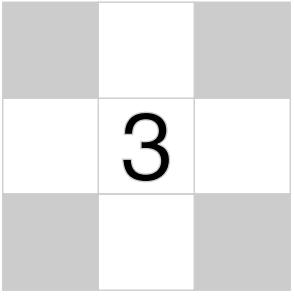

3 Raster Algorithms

Most computer graphics images are presented to the user on a *raster* display. Such systems show images as rectangular arrays of *pixels*, which is short for "picture elements." These pixels are set using RGB (red-green-blue) color. In this chapter, we discuss the basics of raster displays, emphasizing the RGB color system and the non-linearities of standard image display.

3.1 Raster Displays

There are a variety of display technologies for desktop and projected display. These displays vary in *resolution* (the number of pixels) and physical size. Programmers can usually assume that the pixels are laid out in a rectangular array, also called a *raster*.

3.1.1 Pixels

Each displayable element in a raster display is called a *pixel*. Displays usually index pixels by an ordered pair (i, j) indicating the row and column of the pixel. If a display has n_x columns and n_y rows of pixels, the bottom-left element is pixel $(0, 0)$ and the top-right is pixel $(n_x - 1, n_y - 1)$.[1]

[1] In many APIs the rows of an image will be addressed in the less intuitive manner from the top-to-bottom, so the top-left pixel has coordinates $(0, 0)$. This convention is common for historical reasons; it is the order that rows come in a standard television transmission.

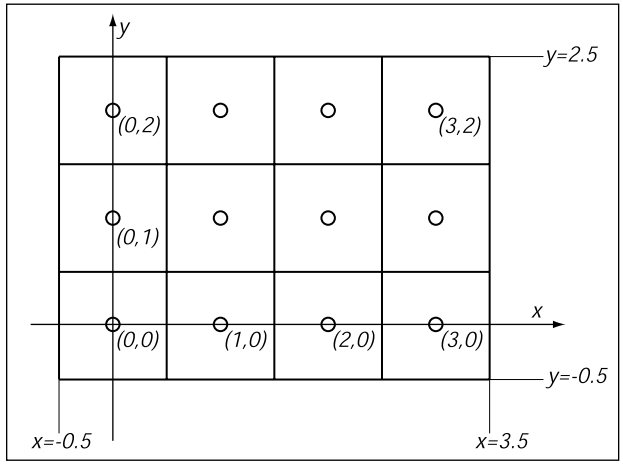

Figure 3.1. Coordinates of a four pixel by three pixel screen. Note that in some APIs the *y*-axis will point downwards.

We need 2D real screen coordinates to specify pixel positions. The details of such systems vary among APIs, but the most common is to use the integer lattice for pixel centers, as shown by the 4 by 3 screen in Figure 3.1. Because pixels have nite extent, note the 0.5 unit overshoot from the pixel centers.

Physical pixels, i.e., the actual displayed elements in hardware, will vary in shape from system to system. In CRTs (cathode ray tubes), the pixel is associated with a patch of phosphor in the CRT, and this phosphor glows based on how much an electron beam stimulates the phosphor. The shape of the pixel depends both on the details of how the electron beam sweeps the pixel, as well as the details of how the phosphor is distributed in the monitor. As a rst approximation, we can assume the phosphor will have a "blobby" shape on the screen, with the highest intensity in the center and a gradual falloff toward the sides of the pixel. On an LCD (liquid crystal display) system, the pixels are approximately square lters which vary their opacity to darken a backlight. These pixels are almost perfect squares, and there is a small gap between squares to allow the control circuitry to get to the pixels. Most display systems other than CRTs or LCDs will behave somewhat like these two, with either blobby or square pixels.

3.2 Monitor Intensities and Gamma

All modern monitors take digital input for the "value" of a pixel and convert this to an intensity level. Real monitors have some non-zero intensity when they are

off because the screen re ects some light. For our purposes we can consider this "black" and the monitor fully on as "white." We assume a numeric description of pixel color that ranges from zero to one. Black is zero, white is one, and a grey halfway between black and white is 0.5. Note that here "halfway" refers to the physical amount of light coming from the pixel, rather than the appearance. The human perception of intensity is non-linear and will not be part of the present discussion.

There are two key issues that must be understood to produce images on monitors. The rst is that monitors are non-linear with respect to input. For example, if you give a monitor 0, 0.5, and 1.0 as inputs for three pixels, the intensities displayed might be 0, 0.25, and 1.0 (i.e., zero, one-quarter fully on, and fully on). As an approximate characterization of this non-linearity, most monitors are characterized by a γ ("gamma") value. This value is the degree of freedom in the formula

$$\text{displayed intensity} = (\text{maximum intensity})a^{\gamma}, \tag{3.1}$$

where a is the input intensity between zero and one. For example, if a monitor has a gamma of 2.0, and we input a value of $a = 0.5$, the displayed intensity will be one fourth the maximum possible intensity because $0.5^2 = 0.25$. Note that $a = 0$ maps to zero intensity and $a = 1$ maps to the maximum intensity regardless of the value of γ. Describing a display's non-linearity using γ is just a rst-order approximation; we do not need a great deal of accuracy in estimating the γ of a device. A nice visual way to gauge the non-linearity is to nd what value of a gives an intensity halfway between black and white. This a will be

$$0.5 = a^{\gamma}.$$

If we can nd that a, we can deduce γ by taking logs of both sides which yields

$$\gamma = \frac{\ln 0.5}{\ln a}.$$

We can nd this a by a standard technique where we display a checkerboard pattern of black and white pixels next to a square of grey pixels with input a (Figure 3.2). When you look at this image from a distance (or without glasses if you are nearsighted), the two sides of the image will look about the same when a is halfway between black and white. This is because the blurred checkerboard is mixing even numbers of white and black pixels so the overall effect is a uniform color halfway between white and black. To make this work, we must be able to try many values for a until one matches. This can be done by giving the user a slider to control a, or by using many different gray squares simultaneously against a

