THE UNIVERSITY OF CALGARY

Scan-Conversion of Important Graphics Primitives

by

Chengfu Yao

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

MAY, 1996

THE UNIVERSITY OF CALGARY
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled *"SCAN-CONVERSION OF IMPORTANT GRAPHICS PRIMITIVES"* submitted by CHENGFU YAO in partial fulfillment of the requirements for the Doctor of Philosophy Degree.

Supervisor, Dr. J. Rokne
Computer Science

Dr. B. Wyvill
Computer Science

Dr. R. Cleve
Computer Science

Dr. E. Enns
Mathematics and Statistics

External Reader
Dr. J. Bresenham
Computer Science
Winthrop University

Date May 20, 1996

Abstract

In this thesis new methods and new algorithms for the scan-conversion of important graphics primitives are presented. The fast computation of linear interpolation in discrete context is first investigated since it is widely used in digitization of graphical objects in computer graphics. Three types of integral linear interpolation are defined, and new algorithms for the fast computation of these three types of integral linear interpolation are presented. The applications of integral linear interpolation in scan-conversion of lines is further discussed which provides a unified framework for the design of various incremental line scan-conversion algorithms. It is also illustrated in the thesis how the fast computation of a specific type of integral linear interpolation can be incorporated into the scan-line algorithm for filled polygons. The run-length slice methodology in scan-conversion is another main topic of this thesis. To apply this approach to the scan-conversion of ellipses and circles, the run-length properties of digitized elliptical and circular arcs are studied, and the use of adaptive forward differencing for the run-length computation is explored. These investigations result in new and faster run-length slice ellipse and circle algorithms. Indications for further research is given in the conclusion part.

Acknowledgements

Sincerest thanks to my supervisor Dr. Jon Rokne for his support and encouragement throughout the course of this research. My thesis work started with his deep-insighted encouragement on my initial idea of unifying the derivation of various line scan-conversion algorithms by integral linear interpolation. He has generously provided me with his library of literatures relating to my research. He has read and commented a number of my research papers which have been incorporated into this thesis. His careful and prompt editorial feedback in the final months of my PhD greatly improved the final thesis.

Thanks also go to the staff, students and faculty with whom I have worked during my PhD.

I owe much to my wife and my daughter. Without their love and support the completion of this thesis would have been impossible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Scan-conversion of continuous images is a fundamental operation in any graphics display [Foley 90]. It is required since a raster display consists of a rectangular grid with pixels centered at grid points. Scan-conversion then means the approximation of a continuous image by pixels. Although images to be displayed may be very complex, they can usually be decomposed into or approximated by relatively simple components. By graphics primitives is meant the components which are simple and most commonly used, such as line segments, polygons, circles, ellipses, and spline curves and surfaces. Correct and efficient scan-conversion of graphics primitives is therefore of fundamental importance in computer graphics.

Scan-conversion of graphics primitives has a long history in the field of computer graphics. A variety of papers have been generated and many solutions have been suggested for the scan-conversion of various primitives. The first important paper dealing with the subject was due to Bresenham. He proposed a very simple method for scan-converting line segments with integral endpoints [Bresenham 65]. Following his fundamental paper, a number of other papers proposed a variety of algorithms for scan-converting line segments and other primitives. Some of the early papers were [McIlroy 83, McIlroy 84, Reggiori 72]. These papers often pursued different paths and used different methods for developing their ideas. New ideas did appear: for

example, the idea of double-step scan-conversion of lines was proposed in a paper by Wu and Rokne [Wu 87]. Following this paper, a number of algorithms were proposed for multiple-step scan-conversion of lines [Bao 89, Gill 94, Graham 93, Rokne 90]. Other ideas were those of the run-length algorithms discussed mainly for the scan-conversion of line segments, where the so-called run-lengths of a line segment were computed so that a variable number of pixels forming a run could be generated in each iterative step [Bresenham 85, Fung 92].

Similar developments could be seen for circle algorithms [Bresenham 77, Horn 76, Hsu 93, McIlroy 83, Wu 87], for ellipse algorithms [Kappel 85, Van Aken 84, Wu 89], and for scan-conversion of higher-order algebraic curves [S-L. Chang 89, Hobby 90].

Since the grid points on a raster display can be labelled by pairs of integers, we call it an *integer-grid*. The work relating to approximating lines on an integer-grid is closely related to number theory. Research into scan-conversion based on number theory was done by Dorst [Dorst 85] and by McIlroy [McIlroy 84], who developed properties relating to lines on an integer-grid [Dorst 85, McIlroy 84]. By employing the number theoretic methods, McIlroy also studied the properties of circles and ellipse on an integer-grid [McIlroy 83, McIlroy 92]

In this thesis, we focus on further development of some of the ideas presented in previous papers on scan-conversion of lines, circles and ellipses. Three of these ideas are:

1. Applying linear interpolation in the discrete context to the establishment of a unified framework for the design of incremental line algorithms.

2. Applying linear interpolation in the discrete context to the scan-conversion

filled polygons.

3. Using the run-length properties of digitized circular and elliptic arcs to design more efficient algorithms for scan-converting circles and ellipses.

1.2 The Starting Point for the Research

The efficient computation of integral linear interpolation was first discussed by Field in [Field 85]. Rokne and the author of this thesis then presented a double-step integral linear interpolation algorithm [Rokne 92]. Graham and Iyengar developed a double- and triple-step algorithm based on the double-step algorithm [Graham 94].

The scan-conversion of line segments has been discussed by many researchers since Bresenham's fundamental work in this area [Bresenham 65]. Most of the algorithms are of incremental type, and are derived based on the geometry of a continuous line segment with integer endpoints on an integer-grid. In the field of digital image analysis, a complementary problem to the scan-conversion of line segments is to determine if a set of grid points is the digitization of a straight line segment. This work was done by Rosenfeld [Rosenfeld 74] and others.

The scan-line method for the scan-conversion of filled polygons is presented in common computer graphics text books such as [Newman 79, Foley 90]. Relating this method to the method of linear interpolation has been done in [Narayanswam 95].

The scan-conversion of circles and ellipses has also been a popular subject for investigation since the introduction of raster devices. Most of the existing algorithms for scan-conversion of circles and ellipses are pixel-based in the sense that they choose individual pixels in incremental steps. Run-length slice method for the

scan-conversion of circles was first discussed by Hsu, Chow and Liu [Hsu 93]. No run-length slice algorithm for the scan-conversion ellipses has previously been reported.

1.3 Thesis Organization

The remainder of this thesis is organized as follows.

In chapter 2 the basic concepts in scan-conversion of graphics primitives are introduced, especially the criteria of minimizing the error in scan-converting a continuous curve to a digitized curve which is of fundamental importance to the design of scan-conversion algorithms. Terminology and notational conventions needed for the subsequent chapters are also presented.

In chapter 3 the concept of linear interpolation in discrete setting is discussed which leads to three types of integral linear interpolation. The implications of different types of integral linear interpolation in the context of scan-conversion of graphics primitives will be illustrated in chapters 4 and 5. New algorithms for fast integral linear interpolation are developed in chapter 3.

Chapter 4 demonstrates how the unified approach, i.e., the linear interpolation approach can derive all the incremental line algorithms published so far. To illustrate the power of the new approach, a double-step, bi-directional, run-length slice line algorithm is derived. The derivation would have been much more complicated if conventional methods such as methods based on analytic geometry were used.

The application of integral linear interpolation to polygon filling is discussed in chapter 5 where the theme is how the extrema of spans on a scan-line can be

assured to lie within the polygon by using rounding-up and rounding down linear interpolation, the new concepts developed in chapter 3.

Chapter 6 and 7 deals with the scan-conversion of more general curves. We apply the run-length slice method to ellipses and circles and discuss how to compute run-lengths efficiently by exploring the run-length properties of digitized elliptical and circular arcs and by using adaptive forward differencing. For the scan-conversion of circles, we also suggest a hybrid method that combines the run-length slice method and the midpoint method.

Some concluding remarks are made in Chapter 8 including some suggestions for further research.

Chapter 2

Scan-Conversion Basics

Computer graphics is the discipline dealing with the generation of images by means of computers. These images are most often displayed using raster displays. A raster display can be abstracted by a regular rectangular array of grid points with integer coordinates which are at the intersections of horizontal and vertical grid lines and thus is called an integer-grid as noted earlier. The integer-grid is denoted by Z^2. Pixel centers are located at the grid points. We will use *pixel* and *grid point* interchangeably in this thesis depending on the context. The images displayed on a raster display are digital images composed of pixels. Each pixel has an integer or an integer vector associated with it to represent its intensity or color. The objects to be displayed are generally in R^3 or R^2, hence scan-conversion is a mapping from R^3 or R^2 to Z^2. In this thesis we will concern ourselves with the mapping from R^2 to Z^2, i.e., we constrain our discussion to the scan-conversion of 2-D graphics objects. We also restrict the discussion to the scan-conversion of a set of simple objects which are some of the so-called graphics primitives. The result of a mapping into Z^2 is called a digitized image. We therefore use the terms scan-conversion and digitization interchangeably in the sequel.

2.1 Choosing Pixels Optimally

The essence of scan-conversion of a 2-D primitive is to choose a set of pixels that best approximate the original continuous object. The scan-conversion of a 2-D curve amounts to selecting a set of pixels to approximate the curve so that the resulting pixel representation is as close as possible to the original continuous curve, i.e., the error of digitization is minimized. There are generally three minimization criteria summarized in [Foley 90]:

1. minimizing the distance from a pixel to the curve along a grid line which is referred to as *displacement* in [McIlroy 83], *linear error* in [Van Aken 84], and *grid-distance* in [McIlroy 83];

2. minimizing the perpendicular distance from a pixel to the curve;

3. minimizing the magnitude of the residual at a pixel center if the implicit equation of the curve is given.

The three criteria correspond to three norms employed to measure the error of digitization. It turns out that the three criteria agree for straight line segments and circles with integer centers and radii [Bresenham 77, McIlroy 83], but do not necessarily agree for ellipses [Van Aken 84, McIlroy 83] and general curves. The midpoint method of Van Aken [Van Aken 84] is equivalent to applying the minimum grid-distance criterion and it is suggested in [Foley 90] as a possible choice for scan-conversion of general curves.

There may be situations where two vertically or horizontally adjacent pixels are equally close to the curve when the error is measured using a specific norm. This

situation is called an *equal-error* situation. When an equal-error situation happens, any one of the two pixels can be chosen. A consistent scheme to solve the tie is preferable, however, since certain symmetry properties would otherwise be violated. In this thesis we will always choose the pixel with larger x coordinate if the two candidate pixels are horizontally adjacent, and choose the pixel with larger y coordinate if the two candidate pixels are vertically adjacent. This tie solving policy is in fact a common practice in most known scan-conversion algorithms.

