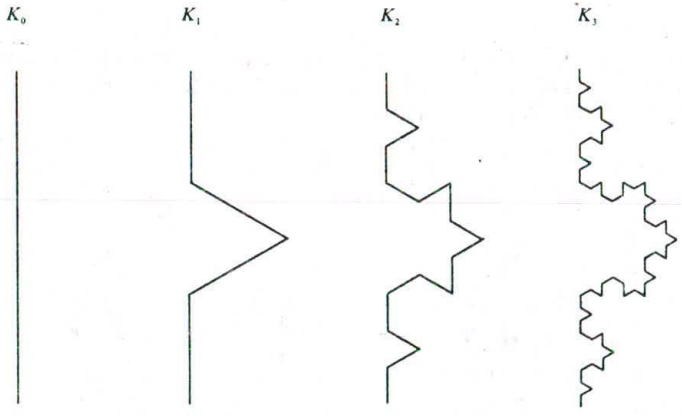


Fig. 3-30 Successive generations of the Koch curve.

The Sierpinski Gasket

This time our graphical primitive is a filled triangle, denoted by S_0 (see Fig. 3-31). The modification rule for constructing successive generations of the Sierpinski gasket is to take out the area defined by the lines connecting the midpoint of the edges of a filled triangle, resulting in three smaller ones that are similar to the original. See Fig. 3-31 for S_1, S_2, and S_3.

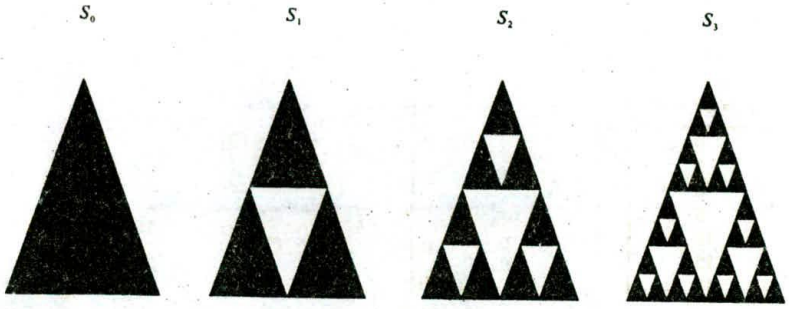


Fig. 3-31 Successive generations of the Sierpinski gasket.

Solved Problems

3.1 The endpoints of a given line are (0, 0) and (6, 18). Compute each value of y as x steps from 0 to 6 and plot the results.

SOLUTION

An equation for the line was not given. Therefore, the equation of the line must be found. The equation of the line ($y = mx + b$) is found as follows. First the slope is found:

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1} = \frac{18 - 0}{6 - 0} = \frac{18}{6} = 3$$

Next, the y intercept b is found by plugging y_1 and x_1 into the equation $y = 3x + b$: $0 = 3(0) + b$. Therefore, $b = 0$, so the equation for the line is $y = 3x$ (see Fig. 3-32).

3.2 What steps are required to plot a line whose slope is between 0° and 45° using the slope–intercept equation?

SOLUTION

1. Compute dx: $dx = x_2 - x_1$.
2. Compute dy: $dy = y_2 - y_1$.
3. Compute m: $m = dy/dx$.
4. Compute b: $b = y_1 - m \times x_1$.
5. Set (x, y) equal to the lower left-hand endpoint and set x_{end} equal to the largest value of x. If $dx < 0$, then $x = x_2$, $y = y_2$, and $x_{end} = x_1$. If $dx > 0$, then $x = x_1$, $y = y_1$, and $x_{end} = x_2$.
6. Test to determine whether the entire line has been drawn. If $x > x_{end}$, stop.
7. Plot a point at the current (x, y) coordinates.
8. Increment x: $x = x + 1$.
9. Compute the next value of y from the equation $y = mx + b$.
10. Go to step 6.

3.3 Use pseudo-code to describe the steps that are required to plot a line whose slope is between 45° and −45° (i.e., $|m| > 1$) using the slope–intercept equation.

$y = 3x + 0$	x
0	0
3	1
6	2
9	3
12	4
15	5
18	6

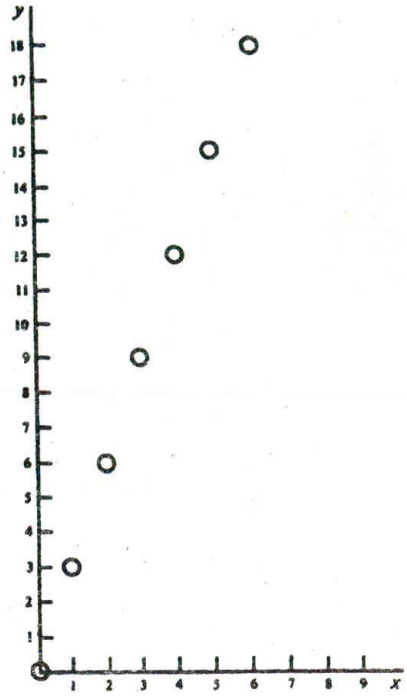


Fig. 3-32

SOLUTION

Presume $y_1 < y_2$ for the two endpoints (x_1, y_1) and (x_2, y_2):

```
int x = x1, y = y1;
float xf, m = (y2 − y1)/(x2 − x1), b = y1 − mx1;
setPixel(x, y);
while (y < y2) {
  y++;
  xf = (y − b)/m;
  x = Floor(xf + 0.5);
  setPixel(x, y);
}
```

3.4 Use pseudo-code to describe the DDA algorithm for scan-converting a line whose slope is between $-45°$ and $45°$ (i.e., $|m| \leq 1$).

SOLUTION

Presume $x_1 < x_2$ for the two endpoints (x_1, y_1) and (x_2, y_2):

```
int x = x1, y;
float yf = y1, m = (y2 − y1)/(x2 − x1);
while (x <= x2) {
  y = Floor(yf + 0.5);
  setPixel(x, y);
  x++;
  yf = yf + m;
}
```

3.5 Use pseudo-code to describe the DDA algorithm for scan-converting a line whose slope is between 45° and −45° (i.e., $|m| > 1$).

SOLUTION

Presume $y_1 < y_2$ for the two endpoints (x_1, y_1) and (x_2, y_2):

```
int x, y = y1;
float xf = x1, minv = (x2 - x1)/(y2 - y1);
while (y <= y2) {
  x = Floor(xf + 0.5);
  setPixel(x, y);
  xf = xf + minv;
  y++;
}
```

3.6 What steps are required to plot a line whose slope is between 0° and 45° using Bresenham's method?

SOLUTION

1. Compute the initial values:

$$dx = x_2 - x_1 \qquad Inc_2 = 2(dy - dx)$$
$$dy = y_2 - y_1 \qquad d = Inc_1 - dx$$
$$Inc_1 = 2dy$$

2. Set (x, y) equal to the lower left-hand endpoint and x_{end} equal to the largest value of x. If $dx < 0$, then $x = x_2$, $y = y_2$, $x_{end} = x_1$. If $dx > 0$, then $x = x_1$, $y = y_1$, $x_{end} = x_2$.
3. Plot a point at the current (x, y) coordinates.
4. Test to see whether the entire line has been drawn. If $x = x_{end}$, stop.
5. Compute the location of the next pixel. If $d < 0$, then $d = d + Inc_1$. If $d \geq 0$, then $d = d + Inc_2$, and then $y = y + 1$.
6. Increment x: $x = x + 1$.
7. Plot a point at the current (x, y) coordinates.
8. Go to step 4.

3.7 Indicate which raster locations would be chosen by Bresenham's algorithm when scan-converting a line from pixel coordinate (1, 1) to pixel coordinate (8, 5).

SOLUTION

First, the starting values must be found. In this case

$$dx = x_2 - x_1 = 8 - 1 = 7 \qquad dy = y_2 - y_1 = 5 - 1 = 4$$

Therefore:

$$Inc_1 = 2dy = 2 \times 4 = 8$$
$$Inc_2 = 2(dy - dx) = 2 \times (4 - 7) = -6$$
$$d = Inc_1 - dx = 8 - 7 = 1$$

The following table indicates the values computed by the algorithm (see also Fig. 3-33).

d	x	y
1	1	1
$1 + Inc_2 = -5$	2	2
$-5 + Inc_1 = 3$	3	2
$3 + Inc_2 = -3$	4	3
$-3 + Inc_1 = 5$	5	3
$5 + Inc_2 = -1$	6	4
$-1 + Inc_1 = 7$	7	4
$7 + Inc_2 = 1$	8	5

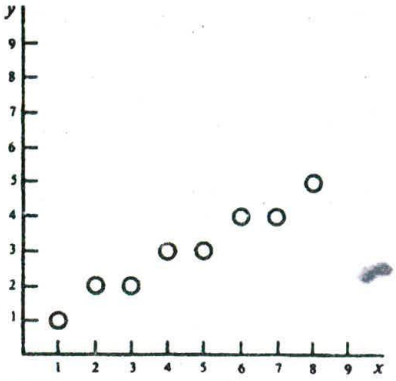


Fig. 3-33

3.8 In the derivation of Bresenham's line algorithm we have used s and t to measure the closeness of pixels S and T to the true line. However, s and t are only distances in the y direction. They are not really distances between a point to a line as defined in geometry. Can we be sure that, when $s = t$, the two pixels S and T are truly equally far away from the true line (hence we can choose either one to approximate the line)?

SOLUTION

As we can see in Fig. 3-34, when $s = t$ the true line intersects the vertical line connecting S and T at midpoint. The true distance from S to the line is dS and that from T to the line is dT. Since the two right-angled triangles are totally equal (they have equal angles and one pair of equal edges s and t), we get $d\text{S} = d\text{T}$.