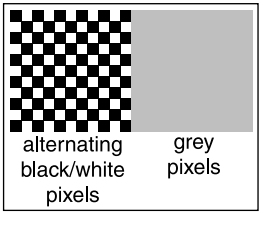

Figure 3.2. Alternating black and white pixels viewed from a distance are halfway between black and white. The gamma of a monitor can be inferred by finding a grey value that appears to have the same intensity as the black and white pattern.

large checkered region. Note that for CRTs, which have difficulty rapidly changing intensity along the horizontal direction, horizontal black and white stripes will work better than a checkerboard. This basic luminance matching strategy can be made more precise by taking advantage of human face recognition ability (Kindlmann, Reinhard, & Creem, 2002).

Once we know γ, we can *gamma correct* our input so that a value of $a = 0.5$ is displayed with intensity halfway between black and white. This is done with the transformation

$$a = a^{\frac{1}{\gamma}}.$$

When this formula is plugged into Equation 3.1 we get

$$\begin{aligned}\text{displayed intensity} &= \left(a^{\frac{1}{\gamma}}\right)^{\gamma} (\text{maximum intensity}) \\ &= a(\text{maximum intensity}).\end{aligned}$$

Another important characteristic of real displays is that they usually take quantized input values. So while we can manipulate intensities in the floating point range $[0, 1]$, the detailed input to a monitor is usually a fixed-size non-negative integer. The most common range for this integer is 0–255 which can be held in 8 bits of storage. This means that the possible values for a are not any number in $[0, 1]$ but instead

$$\text{possible values for } a = \left\{\frac{0}{255}, \frac{1}{255}, \frac{2}{255}, \ldots, \frac{254}{255}, \frac{255}{255}\right\}$$

This means the possible displayed intensity values are approximately

$$\left\{M\left(\frac{0}{255}\right)^{\gamma}, M\left(\frac{1}{255}\right)^{\gamma}, M\left(\frac{2}{255}\right)^{\gamma}, \ldots, M\left(\frac{254}{255}\right)^{\gamma}, M\left(\frac{255}{255}\right)^{\gamma}\right\},$$

where M is the maximum intensity. In applications where the exact intensities need to be controlled, we would have to actually measure the 256 possible intensities, and these intensities might be different at different points on the screen, especially for CRTs. They might also vary with viewing angle. Fortunately few applications require such accurate calibration.

3.3 RGB Color

Most computer graphics images are defined in terms of red-green-blue (RGB) color. RGB color is a simple space that allows straightforward conversion to the controls for most computer screens. In this section RGB color is discussed

from a user's perspective, and operational facility is the goal. A more thorough discussion of color is given in Chapter 20, but the mechanics of RGB color space will allow us to write most graphics programs. The basic idea of RGB color space is that the color is displayed by mixing three *primary* lights: one red, one green, and one blue. The lights mix in an *additive* manner. Additive color mixing is fundamentally different from the more familiar *subtractive* color mixing that governs the mixing of paints and crayons. In those familiar media, red, yellow, and blue are the primaries, and they mix in familiar ways, such as yellow mixed with blue is green. In RGB additive color mixing we have (Figure 3.3):

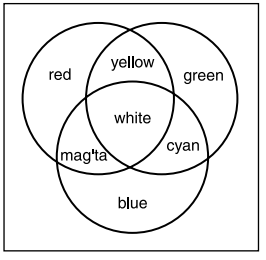

Figure 3.3. The additive mixing rules for colors red/green/blue.

$$\begin{aligned} \text{red} + \text{green} &= \text{yellow} \\ \text{green} + \text{blue} &= \text{cyan} \\ \text{blue} + \text{red} &= \text{magenta} \\ \text{red} + \text{green} + \text{blue} &= \text{white}. \end{aligned}$$

The color "cyan" is a blue-green, and the color "magenta" is a purple.

If we are allowed to dim the primary lights from fully off to fully on, we can create all the colors that can be displayed on an RGB monitor. By convention, we write a color as the fraction of "fully on" it is for each monitor. This creates a three-dimensional *RGB color cube* that has a red, a green, and a blue axis. Allowable coordinates for the axes range from zero to one. The color cube is shown graphically in Figure 3.4.

The RGB coordinates of familiar colors are:

$$\begin{aligned} \text{black} &= (0, 0, 0) \\ \text{red} &= (1, 0, 0) \\ \text{green} &= (0, 1, 0) \\ \text{blue} &= (0, 0, 1) \\ \text{yellow} &= (1, 1, 0) \\ \text{magenta} &= (1, 0, 1) \\ \text{cyan} &= (0, 1, 1) \\ \text{white} &= (1, 1, 1). \end{aligned}$$

Actual RGB levels are often given in quantized form, just like the greyscales discussed in Section 3.2. Each component is speci ed with an integer. The most common size for these integers is one byte each, so each of the three RGB components is an integer between 0 and 255. The three integers together take up three bytes, which is 24 bits. Thus a system that has "24 bit color" has 256 possible levels for each of the three primary colors. Issues of gamma correction discussed in Section 3.2 also apply to each RGB component separately.