The minimum displacement criterion is an application of Freeman's *grid-intersection quantization* scheme for general line drawings [Freeman 74], where a curve to be digitized is intersected with the grid lines, and the digitized curve becomes the set of grid points that are closest to the intersection points. McIlroy [McIlroy 92] therefore uses the term *Freeman approximation* to denote a minimum-displacement approximation where all grid lines are considered and an approximation point is classified as a *minimum-horizontal-displacement* point or a *minimum-vertical-displacement* point according to the direction in which the minimized displacement is measured. In this thesis we will also use the grid-intersection scheme, i.e., the minimum displacement criterion, in discussing the scan-conversion of 2-D graphics primitives; and we will use short-hand abbreviations MHD and MVD for minimum-horizontal-displacement and minimum-vertical-displacement respectively. As has been indicated in [McIlroy 92], the MHD and MVD are not mutually exclusive.

2.2 Speeding Up Scan-Conversion

High efficiency has been a major aim pursued by the designers of scan-conversion algorithms for graphics primitives since they are used frequently in the scan-conversion of an image. One technique applied in accelerating the scan-conversion of lines and curves is to use integer arithmetic. The reason for this is that integer computation is much faster than floating point computation in current computer hardware. Bresenham [Bresenham 65, Bresenham 77] was one of the first authors presenting algorithms to scan-convert lines and circles using integer arithmetic. Sproull [Sproull 82] showed how a floating point line generation algorithm could be converted by programming transformation to a line generation algorithm using integer arithmetic only. Integer arithmetic generally allows fast software implementation and easy hardware implementation.

Most existing scan-conversion algorithms for lines and curves generate pixels in iterative loops. Generating multiple pixels in one iteration with as few computations as possible is also done in another attempt to speed up scan-conversion. Double-step algorithms which generate two pixels in each iteration step have been developed for lines, circles [Wu 87], and ellipses [Wu 89]. Double- and triple-step [Graham 93], quadruple-step [Bao 89], N-step [Gill 94] algorithms for scan-conversion of lines have also been discussed. All these multi-step algorithms are based on the analysis of possible patterns of multiple pixels on an $n \times n$ mesh where n is the step size so that a fixed number of pixels can be generated in each iteration. Another approach to generating multiple pixels in each iteration step is the *run-length slice* method. Unlike n-step algorithms which generate a fixed number of pixels in each incremental

iteration, run-length slice algorithms decompose the digitized images of continuous curves into pixel runs (i.e., pixels with the same ordinate or abscissa) and generate a run of pixels per iteration. The run-length slice method has been employed to the scan-conversion of line segments [Bresenham 85, Fung 92]. The use of this method in scan-conversion of curves is one of the main themes in this thesis, and we therefore first devote the next section to a discussion of the run-length slice method in general.

2.3 The Run-Length Slice Method for Scan-Conversion

We define a *horizontal (vertical) run* of pixels to be a set of contiguous pixels with the same ordinate (abscissa), a *diagonal run* of pixels to be a set of continguous pixels along a path with either 45° slope or 135° slope.

When investigating general digitized curves it is noticed that if a segment of curve is nearly horizontal, then long horizontal pixel runs tend to be present in its digitized image; if it is nearly vertical, then long vertical pixel runs tend to appear; and if it is close to 45°(135°) diagonal, then long diagonal pixels runs tend to appear. Line segments that are nearly horizontal, vertical, or 45°(135°) diagonal exhibit this property most noticably. This leads to an investigation of the run-length properties of digitized line segments, which further leads to run-length slice algorithms for the scan-conversion of line segments.

A run-length slice line algorithm generates a run of pixels in each iteration step. For a line segment which is nearly horizontal, the horizontal run-length slice algorithm can scan-convert the line segment with significantly fewer incremental steps than, for example, the original Bresenham line algorithm. Writing a horizontal run

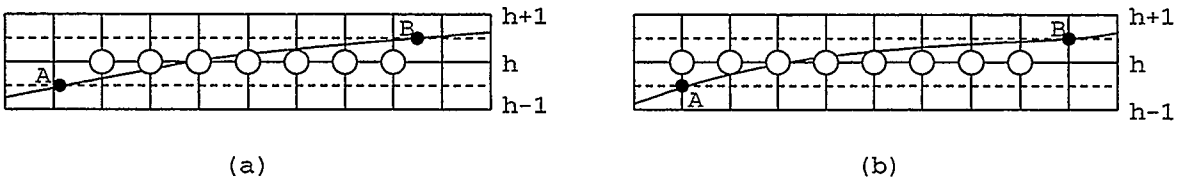


Figure 2.1: Digitized image of an x-dominant, monotonically increasing curve segment on a horizontal grid line.

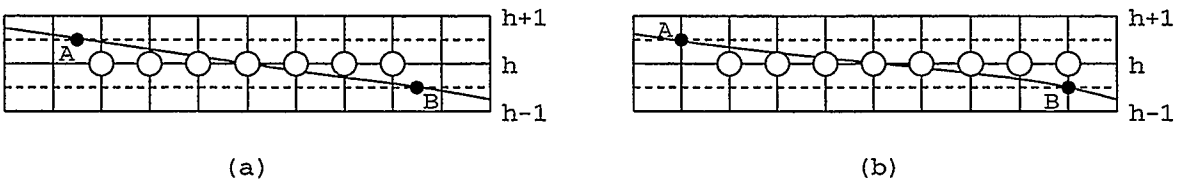


Figure 2.2: Digitized image of an x-dominant, monotonically decreasing curve segment on a horizontal grid line.

of pixels is generally more efficient than writing pixels one at a time. The basic reason for this is that bits have to be set in a word in order to write the pixels to the display medium and setting individual bits in a word is almost as expensive as setting all the bits in the word at one time. Although pixels have to be set one at a time for the hardware structure of the conventional frame buffer for a vertical run of pixels, we can take advantage of the coherence of pixels addresses in the frame buffer to speed up the pixel address computation and therefore speed up the process of pixel writing in this case as well. We can expect further gain in speed for writing a run of pixels if the display hardware supports the writing of horizontal or vertical runs of pixels. In practice, pixel writing is still the bottleneck in the scan-conversion of common graphics primitives such as line segments, circles and ellipses.

We now discuss the application of run-length slice method to the scan-conversion of a general curve. Let $C : y = f(x)$ be a 2-D curve having a continuous first

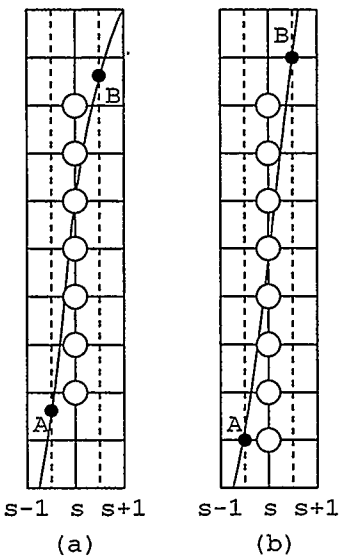


Figure 2.3: Digitized image of a y-dominant, monotonically increasing curve segment on a vertical grid line.

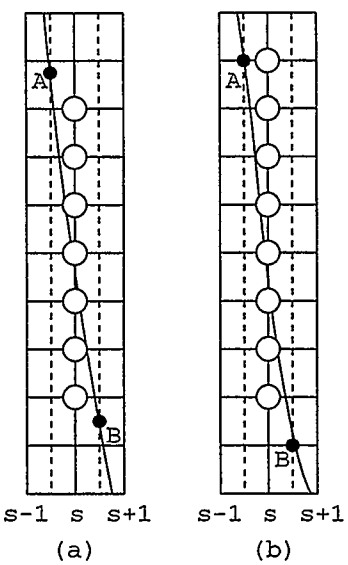


Figure 2.4: Digitized image of a y-dominant, monotonically decreasing curve segment on a vertical grid line.

derivative. $\mathcal{C}$ is said to be $x-$dominant if $|f'(x)| \leq 1$ or $y-$dominant if $|f'(x)| \geq 1$. Let us consider the following four situations:

1. Curve $y = f(x)$ is $x-$dominant and monotonically increasing. Referring to Figure 2.1 (a), the curve intersects with two horizontal lines $y = h - 0.5$ and $y = h + 0.5$ at $(x_1, h - 0.5)$ and $(x_2, h + 0.5)$, where h is an integer and $x_1 < x_2$. Because of the monotonicity of the curve, we have $h - 0.5 < y = f(\xi) < h + 0.5$ for $x_1 < \xi < x_2$. According to the grid-intersection scheme for curve digitization and the forementioned policy to handle equal error cases, the digitized image of the curve on grid line $y = h$ is a horizontal run of pixels having ordinate h and abscissas going from $\dot{x}_s = \lceil x_1 \rceil$, no matter whether x_1 is integral or non-integral, to $\dot{x}_e = \lfloor x_2 \rfloor$ if x_2 is non-integral. In the case that x_2 is integral (see Figure 2.1 (b)) $\dot{x}_e = x_2 - 1$. Both cases (x_2 is integral or non-integral) can be unified by letting $\dot{x}_e = \lceil x_2 \rceil - 1$. Using the similar argument we have the results for the rest three situations.

2. Referring to Figure 2.2 (a) and (b) if the curve is $x-$dominant and monotonically decreasing, the digitized image of the curve on grid line $y = h$ is a horizontal run of pixels having ordinate h and abscissas going from $\dot{x}_s = \lceil x_1 \rceil$, if x_1 is non-integral, or $\dot{x}_s = \lfloor x_1 \rfloor + 1$ if x_1 is integral, to $\dot{x}_e = \lfloor x_2 \rfloor$ no matter whether x_2 is integral or non-integral. Again both cases (x_1 is integral or non-integral) can be unified by letting $\dot{x}_s = \lfloor x_1 \rfloor + 1$.

3. Referring to Figure 2.3 (a) and (b) if curve $y = f(x)$ is $y-$dominant and monotonically increasing, the curve intersects with two vertical lines $x = s-0.5$ and $x = s+0.5$ at $(s-0.5, y_1)$ and $(s+0.5, y_2)$ where s is an integer and $y_1 < y_2$.

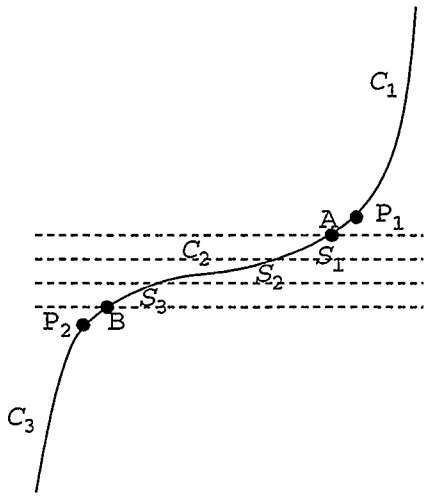


Figure 2.5: Dividing an x-dominant segment of a curve into unit segments by imposing a set of horizontal mid-lines.