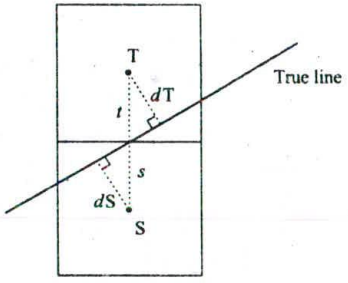


Fig. 3-34

3.9 Modify the description of Bresenham's line algorithm in the text to set all pixels from inside the loop structure.

SOLUTION 1

```
int x = x'1, y = y'1;
int dx = x'2 − x'1, dy = y'2 − y'1, dT = 2(dy − dx), dS = 2dy;
int d = 2dy − dx;
while (x <= x'2) {
  setPixel(x, y);
  x++;
  if (d < 0)
    d = d + dS;
```

```
    else {
      y++;
      d = d + dT;
    }
  }
```

SOLUTION 2

```
int x = x'1 − 1, y = y'1;
int dx = x'2 − x'1, dy = y'2 − y'1, dT = 2(dy − dx), dS = 2dy;
int d = −dx;
while (x < x'2) {
  x++;
  if (d < 0)
    d = d + dS;
  else {
    y++;
    d = d + dT;
  }
  setPixel(x, y);
}
```

3.10 What steps are required to generate a circle using the polynomial method?

SOLUTION

1. Set the initial variables: r = circle radius; (h, k) = coordinates of the circle center; $x = 0$; i = step size; $x_{end} = r/\sqrt{2}$.
2. Test to determine whether the entire circle has been scan-converted. If $x > x_{end}$, stop.
3. Compute the value of the y coordinate, where $y = \sqrt{r^2 - x^2}$.
4. Plot the eight points, found by symmetry with respect to the center (h, k), at the current (x, y) coordinates:

Plot$(x + h, y + k)$	Plot$(-x + h, -y + k)$
Plot$(y + h, x + k)$	Plot$(-y + h, -x + k)$
Plot$(-y + h, x + k)$	Plot$(y + h, -x + k)$
Plot$(-x + h, y + k)$	Plot$(x + h, -y + k)$

5. Increment x: $x = x + i$.
6. Go to step 2.

3.11 What steps are required to scan-convert a circle using the trigonometric method?

SOLUTION

1. Set the initial variables: r = circle radius; (h, k) = coordinates of the circle center; i = step size; $\theta_{end} = \pi/4$ radians = 45°; $\theta = 0$.
2. Test to determine whether the entire circle has been scan-converted. If $\theta > \theta_{end}$, stop.
3. Compute the value of the x and y coordinates:

$$x = r\cos(\theta) \qquad y = r\sin(\theta)$$

4. Plot the eight points, found by symmetry with respect to the center (h, k), at the current (x, y) coordinates:

Plot$(x+h, y+k)$	Plot$(-x+h, -y+k)$
Plot$(y+h, x+k)$	Plot$(-y+h, -x+k)$
Plot$(-y+h, x+k)$	Plot$(y+h, -x+k)$
Plot$(-x+h, y+k)$	Plot$(x+h, -y+k)$

5. Increment θ: $\theta = \theta + i$.
6. Go to step 2.

3.12 What steps are required to scan-convert a circle using Bresenham's algorithm?

SOLUTION

1. Set the initial values of the variables: (h, k) = coordinates of circle center; $x = 0$; y = circle radius r; and $d = 3 - 2r$.
2. Test to determine whether the entire circle has been scan-converted. If $x > y$, stop.
3. Plot the eight points, found by symmetry with respect to the center (h, k), at the current (x, y) coordinates:

Plot$(x+h, y+k)$	Plot$(-x+h, -y+k)$
Plot$(y+h, x+k)$	Plot$(-y+h, -x+k)$
Plot$(-y+h, x+k)$	Plot$(y+h, -x+k)$
Plot$(-x+h, y+k)$	Plot$(x+h, -y+k)$

4. Compute the location of the next pixel. If $d < 0$, then $d = d + 4x + 6$ and $x = x + 1$. If $d \geq 0$, then $d = d + 4(x - y) + 10$, $x = x + 1$, and $y = y - 1$.
5. Go to step 2.

3.13 When eight-way symmetry is used to obtain a full circle from pixel coordinates generated for the 0° to 45° or the 90° to 45° octant, certain pixels are set or plotted twice. This phenomenon is sometimes referred to as overstrike. Identify where overstrike occurs.

SOLUTION

At locations resulted from the initial coordinates $(r, 0)$ or $(0, r)$ since $(0, r) = (-0, r)$, $(0, -r) = (-0, -r)$, $(r, 0) = (r, -0)$, and $(-r, 0) = (-r, -0)$.

In addition, if the last generated pixel is on the diagonal line at $(\alpha r, \alpha r)$ where α approximates $1/\sqrt{(2.0)} = 0.7071$, then overstrike also occurs at $(\alpha r, \alpha r)$, $(-\alpha r, \alpha r)$, $(\alpha r, -\alpha r)$, and $(-\alpha r, -\alpha r)$.

3.14 Is overstrike harmful besides wasting time?

SOLUTION

It is often harmless since resetting a pixel with the same value does not really change the image in the frame buffer. However, if pixel values are sent out directly, for example, to control the exposure of a photographic medium, such as a slide or a negative, then overstrike amounts to double exposure at locations where it occurred.

Furthermore, if we set pixels using their complementary colors, then overstrike would leave them unchanged, since complementing a color twice simply yields the color itself.

3.15 When scan-converting a curve using the polynomial method (see Probs. 3.10, 3.20, and 3.25) or the trigonometric method (see Probs. 3.11, 3.21, and 3.24), a step size i is used to compute successive points on the true curve. These points are then mapped to the image space. What happens if i is too large? What happens if it is too small?

SOLUTION

If i is too large, the computed points will be relatively far from each other and the corresponding pixels will not form a continuous curve.

If i is too small, computed points will be so close to each other that two or more adjacent ones will be mapped to the same pixel, resulting in overstrike.

Note that it is not always possible to find a single step size for a specific scan-conversion task that yields a continuous curve without overstrike. In such cases we may take an adaptive approach in which adjustments are made to step size during scan-conversion based on points that have already been mapped to the image space. For example, if two consecutively computed points have been mapped to two pixels that are not connected to each other, then an additional point between the two points may be computed using half the step size.

3.16 Will the following description of Bresenham's circle algorithm and the one in the text produce the same results?

```
int x = 0, y = r, d = 3 − 2r;
setPixel(x, y);
while (x < y) {
  if (d < 0)
    d = d + 4x + 6;
  else {
    d = d + 4(x − y) + 10;
    y−−;
  }
  x++;
  setPixel(x, y);
}
```

SOLUTION

Let A be the correct set of pixels chosen by Bresenham's circle algorithm. Both versions produce A when the rightmost pixel in A is on the diagonal line $x = y$. However, when the coordinates of the rightmost pixel in A satisfies $x = y - 1$, only the version in the text stops properly. This version will produce one additional pixel beyond the 90° to 45° octant. This extra pixel mirrors the rightmost pixel in A with respect to the diagonal line.

3.17 In the derivation of Bresenham's circle algorithm we have used a decision variable $d_i = D(\mathrm{T}) + D(\mathrm{S})$ to help choose between pixels S and T. However, function D as defined in the text is not a true measure of the distance from the center of a pixel to the true circle. Show that when $d_i = 0$ the two pixels S and T are not really equally far away from the true circle.

SOLUTION

Let $d\mathrm{S}$ be the actual distance from S to the true circle and $d\mathrm{T}$ be the actual distance from T to the true circle (see Fig. 3-35). Also substitute x for $x_i + 1$ and y for y_i in the formula for d_i to make the following proof easier to read:

$$d_i = 2x^2 + y^2 + (y-1)^2 - 2r^2 = 0$$

Since $(r + d\mathrm{T})^2 = x^2 + y^2$ and $(r - d\mathrm{S})^2 = x^2 + (y-1)^2$ we have

$$2rd\mathrm{T} + d\mathrm{T}^2 = x^2 + y^2 - r^2 \qquad \text{and} \qquad -2rd\mathrm{S} + d\mathrm{S}^2 = x^2 + (y-1)^2 - r^2.$$

Hence

$$2rd\mathrm{T} + d\mathrm{T}^2 - 2rd\mathrm{S} + d\mathrm{S}^2 = 0$$
$$d\mathrm{T}(2r + d\mathrm{T}) = d\mathrm{S}(2r - d\mathrm{S})$$

Since $d\mathrm{T}/d\mathrm{S} = (2r - d\mathrm{S})/(2r + d\mathrm{T}) < 1$, we have $d\mathrm{T} < d\mathrm{S}$. This means that, when $d_i = 0$, pixel T is actually closer to the true circle than pixel S.

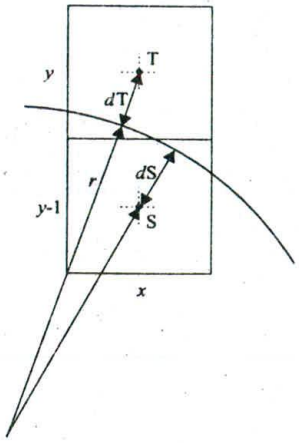


Fig. 3-35

3.18 Write a description of the midpoint circle algorithm in which decision parameter p is updated using x_{i+1} and y_{i+1} instead of x_i and y_i.

SOLUTION

```
int x = 0, y = r, p = 1 − r;
while (x <= y) {
  setPixel(x, y);
  x++;
  if (p < 0)
    p = p + 2x + 1;
  else {
    y−−;
    p = p + 2(x − y) + 1;
  }
}
```

3.19 Will the following description of the midpoint circle algorithm and the one in Prob. 3.18 produce the same results?

```
int x = 0, y = r, p = 1 − r;
setPixel(x, y);
while (x < y) {
  x++;
  if (p < 0)
    p = p + 2x + 1;
  else {
    y−−;
    p = p + 2(x − y) + 1;
  }
  setPixel(x, y);
}
```

SOLUTION

Similar to the solution for Prob. 3.16. Only the version in Prob. 3.18 produces the correct set of pixels under all circumstances.

3.20 What steps are required to generate an ellipse using the polynomial method?

SOLUTION

1. Set the initial variables: a = length of major axis; b = length of minor axis; (h, k) = coordinates of ellipse center; $x = 0$; i = step size; $x_{\text{end}} = a$.
2. Test to determine whether the entire ellipse has been scan-converted. If $x > x_{\text{end}}$, stop.
3. Compute the value of the y coordinate:

$$y = b\sqrt{1 - \frac{x^2}{a^2}}$$

4. Plot the four points, found by symmetry, at the current (x, y) coordinates:

$$\begin{array}{ll} \text{Plot}(x+h, y+k) & \text{Plot}(-x+h, -y+k) \\ \text{Plot}(-x+h, y+k) & \text{Plot}(x+h, -y+k) \end{array}$$

5. Increment x: $x = x + i$.
6. Go to step 2.

3.21 What steps are required to scan-convert an ellipse using the trigonometric method?

SOLUTION

1. Set the initial variables: a = length of major axis; b = length of minor axis; (h, k) = coordinates of ellipse center; i = counter step size; $\theta_{\text{end}} = \pi/2$; $\theta = 0$.
2. Test to determine whether the entire ellipse has been scan-converted. If $\theta > \theta_{\text{end}}$, stop.
3. Compute the values of the x and y coordinates:

$$x = a\cos(\theta) \qquad y = b\sin(\theta)$$

4. Plot the four points, found by symmetry, at the current (x, y) coordinates:

$$\begin{array}{ll} \text{Plot}(x+h, y+k) & \text{Plot}(-x+h, -y+k) \\ \text{Plot}(-x+h, y+k) & \text{Plot}(x+h, -y+k) \end{array}$$

5. Increment θ: $\theta = \theta + i$.
6. Go to step 2.

3.22 When four-way symmetry is used to obtain a full ellipse from pixel coordinates generated for the first quadrant, does overstrike occur? Where?