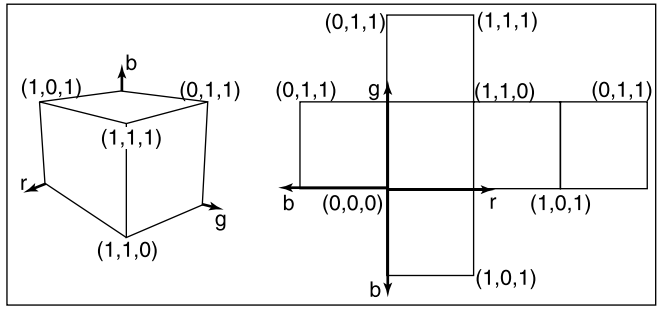

Figure 3.4. The RGB color cube in 3D and its faces unfolded. Any RGB color is a point in the cube. (See also Plate I.)

3.4 The Alpha Channel

Often we would like to only partially overwrite the contents of a pixel. A common example of this occurs in *compositing*, where we have a background and want to insert a foreground image over it. For opaque pixels in the foreground, we just replace the background pixel. For entirely transparent foreground pixels, we do not change the background pixel. For *partially* transparent pixels, some care must be taken. Partially transparent pixels can occur when the foreground object has partially transparent regions such as glass, or when there are sub-pixel holes in

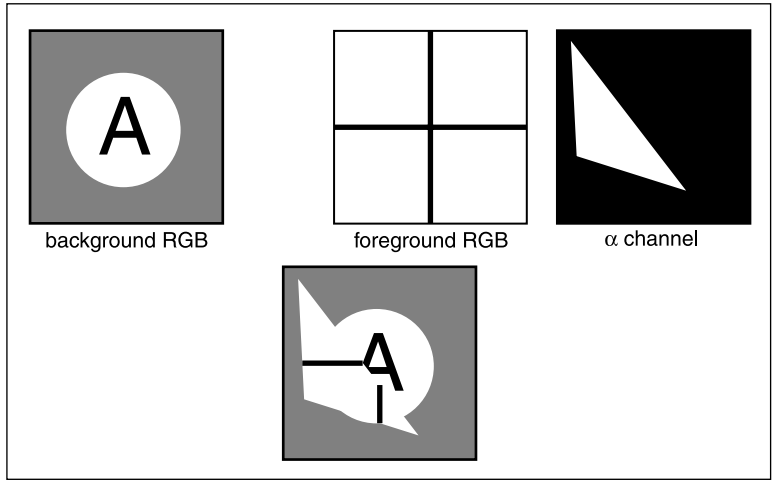

Figure 3.5. An example of compositing using Equation 3.2. The foreground image is in effect cropped by the α channel before being put on top of the background image. The resulting composite is shown on the bottom.

the foreground object such as in the leaves of a distant tree. To blend foreground and background in the case of holes, we want to measure the fraction of the pixel that should be foreground. We can call this fraction α. If we want to composite a foreground color $\mathbf{c}_f$ over background color $\mathbf{c}_b$, and the fraction of the pixel covered by foreground is α, then we can use the formula

$$\mathbf{c} = \alpha\mathbf{c}_f + (1 - \alpha)\mathbf{c}_b. \tag{3.2}$$

An example of using Equation 3.2 is shown in Figure 3.5. Note that the α image might be stored with the RGB image, or it might be stored as a separate greyscale (single channel) image.

Although Equation 3.2 is what is usually used, there are a variety of situations where α is used differently (Porter & Duff, 1984).

3.5 Line Drawing

Most graphics packages contain a line drawing command that takes two endpoints in screen coodinates (Figure 3.1) and draws a line between them. For example, the call for endpoints (1,1) and (3,2) would turn on pixels (1,1) and (3,2) and ll in one pixel between them. For general screen coordinate endpoints (x_0, y_0) and (x_1, y_1), the routine should draw some "reasonable" set of pixels that approximate a line between them. The values x_0, x_1, y_0, y_1 are often restricted to be integers (pixel centers) for simplicity, and because the lines themselves are coarse enough entities that subpixel accuracy is not appropriate. If you are implementing an API which calls for real number endpoint coordinates, rounding them to the nearest integer is usually a reasonable strategy that application programmers are unlikely to notice. Because the endpoint coordinates are integers, care should be taken to understand implicit conversions when these integers interact with oating point variables. Drawing such lines is based on line equations, and we have two types of equations to choose from: implicit and parametric. This section describes the two algorithms that result from these two types of equations.

3.5.1 Line Drawing Using Implicit Line Equations

The most common way to draw lines using implicit equations is the *midpoint* algorithm (Pitteway (1967); Van Aken and Novak (1985)). The midpoint algorithm ends up drawing the same lines as the *Bresenham algorithm* (Bresenham, 1965) but is somewhat more straightforward.

The first thing to do is find the implicit equation for the line as discussed in Section 2.5.2:

$$f(x, y) \equiv (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0. \tag{3.3}$$

We assume that $x_0 \leq x_1$. If that is not true, we swap the points so that it is true. The slope m of the line is given by

$$m = \frac{y_1 - y_0}{x_1 - x_0}.$$

The following discussion assumes $m \in (0, 1]$. Analogous discussions can be derived for $m \in (-\infty, -1]$, $m \in (-1, 0]$, and $m \in (1, \infty)$. The four cases cover all possibilities.

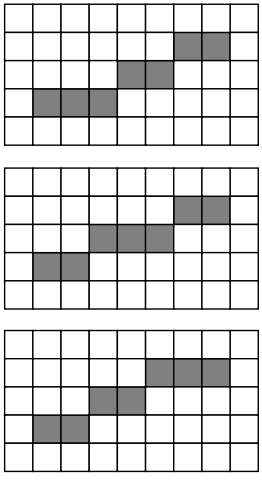

Figure 3.6. Three "reasonable" lines that go seven pixels horizontally and three pixels vertically.