The digitized image of the curve on the grid line $x = s$ is a vertical run of pixels having abscissa s and ordinates going from $\dot{y}_s = \lceil y_1 \rceil$, no matter whether y_1 is integral or non-integral, to $\dot{y}_e = \lfloor y_2 \rfloor$ if y_2 is non-integral, or $\lceil y_2 \rceil - 1$ no matter whether y_2 is integral or not.

4. Referring to Figure 2.4 (a) and (b) if the curve is $y-$dominant and monotonically decreasing, the digitized image of the curve on the grid line $x = s$ is a vertical run of pixels having abscissa s and ordinates going from $\dot{y}_s = \lfloor y_1 \rfloor$ no matter whether y_1 is integral or non-integral, and to $\dot{y}_e = \lfloor y_2 \rfloor + 1$ no matter whether y_2 is integral or not.

The above discussion suggests a way to find the horizontal and vertical pixel runs of the digitized image of the curve $y = f(x)$: Divide the curve into segments where each segment is either x-dominant or y-dominant, either monotonically increasing or monotonically decreasing. We call these segments *monotonic segments*. For each

monotonic segment a set of horizontal mid-lines is imposed if it is x-dominant, or a set of vertical mid-lines if it is y-dominant. Each of the mid-lines lies midway between two adjacent horizontal/vertical grid lines and intersects with the curve segment. The mid-lines therefore subdivide a monotonic segment into subsegment where each subsegment is sandwiched by two adjacent mid-lines. We call these subsegments *unit segments.* The digitized image of a unit segment is therefore a run of pixels on a grid line lying between two adjacent mid-lines. Knowing the two endpoints of a unit segment, a run of pixels can be obtained without further computation. Generally, the two endpoints of a monotonic segment do not lie on two mid-lines which causes two subsegments at two ends of a monotonic segment not to be unit segments. Referring to Figure 2.5, C_1, C_2 and C_3 are three monotonic segments of curve C divided by points P_1 and P_2, where C_2 is x-dominant while C_1 and C_3 are y-dominant. A set of horizontal mid-lines is imposed to subdivide C_2 into unit segments S_1, S_2, and S_3 whose digitized images are three horizontal pixel runs. Two end subsegments, one from P_1 to A and the other fron B to P_2 are not unit segments, and the digitization of these two subsegments should be treated in connection with monotonic segment C_1 for subsegment P_1A, and with monotonic segment C_3 for subsegment BP_2. We will give a detailed treatment of the handling of the end subsegments when discussing the run-length slice method for scan-conversion of specific curves, i.e., circles and ellipses.

2.4 Summary

In this chapter we discussed some general properties of scan-conversion with emphasis on correctness and efficiency. The correctness of a scan-conversion algorithm relates to the criterion chosen to minimize the error in digitization. The policy to handle the particular case of the equal-error situation is discussed and a policy is proposed. This policy will be applied throughout the thesis. Two major approaches discussed which increase the efficiency of scan-conversion are integerizing the computation and generating multiple pixels in each iteration step.

We will focus on scan-conversion of 2-D graphics primitives. The scheme of digitization that will be used is the Freeman's grid-intersection scheme. This minimizes the grid distance of the chosen pixels to grid points for digitized curves. The run-length methodology was furthermore introduced. It will be of central importance as we develop the run-length slice method for the scan-conversion of circles and ellipses.

Chapter 3

Integral Linear Interpolation

Linear interpolation is widely used in computer graphics algorithms. One application is the scan-conversion of line segments which is a special case of linear interpolation where, given two endpoints of a line segment, the x and y coordinates of a set of pixels to approximate the true line are actually linear interpolations of two intervals representing the x-extent and y-extent of the line segment. Other applications of linear interpolation in computer graphics can be seen in the synthesis of realistic imagery [Gouraud 71, Phong 75], and in the scan-conversion of filled polygons [Foley 90]. Speeding up the process of linear interpolation will speed up the relevant graphics algorithms. The fast computation of linear interpolation is also a research topic in its own right.

3.1 Three Types of Integral Linear Interpolation

The problem of linear interpolation is posed in a discrete setting because of the discrete nature of raster graphics: Given an interval $[a, b]$ with integral a and b, find $n+1$ integers which are approximations to $n+1$ equidistant points $x_i = a + \frac{b-a}{n} i$, $i = 0, 1, \ldots, n$ on $[a, b]$ including a and b. The integer points are usually obtained by rounding the real interpolation points to the nearest integers so that the error is minimized. When an interpolation point is exactly halfway between two consecutive integers (the *equal error case*) then a choice has to be made whether to round it up

or to round it down to the nearest integer. In this case either of the two integers can be chosen. The choice made must be, however, consistent. It is a common practice to always choose the larger one. This gives the following representation of the interpolation points:

$$\begin{aligned} \dot{x}_i &= \lfloor a + \frac{b-a}{n} i + 0.5 \rfloor \\ &= \lfloor x_i + 0.5 \rfloor \quad i = 0, 1, \ldots, n. \end{aligned} \tag{3.1}$$

Since the resulting interpolation points should be integral, we use the term integral linear interpolation to distinguish it from the usual continuous linear interpolation as was noted in chapter 1 . Furthermore, because the interpolation points defined by Eq. (3.1) minimizes the error, we call this type of integral linear interpolation *least error integral linear interpolation.*

If, instead of defining the integral interpolation points using Eq. (3.1), we round up each interpolation point x_i to the smallest integer which is greater than or equal to x_i, i.e., we define

$$\dot{x}_i = \lceil x_i \rceil = \lceil a + \frac{b-a}{n} i \rceil, \tag{3.2}$$

then we obtain *rounding-up integral linear interpolation.* Analogously, we define *rounding-down integral linear interpolation* by the following equation:

$$\dot{x}_i = \lfloor x_i \rfloor = \lfloor a + \frac{b-a}{n} i \rfloor, \tag{3.3}$$

i.e., we use the largest integer which is less than or equal to x_i to approximate x_i.

Note that for all three types of integral linear interpolation we use the same

symbol $\dot{x}_i$ to denote an interpolation point. The type of integral linear interpolation it represents can be determined by context.

Fast computation of least-error integral linear interpolation has been discussed by several authors [Field 85, Graham 94, Rokne 92], while rounding-up integral linear interpolation and rounding-down integral linear interpolation have not been defined and discussed by other authors. The motivation for defining and discussing these two additional types of integral linear interpolation is also based on their relation to some of the graphics algorithms which will be illustrated in chapters 4 and 5.

In the remainder of this chapter we will first give notational conventions which will be used in this chapter as well as in chapter 4 and chapter 5. We then derive algorithms for the fast computation of integral linear interpolation. The derivations will follow the pattern that was used for the double-step integral linear interpolation in [Rokne 92]. The idea is further developed using symmetry to yield a bi-directional integral linear interpolation algorithm. We finally compare a double-step, bi-directional integral linear interpolation algorithm with our previously developed double-step algorithm and one of Field's integral linear interpolation algorithm [Field 85].

3.2 Notational Conventions

In this section we first give notational conventions for some quantities which will be used to develop the interpolation algorithms. Efficient computation of these quantities, which relates to the development of efficient interpolation algorithms, is then discussed.

Let $m > 0$ and $n > 0$ be two integers. We then let c and r denote the quotient and remainder when the first integer is divided by the second integer as follows:

$$c = \lfloor \frac{m}{n} \rfloor,$$

$$r = m \bmod n.$$

We use C and R to denote the quotient and the remainder of twice the first integer divided by the second integer:

$$C = \lfloor \frac{2m}{n} \rfloor,$$

$$R = 2m \bmod n.$$

Furthermore we use $\tilde{c}$ and $\tilde{r}$ to denote the quotient and the remainder of the first integer divided by twice the second integer:

$$\tilde{c} = \lfloor \frac{m}{2n} \rfloor,$$

$$\tilde{r} = m \bmod 2n,$$

and $\hat{c}$ is defined as

$$\hat{c} = \lceil \frac{m}{2n} \rceil.$$

Most modern computers provide a "divide with remainder" instruction. This means that c and r can be calculated by one CPU instruction when m and n are given. Even though the other two pairs C, R and $\tilde{c}$, $\tilde{r}$ can also be calculated by "divide with remainder" instruction, their values can be derived with less computational cost

given c and r as seen below. Since

$$m = nc + r, \qquad 0 \leq r < n,$$

we have the following derivations:

$$C = \lfloor \frac{2m}{n} \rfloor = \lfloor \frac{2nc + 2r}{n} \rfloor = 2c + \lfloor \frac{2r}{n} \rfloor = \begin{cases} 2c & \text{if } 2r < n \\ 2c + 1 & \text{if } 2r \geq n, \end{cases}$$

$$R = 2m \bmod n = (2cn + 2r) \bmod n = 2r \bmod n = \begin{cases} 2r & \text{if } 2r < n \\ 2r - n & \text{if } 2r \geq n, \end{cases}$$

$$\begin{aligned} \tilde{c} &= \lfloor \frac{m}{2n} \rfloor = \lfloor \frac{cn + r}{2n} \rfloor \\ &= \lfloor \frac{c}{2} + \frac{r}{2n} \rfloor \\ &= \begin{cases} \frac{c}{2} & \text{if } c \text{ is even} \\ \lfloor \frac{c-1}{2} + \frac{r+n}{2n} \rfloor = \lfloor \frac{c}{2} \rfloor & \text{if } c \text{ is odd} \end{cases} \\ &= c \gg 1, \end{aligned}$$

where $\gg$ denotes a binary right shift. We also have:

$$\begin{aligned} \tilde{r} &= m \bmod 2n = (cn + r) \bmod 2n \\ &= \begin{cases} r \bmod 2n = r & \text{if } c \text{ is even} \\ n + r \bmod 2n = n + r & \text{if } c \text{ is odd}, \end{cases} \end{aligned}$$

$$\hat{c} = \lceil \frac{m}{2n} \rceil = \begin{cases} \tilde{c} & \text{if } \tilde{r} = 0 \\ \tilde{c}+1 & \text{if } \tilde{r} > 0. \end{cases}$$

In some special cases the values of c and r can be obtained in a straightforward manner without performing a division operation. For example, if $m < n$ then $c = 0$ and $r = m$; if $n \leq m < 2n$ then $c = 1$ and $r = m - n$. These situations occur when we transform integral linear interpolation algorithms to line scan-conversion algorithm. The meanings of $c, r, C, R, \tilde{c}$, $\tilde{r}$ and $\hat{c}$ will remain unchanged although m and n may be substituted by specific integers.