SOLUTION

Overstrike occurs at $(0, b)$, $(0, -b)$, $(a, 0)$, and $(-a, 0)$ since $(0, b) = (-0, b)$, $(0, -b) = (-0, -b)$, $(a, 0) = (a, -0)$, and $(-a, 0) = (-a, -0)$.

3.23 In the midpoint ellipse algorithm we have used only the coordinates (x_k, y_k) of the last pixel chosen for part 1 of the curve to compute the initial value q_1 of the decision parameter q_j for part 2. Can we also make use of the last value of the decision parameter p_i?

SOLUTION

The last computed value of the decision parameter p_i for part 1 of the curve is

$$p_k = f(x_k + 1, y_k - \tfrac{1}{2}) = b^2(x_k + 1)^2 + a^2(y_k - \tfrac{1}{2})^2 - a^2b^2$$
$$= b^2(x_k^2 + 2x_k + 1) + a^2(y_k^2 - y_k + \tfrac{1}{4}) - a^2b^2$$

Since

$$q_1 = f(x_k + \tfrac{1}{2}, y_k - 1) = b^2(x_k + \tfrac{1}{2})^2 + a^2(y_k - 1)^2 - a^2b^2$$
$$= b^2(x_k^2 + x_k + \tfrac{1}{4}) + a^2(y_k^2 - 2y_k + 1) - a^2b^2$$

We have

$$q_1 = p_k - b^2(x_k + \tfrac{3}{4}) - a^2(y_k - \tfrac{3}{4})$$

3.24 What steps are required to scan-convert an arc using the trigonometric method?

SOLUTION

1. Set the initial variables: a = major axis; b = minor axis; (h, k) = coordinates of arc center; i = step size; θ = starting angle; θ_1 = ending angle.
2. Test to determine whether the entire arc has been scan-converted. If $\theta > \theta_1$, stop.
3. Compute the values of the x and y coordinates:

$$x = a\cos(\theta) + h \qquad y = a\sin(\theta) + k$$

4. Plot the points at the current (x, y) coordinates: Plot(x, y).
5. Increment θ: $\theta = \theta + i$.
6. Go to step 2.

(Note: for the arc of a circle $a = b$ = circle radius r.)

3.25 What steps are required to generate an arc of a circle using the polynomial method?

SOLUTION

1. Set the initial variables: r = radius; (h, k) = coordinates of arc center; $x = x$ coordinate of start of arc; $x_1 = x$ coordinate of end of arc; i = counter step size.
2. Test to determine whether the entire arc has been scan-converted. If $x > x_1$, stop.
3. Compute the value of the y coordinate:

$$y = \sqrt{r^2 - x^2}$$

4. Plot at the current (x, y) coordinates:

$$\text{Plot}(x + h, y + k)$$

5. Increment x: $x = x + i$.
6. Go to step 2.

3.26 What steps are required to scan-convert a rectangle whose sides are parallel to the coordinate axes?

SOLUTION

1. Set initial variables: (x_1, y_1) = coordinates of first point specified; (x_2, y_2) = coordinates of second point specified.

2. Plot the rectangle:

$$\text{Plot}(x_1, y_1) \text{ to } (x_2, y_1) \quad \text{Plot}(x_2, y_2) \text{ to } (x_1, y_2)$$
$$\text{Plot}(x_2, y_1) \text{ to } (x_2, y_2) \quad \text{Plot}(x_1, y_2) \text{ to } (x_1, y_1)$$

3.27 How would a flood-fill algorithm fill the region shown in Fig. 3-36, using the 8-connected definition for region pixels?

SOLUTION

1. Assume that a seed is given at coordinate 3, 3. The flood-fill algorithm will inspect the eight points surrounding the seed (4, 4; 3, 4; 2, 4; 2, 3; 2, 2; 3, 2; 4, 2; 4, 3). Since all the points surrounding the seed have the region's original color, each point will be filled (see Fig. 3-37).
2. Each of the eight points found in step 1 becomes a new seed, and the points surrounding each new seed are inspected and filled. This process continues until all the points surrounding all the seeds are rid of the region's original color (see Fig. 3-38).

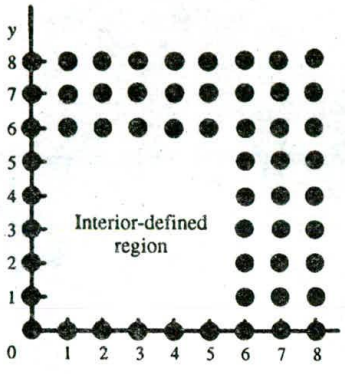


Fig. 3-36

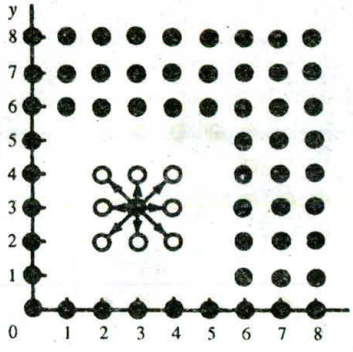


Fig. 3-37

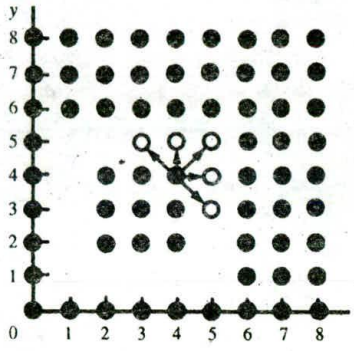


Fig. 3-38

3.28 Write a pseudo-code procedure to implement the boundary-fill algorithm in the text in its basic form, using the 4-connected definition for region pixels.

SOLUTION

```
BoundaryFill (int x, y, fill_color, boundary_color)
{
  int color;
  getPixel(x, y, color);
  if (color != boundary_color && color != fill_color) {
    setPixel(x, y, fill_color);
    BoundaryFill(x + 1, y, fill_color, boundary_color);
    BoundaryFill(x, y + 1, fill_color, boundary_color);
    BoundaryFill(x − 1, y, fill_color, boundary_color);
    BoundaryFill(x, y − 1, fill_color, boundary_color);
  }
}
```

3.29 Write a pseudo-code procedure to implement the flood-fill algorithm in the text in its basic form, using the 4-connected definition for region pixels.

SOLUTION

```
FloodFill (int x, y, fill_color, original_color)
{
  int color;
  getPixel(x, y, color);
  if (color == original_color) {
    setPixel(x, y, fill_color);
    FloodFill(x + 1, y, fill_color, original_color);
    FloodFill(x, y + 1, fill_color, original_color);
    FloodFill(x − 1, y, fill_color, original_color);
    FloodFill(x, y − 1, fill_color, original_color);
  }
}
```

3.30 The coordinates of the vertices of a polygon are shown in Fig. 3-39. (*a*) Write the initial edge list for the polygon. (*b*) State which edges will be active on scan lines $y = 6$, 7, 8, 9, and 10.

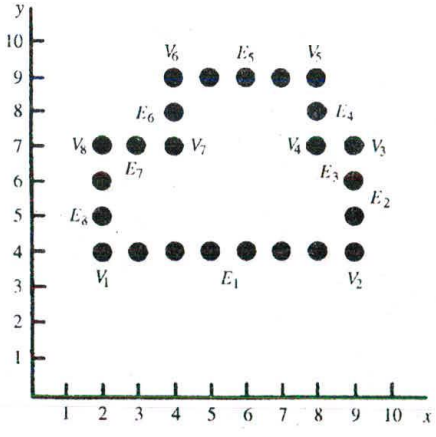


Fig. 3-39

SOLUTION

(*a*) Column x contains the x coordinate of the corresponding edge's lower endpoint. Horizontal edges are not included.

Edge	y_{min}	y_{max}	x	$1/m$
E_2	4	7	9	0
E_8	4	7	2	0
E_4	7	9	8	0
E_6	7	9	4	0

(*b*) An edge becomes active when the scan line value y equals the edge's y_{min} value. The edge remains active until the scan line value y goes beyond the edge's y_{max} value. Therefore, the active edges for $y = 6, 7, 8, 9$, and 10 appears as follows.

At $y = 6$, E_2 and E_8.

At $y = 7$, $y = y_{max}$ for both edges E_2 and E_8 so they remain active. Also at $y = 7$, edges E_4 and E_6 become active.

At $y = 8$, E_2 and E_8 are removed from the edge list. E_4 and E_6 remain active.

At $y = 9$, the active edges remain the same. At $y = 10$, edges E_2 and E_4 are removed from the edge list and the edge list becomes empty.

3.31 What are the three major adverse side effects of scan conversion?

SOLUTION

The three major adverse effects of scan conversion are staircase appearance, unequal brightness of slanted lines, and the picket fence problem.

3.32 Suppose that in 3×3 super sampling a pixel has three of its subpixels in a red area, three in a green area, and three in a blue area, what is the pixel's overall color?

SOLUTION

Each of the three areas is responsible for one third of the pixel's overall intensity value, which is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

3.33 Write a pseudo-code procedure for generating the Koch curve K_n (after the one in the text for generating C_n).

SOLUTION

```
Koch-curve (float x, y, len, alpha; int n)
{
  if (n > 0) {
    len = len/3;
    Koch-curve(x, y, len, alpha, n − 1);
    x = x + len*cos(alpha);
    y = y + len*sin(alpha);
    Koch-curve(x, y, len, alpha − 60, n − 1);
    x = x + len*cos(alpha − 60);
    y = y + len*sin(alpha − 60);
    Koch-curve(x, y, len, alpha + 60, n − 1);
    x = x + len*cos(alpha + 60);
    y = y + len*sin(alpha + 60);
    Koch-curve(x, y, len, alpha, n − 1);
  } else
    line(x, y, x + len*cos(alpha), y + len*sin(alpha));
}
```

3.34 Presume that the following statement produces a filled triangle with vertices at (x_1, y_1), (x_2, y_2), and (x_3, y_3):

triangle($x_1, y_1, x_2, y_2, x_3, y_3$)

Write a pseudo-code procedure for generating the Sierpinski gasket S_n (after the procedure in the text for generating C_n).