For the case $m \in (0, 1]$, there is more "run" than "rise", i.e., the line is moving faster in x than in y. If we have an API where the y-axis points downwards, we might have a concern about whether this makes the process harder, but, in fact, we can ignore that detail. We can ignore the geometric notions of "up" and "down," because the algebra is exactly the same for the two cases. Cautious readers can confirm that the resulting algorithm works for the y-axis downwards case. The key assumption of the midpoint algorithm is that we draw the thinnest line possible that has no gaps. A diagonal connection between two pixels is not considered a gap.

As the line progresses from the left endpoint to the right, there are only two possibilities: draw a pixel at the same height as the pixel drawn to its left, or draw a pixel one higher. There will always be exactly one pixel in each column of pixels between the endpoints. Zero would imply a gap, and two would be too thick a line. There may be two pixels in the same row for the case we are considering; the line is more horizontal than vertical so sometimes it will go right, and sometimes up. This concept is shown in Figure 3.6, where three "reasonable" lines are shown, each advancing more in the horizontal direction than in the vertical direction.

The midpoint algorithm for $m \in (0, 1]$ first establishes the leftmost pixel and the column number (x-value) of the rightmost pixel and then loops horizontally establishing the row (y-value) of each pixel. The basic form of the algorithm is:

$y = y_0$
for $x = x_0$ to x_1 **do**
  draw(x, y)
  if (some condition) **then**
    $y = y + 1$

Note that x and y are integers. In words this says, "keep drawing pixels from left to right and sometimes move upwards in the y-direction while doing so." The key is to establish efficient ways to make the decision in the *if* statement.

An effective way to make the choice is to look at the *midpoint* of the line between the two potential pixel centers. More speci cally , the pixel just drawn is pixel (x, y) whose center in real screen coordinates is at (x, y). The candidate pixels to be drawn to the right are pixels $(x+1, y)$ and $(x+1, y+1)$.The midpoint between the centers of the two candidate pixels is $(x + 1, y + 0.5)$. If the line passes below this midpoint we draw the bottom pixel, and otherwise we draw the top pixel (Figure 3.7).

To decide whether the line passes above or below $(x+1, y+0.5)$, we evaluate $f(x, y + 0.5)$ in Equation 3.3. Recall from Section 2.5 that $f(x, y) = 0$ for points (x, y) on the line, $f(x, y) > 0$ for points on one side of the line, and $f(x, y) < 0$ for points on the other side of the line. Because $-f(x, y) = 0$ and $f(x, y) = 0$ are both perfectly good equations for the line, it is not immediately clear whether $f(x, y)$ being positive indicates that (x, y) is above the line, or whether it is below. However, we can gure it out; the key term in Equation 3.3 is the y term $(x_1 - x_0)y$. Note that $(x_1 - x_0)$ is de nitely positive because $x_1 > x_0$. This means that as y increases, the term $(x_1 - x_0)y$ gets larger (i.e., more positive or less negative). Thus, the case $f(x, +\infty)$ is de nitely positive, and de nitely above the line, implying points above the line are all positive. Another way to look at it is that the y component of the gradient vector is positive. So above the line, where y can increase arbitrarily, $f(x, y)$ must be positive. This means we can make our code more speci c by lling in the *if* statement:

if $f(x + 1, y + 0.5) < 0$ **then**
  $y = y + 1$

The above code will work nicely for lines of the appropriate slope (i.e., between zero and one). The reader can work out the other three cases which differ only in small details.

If greater ef cienc y is desired, using an *incremental* method can help. An incremental method tries to make a loop more ef cient by reusing computation from the previous step. In the midpoint algorithm as presented, the main computation is the evaluation of $f(x + 1, y + 0.5)$. Note that inside the loop, after the rst iteration, either we already evaluated $f(x - 1, y + 0.5)$ or $f(x - 1, y - 0.5)$ (Figure 3.8). Note also this relationship:

$$f(x + 1, y) = f(x, y) + (y_0 - y_1)$$
$$f(x + 1, y + 1) = f(x, y) + (y_0 - y_1) + (x_1 - x_0).$$

This allows us to write an incremental version of the code:

$y = y_0$
$d = f(x_0 + 1, y_0 + 0.5)$

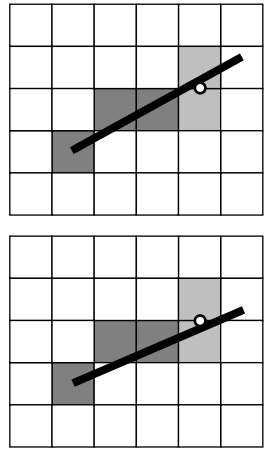

Figure 3.7. Top: the line goes above the midpoint so the top pixel is drawn. Bottom: the line goes below the midpoint so the bottom pixel is drawn.

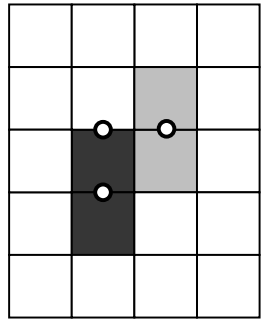

Figure 3.8. When using the decision point shown between the two light grey pixels, we just drew one of the dark grey pixels, so we evaluated *f* at one of the two left points shown.

for $x = x_0$ to x_1 **do**
  draw(x, y)
  if $d < 0$ **then**
    $y = y + 1$
    $d = d + (x_1 - x_0) + (y_0 - y_1)$
  else
    $d = d + (y_0 - y_1)$

This code should run faster since it has little extra setup cost compared to the non-incremental version (that is not always true for incremental algorithms), but it may accumulate more numeric error because the evaluation of $f(x, y + 0.5)$ may be composed of many adds for long lines. However, given that lines are rarely longer than a few thousand pixels, such error is unlikely to be critical. Slightly longer setup cost, but faster loop execution, can be achieved by storing $(x_1 - x_0) + (y_0 - y_1)$ and $(y_0 - y_1)$ as variables. We might hope a good compiler would do that for us, but if the code is critical, it would be wise to examine the results of compilation to make sure.

