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.

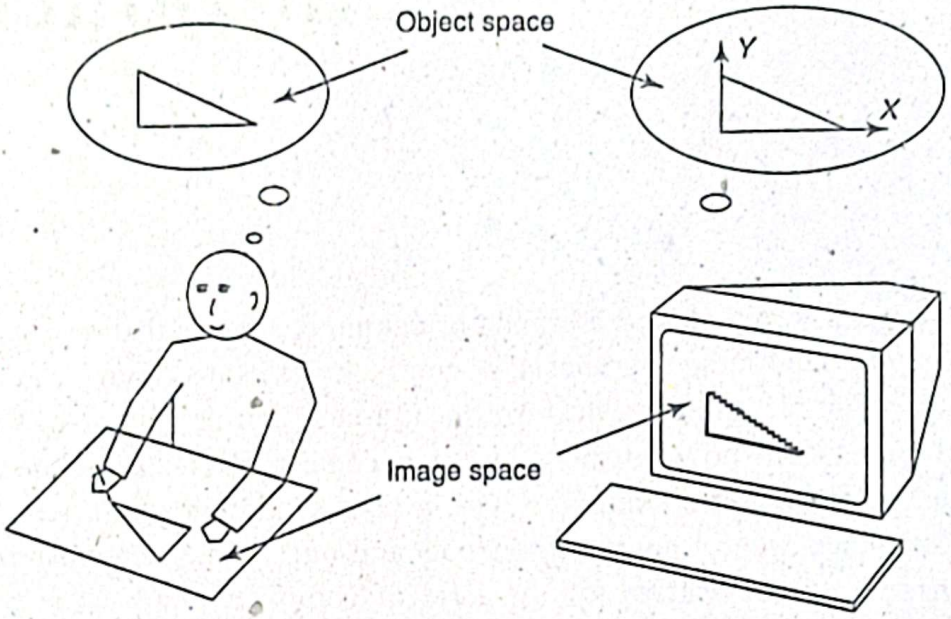

Fig. 1.1 *Drawing a Triangle*

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. Furthe ore, by using multiple

settings for the 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 Figure 1.2 serves as an intermediary between the object coordinate system on the 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.

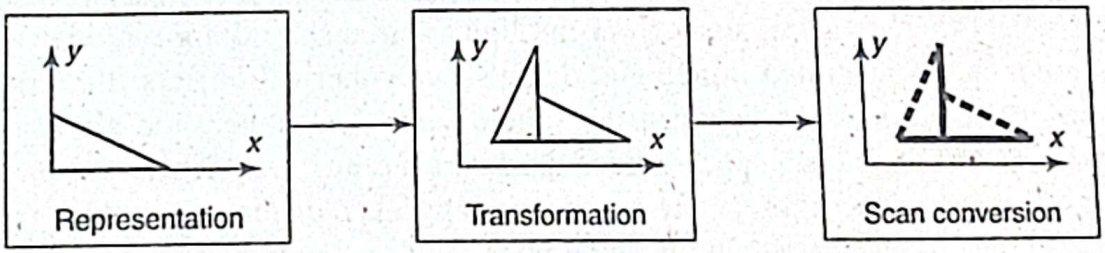

Fig. 1.2 *A Simple Graphics Pipeline*

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.

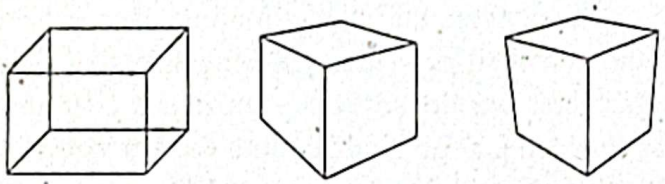

Fig. 1.3 *Several Ways to Depict a Cube*

Did you notice that each drawing in Figure 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 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 even 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

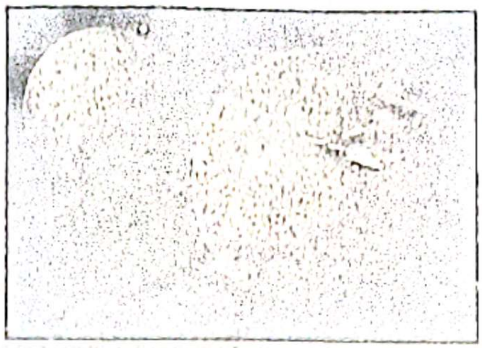

Fig. 1.4 *Two Shaded Spheres*

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 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.