SOLUTION

```
S-Gasket (float x1, y1, x2, y2, x3, y3; int n)
{
  float x12, y12, x13, y13, x23, y23;
  if (n > 0) {
    x12 = (x1 + x2)/2;
    y12 = (y1 + y2)/2;
    x13 = (x1 + x3)/2;
    y13 = (y1 + y3)/2;
    x23 = (x2 + x3)/2;
    y23 = (y2 + y3)/2;
    S-Gasket(x1, y1, x12, y12, x13, y13, n − 1);
    S-Gasket(x12, y12, x2, y2, x23, y23, n − 1);
    S-Gasket(x13, y13, x23, y23, x3, y3, n − 1);
  } else
    triangle(x1, y1, x2, y2, x3, y3);
}
```

Supplementary Problems

3.35 Given the following equations, find the corresponding values of y for each value of x ($x = 2, 7, 1$): (*a*) $y = 4x + 3$, (*b*) $y = 1x + 0$, (*c*) $y = -3x - 4$, and (*d*) $y = -2x + 1$.

3.36 What steps are required to plot a line whose slope is between 45° and 90° using Bresenham's method?

3.37 What steps are required to plot a dashed line?

3.38 Show graphically that an ellipse has four-way symmetry by plotting four points on the ellipse:

$$x = a\cos(\theta) + h \qquad y = b\sin(\theta) + k$$

where $a = 2$
$b = 1$
$h = 0$
$k = 0$
$\theta = \pi/4, 3\pi/4, 5\pi/4, 7\pi/4$

3.39 How must Prob. 3.21 be modified if an ellipse is to be rotated (*a*) $\pi/4$, (*b*) $\pi/9$, and (*c*) $\pi/2$ radians?

3.40 What steps are required to scan-convert a sector using the trigonometric method?

3.41 What steps must be added to a fill algorithm if a region is to be filled with a pattern?

3.42 Why is it important for the designer to remain consistent when choosing either local or global aliasing?

3.43 What steps are required to scan-convert a polygonal area using the scan-line algorithm?

3.44 How can we eliminate overstrike?

CHAPTER 4

Two-Dimensional Transformations

Fundamental to all computer graphics system is the ability to simulate the manipulation of objects in space. This simulated spatial manipulation is referred to as *transformation*. The need for transformation arises when several objects, each of which is independently defined in its own coordinate system, need to be properly positioned into a common scene in a master coordinate system. Transformation is also useful in other areas of the image synthesis process (e.g. viewing transformation in Chap. 5).

There are two complementary points of view for describing object transformation. The first is that the object itself is transformed relative to a stationary coordinate system or background. The mathematical statement of this viewpoint is described by *geometric transformations* applied to each point of the object. The second point of view holds that the object is held stationary while the coordinate system is transformed relative to the object. This effect is attained through the application of *coordinate transformations*. An example that helps to distinguish these two viewpoints involves the movement of an automobile against a scenic background. We can simulate this by moving the automobile while keeping the backdrop fixed (a geometric transformation). Or we can keep the car fixed while moving the backdrop scenery (a coordinate transformation).

This chapter covers transformations in the plane, i.e., the two-dimensional (2D) space. We detail three basic transformations: translation, rotation, and scaling, along with other transformations that can be accomplished in terms of a sequence of basic transformations. We describe these operations in mathematical form suitable for computer processing and show how they are used to achieve the ends of object manipulation.

4.1 GEOMETRIC TRANSFORMATIONS

Let us impose a coordinate system on a plane. An object *Obj* in the plane can be considered as a set of points. Every object point P has coordinates (x, y), and so the object is the sum total of all its coordinate points (Fig. 4-1). If the object is moved to a new position, it can be regarded as a new object *Obj'*, all of whose coordinate points P' can be obtained from the original points P by the application of a geometric transformation.

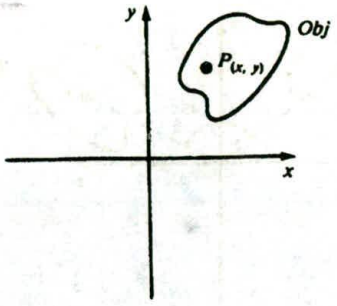


Fig. 4-1

Translation

In *translation*, an object is displaced a given distance and direction from its original position. If the displacement is given by the vector $\mathbf{v} = t_x\mathbf{I} + t_y\mathbf{J}$, the new object point $P'(x', y')$ can be found by applying the transformation $T_{\mathbf{v}}$ to $P(x, y)$ (see Fig. 4-2).

$$P' = T_{\mathbf{v}}(P)$$

where $x' = x + t_x$ and $y' = y + t_y$.

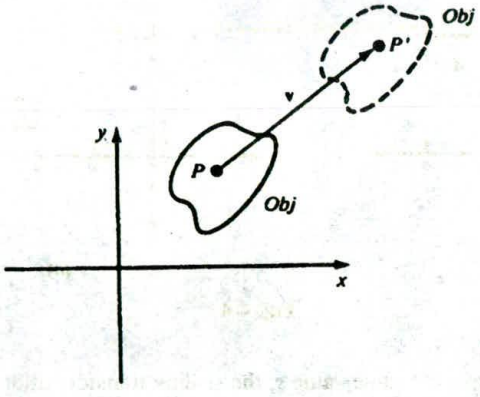


Fig. 4-2

Rotation about the Origin

In *rotation*, the object is rotated $\theta°$ about the origin. The convention is that the direction of rotation is counterclockwise if θ is a positive angle and clockwise if θ is a negative angle (see Fig. 4-3). The transformation of rotation R_θ is

$$P' = R_\theta(P)$$

where $x' = x\cos(\theta) - y\sin(\theta)$ and $y' = x\sin(\theta) + y\cos(\theta)$.

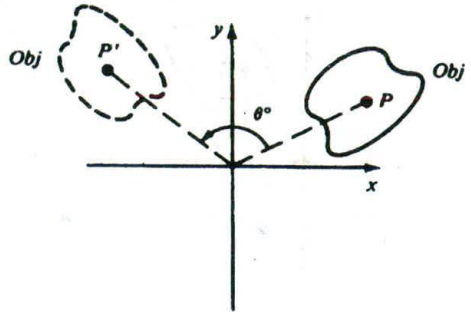


Fig. 4-3

Scaling with Respect to the Origin

Scaling is the process of expanding or compressing the dimensions of an object. Positive scaling constants s_x and s_y are used to describe changes in length with respect to the x direction and y direction, respectively. A scaling constant greater than one indicates an expansion of length, and less than one, compression of length. The scaling transformation S_{s_x,s_y} is given by $P' = S_{s_x,s_y}(P)$ where $x' = s_x x$ and $y' = s_y y$. Notice that, after a scaling transformation is performed, the new object is located at a different position relative to the origin. In fact, in a scaling transformation the only point that remains fixed is the origin. Figure 4-4 shows scaling transformation with scaling factors $s_x = 2$ and $s_y = \frac{1}{2}$.

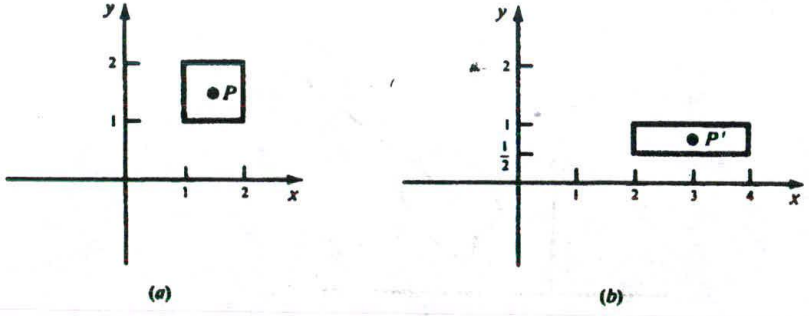


Fig. 4-4

If both scaling constants have the same value s, the scaling transformation is said to be *homogeneous* or *uniform*. Furthermore, if $s > 1$, it is a *magnification* and for $s < 1$, a *reduction*.

Mirror Reflection about an Axis

If either the x or y axis is treated as a mirror, the object has a mirror image or reflection. Since the reflection P' of an object point P is located the same distance from the mirror as P (Fig. 4-5), the mirror reflection transformation M_x about the x axis is given by

$$P' = M_x(P)$$

where $x' = x$ and $y' = -y$.

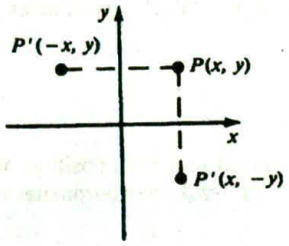


Fig. 4-5

Similarly, the mirror reflection about the y axis is

$$P' = M_y(P)$$

where $x' = -x$ and $y' = y$.

Note that $M_x = S_{1,-1}$ and $M_{xy} = S_{-1,1}$. The two reflection transformations are simply special cases of scaling.

Inverse Geometric Transformations

Each geometric transformation has an inverse (see App. 1) which is described by the opposite operation performed by the transformation:

Translation: $T_{\mathbf{v}}^{-1} = T_{-\mathbf{v}}$, or translation in the opposite direction

Rotation: $R_\theta^{-1} = R_{-\theta}$, or rotation in the opposite direction

Scaling: $S_{s_x,s_y}^{-1} = S_{1/s_x,1/s_y}$

Mirror reflection: $M_x^{-1} = M_x$ and $M_y^{-1} = M_y$.

4.2 COORDINATE TRANSFORMATIONS

Suppose that we have two coordinate systems in the plane. The first system is located at origin O and has coordinates axes xy. The second coordinate system is located at origin O' and has coordinate axes $x'y'$ (Fig. 4-6). Now each point in the plane has two coordinate descriptions: (x, y) or (x', y'), depending on which coordinate system is used. If we think of the second system $x'y'$ as arising from a transformation applied to the first system xy, we say that a *coordinate transformation* has been applied. We can describe

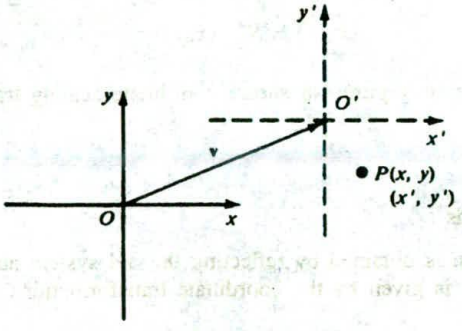


Fig. 4-6

this transformation by determining how the (x', y') coordinates of a point P are related to the (x, y) coordinates of the same point.

Translation

If the xy coordinate system is displaced to a new position, where the direction and distance of the displacement is given by the vector $\mathbf{v} = t_x\mathbf{I} + t_y\mathbf{J}$, the coordinates of a point in both systems are related by the translation transformation $\bar{T}_{\mathbf{v}}$:

$$(x', y') = \bar{T}_{\mathbf{v}}(x, y)$$

where $x' = x - t_x$ and $y' = y - t_y$.