3.3 Fast Computation of Integral Linear Interpolation

The computation of the three types of integral linear interpolation can be performed in a straightforward manner using floating point arithmetic according to their definitions in Eq. (3.1), (3.2) and (3.3). Floating point computation is usually expensive, so the aim is to replace floating point computation by integer computation. In our derivations of the algorithms for the computation of three types of integral linear interpolation we follow the following methodology: The integral interpolation points are calculated incrementally based on recurrence formulas which are derived from the equations which define the corresponding types of integral linear interpolation while eliminating floating point calculations in the recurrence formulas so that the computation of integral linear interpolation uses integer arithmetic only. The incremental algorithms can thus be derived from recurrence formulas. Algorithms that produce one interpolation point in each iteration step are called *single-step algorithms*, and algorithms that generate two interpolation points in each iteration step are called

double-step algorithms. The approach used to derive recurrence formulas for integral linear interpolation was used in our previous work on double-step least-error integral linear interpolation [Rokne 92]. In this thesis we will follow this approach for all three types of integral linear interpolation. We will first deal with single-step algorithms for illustrative purposes, and then we continue with double-step algorithms, and finally we discuss double-step bi-directional algorithms using symmetry. For least-error integral linear interpolation an interesting property was revealed by our previous work [Rokne 92] that when advancing from one interpolation point to the next, the step size is confined to two consecutive integers; this is also true for the double-step advance, i.e., the size of a double-step is also confined to two consecutive integers. This allows the use of a variable called discriminator whose sign determines the step size, either single or double. We will see that this property is shared by rounding-up and rounding-down integral linear interpolation as well.

3.3.1 Single-Step Algorithms

Least-Error Integral Linear Interplation

The recurrence formulas for the single-step least-error integral linear interpolation can be derived as follows. Let $m = b - a$. Since a, the lower bound of the interval, and b, the upper bound of the interval are both integers, m is a positive integer. The integral interpolation points are given by

$$\dot{x}_i = \lfloor x_i + 0.5 \rfloor, \quad i = 0, 1, \ldots, n.$$

We therefore have

$$x_i - 0.5 < \dot{x}_i \leq x_i + 0.5,$$

$$x_{i+1} - 0.5 < \dot{x}_{i+1} \leq x_{i+1} + 0.5,$$

i.e.,

$$a + i\frac{m}{n} - 0.5 < \dot{x}_i \leq a + i\frac{m}{n} + 0.5, \tag{3.4}$$

$$a + (i+1)\frac{m}{n} - 0.5 < \dot{x}_{i+1} \leq a + (i+1)\frac{m}{n} + 0.5. \tag{3.5}$$

Subtracting Eq (3.4) from Eq. (3.5) yields

$$\frac{m}{n} - 1 < \dot{x}_{i+1} - \dot{x}_i < \frac{m}{n} + 1.$$

Using our notational convention we can easily see that the value of $\dot{x}_{i+1} - \dot{x}_i$ is either c or $c+1$ where $c = \lfloor \frac{m}{n} \rfloor$, i.e., the difference of two consecutive interpolation points can only assume the values of two consecutive integers. Letting $\varepsilon_i = x_{i+1} - \dot{x}_i - (c+0.5)$, we have

$$\dot{x}_{i+1} = \begin{cases} \dot{x}_i + c & \varepsilon_i < 0 \\ \dot{x}_i + c + 1 & \varepsilon_i \geq 0. \end{cases}$$

It follows from $n > 0$ that $D_i = 2n\varepsilon_i$ retains the sign of ε_i. Since D_i turns out to be a conveniently calculated quantity, we choose it to be the discriminator. We thus have

$$\dot{x}_{i+1} = \begin{cases} \dot{x}_i + c & D_i < 0 \\ \dot{x}_i + c + 1 & D_i \geq 0. \end{cases} \tag{3.6}$$

Noting that

$$\begin{aligned}\varepsilon_i &= x_{i+1} - \dot{x}_i - (c + 0.5) \\ &= a + (i+1) \cdot \frac{b-a}{n} - \dot{x}_i - (c + 0.5),\end{aligned}$$

it follows that

$$D_i = 2na + 2(i+1)(b-a) - 2n\dot{x}_i - n(2c+1). \tag{3.7}$$

Subtracting D_i from D_{i+1} yields

$$D_{i+1} - D_i = 2(b-a) - 2n(\dot{x}_{i+1} - \dot{x}_i).$$

Hence,

$$D_{i+1} = \begin{cases} D_i + 2(b-a) - 2nc & D_i < 0 \\ D_i + 2(b-a) - 2n(c+1) & D_i \geq 0. \end{cases} \tag{3.8}$$

The initial value for $\dot{x}_i$ is

$$\dot{x}_0 = a. \tag{3.9}$$

Evaluating Eq. (3.7) for $i = 0$ determines the initial value of the discriminator

$$D_0 = 2(b-a) - n(2c+1). \tag{3.10}$$

Noting that $b-a=nc+r$, it follows that Eq. (3.8) and (3.10) can be simplified to

$$D_{i+1}=\begin{cases} D_i+2r & D_i<0 \\ D_i+2(r-n) & D_i\geq 0, \end{cases} \tag{3.11}$$

and

$$D_0=2r-n. \tag{3.12}$$

An integerized algorithm for least error integral linear interpolation can be implemented according to the initial values given by Eq. (3.9), (3.12) and the recurrence formulas Eq. (3.6), (3.11).

Rounding-up and Rounding-down Integral Linear Interpolation

We first consider single-step rounding-up integral linear interpolation. According to the definition we have

$$x_i\leq \dot{x}_i<x_i+1, \tag{3.13}$$

$$x_{i+1}\leq \dot{x}_{i+1}<x_{i+1}+1. \tag{3.14}$$

Subtracting Eq.(3.13) from Eq.(3.14) yields

$$x_{i+1}-x_i-1<\dot{x}_{i+1}-\dot{x}_i<x_{i+1}-x_i+1.$$

Again we have

$$\frac{m}{n}-1<\dot{x}_{i+1}-\dot{x}_i<\frac{m}{n}+1,$$

which implies that the value of $\dot{x_{i+1}} - \dot{x_i}$ is either c or $c+1$ depending on how far away x_{i+1} is from $\dot{x_i}$:

$$\dot{x}_{i+1} - \dot{x}_i = \begin{cases} c & x_{i+1} - \dot{x}_i - c \leq 0 \\ c+1 & x_{i+1} - \dot{x}_i - c > 0. \end{cases}$$

Based on a similar argument as in the least-error case the discriminator is defined to be

$$\begin{aligned} D_i &= n(x_{i+1} - \dot{x}_i - c) \\ &= na + (i+1)(b-a) - n\dot{x}_i - nc. \end{aligned}$$

Subtracting D_i from D_{i+1} yields

$$\begin{aligned} D_{i+1} - D_i &= (b-a) - n(\dot{x}_{i+1} - \dot{x}_i) \\ &= \begin{cases} b-a-nc & D_i \leq 0 \\ b-a-n(c+1) & D_i > 0, \end{cases} \end{aligned}$$

and we finally have the recurrence formulas

$$\dot{x}_{i+1} = \begin{cases} \dot{x}_i + c & D_i \leq 0 \\ \dot{x}_i + c + 1 & D_i > 0, \end{cases} \tag{3.15}$$

and

$$D_{i+1} = \begin{cases} D_i + b - a - nc & D_i \leq 0 \\ D_i + b - a - n(c+1) & D_i > 0, \end{cases} \tag{3.16}$$

The initial values of the interpolation point and the discriminator are

$$\dot{x}_0 = a \tag{3.17}$$

and

$$D_0 = b - a - nc \tag{3.18}$$

respectively. Using the equality $b-a = nc+r$, Eq. (3.16) and (3.18) can be simplified to

$$D_{i+1} = \begin{cases} D_i + r & D_i \leq 0 \\ D_i + r - n & D_i > 0. \end{cases} \tag{3.19}$$

and

$$D_0 = r. \tag{3.20}$$

Similarly, we have the following formulas for the the single-step rounding-down integral linear interpolation:

$$\dot{x}_{i+1} = \begin{cases} \dot{x}_i + c & D_i < 0 \\ \dot{x}_i + c + 1 & D_i \geq 0, \end{cases} \tag{3.21}$$

and

$$D_{i+1} = \begin{cases} D_i + r & D_i < 0 \\ D_i + r - n & D_i \geq 0, \end{cases} \tag{3.22}$$

$$\dot{x}_0 = a, \tag{3.23}$$

and

$$D_0 = r - n. \tag{3.24}$$

Again, the integerized single-step algorithms for rounding-up and rounding-down integral linear interpolation can be implemented in a very straightforward manner from the respective recurrence formulas.

3.3.2 Double-Step Algorithms

Double-step algorithms generate two interpolation points in each iteration. Supposing we only need to calculate the interpolation points with even number indices, we can derive the recurrence formulas in the same manner as we did for the single-step algorithms, and the length of each double-step is confined to two consecutive integers. To produce the intermediate interpolation point after each double-step advance, further computation is needed. Since the length of a single-step is also confined to two consecutive integers, the intermediate point can be decided by a two-state logic. The derivation of recurrence formulas for double-step calculation of all three types of integral linear interpolation follows the same line of thought as that of single-step interpolation. We thus give a brief derivation of the recurrence formulas for the double-step least-error integral linear interpolation here. This was published in our previous paper [Rokne 92] and will be presented here in a manner complying with our notational convention for this thesis. We will leave the derivation of recurrence formulas for double-step rounding-up and rounding-down integral linear interpolation to chapter 4 where we illustrate how they can be used for the derivation of run-length line scan-conversion algorithms. Again we use m to denote $b-a$. Let

$$X_i \;=\; x_{2i} = a + 2i\frac{m}{n}$$

$$= \quad a + i\frac{2m}{n}, \qquad i = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor,$$

and

$$\dot{X}_i = \lfloor X_i + 0.5 \rfloor = \dot{x}_{2i}.$$

Using the same method as we used for the single-step case, we have

$$\frac{2m}{n} - 1 < \dot{X}_{i+1} - \dot{X}_i < \frac{2m}{n} + 1.$$

This results in the conclusion that the value of $\dot{X}_{i+1} - \dot{X}_i$ is either C or $C+1$ where $C = \lfloor \frac{2m}{n} \rfloor$ according to our notation. The double-step size from $\dot{X}_i$ to $\dot{X}_{i+1}$ can be determined by testing the sign of

$$X_{i+1} - \dot{X}_i - (C + 0.5) = a + 2(i+1)\frac{b-a}{n} - \dot{X}_i - (C + 0.5).$$

We therefore define the discriminator to be

$$D_i = 2an + 4(i+1)(b-a) - 2n\dot{X}_i - n(2C+1),$$

and we have the recurrence formula

$$\dot{X}_{i+1} = \begin{cases} \dot{X}_i + C & D_i < 0 \\ \dot{X}_i + C + 1 & D_i \geq 0. \end{cases} \tag{3.25}$$

Using the same technique as we used for the single-step case we have the following recurrence formula for the discriminator:

$$D_{i+1} = \begin{cases} D_i + 4(b-a) - 2nC & D_i < 0 \\ D_i + 4(b-a) - 2n(C+1) & D_i \geq 0. \end{cases} \tag{3.26}$$

Since the the single-step size can be either c or $c+1$, the interpolation point between $\dot{X}_i$ and $\dot{X}_{i+1}$, i.e., $\dot{x}_{2i+1}$ can be determined by testing the sign of

$$x_{2i+1} - \dot{X}_i - (c+0.5) = a + (2i+1)\frac{b-a}{n} - \dot{X}_i - (c+0.5).$$