In some cases, it is faster if an algorithm uses only integer operations. Because we have imposed the constraint that x_0, x_1, y_0, y_1 are all integers, the algorithm above is almost an integer-only algorithm. However, it does require the initialization $d = f(x_0 + 1, y_0 + 0.5)$. Note that this can be expanded as

$$f(x_0 + 1, y_0 + 0.5) = (y_0 - y_1)(x_0 + 1) + (x_1 - x_0)(y_0 + 0.5) + x_0y_1 - x_1y_0.$$

The $y_0 + 0.5$ is not an integer operation and results in a non-integer multiplier. But we can x this: if $f(x, y) = 0$ is the equation of the line, then $2f(x, y) = 0$ is also a valid equation for the same line. So if we use $2f(x, y)$ instead of $f(x, y)$, the expression for d becomes

$$\begin{aligned} 2f(x_0 + 1, y_0 + 0.5) = {} & 2(y_0 - y_1)(x_0 + 1) + (x_1 - x_0)(2y_0 + 1) \\ & + 2x_0y_1 - 2x_1y_0, \end{aligned}$$

which has all integer terms. The resulting code is:

$y = y_0$
$d = 2(y_0 - y_1)(x_0 + 1) + (x_1 - x_0)(2y_0 + 1) + 2x_0y_1 - 2x_1y_0$
for $x = x_0$ to x_1 **do**
  draw(x, y)
  if $d < 0$ **then**
    $y = y + 1$
    $d = d + 2(x_1 - x_0) + 2(y_0 - y_1)$
  else
    $d = d + 2(y_0 - y_1)$

The careful reader will note that before we can execute the all-integer code above, we must check whether $m \in [0, 1)$, and explicitly computing m requires a divide. However, the following code, which is equivalent to the explicit slope check, can be used:

$$((y_1 \geq y_0) \text{ and } (x_1 - x_0 > y_1 - y_0)) \quad \equiv \quad (m \in [0, 1))$$

3.5.2 Line Drawing Using Parametric Line Equations

As derived in Section 2.6.1 a parametric line in 2D that goes through points $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$ can be written

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_0 + t(x_1 - x_0) \\ y_0 + t(y_1 - y_0) \end{bmatrix},$$

or equivalently in vector form,

$$\mathbf{p}(t) = \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0).$$

If the slope of the line $m \in [-1, 1]$, then the "for" loop can advance in x, and we get remarkably compact code. A similar algorithm results for m outside $[-1, 1]$, so only two cases are needed. Because t progresses constantly along the distance of the line, so do the x and y components of the line; thus, we can compute t as a function of x:

$$t = \frac{x - x_0}{x_1 - x_0}.$$

The resulting code is:

for $x = x_0$ to x_1 **do**
  $t = (x - x_0)/(x_1 - x_0)$
  $y = y_0 + t(y_1 - y_0)$
  draw$(x, \text{round}(y))$

A nice property of this algorithm is that it works whether or not $x_1 > x_0$. So the code for parametric lines turns out to be very simple. However, it cannot be made integer-only as we shall see. In many computer languages, conversion from float to integer is implemented as truncation, so the term round(y) can be implemented as an integer conversion of $(y + 0.5)$.

This code can also be made incremental because t and therefore y change by a constant amount in each iteration:

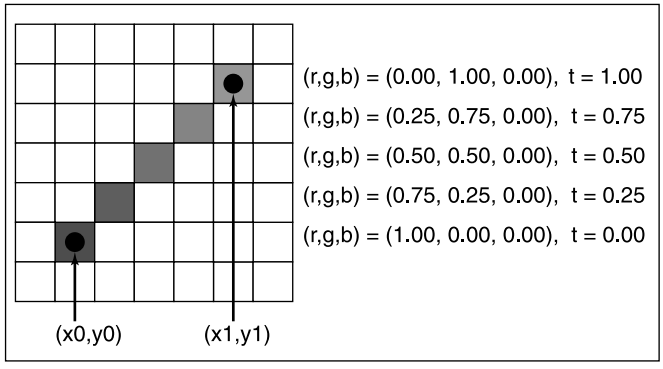

Figure 3.9. A colored line switching from red to green. The middle pixel is half red and half green which is a "dark yellow". (See also Plate II.)

$\Delta y = (y_1 - y_0)/(x_1 - x_0)$
$y = y_0$
for $x = x_0$ to x_1 **do**
  draw(x,round(y))
  $y = y + \Delta y$

Sometimes lines are speci ed with RGB colors $\mathbf{c}_0$ and $\mathbf{c}_1$ at either end, and we would like to change the color smoothly along the line. If we can parameterize the line segment in terms of a $t \in [0, 1]$, we can use the formula

$$\mathbf{c} = (1 - t)\mathbf{c}_0 + t\mathbf{c}_1.$$

This allows us to compute a color at each pixel. An example of such a colored line which shifts from red to green is shown in Figure 3.9. Note that the expression for $\mathbf{c}$ and for t can also be computed incrementally if desired. The code for this is:

$\Delta y = (y_1 - y_0)/(x_1 - x_0)$
$\Delta r = (r_1 - r_0)/(x_1 - x_0)$
$\Delta g = (g_1 - g_0)/(x_1 - x_0)$
$\Delta b = (b_1 - b_0)/(x_1 - x_0)$
$y = y_0$, $r = r_0$, $g = g_0$, $b = b_0$
for $x = x_0$ to x_1 **do**
  draw(x,round(y), r, g, b)
  $y = y + \Delta y$
  $r = r + \Delta r$
  $g = g + \Delta g$
  $b = b + \Delta b$

A similar change can be made to the midpoint (implicit) algorithm, but it would be dif cult to do using only integer operations. In infrastructures where oating point division is expensive, the four divides above can be replaced by one divide and four multiplies.