Rotation about the Origin

The xy system is rotated $\theta°$ about the origin (see Fig. 4-7). Then the coordinates of a point in both systems are related by the rotation transformation $\bar{R}_\theta$:

$$(x', y') = \bar{R}_\theta(x, y)$$

where $x' = x\cos(\theta) + y\sin(\theta)$ and $y' = -x\sin(\theta) + y\cos(\theta)$.

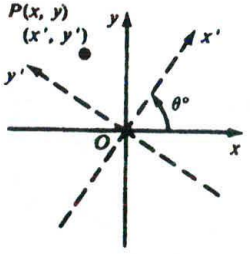


Fig. 4-7

Scaling with Respect to the Origin

Suppose that a new coordinate system is formed by leaving the origin and coordinate axes unchanged, but introducing different units of measurement along the x and y axes. If the new units are obtained from the old units by a scaling of s_x along the x axis and s_y along the y axis, the coordinates in the new system are related to coordinates in the old system through the scaling transformation $\bar{S}_{s_x,s_y}$:

$$(x', y') = \bar{S}_{s_x,s_y}(x, y)$$

where $x' = (1/s_x)x$ and $y' = (1/s_y)y$. Figure 4-8 shows coordinate scaling transformation using scaling factors $s_x = 2$ and $s_y = \frac{1}{2}$.

Mirror Reflection about an Axis

If the new coordinate system is obtained by reflecting the old system about either x or y axis, the relationship between coordinates is given by the coordinate transformations $\bar{M}_x$ and $\bar{M}_y$. For reflection about the x axis [Fig. 4-9(a)]

$$(x', y') = \bar{M}_x(x, y)$$

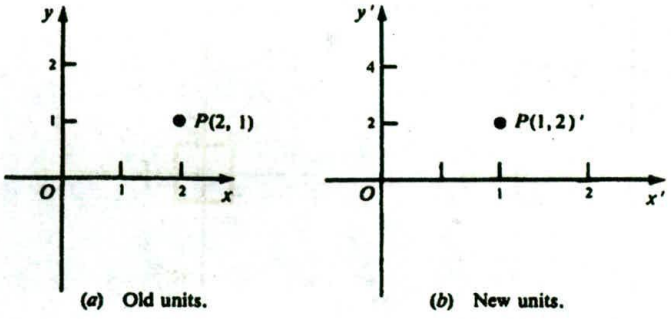


Fig. 4-8

where $x' = x$ and $y' = -y$. For reflection about the y axis [Fig. 4-9(b)]

$$(x', y') = \bar{M}_y(x, y)$$

where $x' = -x$ and $y' = y$.

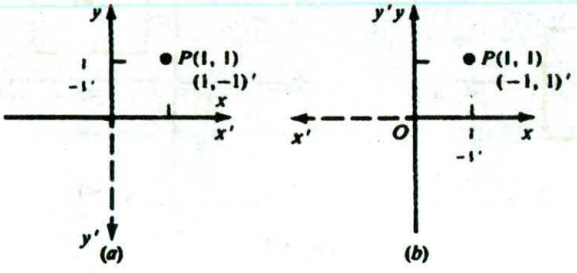


Fig. 4-9

Notice that the reflected coordinate system is left-handed; thus reflection changes the orientation of the coordinate system. Also note that $\bar{M}_x = \bar{S}_{1,-1}$ and $\bar{M}_y = \bar{S}_{-1,1}$.

Inverse Coordinate Transformations

Each coordinate transformation has an inverse (see App. 1) which can be found by applying the opposite transformation:

Translation: $\bar{T}_v^{-1} = \bar{T}_{-v}$, translation in the opposite direction
Rotation: $\bar{R}_\theta^{-1} = \bar{R}_{-\theta}$, rotation in the opposite direction
Scaling: $\bar{S}_{s_x,s_y}^{-1} = \bar{S}_{1/s_x,1/s_y}$
Mirror reflection: $\bar{M}_x^{-1} = \bar{M}_x$ and $\bar{M}_y^{-1} = \bar{M}_y$.

4.3 COMPOSITE TRANSFORMATIONS

More complex geometric and coordinate transformations can be built from the basic transformations described above by using the process of *composition of functions* (see App. 1). For example, such operations as rotation about a point other than the origin or reflection about lines other than the axes can be constructed from the basic transformations.

EXAMPLE 1. Magnification of an object while keeping its center fixed (see Fig. 4-10). Let the geometric center be located at $C(h, k)$ [Fig. 4-10(a)]. Choosing a magnification factor $s > 1$, we construct the transformation by

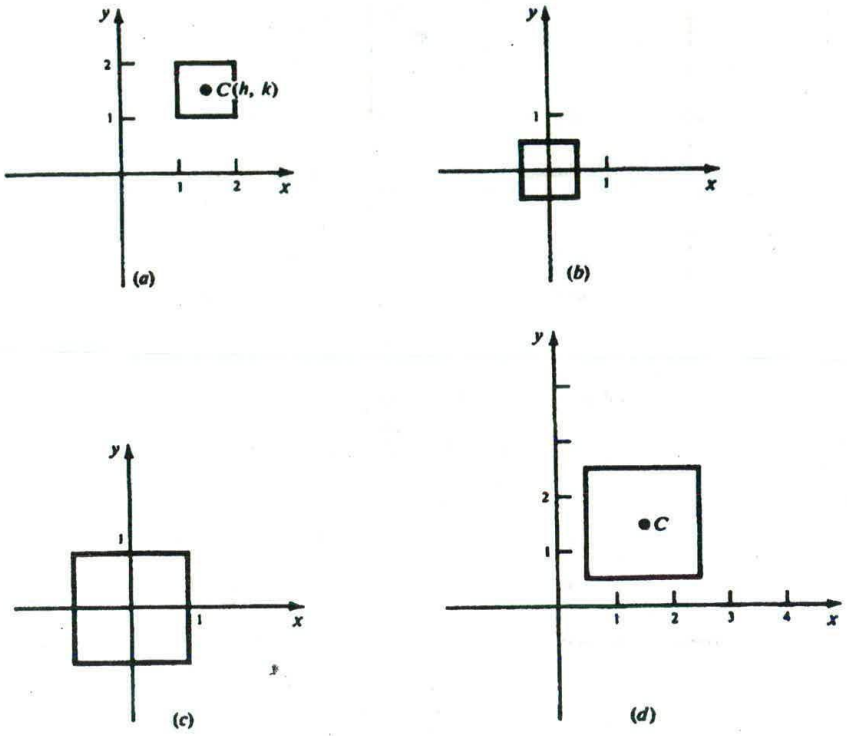


Fig. 4-10

performing the following sequence of basic transformations: (1) translate the object so that its center coincides with the origin [Fig. 4-10(*b*)], (2) scale the object with respect to the origin [Fig. 4-10(*c*)], and (3) translate the scaled object back to the original position [Fig. 4-10(*d*)].

The required transformation $S_{s,C}$ can be formed by compositions $S_{s,C} = T_{\mathbf{v}} \cdot S_{s,s} \cdot T_{\mathbf{v}}^{-1}$ where $\mathbf{v} = h\mathbf{I} + k\mathbf{J}$. By using composition, we can build more general scaling, rotation, and reflection transformations. For these transformations, we shall use the following notations: (1) $S_{s_x,s_y,P}$—scaling with respect to a fixed point P; (2) $R_{\theta,P}$—rotation about a point P; and (3) M_L—reflection about a line L.

The matrix description of these transformations can be found in Probs. 4.4, 4.7, and 4.10.

Matrix Description of the Basic Transformations

The transformations of rotation, scaling, and reflection can be represented as matrix functions:

Geometric transformations	Coordinate transformations
$R_\theta = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$	$\bar{R}_\theta = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$
$S_{s_x,s_y} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$	$\bar{S}_{s_x,s_y} = \begin{pmatrix} \frac{1}{s_x} & 0 \\ 0 & \frac{1}{s_y} \end{pmatrix}$
$M_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	$\bar{M}_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
$M_y = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$	$\bar{M}_y = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$

The translation transformation cannot be expressed as a 2×2 matrix function. However, a certain artifice allows us to introduce a 3×3 matrix function which performs the translation transformation.

We represent the coordinate pair (x, y) of a point P by the triple $(x, y, 1)$. This is simply the homogeneous representation of P (App. 2). Then translation in the direction $\mathbf{v} = t_x\mathbf{I} + t_y\mathbf{J}$ can be expressed by the matrix function

$$T_{\mathbf{v}} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix}$$

Then

$$\begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$$

From this we extract the coordinate pair $(x + t_x, y + t_y)$.

Concatenation of Matrices

The advantage of introducing a matrix form for translation is that we can now build complex transformations by multiplying the basic matrix transformations. This process is sometimes called *concatenation of matrices* and the resulting matrix is often referred to as the *composite transformation matrix* (CTM). Here, we are using the fact that the composition of matrix functions is equivalent to matrix multiplication (App. 1). We must be able to represent the basic transformations as 3×3 *homogeneous coordinate matrices* (App. 2) so as to be compatible (from the point of view of matrix multiplication) with the matrix of translation. This is accomplished by augmenting the 2×2 matrices with a third column $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ and a third row (0 0 1). That is

$$\begin{pmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

EXAMPLE 2. Express as a matrix (i.e., CTM) the transformation which magnifies an object about its center $C(h, k)$. From Example 1, the required transformation $S_{s,C}$ can be written as

$$\begin{aligned} S_{s,C} &= T_{\mathbf{v}} \cdot S_{s,s} \cdot T_{\mathbf{v}}^{-1} \\ &= \begin{pmatrix} 1 & 0 & h \\ 0 & 1 & k \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -h \\ 0 & 1 & -k \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} s & 0 & -sh + h \\ 0 & s & -sk + k \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

Caution on Matrix Notations

The reader should be alerted to the fact that, within the field of computer graphics, there are two different matrix notations that are used. This book represents points by column vectors and applies transformations by left-multiplying by the transformation matrix. We have chosen this approach because it is the standard used in mathematics and computer science texts. The other notation represents points by row vectors and applies transformations by right-multiplying by the transformation matrix. It is used in much of the computer graphics literature.