This amounts to the testing of sign of

$$\begin{aligned} d_i &= 2na + 2(2i+1)(b-a) - 2n\dot{X}_i - n(2c+1) \\ &= D_i - 2(b-a) + 2n(C-c). \end{aligned}$$

We therefore have

$$\dot{x}_{2i+1} = \begin{cases} \dot{X}_i + c & D_i < 2(b-a) - 2n(C-c) \\ \dot{X}_i + c + 1 & D_i \geq 2(b-a) - 2n(C-c). \end{cases} \tag{3.27}$$

The initial values of $\dot{X}_i$ and D_i are

$$\dot{X}_0 = a, \tag{3.28}$$

and

$$D_0 = 4(b-a) - n(2C+1). \tag{3.29}$$

Noting the equalities

$$b - a = nc + r,$$

and

$$2(b-a) = nC + R,$$

the recurrence formula for D_i can be simplified to

$$D_{i+1} = \begin{cases} D_i + 2R & D_i < 0 \\ D_i + 2(R-n) & D_i \geq 0, \end{cases} \tag{3.30}$$

and the recurrence formula for x_{2i+1} can be simplified to

$$\dot{x}_{2i+1} = \begin{cases} \dot{X}_i + c & D_i < 2(R-r) \\ \dot{X}_i + c + 1 & D_i \geq 2(R-r). \end{cases} \tag{3.31}$$

The initial value of D_i is

$$D_0 = 2R - n. \tag{3.32}$$

A double-step least-error integral linear interpolation algorithm can be implemented in a straightforward manner based on Eq. (3.28), (3.32), (3.25), (3.30), (3.31). One single-step advance after the last double-step advance is needed if n is odd. One implementation is given in [Rokne 92], where the point $\dot{x}_{2i+1}$ is obtained by adding half that length to $\dot{X}_i$ without comparing D_i with $2(R-r)$ to further improve the efficiency when the length of a double-step from $\dot{X}_i$ to $\dot{X}_{i+1}$ (C or $C+1$) is even.

The reason is that one double-step equals the sum of two consecutive single-steps, so if the length of the double-step is even, the lengths of the two single-steps must be equal.

3.3.3 Bi-Directional Interpolation

Integral linear interpolation can be further sped up by utilizing symmetry, i.e., the computation is performed from both direction simultaneously. Similar situation exists in line scan-conversion where symmetry is utilized for bi-directional line drawing. This was first mentioned by Gardner [Gardner 92]. Here we note that it will reduce the number of iterations to 1/4 of that needed by single-step algorithms when combined with the double-step technique. The forementioned algorithms for the computation of integral linear interpolation are all performed from a to b, and we thus call them *forward incremental algorithms*. To show how interpolation can be performed bi-directionally, we first illustrate how it can be performed in the reversed direction. Again, we use least-error integral linear interpolation as example. Define

$$x'_i = b - \frac{b-a}{n}i, \quad i = 0, 1, \ldots, n,$$

and

$$\dot{x}'_i = \lfloor x'_i + 0.5 \rfloor, \quad i = 0, 1, \ldots, n,$$

then $x'_i = x_{n-i}, \dot{x}'_i = \dot{x}_{n-i}$. We further define

$$X'_i = x'_{2i}, \quad i = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor,$$

$$\dot{X}'_i = \lfloor X'_i + 0.5 \rfloor, \quad i = 0, 1, \ldots, \lfloor \frac{n}{2} \rfloor.$$

Using similar derivations as used for the forward double-step incremental algorithm we have the following recurrence formulas:

$$\dot{X}'_0 = b, \tag{3.33}$$

$$D'_0 = 2R - n, \tag{3.34}$$

$$\dot{X}'_{i+1} = \begin{cases} \dot{X}'_i - C & D'_i \leq 0 \\ \dot{X}'_i - C - 1 & D'_i > 0, \end{cases} \tag{3.35}$$

$$D'_{i+1} = \begin{cases} D'_i + 2R & D'_i \leq 0 \\ D'_i + 2(R - n) & D'_i > 0, \end{cases} \tag{3.36}$$

$$\dot{x}'_{2i+1} = \begin{cases} \dot{X}'_i - c & D'_i \leq 2(R - r) \\ \dot{X}'_i - c - 1 & D'_i > 2(R - r). \end{cases} \tag{3.37}$$

Based on Eq. (3.33)-(3.37) we can present the backward version of double-step least-error integral linear interpolation algorithm which will generate exactly the same set of points as its forward counterpart except that the order of the points generated is reversed. Compare two sets of formulas for the forward and the backward double-step least error integral linear interpolation, we find that

1. The initial values of the discriminators are the same, i.e., $D_0 = D'_0$.

2. The recurrence formulas for obtaining $\dot{X}_{i+1}$ and $\dot{X}'_{i+1}$ from $\dot{X}_i$ and $\dot{X}'_i$ respectively (Eq. (3.24) and (3.35)) are basically symmetric except for the difference

in the equality case for updating the formulas: incrementing $\dot{X}_i$ by C or $C+1$ to yield $\dot{X}_{i+1}$ according to the sign of D_i, and decrementing $\dot{X}'_i$ by C or $C+1$ to yield $\dot{X}'_{i+1}$ according to the sign of D'_i. A similar situation exists for the formulas to calculate $\dot{x}_{2i+1}$ and $\dot{x}'_{2i+1}$ (Eq. (3.31) and (3.37)).

3. After $\dot{X}_{i+1}$ and $\dot{X}'_{i+1}$ are obtained from $\dot{X}_i$ and $\dot{X}'_i$ respectively, the formulas to update D_i and D'_i to D_{i+1} and D'_{i+1} (Eq. (3.30) and (3.36)) are again essentially the same except for the equality case for updating the formulas.

The similarity between the two sets of formulas suggests that only one discriminator is necessary in order to perform the bi-directional calculation if the asymmetric portions of the two sets of formulas can be handled properly. If D_i is never zero, then $D'_i = D_i$ for all i. Therefore starting from $\dot{X}_0 = a$ and $\dot{X}'_0 = b$ we only test the sign of D_i to increment $\dot{X}_i$ and to decrement $\dot{X}'_i$ by the same amount (C or $C+1$) to yield $\dot{X}_{i+1}$ and $\dot{X}'_{i+1}$, and update D_i to D_{i+1}. If, for some i, $D_i = D'_i = 0$, then D_i is updated by adding $2(R-n)$ and D'_i is updated by adding $2R$, which means that D_{i+1} and D'_{i+1} assume different values. The use of only D_i as discriminator to perform bi-directional interpolation computation fails in this scenario. The following lemma provides a remedy for the asymmetry should it happen.

Lemma 1 *If $D_i = D'_i = 0$ for some i, and therefore $D_{i+1} \neq D'_{i+1}$ then equality of D_j and D'_j is restored for $j = i+2$, i.e., $D_{i+2} = D'_{i+2}$, hence symmetry of the interpolation points is restored at $i+2$ if $D_{i+2} \neq 0$.*

Proof: If $R = 0$, then $D_0 = D'_0 = -n$ according to Eq. (3.32) and (3.34). Therefore $D_i = D'_i = -n$ for all i, and $D_i = 0$ will never happen. The above discussion indicates

that if $D_i = 0$ for some i then the inequality $R > 0$ must hold, which implies $2R > 0$. Meanwhile, inequality $2(R - n) < 0$ always holds. Suppose $D_i = D'_i = 0$ for some i, then

$$D_{i+1} = 2(R - n) < 0,$$

$$D'_{i+1} = 2R > 0.$$

Adding $2R$ to D_{i+1} and $2(R - n)$ to D'_{i+1} yields

$$D_{i+2} = D'_{i+2} = 4R - 2n.$$

□

Combining the forward and backward double-step algorithms yields a bi-directional interpolation algorithm in an obvious manner. The bi-directional interpolation algorithm first outputs the first point $X_0 = a$ and the last point $X'_0 = b$, then the interpolation computation is performed bi-directionally with $N = \lfloor \frac{n-1}{4} \rfloor$ iteration steps. If $D_i \neq 0$ the algorithm simply increments $\dot{X}_i$ and decrements $\dot{X}'_i$ symmetrically to yield $\dot{X}_{i+1}$ and $\dot{X}'_{i+1}$. The point $\dot{x}_{2i+1}$ between $\dot{X}_i$ and $\dot{X}_{i+1}$ and the point $\dot{x}'_{2i+1}$ between $\dot{X}'_i$ and $\dot{X}'_{i+1}$ can be obtained by Eq. (3.31) and (3.37). If the number to increment (decrement) from $\dot{X}_i$ ($\dot{X}'_i$) is even, then $\dot{x}_{2i+1}$ ($\dot{x}'_{2i+1}$) can be obtained by incrementing (decrementing) $\dot{X}_i$ ($\dot{X}'_i$) by half that amount. If $D_i = D'_i$ in one of the iteration steps, the algorithm increments $\dot{X}_i$ by $C + 1$ and decrements $\dot{X}'_i$ by C to yield $\dot{X}_{i+1}$ and $\dot{X}'_{i+1}$ and then sets the discriminator D_{i+2} to be $4R - 2n$. The iteration count should be decreased by one in this case since we actually skip one iteration to update the formulas. After the iteration stops, the number of leftover

points is $s = (n-1) \bmod 4 = 0, 1, 2, 3$. If $s = 0$, all the interpolation points have been generated, otherwise the leftover points are generated in the following manner:

- If $s = 1$, move forward one single step. This can be done by comparing D and $2(R-r)$ to determine the step length (c or $c+1$).
- If $s = 2$, move forward a double-step to fill in two points.
- If $s = 3$, move forward a double-step in the forward algorithm and move backward one single step to fill in three points in the backward algorithm.

In this chapter the mathematical and logical foundations were given for a number of algorithms. The complete algorithms were, however, not developed. As a conclusion to this chapter we therefore present a complete algorithm using a C like pseudo-code for a double-step, bi-directional least-error integral linear interpolation algorithm.