3.6 Triangle Rasterization

We often want to draw a 2D triangle with 2D points $\mathbf{p}_0 = (x_0, y_0)$, $\mathbf{p}_1 = (x_1, y_1)$, and $\mathbf{p}_2 = (x_2, y_2)$ in screen coordinates. This is similar to the line drawing problem, but it has some of its own subtleties. It will turn out that there is no advantage to integer coordinates for endpoints, so we will allow the (x_i, y_i) to have oating point values. As with line drawing, we may wish to interpolate color or other properties from values at the vertices. This is straightforward if we have the barycentric coordinates (Section 2.11). For example, if the vertices have colors $\mathbf{c}_0$, $\mathbf{c}_1$, and $\mathbf{c}_2$, the color at a point in the triangle with barycentric coordinates (α, β, γ) is

$$\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2.$$

This type of interpolation of color is known in graphics as *Gouraud* interpolation after its inventor (Gouraud, 1971).

Another subtlety of rasterizing triangles is that we are usually rasterizing triangles that share vertices and edges. This means we would like to rasterize adjacent triangles so there are no holes. We could do this by using the midpoint algorithm to draw the outline of each triangle and then ll in the interior pixels. This would mean adjacent triangles both draw the same pixels along each edge. If the adjacent triangles have different colors, the image will depend on the order in which the two triangles are drawn. The most common way to rasterize triangles that avoids the order problem and eliminates holes is to use the convention that pixels are drawn if and only if their centers are inside the triangle, i.e., the barycentric coordinates of the pixel center are all in the interval $(0, 1)$. This raises the issue of what to do if the center is exactly on the edge of the triangle. There are several ways to handle this as will be discussed later in this section. The key observation is that barycentric coordinates allow us to decide whether to draw a pixel and what color that pixel should be if we are interpolating colors from the vertices. So our problem of rasterizing the triangle boils down to ef ciently nding the barycentric coordinates of pixel centers (Pineda, 1988). The brute-force rasterization algorithm is:

for all x **do**
  for all y **do**

compute (α, β, γ) for (x, y)
if ($\alpha \in [0,1]$ and $\beta \in [0,1]$ and $\gamma \in [0,1]$) **then**
$\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
drawpixel (x, y) with color $\mathbf{c}$

The rest of the algorithm limits the outer loops to a smaller set of candidate pixels and makes the barycentric computation ef cient.

We can add a simple ef cienc y by nding the bounding rectangle of the three vertices and only looping over this rectangle for candidate pixels to draw. We can compute barycentric coordinates using Equation 2.32. This yields the algorithm:

$x_{\text{min}} =$ oor (x_i)
$x_{\text{max}} =$ ceiling (x_i)
$y_{\text{min}} =$ oor (y_i)
$y_{\text{max}} =$ ceiling (y_i)
for $y = y_{\text{min}}$ to y_{max} **do**
for $x = x_{\text{min}}$ to x_{max} **do**
$\alpha = f_{12}(x, y)/f_{12}(x_0, y_0)$
$\beta = f_{20}(x, y)/f_{20}(x_1, y_1)$
$\gamma = f_{01}(x, y)/f_{01}(x_2, y_2)$
if ($\alpha > 0$ and $\beta > 0$ and $\gamma > 0$) **then**
$\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
drawpixel (x, y) with color $\mathbf{c}$

Here f_{ij} is the line given by Equation 3.3 with the appropriate vertices:

$$f_{01}(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0,$$
$$f_{12}(x, y) = (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1,$$
$$f_{20}(x, y) = (y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2.$$

Note that we have exchanged the test $\alpha \in (0, 1)$ with $\alpha > 0$ etc., because if all of α, β, γ are positive, then we know they are all less than one because $\alpha + \beta + \gamma = 1$. We could also compute only two of the three barycentric variables and get the third from that relation, but it is not clear that this saves computation once the algorithm is made incremental, which is possible as in the line drawing algorithms; each of the computations of α, β, and γ does an evaluation of the form $f(x, y) = Ax + By + C$. In the inner loop, only x changes, and it changes by one. Note that $f(x + 1, y) = f(x, y) + A$. This is the basis of the incremental algorithm. In the outer loop, the evaluation changes for $f(x, y)$ to $f(x, y + 1)$, so a similar ef cienc y can be achieved. Because α, β, and γ change by constant

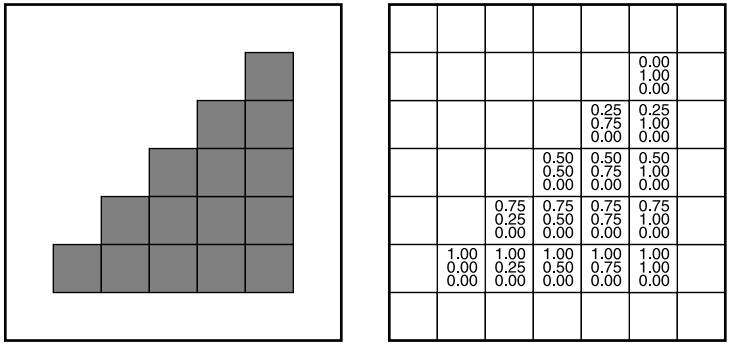

Figure 3.10. A colored triangle with barycentric interpolation. Note that the changes in color components are linear in each row and column as well as along each edge. In fact it is constant along every line, such as the diagonals, as well. (See also Plate III.)

increments in the loop, so does the color $\mathbf{c}$. So this can be made incremental as well. For example, the red value for pixel $(x + 1, y)$ differs from the red value for pixel (x, y) by a constant amount that can be precomputed. An example of a triangle with color interpolation is shown in Figure 3.10.