User Interface

While the main focus of computer graphics is the production of images, the field of user interface 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 OVERVIEW OF 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 62/3l 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 × 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 × 41/2 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), the pixel at the upper left corner would have coordinates (0, 479), and the pixel at the upper right corner would have coordinates (639, 479).

1.3 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 detailed discussion on this topic later in another chapter.

Figure 1.5 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

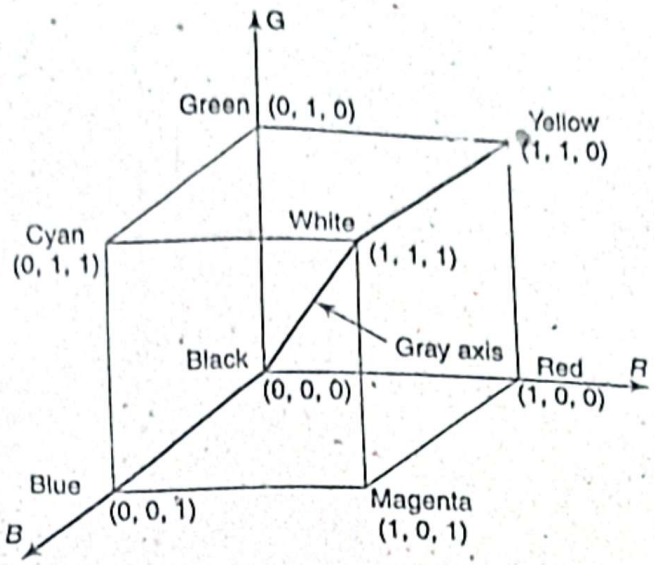

Fig. 1.5 *The RGB Color Space*

(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.

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 Section 1.6). 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 Section 1.7).

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 1.6 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 formulae summarize the conversion between the two color models:

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

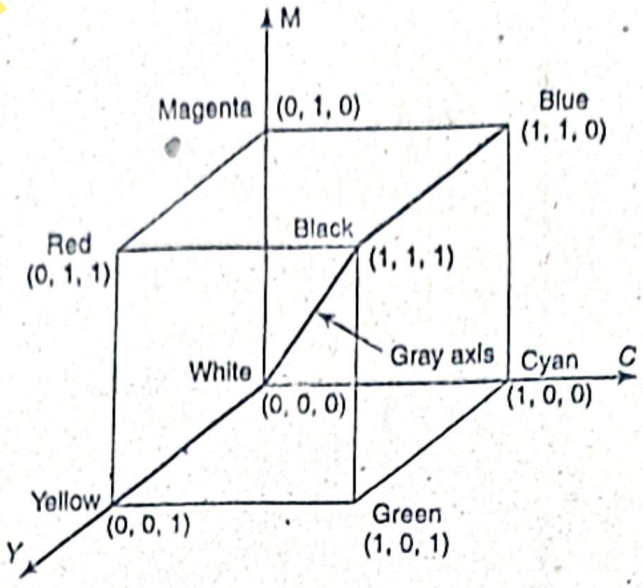

Fig. 1.6 *The CMY Color Space*

1.4 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 Table 1.1). 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.

Table 1.1 *Direct Coding of Colors using 3 bits*

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

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 $256 \times 256 \times 256$ or 16.7 million possible choices. This 24-bit format is commonly referred to

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 (bi-level) 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.

1.5 LOOKUP TABLE

Image representation using a lookup table can be viewed as a compromise between the desire to have a lower storage requirement and the 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 1.7 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 $0 < i < 255$, is determined by the color value in the