To change from one notational style to another, it is necessary to take the transpose of the matrices that appear in any expression. For example, translation of point (x, y) in the direction $\mathbf{v} = t_x\mathbf{I} + t_y\mathbf{J}$ can also be expressed as

$$(x \quad y \quad 1)\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{pmatrix} = (x + t_x \quad y + t_y \quad 1)$$

EXAMPLE 2 continued. Using the row-vector notation, we have

$$S_{s,C} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -h & -k & 1 \end{pmatrix}\begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ h & k & 1 \end{pmatrix} = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ -sh + h & -sk + k & 1 \end{pmatrix}$$

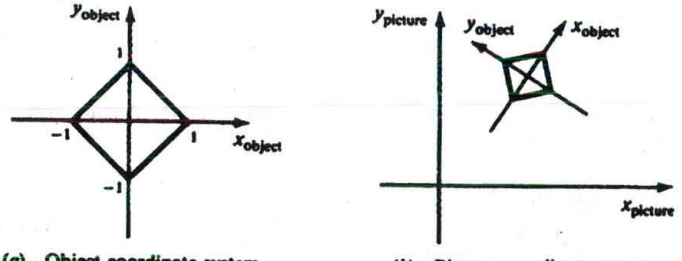
4.4 INSTANCE TRANSFORMATIONS

Quite often a picture or design is composed of many objects used several times each. In turn, these objects may also be composed of other symbols and objects. We suppose that each object is defined, independently of the picture, in its own coordinate system. We wish to place these objects together to form the picture or at least part of the picture, called a *subpicture*. We can accomplish this by defining a transformation of coordinates, called an *instance transformation*, which converts object coordinates to picture coordinates so as to place or create an *instance* of the object in the picture coordinate system.

The *instance transformation* $N_{\text{picture,object}}$ is formed as a composition or concatenation of scaling, rotation, and translation operations, usually performed in this order (although any order can be used):

$$N_{\text{picture,object}} = T_{\mathbf{v}} \cdot R_{\theta,P} \cdot S_{a,b,P}$$

With the use of different instance transformations, the same object can be placed in different positions, sizes, and orientations within a subpicture. For instance, Fig. 4-11(*a*) is placed in the picture coordinate system of Fig. 4-11(*b*) by using the instance transformations $N_{\text{picture,object}}$.



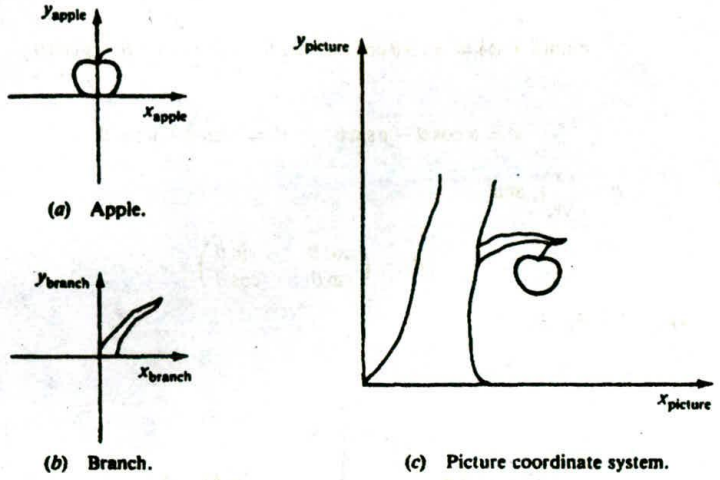
(*a*) Object coordinate system. (*b*) Picture coordinate system.

Fig. 4-11

Nested Instances and Multilevel Structures

A subpicture or picture may exhibit a multilevel or nested structure by being composed of objects which are, in turn, composed of still other objects, and so on. Separate instance transformations must then be applied, in principle, at each level of the picture structure for each picture component.

EXAMPLE 3. A picture of an apple tree contains branches, and an apple hangs on each branch. Suppose that each branch and apple is described in its own coordinate system [Figs. 4-12(*a*) and 4-12(*b*)]. Then a subpicture call to place



(*a*) Apple. (*b*) Branch. (*c*) Picture coordinate system.

Fig. 4-12

an instance of this branch in the picture of the tree requires an additional subpicture call to place an instance of the apple into the branch coordinate system.

We can perform each instance transformation separately, i.e., instance the apple in the branch coordinate system and then instance both branch and apple from the branch coordinate system to the picture coordinate system. However, it is much more efficient to transform the apple directly into picture coordinates [see Fig. 4-12(*c*)]. This is accomplished by defining the composite transformation matrix $C_{\text{picture,object}}$ to be the composition of the nested instance transformations from apple coordinates to branch coordinates and then from branch coordinates to picture coordinates:

$$C_{\text{picture,apple}} = N_{\text{picture,branch}} \cdot N_{\text{branch,apple}}$$

Since the branch subpicture is only one level below the picture

$$C_{\text{picture,branch}} = N_{\text{picture,branch}}$$

Solved Problems

4.1 Derive the transformation that rotates an object point $\theta°$ about the origin. Write the matrix representation for this rotation.

SOLUTION

Refer to Fig. 4-13. Definition of the trigonometric functions sin and cos yields

$$x' = r\cos(\theta + \phi) \qquad y' = r\sin(\theta + \phi)$$

and

$$x = r\cos\phi \qquad y = r\sin\phi$$

Using trigonometric identities, we obtain

$$r\cos(\theta + \phi) = r(\cos\theta\cos\phi - \sin\theta\sin\phi) = x\cos\theta - y\sin\theta$$

and

$$r\sin(\theta+\phi) = r(\sin\theta\cos\phi + \cos\theta\sin\phi) = x\sin\theta - y\cos\theta$$

or

$$x' = x\cos\theta - y\sin\theta \qquad y' = x\sin\theta + y\cos\theta$$

Writing $P' = \begin{pmatrix} x' \\ y' \end{pmatrix}$, $P = \begin{pmatrix} x \\ y \end{pmatrix}$, and

$$R_\theta = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

we can now write $P' = R_\theta \cdot P$.

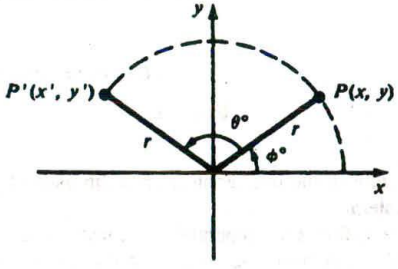


Fig. 4-13

4.2 (*a*) Find the matrix that represents rotation of an object by 30° about the origin.
(*b*) What are the new coordinates of the point $P(2, -4)$ after the rotation?

SOLUTION

(*a*) From Prob. 4.1:

$$R_{30^\circ} = \begin{pmatrix} \cos 30^\circ & -\sin 30^\circ \\ \sin 30^\circ & \cos 30^\circ \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix}$$

(*b*) So the new coordinates can be found by multiplying:

$$\begin{pmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix} \begin{pmatrix} 2 \\ -4 \end{pmatrix} = \begin{pmatrix} \sqrt{3}+2 \\ 1-2\sqrt{3} \end{pmatrix}$$

4.3 Describe the transformation that rotates an object point, $Q(x, y)$, θ° about a fixed center of rotation $P(h, k)$ (Fig. 4-14).

SOLUTION

We determine the transformation $R_{\theta,P}$ in three steps: (1) translate so that the center of rotation P is at the origin, (2) perform a rotation of θ degrees about the origin, and (3) translate P back to (h, k).

Using $\mathbf{v} = h\mathbf{I} + k\mathbf{J}$ as the translation vector, we build $R_{\theta,P}$ by composition of transformations:

$$R_{\theta,O} = T_{\mathbf{v}} \cdot R_\theta \cdot T_{-\mathbf{v}}$$

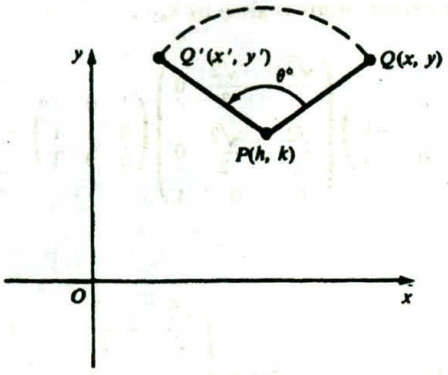


Fig. 4-14

4.4 Write the general form of the matrix for rotation about a point $P(h, k)$.

SOLUTION

Following Prob. 4.3, we write $R_{\theta,P} = T_{\mathbf{v}} \cdot R_{\theta} \cdot T_{-\mathbf{v}}$, where $\mathbf{v} = h\mathbf{I} + k\mathbf{J}$. Using the 3×3 homogeneous coordinate form for the rotation and translation matrices, we have

$$R_{\theta,P} = \begin{pmatrix} 1 & 0 & h \\ 0 & 1 & k \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -h \\ 0 & 1 & -k \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos(\theta) & -\sin(\theta) & [-h\cos(\theta) + k\sin(\theta) + h] \\ \sin(\theta) & \cos(\theta) & [-h\sin(\theta) - k\cos(\theta) + k] \\ 0 & 0 & 1 \end{pmatrix}$$

4.5 Perform a 45° rotation of triangle $A(0, 0)$, $B(1, 1)$, $C(5, 2)$ (*a*) about the origin and (*b*) about $P(-1, -1)$.

SOLUTION

We represent the triangle by a matrix formed from the homogeneous coordinates of the vertices:

$$\begin{matrix} A & B & C \end{matrix}$$
$$\begin{pmatrix} 0 & 1 & 5 \\ 0 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix}$$

(*a*) The matrix of rotation is

$$R_{45°} = \begin{pmatrix} \cos 45° & -\sin 45° & 0 \\ \sin 45° & \cos 45° & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

So the coordinates $A'B'C'$ of the rotated triangle ABC can be found as

$$[A'B'C'] = R_{45°} \cdot [ABC] = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 5 \\ 0 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix} = \begin{matrix} A' & B' & C' \end{matrix} \begin{pmatrix} 0 & 0 & \frac{3\sqrt{2}}{2} \\ 0 & \sqrt{2} & \frac{7\sqrt{2}}{2} \\ 1 & 1 & 1 \end{pmatrix}$$

Thus $A' = (0, 0)$, $B' = (0, \sqrt{2})$, and $C' = (\frac{3}{2}\sqrt{2}, \frac{7}{2}\sqrt{2})$.