3.6.1 Dealing With Pixels on Triangle Edges

We have still not discussed what to do for pixels whose centers are exactly on the edge of a triangle. If a pixel is exactly on the edge of a triangle, then it is also on the edge of the adjacent triangle if there is one. There is no obvious way to award the pixel to one triangle or the other. The worst decision would be to not draw the pixel because a hole would result between the two triangles. Better, but still not good, would be to have both triangles draw the pixel. If the triangles are transparent, this will result in a double-coloring. We would really like to award the pixel to exactly one of the triangles, and we would like this process to be simple; which triangle is chosen does not matter as long as the choice is well de ned.

One approach is to note that any off-screen point is de nitely on exactly one side of the shared edge and that is the edge we will draw. For two non-overlapping triangles, the vertices not on the edge are on opposite sides of the edge from each other. Exactly one of these vertices will be on the same side of the edge as the off-screen point (Figure 3.11). This is the basis of the test. The test if numbers p and q have the same sign can be implemented as the test $pq > 0$, which is very ef cient in most environments.

Note that the test is not perfect because the line through the edge may also go through the offscreen point, but we have at least greatly reduced the number

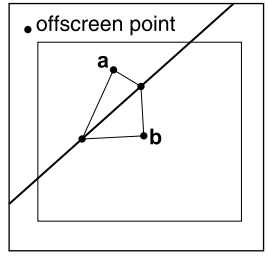

Figure 3.11. The off-screen point will be on one side of the triangle edge or the other. Exactly one of the non-shared vertices **a** and **b** will be on the same side.

of problematic cases. Which off-screen point is used is arbitrary, and $(x, y) = (-1, -1)$ is as good a choice as any. We will need to add a check for the case of a point exactly on an edge. We would like this check not to be reached for common cases, which are the completely inside or outside tests. This suggests:

$x_{\text{min}} =$ oor (x_i)
$x_{\text{max}} =$ ceiling (x_i)
$y_{\text{min}} =$ oor (y_i)
$y_{\text{max}} =$ ceiling (y_i)
$f_\alpha = f_{12}(x_0, y_0)$
$f_\beta = f_{20}(x_1, y_1)$
$f_\gamma = f_{01}(x_2, y_2)$
for $y = y_{\text{min}}$ to y_{max} **do**
  for $x = x_{\text{min}}$ to x_{max} **do**
    $\alpha = f_{12}(x, y)/f_\alpha$
    $\beta = f_{20}(x, y)/f_\beta$
    $\gamma = f_{01}(x, y)/f_\gamma$
    if ($\alpha \geq 0$ and $\beta \geq 0$ and $\gamma \geq 0$) **then**
      if ($\alpha > 0$ or $f_\alpha f_{12}(-1, -1) > 0$) and ($\beta > 0$ or $f_\beta f_{20}(-1, -1) > 0$) and ($\gamma > 0$ or $f_\gamma f_{01}(-1, -1) > 0$) **then**
        $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
        drawpixel (x, y) with color $\mathbf{c}$

We might expect that the above code would work to eliminate holes and double-draws only if we use exactly the same line equation for both triangles. In fact, the line equation is the same only if the two shared vertices have the same order in the draw call for each triangle. Otherwise the equation might ip in sign. This could be a problem depending on whether the compiler changes the order of operations. So if a robust implementation is needed, the details of the compiler and arithmetic unit may need to be examined. The rst four lines in the pseudocode above must be coded carefully to handle cases where the edge exactly hits the pixel center.

In addition to being amenable to an incremental implementation, there are several potential early exit points. For example, if α is negative, there is no need to compute β or γ. While this may well result in a speed improvement, pro ling is always a good idea; the extra branches could reduce pipelining or concurrency and might slow down the code. So as always, test any attractive looking optimizations if the code is a critical section.

Another detail of the above code is that the divisions could be divisions by zero for degenerate triangles, i.e., if $f_\gamma = 0$. Either the oating point error conditions should be accounted for properly, or another test will be needed.

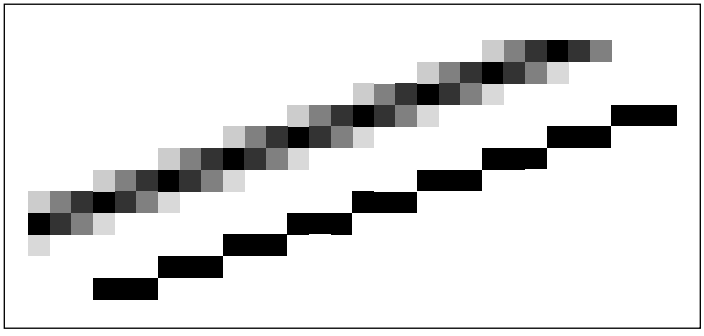

Figure 3.12. An antialiased and a jaggy line viewed at close range so individual pixels are visible.

3.7 Simple Antialiasing

One problem with the line and triangle drawing algorithms presented earlier is that they have fairly jaggy appearances. We can lessen this visual artifact by allowing a pixel to be "partially" on (Crow, 1978). For example, if the center of a pixel is *almost* inside a black triangle on a white background, we can color it halfway between white and black. The top line in the gure is drawn this way. In practice this form of blurring helps visual quality, especially in animations. This is shown as the bottom line of Figure 3.12.