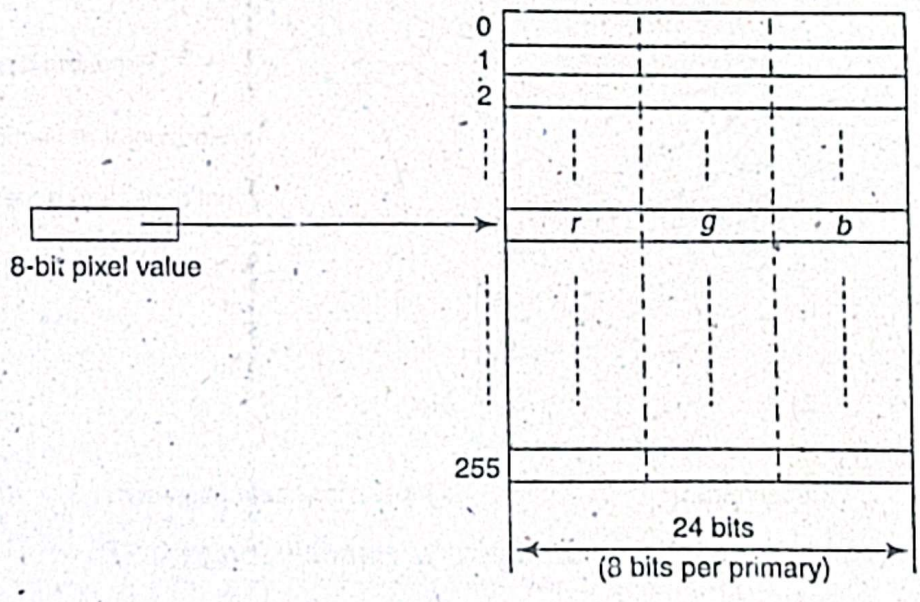

Fig. 1.7 *A 24-bit 256-entry Lookup Table*

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.

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.

1.6 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. 1.8). 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 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*.

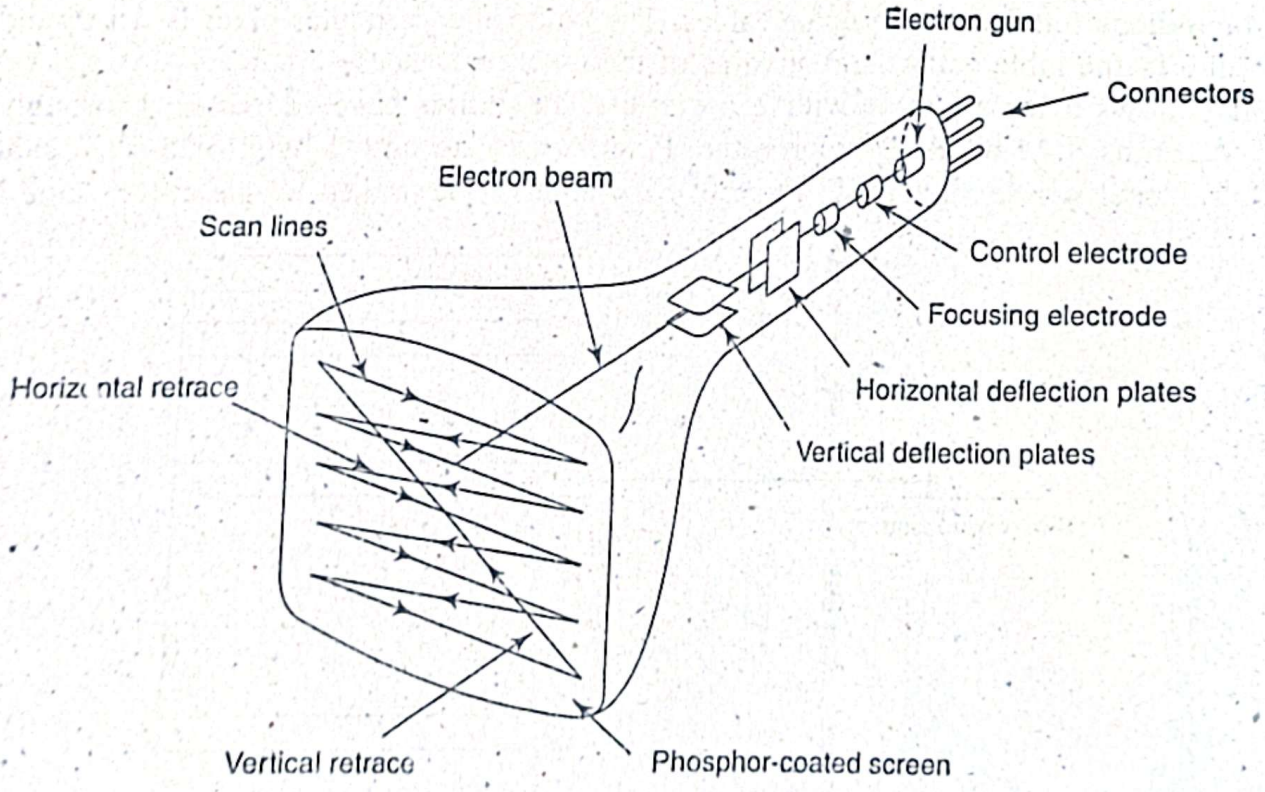

Fig. 1.8 *Anatomy of a Monochromatic CRT*

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 Figure 1.9), 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.