(*b*) From Prob. 4.4, the rotation matrix is given by $R_{45°,P} = T_{\mathbf{v}} \cdot R_{45°} \cdot T_{-\mathbf{v}}$, where $\mathbf{v} = -\mathbf{I} - \mathbf{J}$. So

$$R_{45°,P} = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & 0 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & -1 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & (\sqrt{2}-1) \\ 0 & 0 & 1 \end{pmatrix}$$

Now

$$[A'B'C'] = R_{45°,P} \cdot [ABC] = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} & -1 \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & (\sqrt{2}-1) \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 5 \\ 0 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} -1 & -1 & (\frac{3}{2}\sqrt{2}-1) \\ (\sqrt{2}-1) & (2\sqrt{2}-1) & (\frac{9}{2}\sqrt{2}-1) \\ 1 & 1 & 1 \end{pmatrix}$$

So $A' = (-1, \sqrt{2}-1)$, $B' = (-1, 2\sqrt{2}-1)$, and $C' = (\frac{3}{2}\sqrt{2}-1, \frac{9}{2}\sqrt{2}-1)$.

4.6 Find the transformation that scales (with respect to the origin) by (*a*) a units in the X direction, (*b*) b units in the Y direction, and (*c*) simultaneously a units in the X direction and b units in the Y direction.

SOLUTION

(*a*) The scaling transformation applied to a point $P(x, y)$ produces the point (ax, y). We can write this in matrix form as $S_{a,1} \cdot P$, or

$$\begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax \\ y \end{pmatrix}$$

(*b*) As in part (*a*), the required transformation can be written in matrix form as $S_{1,b} \cdot P$. So

$$\begin{pmatrix} 1 & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ by \end{pmatrix}$$

(*c*) Scaling in both directions is described by the transformation $x' = ax$ and $y' = by$. Writing this in matrix form as $S_{a,b} \cdot P$, we have

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax \\ by \end{pmatrix}$$

4.7 Write the general form of a scaling matrix with respect to a fixed point $P(h, k)$.

SOLUTION

Following the same general procedure as in Probs. 4.3 and 4.4, we write the required transformation with

$\mathbf{v} = h\mathbf{I} + k\mathbf{J}$ as

$$S_{a,b,P} = T_{\mathbf{v}} \cdot S_{a,b} \cdot T_{-\mathbf{v}}$$
$$= \begin{pmatrix} 1 & 0 & h \\ 0 & 1 & k \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -h \\ 0 & 1 & -k \\ 0 & 0 & 1 \end{pmatrix}$$
$$= \begin{pmatrix} a & 0 & -ah + h \\ 0 & b & -bk + k \\ 0 & 0 & 1 \end{pmatrix}$$

4.8 Magnify the triangle with vertices $A(0, 0)$, $B(1, 1)$, and $C(5, 2)$ to twice its size while keeping $C(5, 2)$ fixed.

SOLUTION

From Prob. 4.7, we can write the required transformation with $\mathbf{v} = 5\mathbf{I} + 2\mathbf{J}$ as

$$S_{2,2,C} = T_{\mathbf{v}} \cdot S_{2,2} \cdot T_{-\mathbf{v}}$$
$$= \begin{pmatrix} 1 & 0 & 5 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -5 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0 & -5 \\ 0 & 2 & -2 \\ 0 & 0 & 1 \end{pmatrix}$$

Representing a point P with coordinates (x, y) by the column vector $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$, we have

$$S_{2,2,C} \cdot A = \begin{pmatrix} 2 & 0 & -5 \\ 0 & 2 & -2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -5 \\ -2 \\ 1 \end{pmatrix}$$
$$S_{2,2,C} \cdot B = \begin{pmatrix} 2 & 0 & -5 \\ 0 & 2 & -2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} -3 \\ 0 \\ 1 \end{pmatrix}$$
$$S_{2,2,C} \cdot C = \begin{pmatrix} 2 & 0 & -5 \\ 0 & 2 & -2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 5 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 2 \\ 1 \end{pmatrix}$$

So $A' = (-5, -2)$, $B' = (-3, 0)$, and $C' = (5, 2)$. Note that, since the triangle ABC is completely determined by its vertices, we could have saved much writing by representing the vertices using a 3×3 matrix

$$[ABC] = \begin{pmatrix} 0 & 1 & 5 \\ 0 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix}$$

and applying $S_{2,2,C}$ to this. So

$$S_{2,2,C} \cdot [ABC] = \begin{pmatrix} 2 & 0 & -5 \\ 0 & 2 & -2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 5 \\ 0 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} -5 & -3 & 5 \\ -2 & 0 & 2 \\ 1 & 1 & 1 \end{pmatrix} = [A'B'C']$$

4.9 Describe the transformation M_L which reflects an object about a line L.

SOLUTION

Let line L in Fig. 4-15 have a y intercept $(0, b)$ and an angle of inclination $\theta°$ (with respect to the x axis). We reduce the description to known transformations:

1. Translate the intersection point B to the origin.
2. Rotate by $-\theta^\circ$ so that line L aligns with the x axis.
3. Mirror-reflect about the x axis.
4. Rotate back by θ°.
5. Translate B back to $(0, b)$.

In transformation notation, we have

$$M_L = T_{\mathbf{v}} \cdot R_\theta \cdot M_x \cdot R_{-\theta} \cdot T_{-\mathbf{v}}$$

where $\mathbf{v} = b\mathbf{J}$.

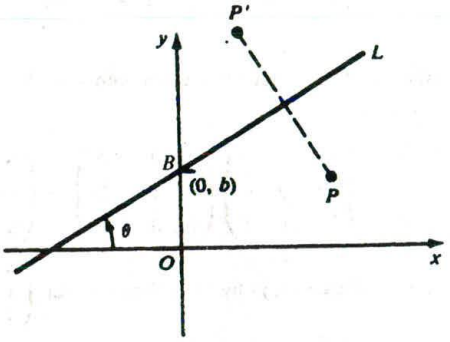


Fig. 4-15

4.10 Find the form of the matrix for reflection about a line L with slope m and y intercept $(0, b)$.

SOLUTION

Following Prob. 4.9 and applying the fact that the angle of inclination of a line is related to its slope m by the equation $\tan(\theta) = m$, we have with $\mathbf{v} = b\mathbf{J}$,

$$M_L = T_{\mathbf{v}} \cdot R_\theta \cdot M_x \cdot R_{-\theta} \cdot T_{-\mathbf{v}}$$

$$= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{pmatrix}$$

Now if $\tan(\theta) = m$, standard trigonometry yields $\sin(\theta) = m/\sqrt{m^2+1}$ and $\cos(\theta) = 1/\sqrt{m^2+1}$. Substituting these values for $\sin(\theta)$ and $\cos(\theta)$ after matrix multiplication, we have

$$M_L = \begin{pmatrix} \dfrac{1-m^2}{m^2+1} & \dfrac{2m}{m^2+1} & \dfrac{-2bm}{m^2+1} \\ \dfrac{2m}{m^2+1} & \dfrac{m^2-1}{m^2+1} & \dfrac{2b}{m^2+1} \\ 0 & 0 & 1 \end{pmatrix}$$

4.11 Reflect the diamond-shaped polygon whose vertices are $A(-1, 0)$, $B(0, -2)$, $C(1, 0)$, and $D(0, 2)$ about (*a*) the horizontal line $y = 2$, (*b*) the vertical line $x = 2$, and (*c*) the line $y = x + 2$.

SOLUTION

We represent the vertices of the polygon by the homogeneous coordinate matrix

$$V = \begin{pmatrix} -1 & 0 & 1 & 0 \\ 0 & -2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

From Prob. 4.9, the reflection matrix can be written as

$$M_L = T_{\mathbf{v}} \cdot R_\theta \cdot M_x \cdot R_{-\theta} \cdot T_{-\mathbf{v}}$$

(*a*) The line $y = 2$ has y intercept $(0, 2)$ and makes an angle of $0°$ with the x axis. So with $\theta = 0$ and $\mathbf{v} = 2\mathbf{J}$, the transformation matrix is

$$M_L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 4 \\ 0 & 0 & 1 \end{pmatrix}$$

This same matrix could have been obtained directly by using the results of Prob. 4.10 with slope $m = 0$ and y intercept $b = 2$. To reflect the polygon, we set

$$M_L \cdot V = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 4 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} -1 & 0 & 1 & 0 \\ 0 & -2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix} = \begin{matrix} \\ \\ \\ \end{matrix}\begin{matrix} A' & B' & C' & D' \\ \end{matrix}\begin{pmatrix} -1 & 0 & 1 & 0 \\ 4 & 6 & 4 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Converting from homogeneous coordinates, $A' = (-1, 4)$, $B' = (0, 6)$, $C' = (1, 4)$, and $D' = (0, 2)$.

(*b*) The vertical line $x = 2$ has no y intercept and an infinite slope! We can use M_y, reflection about the y axis, to write the desired reflection by (1) translating the given line two units over to the y axis, (2) reflect about the y axis, and (3) translate back two units. So with $\mathbf{v} = 2\mathbf{I}$,

$$\begin{aligned} M_L &= T_{\mathbf{v}} \cdot M_y \cdot T_{-\mathbf{v}} \\ &= \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

Finally

$$M_L \cdot V = \begin{pmatrix} -1 & 0 & 4 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} -1 & 0 & 1 & 0 \\ 0 & -2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 5 & 4 & 3 & 4 \\ 0 & -2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

or $A' = (5, 0)$, $B' = (4, -2)$, $C' = (3, 0)$, and $D' = (4, 2)$.

(*c*) The line $y = x + 2$ has slope 1 and a y intercept $(0, 2)$. From Prob. 4.10, with $m = 1$ and $b = 2$, we find

$$M_L = \begin{pmatrix} 0 & 1 & -2 \\ 1 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

The required coordinates A', B', C', and D' can now be found.

$$M_L \cdot V = \begin{pmatrix} 0 & 1 & -2 \\ 1 & 0 & 2 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} -1 & 0 & 1 & 0 \\ 0 & -2 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} -2 & -4 & -2 & 0 \\ 1 & 2 & 3 & 2 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

So $A' = (-2, 1)$, $B' = (-4, 2)$, $C' = (-2, 3)$, and $D' = (0, 2)$.

4.12 The matrix $\begin{pmatrix} 1 & a \\ b & 1 \end{pmatrix}$ defines a transformation called a *simultaneous shearing* or *shearing* for short.

The special case when $b = 0$ is called *shearing in the x direction*. When $a = 0$, we have *shearing in the y direction*. Illustrate the effect of these shearing transformations on the square $A(0, 0)$, $B(1, 0)$, $C(1, 1)$, and $D(0, 1)$ when $a = 2$ and $b = 3$.

SOLUTION

Figure 4-16(*a*) shows the original square, Fig. 4-16(*b*) shows shearing in the x direction, Fig. 4-16(*c*) shows shearing in the y direction, and Fig. 4-16(*d*) shows shearing in both directions.