ALGORITHM 3.1: Double-step, bi-directional least-error integral linear interpolation

$c = (b-a)/n; \quad r = (b-a)\%n;$

$C = c << 1; \quad R = r << 1;$

if $(R >= n)$ {

$\quad C = C + 1; \quad R = R - n;$

}

$Inc1 = R << 1; \quad Inc2 = (R-n) << 1;$

$V = (R-r) << 1;$

$D = Inc1 - n;$

$N = (n-1) >> 2; \quad LeftOver = n - 1 - (N << 2);$

```
X1 = a;    X2 = b;

output2(X1, X2);
if (C is even) {
  Ch = C >> 1;
  for (i = 1; i <= N; i++) {
    if (D < 0) {
      x1 = X1 + Ch;   x2 = X2 - Ch;
      X1 = X1 + C;   X2 = X2 - C;
      output4(x1, X1, x2, X2);
      D = D + Incl;
    }
    else if (D > 0) {
      if (D < V) {
        x1 = X1 + c;   x2 = X2 - c;
      }
      else if (D > V) {
        x1 = X1 + c + 1;   x2 = X2 - c - 1;
      }
      else {
        x1 = X1 + c + 1;   x2 = X2 - c;
      };
      X1 = X1 + C + 1;   X2 = X2 - C - 1;
```

```
        output4(x1, X1, x2, X2);
        D = D + Inc2;
      }
      else {        /* D = 0 */
        if (D < V) x1 = X1 + c;
        else x1 = X1 + c + 1;
        X1 = X1 + C + 1;
        x2 = X2 - Ch;   X2 = X2 - C;
        output4(x1, X1, x2, X2);
        i = i + 1;
        if (i > N) {
          D = Inc2;  goto L;
        }
        x1 = X1 + Ch;
        if (Inc1 > V) x2 = X2 - c - 1;
        else x2 = X2 - c;
        X1 = X1 + C;   X2 = X2 - C - 1;
        output4(x1, X1, x2, X2);
        D = Inc1 + Inc2;
      }
    }
  }
  else {     /* C + 1 is even */
```

```
Ch = (C + 1) >> 1;
for (i = 1; i <= N; i++) {
    if (D > 0) {
      x1 = X1 + Ch;   x2 = X2 - Ch;
      X1 = X1 + C + 1;   X2 = X2 - C - 1;
      output4(x1, X1, x2, X2);
      D = D + Inc2;
    }
    else if (D < 0) {
      if (D < V) {
        x1 = X1 + c;   x2 = X2 - c;
      }
      else if (D > V) {
        x1 = X1 + c + 1;   x2 = X2 - c - 1;
      }
      else {
        x1 = X1 + c + 1;   x2 = X2 - c;
      }
      X1 = X1 + C;   X2 = X2 - C;
      output4(x1, X1, x2, X2);
      D = D + Inc1;
    }
    else{        /* D = 0 */
```

```
            x1 = X1 + Ch;   X1 = X1 + C + 1;
            if (D > V) x2 = X2 - c - 1;
            else x2 = X2 - c;
            X2 = X2 - C;
            output4(x1, X1, x2, X2);
            i = i + 1;
            if (i > N) {
              D = Inc2;  goto L;
            }
            if (Inc2 < V) x1 = X1 + c;
            else x1 = X1 + c + 1;
            X1 = X1 + C;
            x2 = X2 - Ch;   X2 = X2 - C - 1;
            output4(x1, X1, x2, X2);
            D = Inc1 + Inc2;
          }
        }
      }
L:
      if (LeftOver == 0) return;
      if (LeftOver > 1) {
        if (D < V) x1 = X1 + c;
        else x1 = X1 + c + 1;
```

```
        if (D < 0) {
            X1 = X1 + C;    D = D + Inc1;
        }
        else {
            X1 = X1 + C + 1;    D = D + Inc2;
        }
        output2(x1, X1);
        if (LeftOver == 2) return;
    }
    if (D < V) x1 = X1 + c;
    else x1 = X1 + c + 1;
    output1(x1);
}
```

3.3.4 Complexity Analysis and Numerical Results

In this section we perform a complexity analysis for the double-step, bi-directional least-error integral interpolation algorithm so that we can compare the time efficiency of three least-error integral linear interpolation algorithms: Field's single-step algorithm, the double-step algorithm, and the double-step, bi-directional algorithm.

Since the complexity of the algorithm is dominated by the repetitive loops that perform the iterations, we count the operations in the loops. Referring to the pseudo code of the double-step, bi-directional interpolation algorithm, the algorithm has different **for** loops for C being even and odd, and we need choose only one case to analyze since the code segments for C being even and odd actually involve the

same amount of computation. Let us choose the case of C being even. In each iteration, the sign of the discriminator D is tested. Unlike the single-step and the double-step algorithms where we only need to test if $D \geq 0$ or $D < 0$, we have to discriminate between the cases of $D < 0$, $D > 0$ and $D = 0$. The algorithm first tests if $D < 0$. If this test is not passed, then it tests if $D > 0$. An average case analysis of the number of tests of D is difficult to accomplish. If we assume, however, that the $D > 0$ comparison is performed in half of the loop iterations, then we get a rough estimation of $\frac{3}{2}$ comparisons of D in each iteration. If $D < 0$ then there are 4 additions to obtain four new interpolation points, and 1 addition to update D. If $D > 0$ we have to compare D with V. Again a three-state logic is required to discriminate between the cases of $D < V$, $D > V$, and $D = V$. We therefore estimate the number of comparisons of D with V to be $\frac{3}{2}$ if $D > 0$. As with the case of $D < 0$ there are 4 additions to obtain 4 new interpolation points, and 1 addition to update D. The case of $D = 0$ is more complex. It actually performs two iterations and produces 8 new interpolation points, but updates D only once. There are two comparisons. One comparison of D with V and one comparison of $Inc2$ with V. If we approximate the cost of two comparisons to one addition, then there are a total of 5 additions in each iteration when $D = 0$ as in the case of $D < 0$. Finally we estimate the operations in each iteration by considering all the possible cases. As has been indicated, there are always 5 additions to produce 4 new interpolation points and to update D, and there are $\frac{3}{2}$ comparisons for discriminating between the cases of $D < 0$, $D > 0$, and $D = 0$. There are additionally $\frac{3}{2}$ comparisons if $D > 0$. Assume that the case of $D > 0$ occurs in half of the total loop iterations, then the estimated number of comparisons in each iteration is $\frac{3}{2} + \frac{3}{2} \cdot \frac{1}{2} = \frac{9}{4}$. Considering

that there are a total of $\frac{n}{4}$ iterations and that loop control for each iteration requires one comparison, the number of operations in repetitive loops are $\frac{5n}{4}$ additions and $\frac{9n}{16} + \frac{n}{4} = \frac{13n}{16}$ comparisons.

Complexity analyses for Field's single-step algorithm and for the double-step algorithm were previously performed in [Rokne 92]. We therefore refer to the previous results: The single-step algorithm requires $2n$ comparisons and $2n$ additions, and the double-step algorithm requires $\frac{5}{4}n$ comparisons and $\frac{3}{2}n$ additions. The ratios of operations between the double-step, bi-directional algorithm and the single-step algorithm are therefore 0.41 for comparisons and 0.63 for additions, and the ratios of operations between the double-step, bi-directional algorithm with the double-step algorithm are 0.65 for comparisons and 0.83 for additions.

We implemented and tested a number of algorithms using the programming language C. The algorithms were Field's single-step linear interpolation algorithm B5 (abbreviated as SS) in [Field 85], our previously presented double-step linear interpolation algorithm (abbreviated as DS) in [Rokne 92], and the double-step bi-directional linear interpolation algorithm (abbreviated as SYMDS) which was just discussed, as well as Graham and Iyengar's double- and triple-step linear interpolation algorithm (abbreviated as DTS) in [Graham 94]. These algorithms all perform least-error integral linear interpolation. The output was not implemented for each of the above algorithms so that only the arithmetic operations were counted. The algorithms were tested on an SGI INDIGO2 workstation. Timing measurements were made by using the UNIX command *time*. The comparison of the efficiencies was based on the accumulated running times in seconds. The interval over which the linear interpolation was performed was $[0, 500]$, i.e., $a = 0, b = 500$. When we tested

n	SS	DS	SYMDS	DTS	SYMDS/SS	SYMDS/DS
$50 \sim 53$	0.440	0.303	0.244	0.330	0.56	0.81
$100 \sim 103$	0.823	0.485	0.374	0.572	0.45	0.77
$200 \sim 203$	1.576	1.042	0.851	1.208	0.54	0.82
$400 \sim 403$	3.073	1.831	1.382	2.181	0.45	0.78
$800 \sim 803$	6.164	3.803	2.957	3.915	0.48	0.79
$1600 \sim 1603$	12.129	7.259	5.638	7.557	0.47	0.78
$3200 \sim 3203$	24.087	13.713	10.165	15.930	0.42	0.74

Table 3.1: Runtime comparison of four integral linear interpolation algorithms.

an algorithm, we used four consecutive values for n, i.e, the algorithm first ran with $n = k$, then $n = k + 1$, $k + 2$ and finally with $k + 3$. This process was repeated 5000 times to get the accumulated running time. These empirical results are tabulated in Table 3.1. The table also lists the ratios of running times of SYMDS over SS and SYMDS over DS for different n's. Noting that from the table the average ratio of the running time of SYMDS over the running time of SS is 0.48, and the average ratio of the running time of SYMDS over the running time of DS is 0.78, the results of theoretical complexity analysis therefore convincingly indicates the efficiency improvement by applying double-step, bi-directional interpolation.

In our test the double- and triple-step algorithm cannot beat the double-step algorithm. We believe that this is due to: 1) the increased overhead because of computing the value of $incr3$ which involves multiplication; 2) the use of counting variable i introduces one more addition in each iteration.

3.4 Summary

In this chapter three types of integral linear interpolation were defined. It was then shown how to derive recurrence formulas for three interpolation types based on incremental computation of integral linear interpolation which can be performed using integer arithmetic. Double-step algorithms reduce the number of iteration steps by half compared to single-step algorithms, and double-step, bi-directional algorithms reduce the number of iteration steps to 1/4 of the number of iteration steps of single-step algorithms. Because of increased overhead and code complexity in the iteration loop, double-step algorithms cannot double the speed, nor can the double-step, bi-directional algorithms quadruple the speed.

Chapter 4

Integral Linear Interpolation Approach to the Design of Incremental Line Algorithms

4.1 Introduction

Scan-conversion of line segments (called lines in the sequel) is an important basic graphics primitive. This is evidenced by the fact that graphics hardware tends to be benchmarked by the speed by which it can generate lines. Considerable interest has therefore been shown in designing efficient line scan-conversion algorithms. These algorithms select the pixels nearest to the line based on the geometry of a line relative to a coordinate grid as discussed in section 2.1. Most of the algorithms are incremental algorithms [Bao 89, Bresenham 65, Bresenham 82, Bresenham 85, Fung 92, Graham 93, Rokne 90, Wu 87]. Incremental algorithms are distinguished by the fact that they generate the digitized image of a line from one endpoint of a line to the other by selecting one or multiple pixels in each incremental step along a certain axis (x-axis for lines with slope between 0 and 1). The choice of which pixel(s) to set is made by testing the sign of a function called the discriminator. The discriminator obeys a simple recurrence formula which may be evaluated using only integer arithmetic. The first such algorithm was due to Bresenham [Bresenham 65] as mentioned earlier. His algorithm was easy to implement and it effectively set a standard for subsequent line scan conversion algorithms.