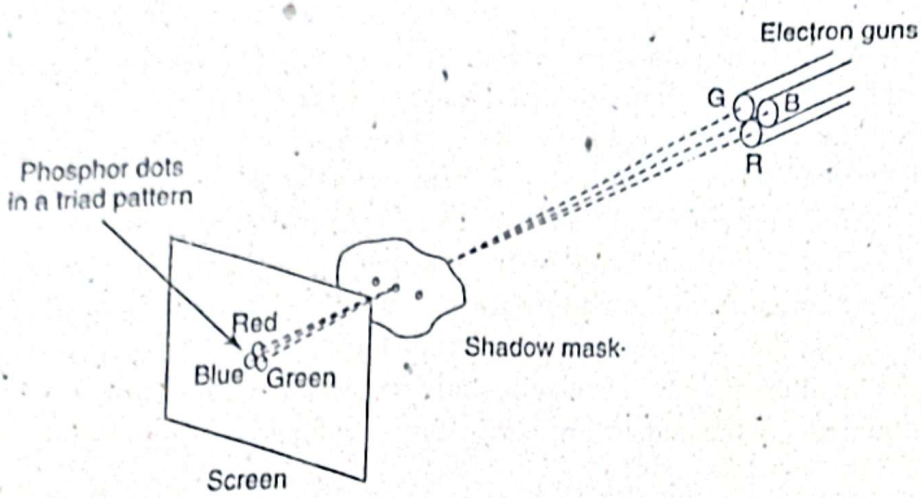

Fig. 1.9 *Color CRT using a Shadow Mask*

1.7 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 Section 1.3). 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. 1.10 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. 1.10 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 (dpi), whereas book and magazine halftones use 120 to 200 dots per inch.

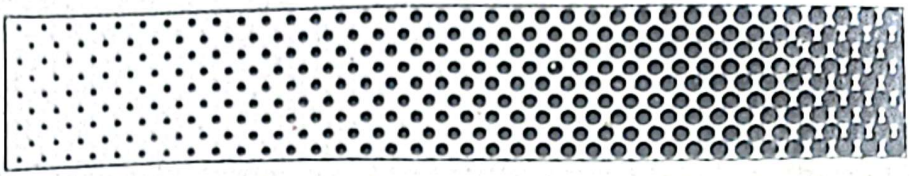

Fig. 1.10 *A Halftone Stripe*

Halftone Approximation

Instead of changing dot size we can approximate the halftone technique using pixel-grid patterns. For example, with a 2 × 2 bi-level pixel grid we can construct five grid patterns to produce five overall intensity levels (see Fig. 1.11). 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. 1.11 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. 1.12).

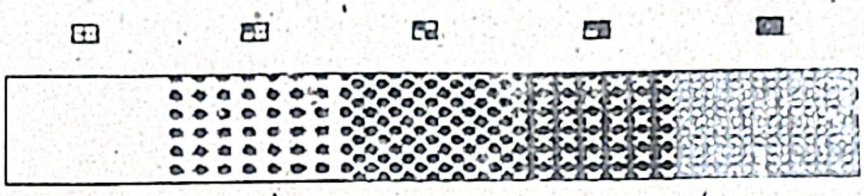

Fig. 1.11 *Halftone Approximation*

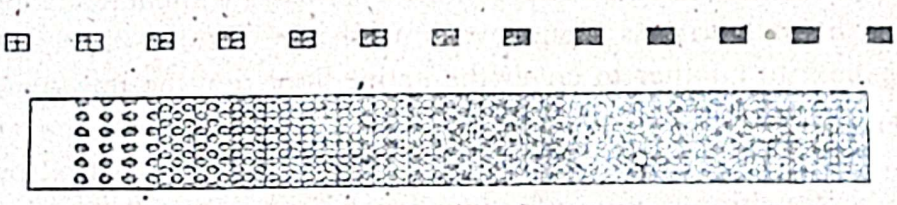

Fig. 1.12 *Halftone Approximation with 13 Intensity Levels*

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 bi-level 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 denned 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. 1.9). 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 that 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:

$$S(x + 1, y) = S(x + 1, y) + ae$$
$$S(x - 1, y - 1) = S(x - 1, y - 1) + be$$
$$S(x, y - 1) = S(x, y - 1) + ce$$
$$S(x + 1, y - 1) = S(x + 1, y - 1) + de$$

where parameters a through d often take values 7/16, 3/16, 5/16, and 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 bi-level 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 $96 \times 7/16 = 42$ 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).