The most straightforward way to create such "unjaggy" images is to use a *box filter*, where the pixel is set to the average color of the regions inside it. This means we have to think of all drawable entities as having well-de ned areas. For example, a line is just a rectangle as shown in Figure 3.13. More sophisticated methods of blurring for visual quality are discussed in Chapter 4. Jaggy artifacts, like the sawtoothed lines the midpoint algorithm generates, are a result of *aliasing*, a term from signal processing. Thus, the general technique of carefully selecting pixel values to avoid jaggy artifacts is called *antialiasing*. The box lter will suf ce for most applications that do not have extremely high visual quality requirements.

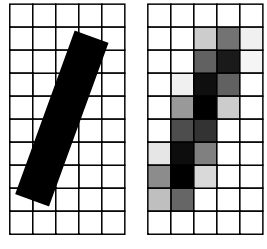

Figure 3.13. An antialiased line can be created by using an underlying rectangle.

The easiest way to implement box- lter antialiasing is to create images at very high resolutions and then downsample. For example, if our goal is a 256 by 256 pixel image of a line with width 1.2 pixels, we could rasterize a rectangle version of the line with width 4.8 pixels on a 1024 by 1024 screen, and then average 4 by 4 groups of pixels to get the colors for each of the 256 by 256 pixels in the "shrunken" image. This is an approximation of the actual box- ltered image, but works well when objects are not extremely small relative to the distance between pixels.

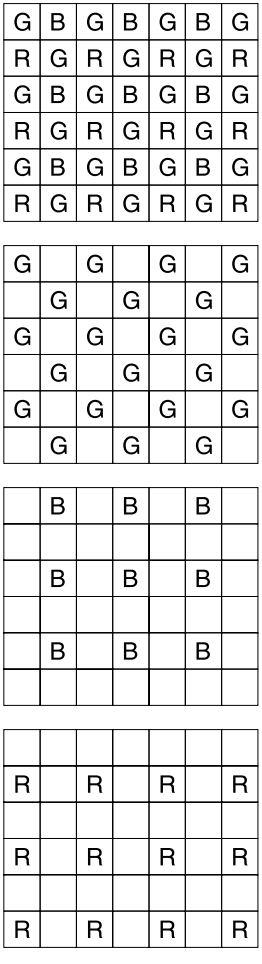

Figure 3.14. The top image shows the mosaic of RGB sensors in a typical digital camera. The bottom three images show the same pattern with individual colors highlighted.

3.8 Image Capture and Storage

Almost all graphics software deals with some "real" images that are captured using digital cameras or atbed scanners. This section deals with the practicalities of acquiring, storing, and manipulating such images.

3.8.1 Scanners and Digital Cameras

Scanners and digital cameras use some type of light-sensitive chip to record light. The dominant technologies are CCD and CMOS arrays. These devices are sensitive to light intensity across all wavelengths. To get color images, either the light is split into three components and then ltered through red, green, and blue lters (so three chips are needed), three passes are made with different lters and the same chip, or the sensors on the chip are individually coated with different colored lters.

In most current digital cameras, a single light-sensitive CCD is used with colored lters in the *Bayer mosaic* (Bayer, 1976) (Figure 3.14). This pattern devotes half of the sensors to the green channel and a quarter each to red and blue. The pattern makes the green channels a regular array at a forty- ve degree angle to the chip lattice. For natural scenes this pattern works well in practice, but for some man-made scenes color aliasing can result. Camera manufacturers have different proprietary algorithms for creating a single RGB image from an image captured using the Bayer mosaic that are somewhat more complicated than the obvious strategy of linear interpolation in each separate channel.

3.8.2 Image Storage

Most RGB image formats use eight bits for each of the red, green, and blue channels. This results in approximately three megabytes of raw information for a single million-pixel image. To reduce the storage requirement, most image formats allow for some kind of compression. At a high level, such compression is either *lossless* or *lossy*. No information is discarded in lossless compression, while some information is lost unrecoverably in a lossy system. Popular image storage formats include:

gif This lossy format indexes only 256 possible colors. The quality of the image depends on how carefully these 256 colors are chosen. This format typically works well for natural diagrams.

jpeg This lossy format compresses image blocks based on thresholds in the human visual system. This format works well for natural images.

tiff This lossless format is usually a compressed 24-bit per pixel although many other options exist.

ppm This lossless format is typically a 24-bit per pixel uncompressed format although many options exist.

png This is a set of lossless formats with a good set of open source management tools.

Because of compression and variants, writing input/output routines for images can be involved. For non-commercial applications it is advisable to just use raw ppm if no read/write libraries are readily available.

Frequently Asked Questions

• Why don't they just make monitors linear and avoid all this gamma business?

Ideally the 256 possible intensities of a monitor should *look* evenly spaced as opposed to being linearly spaced in energy. Because human perception of intensity is itself non-linear, a gamma between 1.5 and 3 (depending on viewing conditions) will make the intensities approximately uniform in a subjective sense. In this way gamma is a feature. Otherwise the manufacturers would make the monitors linear.

• How are polygons that are not triangles rasterized?

These can either be done directly scan-line by scan-line, or they can be broken down into triangles. The latter appears to be the more popular technique.

• Is it always better to antialias?

No. Some images look crisper without antialiasing. Many programs use unantialiased "screen fonts" because they are easier to read.

Exercises

1. Derive the incremental form of the midpoint line drawing algorithm with colors at endpoints for $0 < m \leq 1$.

2. Modify the triangle drawing algorithm so that it will draw exactly one pixel for points on a triangle edge which goes through $(x, y) = (-1, -1)$.

3. Simulate an image acquired from the Bayer mosaic by taking a natural image (preferably a scanned photo rather than a digital photo where the Bayer mosaic may already have been applied) and creating a greyscale image composed of interleaved red/green/blue channels. This simulates the raw output of a digital camera. Now create a true RGB image from that output and compare with the original.