The number of pixels generated in each incremental step may be either fixed or variable. The algorithms which generate a fixed number of pixels in each incremental step are usually named according to the length of the incremental step along a chosen axis. We thus have single-step line algorithms which are represented by the first incremental line algorithm due to Bresenham; the double-step line algorithm of Wu and Rokne [Wu 87] and the quadruple-step line algorithm of Bao and Rokne [Bao 89]. The double-step algorithm [Wu 87] is similar to Bresenham's algorithm but it takes advantage of special double-step pixel patterns and therefore reduces the number of incremental steps by one half. The quadruple-step algorithm [Bao 89] generates four pixels in each incremental step at the cost of one to three decision tests, with the average being slightly less than two. More recently, Graham and Iyengar [Graham 94] presented a double- and triple-step incremental line algorithm with the main feature being a double-step line generator but setting a third pixel in some of the loop iterations. Gill [Gill 94] suggested N-step incremental line algorithms based on Bresenham's line algorithm. In Bresenham's algorithm, the sign of the discriminator predicts the pixel to be chosen at any step. In the N-step move, the changes to the discriminator for each step in the scan-conversion steps gives a set of equations that must be satisfied. These equations form a test set that predicts the N-step move in advance.

The incremental line algorithms which generate a variable number of pixels in each incremental step are based on the observation that the digitized image of a line with slope between 0 and 1 can be divided into slices of horizontal pixel runs. A run of pixels is generated in each incremental step. Line algorithms of this type are usually referred to as run-length slice algorithms [Bresenham 85, Fung 92]. A two-

state discriminator can be used to determine the length of the next run due to the property that the lengths of the runs are confined to two successive integers. This results in a scheme similar to the one used in algorithms generating a fixed number of pixels in each step.

The run-length properties of digitized lines were first investigated by Reggiori [Reggiori 72] who also obtained some theoretical results. Rosenfeld derived them from the chord property of a straight digital arc, the digitization of a straight line segment [Rosenfeld 74]. In the horizontal run-length slice line algorithm, the incremental direction is along the y-axis for calculating the horizontal runs of the line with slope between 0 and 1 since two consecutive horizontal runs have an increment of one in their ordinates. In [Bresenham 85] Bresenham used the property of the so-called complementary line of a line to calculate the diagonal runs of lines with slope between 0.5 and 1. The run-length slice algorithms in [Bresenham 85] can be viewed as a single-step algorithm in the sense that the incremental step is one in the direction of y-axis. Fung, Nicholl and Dewdney presented a double-step version of the run-length slice algorithm in [Fung 92] to further increase the efficiency of the line generation.

The structural approach is another major approach for digitized line generation discussed in the literature. These algorithms construct a string representing a digitized line on a grid in Freeman's coding scheme [Freeman 61]. The runs are derived from a structural orientation in these algorithms due to the interesting structural properties of the chain code representation of lines. Brons [Brons 74] and Cederberg [Cederberg 79] presented recursive algorithms to produce the chain code representations of digitized lines. Castle and Pitteway [Castle 87] presented another structural

line algorithm based on the palindromic symmetry in the chain code representation of a discrete line. Their algorithm uses Euclid's algorithm to control two symmetric production rules which construct the "best-fit" line.

This chapter discusses a new approach to the design of line algorithms of incremental type called the integral linear interpolation approach As has been mentioned above, the core of the existing incremental line algorithms consists of choosing a pixel or a group of pixels in each iteration step. The analyses leading to the algorithms are based on the geometry of digitized lines on the raster grid. Pixel patterns are studied individually for the derivation of single-step, double-step, and quadruple-step algorithms. From the view point of numerical computation, selecting pixels to approximate a continuous line amounts to a problem of linear interpolation. Because the pixels have integer coordinates, the integral linear interpolation is the natural computation model. In the simplest case, we first discuss the relationship between Bresenham's line algorithm and the least-error integral linear interpolation. We then illustrate how the variations of the original Bresenham's line algorithm such as double-step and quadruple-step incremental line algorithms can be derived by using double-step and quadruple-step least-error integral linear interpolation. This method also applies to the derivation of the incremental run-length slice line algorithms where rounding-up or rounding-down integral linear interpolations are used.

The topics in this chapter are well-known from other investigations, however, the approach is different from other relevant work. This results in a new treatment that unifies a considerable body of literature on incremental line drawing. In the remainder of this chapter we will therefore illustrate how existing incremental line algorithms can be derived by means of integral linear interpolation. To demonstrate

the power of this method, we will present a double-step, bi-directional run-length slice line algorithm by using double-step, bi-directional rounding-up integral linear interpolation.

The discussion of the line scan-conversion problem in this chapter is constrained to lines with slopes between 0 and 1 whose endpoint coordinates are integers. More specifically, we denote the two endpoints of the line by (x_s, y_s) and (x_e, y_e) and we assume that $x_e \geq x_s$ and $y_e \geq y_s$. Other lines with integer endpoints can be transformed to meet this condition via sign changes and/or coordinate exchanges. The slope of the line is denoted by $k = \Delta y/\Delta x$ where $\Delta x = x_e - x_s$ and $\Delta y = y_e - y_s$.

4.2 Bresenham Line Algorithm and Integral Linear Interpolation

Starting from $x = x_s, y = y_s$, Bresenham's algorithm generates the digitizeded image of a line by incrementing the integer abscissa by one in each iteration, then deciding which of the two neighboring pixels, $(x+1, y)$ and $(x+1, y+1)$ is closer to the true line and then moving to that position (see Figure 4.1). A discriminator is derived from the geometry of the true line and the two candidate pixels, which allows the algorithm to choose one of the two candidate pixels in each iteration step according to the sign of the discriminator. The decision made effectively rounds the real ordinate value of the point on the real line with integer abscissa $x+1$ to the nearest integer. If an equal error instance occurs, Bresenham's algorithm will round the real ordinate value up, i.e., choose the integer ordinate to be $y+1$. The discriminator is updated according to a simple recurrence formula. The following code constitutes the core of

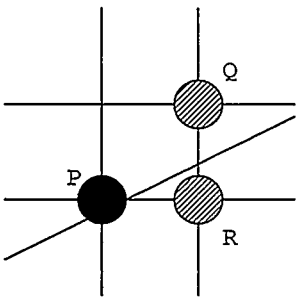


Figure 4.1: Choosing a pixel that is closer to the true line from two candidate pixels Q and R.

Bresenham's line algorithm for generating the lines with slopes between 0 and 1.

```
D = 2Δy − Δx;
for (i = 0; i <= Δx; i = i + 1) {
    plot(x, y);
    if (D >= 0) {
      y = y + 1;
      D = D − 2Δx;
    }
    x = x + 1;
    D = D + 2Δy;
}
```

In the above code D is the discriminator.

Let us now view the line generation problem from a different perspective. Denote the ordered set of abscissas of the pixels on the line from x_s to x_e by $\mathcal{X}$ and the ordered

set of ordinates of the pixels from y_s to y_e by $\mathcal{Y}$. There is a one to one correspondence between the elements of these two sets. Since $\Delta x \geq \Delta y$, x is increased by one each time a pixel is determined. Hence

$$\mathcal{X} = \{x_s, x_s + 1, \ldots, x_e\}.$$

The elements in $\mathcal{Y}$ are the result of the least-error integral linear interpolation performed on the interval $[y_s, y_e]$:

$$\mathcal{Y} = \{y_s = \dot{y}_0, \dot{y}_1, \ldots, \dot{y}_n = y_e\},$$

where $n = \Delta x = x_e - x_s$. Since $\Delta y = y_e - y_s < n$ there exist at least one $i, i = 0, 1, \ldots, n-1$, such that $\dot{y}_i = \dot{y}_{i+1}$. Replacing a, b by y_s, y_e and n by Δx in the least-error integral linear interpolation formulas in chapter 3, we get the following recurrence relations:

$$\dot{y}_{i+1} = \begin{cases} \dot{y}_i + c & D_i < 0 \\ \dot{y}_i + c + 1 & D_i \geq 0, \end{cases} \tag{4.1}$$

$$D_{i+1} = \begin{cases} D_i + 2r & D_i < 0 \\ D_i + 2(r - \Delta x) & D_i \geq 0. \end{cases} \tag{4.2}$$

The initial value of the discriminator becomes

$$D_0 = 2r - \Delta x. \tag{4.3}$$

The relationship between Eq. (4.1),(4.2) and (4.3) and Bresenham line algorithm is revealed by examining the values of c and r, which can be trivially determined:

$$c = \begin{cases} 0 & 0 \leq k < 1\,(\Delta y < \Delta x) \\ 1 & k = 1\,(\Delta x = \Delta y), \end{cases} \tag{4.4}$$

$$r = \begin{cases} \Delta y & c = 0 \\ 0 & c = 1. \end{cases} \tag{4.5}$$

In the case of $c = 0$ Eq. (4.1) and (4.2) can be simplified to

$$\dot{y}_{i+1} = \begin{cases} \dot{y}_i, & D_i < 0 \\ \dot{y}_i + 1, & D_i \geq 0, \end{cases} \tag{4.6}$$

$$D_{i+1} = \begin{cases} D_i + 2\Delta y & D_i < 0 \\ D_i + 2(\Delta y - \Delta x), & D_i \geq 0. \end{cases} \tag{4.7}$$

The initial value of the discriminator becomes

$$D_0 = 2\Delta y - \Delta x. \tag{4.8}$$

These recurrence relations are identical to those employed in Bresenham's algorithm. In the case of $c = 1$, i.e., $\Delta x = \Delta y$, the line forms a 45 degree angle with respect to the positive x-direction, the recurrence formulas become

$$\dot{y}_{i+1} = \begin{cases} \dot{y}_i + 1 & D_i < 0 \\ \dot{y}_i + 2 & D_i \geq 0, \end{cases} \tag{4.9}$$

$$D_{i+1} = \begin{cases} D_i & D_i < 0 \\ D_i - 2\Delta x & D_i \geq 0, \end{cases} \tag{4.10}$$

and the initial value of the discriminator becomes

$$D_0 = -\Delta x. \tag{4.11}$$

These formulas are different from the formulas used in Bresenham's algorithm. But since the initial value of the discriminator is negative, and according to Eq. (4.10) will never change in the process of iteration, therefore, the value of $\dot{y}_i$ will increase by one in each iteration to form a 45 degree move according to Eq. (4.9). In a practical implementation of line scan-conversion, however, the case of $\Delta x = \Delta y$ is typically excluded from the code for arbitrary slope lines and is treated as an exceptional instance for even faster pixel selections. We therefore conclude that performing the least-error integral linear interpolation over the interval $[y_s, y_e]$ results in a line algorithm which is equivalent to Bresenham's line algorithm.