1.8 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 1.13 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

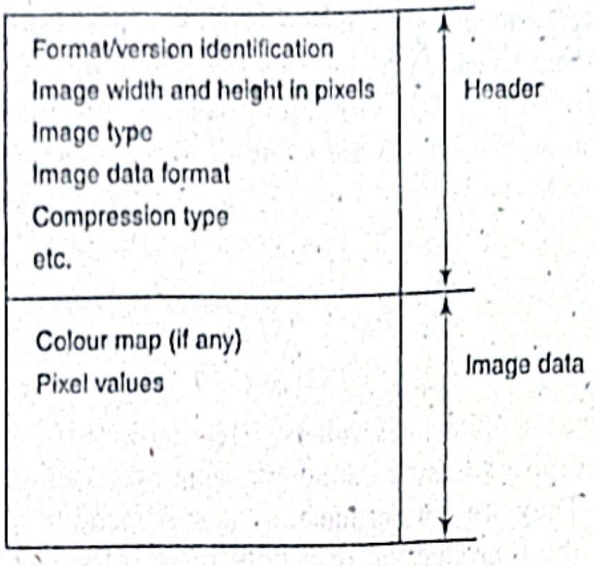

Fig. 1.13 *Typical Image File Format*

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 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.

1.9 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 in array 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.

1.10 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 $x_1 = x_0^2 + z$, $x_2 = x_1^2 + z$, ..., $x_{n+1} = x^2_n + z$, ... If $|x|$ goes towards infinity when n increases, then z is not a member. Otherwise, z belongs to the Mandelbrot set.

Figure 1.14 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

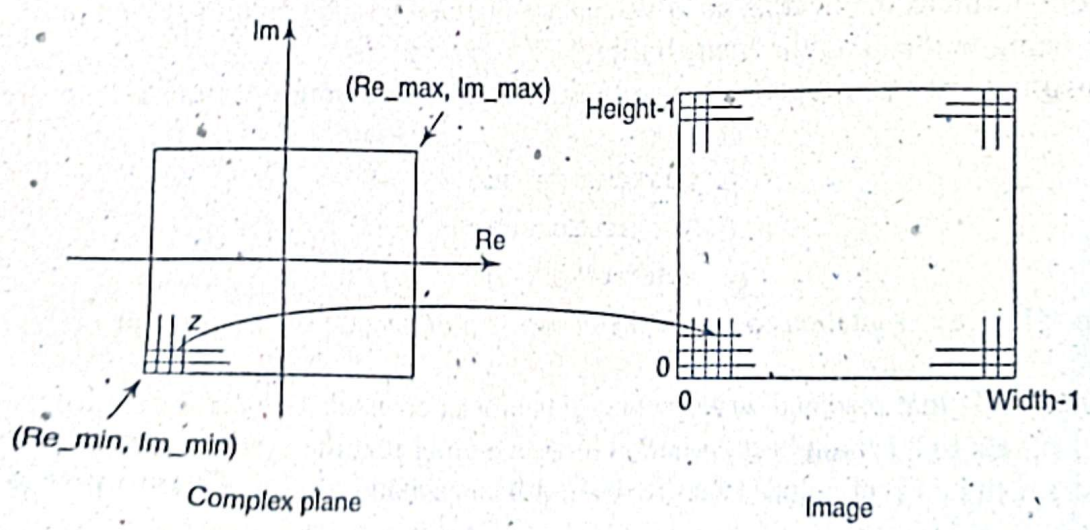

Fig. 1.14 Visualizing the Mandelbrot Set

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 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 l[illegible]an or equal to 2 after a preset maximum number of iterations, we simply set the pixel value t[illegible]lack).