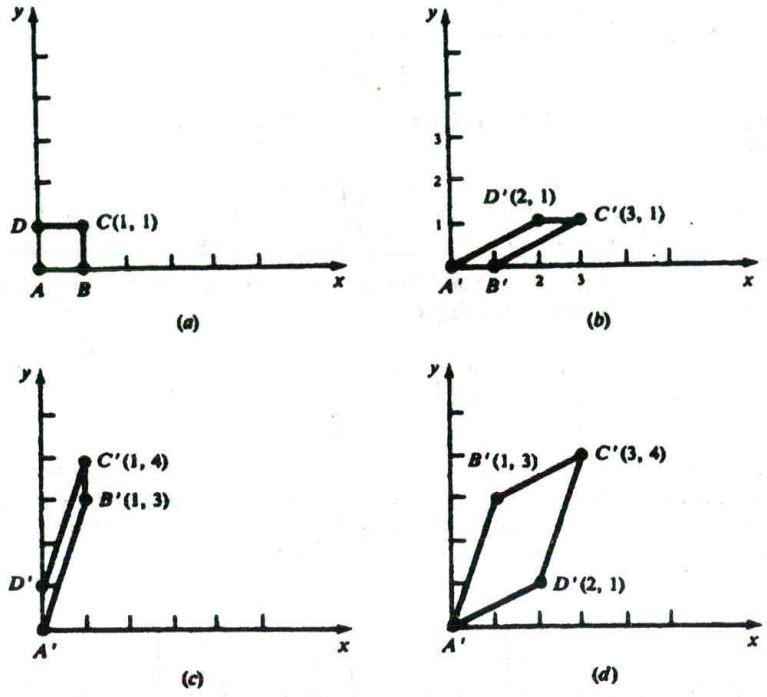


Fig. 4-16

4.13 An observer standing at the origin sees a point $P(1, 1)$. If the point is translated one unit in the direction $\mathbf{v} = \mathbf{I}$, its new coordinate position is $P'(2, 1)$. Suppose instead that the observer stepped back one unit along the x axis. What would be the apparent coordinates of P with respect to the observer?

SOLUTION

The problem can be set up as a transformation of coordinate systems. If we translate the origin O in the direction $\mathbf{v} = -\mathbf{I}$ (to a new position at O') the coordinates of P in this system can be found by the translation $\bar{T}_{\mathbf{v}}$:

$$\bar{T}_{\mathbf{v}} \cdot P = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}$$

So the new coordinates are $(2, 1)'$. This has the following interpretation: a displacement of one unit in a given direction can be achieved by either moving the object forward or stepping back from it.

4.14 An object is defined with respect to a coordinate system whose units are measured in feet. If an observer's coordinate system uses inches as the basic unit, what is the coordinate transformation used to describe object coordinates in the observer's coordinate system?

SOLUTION

Since there are 12 inches to a foot, the required transformation can be described by a coordinate scaling transformation with $s = \frac{1}{12}$ or

$$\bar{S}_{1/12} = \begin{pmatrix} \frac{1}{1/12} & 0 \\ 0 & \frac{1}{1/12} \end{pmatrix} = \begin{pmatrix} 12 & 0 \\ 0 & 12 \end{pmatrix}$$

and so

$$\bar{S}_{1/12} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 12 & 0 \\ 0 & 12 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 12x \\ 12y \end{pmatrix}$$

4.15 Find the equation of the circle $(x')^2 + (y')^2 = 1$ in terms of xy coordinates, assuming that the $x'y'$ coordinate system results from a scaling of a units in the x direction and b units in the y direction.

SOLUTION

From the equations for a coordinate scaling transformation, we find

$$x' = \frac{1}{a}x \qquad y' = \frac{1}{b}y$$

Substituting, we have

$$\left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 = 1$$

Notice that as a result of scaling, the equation of the circle is transformed to the equation of an ellipse in the xy coordinate system.

4.16 Find the equation of the line $y' = mx' + b$ in xy coordinates if the $x'y'$ coordinate system results from a 90° rotation of the xy coordinate system.

SOLUTION

The rotation coordinate transformation equations can be written as

$$x' = x\cos(90°) + y\sin(90°) = y \qquad y' = -x\sin(90°) + y\cos(90°) = -x$$

Substituting, we find $-x = my + b$. Solving for y, we have $y = (-1/m)x - b/m$.

4.17 Find the instance transformation which places a half-size copy of the square $A(0, 0)$, $B(1, 0)$, $C(1, 1)$, $D(0, 1)$ [Fig. 4-17(a)] into a master picture coordinate system so that the center of the square is at $(-1, -1)$ [Fig. 4-17(b)].

SOLUTION

The center of the square $ABCD$ is at $P(\frac{1}{2}, \frac{1}{2})$. We shall first apply a scaling transformation while keeping P fixed (see Prob. 4.7). Then we shall apply a translation that moves the center P to $P'(-1, -1)$. Taking $t_x = (-1) - (\frac{1}{2}) = -\frac{3}{2}$ and similarly $t_y = -\frac{3}{2}$ (so $\mathbf{v} = -\frac{3}{2}\mathbf{I} - \frac{3}{2}\mathbf{J}$), we obtain

$$N_{\text{picture,square}} = T_{\mathbf{v}} \cdot S_{1/2,1/2,P} = \begin{pmatrix} 1 & 0 & -\frac{3}{2} \\ 0 & 1 & -\frac{3}{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{4} \\ 0 & \frac{1}{2} & \frac{1}{4} \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 & -\frac{5}{4} \\ 0 & \frac{1}{2} & -\frac{5}{4} \\ 0 & 0 & 1 \end{pmatrix}$$

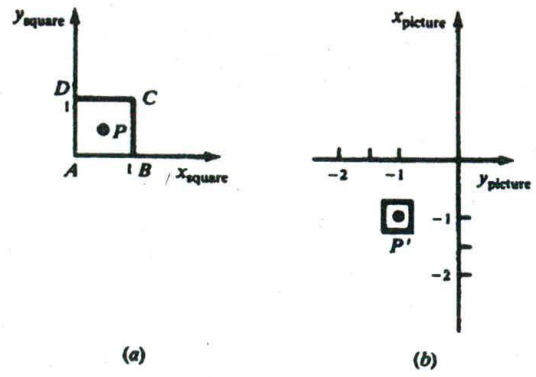


Fig. 4-17

4.18 Write the composite transformation that creates the design in Fig. 4-19 from the symbols in Fig. 4-18.

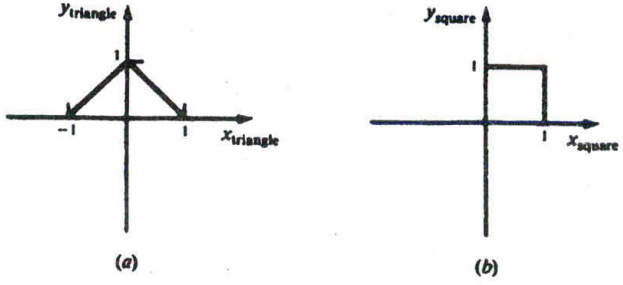


Fig. 4-18

SOLUTION

First we create an instance of the triangle [Fig. 4-18(*a*)] in the square [Fig. 4-18(*b*)]. Since the base of the triangle must be halved while keeping the height fixed at one unit, the appropriate instance transformation is $N_{\text{square,triangle}} = T_{1/2,\mathbf{I}} \cdot S_{1/2,1}$.

The instance transformation needed to place the square at the desired position in the picture coordinate system (Fig. 4-19) is a translation in the direction $\mathbf{v} = \mathbf{I} + \mathbf{J}$:

$$N_{\text{picture,square}} = T_{\mathbf{v}}$$

Then the composite transformation for placing the triangle into the picture is

$$C_{\text{picture,triangle}} = N_{\text{picture,square}} \cdot N_{\text{square,triangle}}$$

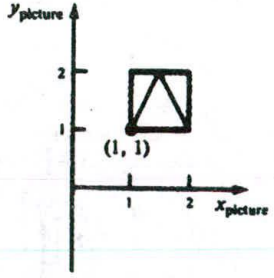


Fig. 4-19

and the composite transformation to place the square into the picture is

$$C_{\text{picture,square}} = N_{\text{picture,square}}$$

Supplementary Problems

4.19 What is the relationship between the rotations R_θ, $R_{-\theta}$, and R_θ^{-1}?

4.20 Describe the transformations used in magnification and reduction with respect to the origin. Find the new coordinates of the triangle $A(0, 0)$, $B(1, 1)$, $C(5, 2)$ after it has been (a) magnified to twice its size and (b) reduced to half its size.

4.21 Show that reflection about the line $y = x$ is attained by reversing coordinates. That is,

$$M_L(x, y) = (y, x)$$

4.22 Show that the order in which transformations are performed is important by the transformation of triangle $A(1, 0)$, $B(0, 1)$, $C(1, 1)$, by (a) rotating 45° about the origin and then translating in the direction of vector $\mathbf{I}$, and (b) translating and then rotating.

4.23 An object point $P(x, y)$ is translated in the direction $\mathbf{v} = a\mathbf{I} + b\mathbf{J}$ and simultaneously an observer moves in the direction $\mathbf{v}$. Show that there is no apparent motion (from the point of view of the observer) of the object point.

4.24 Assuming that we have a mathematical equation defining a curve in $x'y'$ coordinates, and the $x'y'$ coordinate system is the result of a coordinate transformation from the xy coordinate system, write the equation in terms of xy coordinates.

4.25 Show that

$$T_{\mathbf{V}_1} \cdot T_{\mathbf{V}_2} = T_{\mathbf{V}_2} \cdot T_{\mathbf{V}_1} = T_{\mathbf{V}_1 + \mathbf{V}_2}$$

4.26 Show that $S_{a,b} \cdot S_{c,d} = S_{c,d} \cdot S_{a,b} = S_{ac,bd}$.

4.27 Show that $R_\alpha \cdot R_\beta = R_\beta \cdot R_\alpha = R_{\alpha+\beta}$.

4.28 Find the condition under which we have

$$S_{s_x,s_y} \cdot R_\theta = R_\theta \cdot S_{s_x,s_y}$$

4.29 Is a simultaneous shearing the same as a shearing in one direction followed by a shearing in another direction? Why?

4.30 Find the condition under which we can switch the order of a rotation and a simultaneous shearing and still get the same result.

4.31 Express a simultaneous shearing in terms of rotation and scaling transformations.

4.32 Express R_θ in terms of shearing and scaling transformations.

4.33 Express R_θ in terms of shearing transformations.

4.34 Prove that the 2D composite transformation matrix is always in the following form:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{pmatrix}$$

4.35 Consider a line from P_1 to P_2 and an arbitrary point P on the line. Prove that for any given composite transformation the transformed P is on the line between the transformations of P_1 and P_2.