The following pseudo-code implements [illegible] we have discussed in the above paragraphs. We use TV 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;y < 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. 1.15 shows what is nicknamed the Mandelbrot bug. It visualizes an area where $-2.0 < z.real < 0.5$ and $-1.25 < z.imag < 1.25$ 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 takes 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.

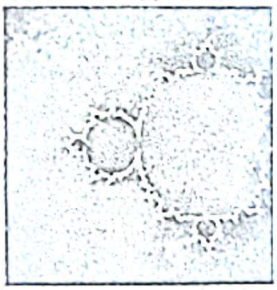

Fig. 1.15 *The Mandelbrot Set*

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. 1.16 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 $-1.2 < x_o \cdot real < 1.2$, $-1.2 < x_o \cdot imag < 1.2$, and $N = 128$.

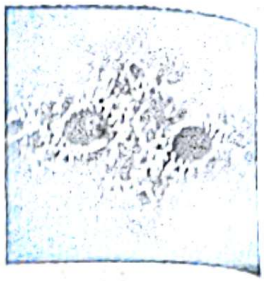

Fig. 1.16 A Julia Set

1.11 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 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.

SOLVED PROBLEMS

1.1 What is the resolution of an image?

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

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

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

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

512/2 or 256 pixels per inch.

1.4 What is an image's aspect ratio?

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

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

width = 1.5 × height = 1.5 × 2 = 3 inches.

1.6 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?

height = 640 × 768/1024 = 480.

1.7 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?

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

1.8 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)?

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

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

(1 − 0.2, 1 − 1, 1 − 0.5) = (0.8, 0, 0.5).

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

(1 − 0.15, 1 − 0.75, 1 − 0) = (0.85, 0.25, 1).

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

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

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

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

1.13 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?

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

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

$2^{12} = 4096$.

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

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

1.16 State whether the following statement is 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.

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.

1.17 What is persistence?

The duration of phosphorescence exhibited by a phosphor.

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

Regulate the intensity of the electron beam.

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

Electrostatic deflection and magnetic deflection.

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

Horizontal retrace.

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

Vertical retrace.

1.22 What is the pitch of a color CRT?

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

1.23 Why do many color printers use black pigment?

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

1.24 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.

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).

1.25 Represent the grid patterns in Fig. 1.11 with a dither matrix.

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

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

$$S(x+1, y) = S(x+1, y) + ae$$
$$S(x-1, y-1) = S(x-1, y-1) + be$$
$$S(x, y-1) = S(x, y-1) + ce$$
$$S(x+1, y-1) = S(x+1, y-1) + de$$

1.27 What is RLE?

RLE stands for Run-Length Encoding, a technique used for image data compression.

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

"8888888884555"

1.29 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?

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

1.30 What if the image in Problem 1.29 is stored in left-to-right and top-to-bottom order?

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

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

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

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

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

1.33 Develop a pseudo-code segment to draw a rectangular area *of wxh* (in number of pi that starts at (x, y) using color *rgb*.

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

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

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

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

```
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);
```

1.36 What if the image in Problem 1.35 is in the 24-bit true color representation?

```
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);
```

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

$(0.5 + 1.0) + (2.0 + (-1.0))i = 1.5 + 1.0i$

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

1.38 Calculate the square of the two complex numbers in Problem 1.37.

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

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

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

Proportional mapping means that we want

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

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

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

```
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);
```

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

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

SUPPLEMENTARY PROBLEMS

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

1.2 Referring to Supplementary Problem 1.1, what if the original is $5\frac{1}{4} \times 3\frac{1}{2}$ inch?

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

1.4 An RGB color image can be converted to a gray-scale image using the formula $0.299R + 0.587G + 0.114B$ for gray level (see Chapter 11, Section 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.

ANSWERS TO SUPPLEMENTARY PROBLEMS

1.1 No, since there is a change in aspect ratio ($5/3.5 \neq 6/4$).

1.2 Yes, since $5.25/3.5 = 6/4 = 1.5$.

1.3 Present the image at an aspect ratio that is lower than the original.

1.4
```
int i, j, c, rgb[3];
for (j = 0; j < height; j++)
  for (i = 0; i < width; i++) {
    getPixel(i, j, rgb);
    c = 0.299*rgb[0] + 0.587*rgb[1]
0.144*rgb[2];
    setPixel(i, j, c);
  }
```