4.3 Double-Step Line Algorithm and Integral Linear Interpolation

4.3.1 Double-Step Line Algorithm

The double-step line algorithm [Wu 87], which improves the efficiency of Bresenham's algorithm, is based on the observation that if a point (x_i, y_i) at the lower left corner of a 2×2 mesh representing an already plotted pixel in the line with slope between 0 and 1 is given, then only four pixel patterns shown in Figure 4.2 can be formed in

a double-step increment in the x-direction under the conditions on the line given in section 4.1. The double-step algorithm therefore proceeds as follows. Starting from (x_s, y_s), the x coordinate is incremented by two raster units. If the pixel at the right-lower (the right-upper) corner of the 2×2 mesh is selected, then it is clear that pattern 1 (4) occurs. This means that in both cases the middle pixel can be plotted with no extra computation. If pattern 2 or 3 occurs (abbreviated pattern 2 (3) in the sequel), then some extra work has to be done in order to distinguish which of the two patterns have to be plotted. It was conjectured by Freeman [Freeman 61, Freeman 70] and proved by Reggiori [Reggiori 72] (see also [Rokne 90, Wu 87]) that only two pattern types may occur simultaneously: either 1 and 2 (3) or 2 (3) and 4. From these results the double-step strategy is given by

1. If $0 \leq k < 0.5$, then

$$D_0 = 4\Delta y - \Delta x, \tag{4.12}$$

$$D_{i+1} = \begin{cases} D_i + 4\Delta y & D_i < 0 \text{ (pattern 1)} \\ D_i + 4\Delta y - 2\Delta x & D_i \geq 0 \text{ (pattern 2 or 3).} \end{cases} \tag{4.13}$$

2. If $0.5 \leq k \leq 1$, then

$$D_0 = 4\Delta y - 3\Delta x, \tag{4.14}$$

$$D_{i+1} = \begin{cases} D_i + 4\Delta y - 2\Delta x & D_i < 0 \text{ (pattern 2 or 3)} \\ D_i + 4(\Delta y - \Delta x) & D_i \geq 0 \text{ (pattern 4).} \end{cases} \tag{4.15}$$

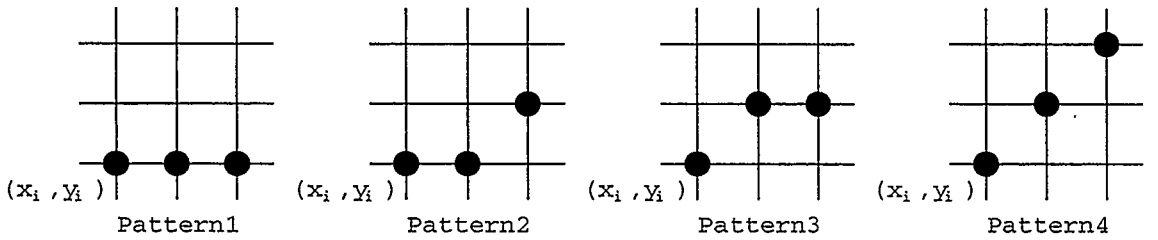


Figure 4.2: The four double-step patterns.

To distinguish between pattern 2 and 3 requires the test

$$D_i < \begin{cases} 2\Delta y & \text{if } 0 \leq k < 0.5 \\ 2(\Delta y - \Delta x) & \text{if } 0.5 \leq k \leq 1 \end{cases} \tag{4.16}$$

resulting in pattern 2 if the test is passed, pattern 3 if not.

4.3.2 Designing Double-Step Line Algorithm Using Double-Step Integral Linear Interpolation

Wu and Rokne's double-step line algorithm can easily be derived by means of double-step least-error integral linear interpolation since what we really need to calculate is the set of y-coordinates of the pixels on the digitized line, and it is readily understood that this can be achieved by performing double-step linear interpolation over the interval $[y_s, y_e]$. If a, b are replaced by y_s, y_e, $\dot{X}, \dot{x}$ by $\dot{Y}, \dot{y}$, and n by Δx in the formulas for double-step least-error integral linear interpolation given in chapter 3 then the following recurrence formulas for calculating y-coordinates of the pixels are obtained.

$$\dot{Y}_{i+1} = \begin{cases} \dot{Y}_i + C & D_i < 0 \\ \dot{Y}_i + C + 1 & D_i \geq 0, \end{cases} \tag{4.17}$$

$$D_{i+1} = \begin{cases} D_i + 2R & D_i < 0 \\ D_i + 2(R - \Delta x) & D_i \geq 0 \end{cases} \tag{4.18}$$

where $\dot{Y}_i = \dot{y}_{2i}$ and where C is interpreted as $\lfloor 2\Delta y/\Delta x \rfloor = \lfloor 2k \rfloor$ so that

$$\dot{y}_{2i+1} = \begin{cases} \dot{Y}_i + c & D_i < 2(R - r) \\ \dot{Y}_i + c + 1 & D_i \geq 2(R - r). \end{cases} \tag{4.19}$$

The initial value of the discriminator becomes

$$D_0 = 2R - \Delta y. \tag{4.20}$$

Noting that the values of c and r are determined according to Eq. (4.4) and (4.5) the values of C and R can be determined by the the following equations:

$$C = \begin{cases} 0 & 0 \leq k < 0.5 \\ 1 & 0.5 \leq k < 1 \\ 2 & k = 1, \end{cases} \tag{4.21}$$

$$R = \begin{cases} 2\Delta y & C = 0 \\ 2\Delta y - \Delta x & C = 1 \\ 2(\Delta y - \Delta x) & C = 2. \end{cases} \tag{4.22}$$

We can now easily see that the above recurrence formulas for double-step integral linear interpolation over the interval $[y_s, y_e]$ are identical to the recurrence formulas used for double-step line algorithm in [Wu 87] for the case of $0 \leq k < 1$, and they are correct in the case of $k = 1$ though the resulting formulas are different from those of

double-step line algorithm. We omit the derivation here since it is quite easy.

The advantage of using linear interpolation technique in designing a double-step line algorithm is obvious. The discussion on double-step pixel patterns and the differentiation of the two cases according to whether the slope of a line is greater or less than 0.5 is no longer needed. Only one set of recurrence formulas is derived which corresponds to different sets of formulas derived in [Wu 87] depending on the value of $k = \frac{\Delta y}{\Delta x}$. In an analogous manner we can design a quadruple-step least-error integral linear interpolation algorithm, and it is readily understood that the quadruple-step line generation [Bao 89] is just an application of the quadruple-step least error integral linear interpolation.

4.4 Run-Length Slice Line Algorithms And Integral Linear Interpolation

4.4.1 Horizontal Run-length Slice Line Algorithms

We have shown that the problem of line generation can be reduced to the problem of least-error integral linear interpolation over the interval $[y_s, y_e]$ with $n = \Delta x$. This results in the conventional incremental line algorithms such as Bresenham's line algorithm and the double-step line algorithm. We now show that the run-length slice line algorithms can also be derived from integral linear interpolation. Referring to Figure 4.3, the digitized image of a line segment can be broken down into $\Delta y + 1 = y_e - y_s + 1$ horizontal runs of pixels with the ordinates $y_s, y_s + 1, \ldots, y_e$. A set of midlines $y = y_s + i - 0.5, i = 0, 1, \ldots, \Delta y + 1$ is imposed on the grid coordinate. Denoting by x_i the x-coordinate of the intersection of the line passing

through (x_s, y_s) and (x_e, y_e) with line $y = y_s + i - 0.5$, we observe that except for the 0-th run which starts from x_s, the i-th run, $i = 1, 2, \ldots, \Delta y$, starts from $\lceil x_i \rceil$, which is a rounding-up integral linear interpolation point on the interval $[x_0, x_{\Delta y+1}]$. This is also true for the case of x_i being integral, which happens if the line passes the midpoint of two vertically adjacent grid points (see Figure 4.4). Thus the calculation of horizontal pixel runs for the line segment from (x_s, y_s) to (x_e, y_e) can be done by performing the rounding-up integral linear interpolation over the interval $[x_0, x_{\Delta y+1}]$ where $x_0 = x_s - \frac{\Delta x}{2\Delta y}$ and $x_{\Delta y+1} = x_e + \frac{\Delta x}{2\Delta y}$. A single-step interpolation scheme will result in a single-step run-length slice line algorithm where a run of pixels is generated in each iteration step such as the algorithm given in [Bresenham 85], and a double-step scheme will result in a double-step run-length slice line algorithm which generates two runs of pixels in each iteration such as the algorithm presented in [Fung 92]. Furthermore, by incorporating a bi-directional interpolation technique, we can derive a double-step, bi-directional run-length slice line algorithm. This algorithm, which has not been presented previously in the literature, will now be developed in detail as a case study. The derivation is similar to what we have done in chapter 3 for the derivation of double-step, bi-directional, least-error integral linear interpolation algorithm. We first derive the recurrence formulas for the double-step forward and backward rounding-up integral linear interpolations for this particular problem, and we then combine these to obtain a double-step, bi-directional run-length line scan-conversion algorithm.

Note that the distance between x_i and x_{i+1} is

$$(x_e + \frac{\Delta x}{2\Delta y} - x_s + \frac{\Delta x}{2\Delta y})/(\Delta y + 1) = \Delta x/\Delta y = k,$$

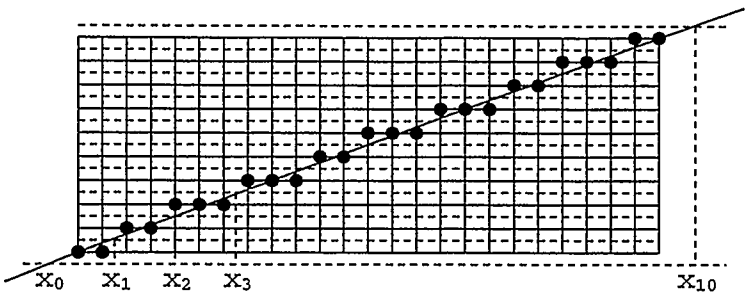


Figure 4.3: Horizontal pixel runs in a digitized line segment. Dashed lines are mid-lines passing midway between vertically adjacent grid points.

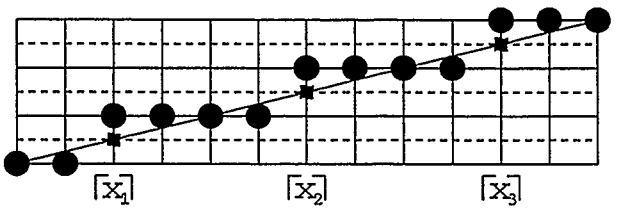


Figure 4.4: A line with equal error cases.

which means that x_i can be represented by

$$x_i = x_0 + \frac{\Delta x}{\Delta y} i = x_s - \frac{\Delta x}{2\Delta y} + i\frac{\Delta x}{\Delta y}.$$

Since both Δx and Δy are integral, we use the notation defined in chapter 3:

$$c = \lfloor \frac{\Delta x}{\Delta y} \rfloor = \lfloor k \rfloor,$$

$$r = \Delta x \bmod \Delta y,$$

$$C = \lfloor \frac{2\Delta x}{\Delta y} \rfloor,$$

$$R = 2\Delta x \bmod \Delta y,$$

$$\tilde{c} = \lfloor \frac{\Delta x}{2\Delta y} \rfloor,$$

$$\tilde{r} = \Delta x \bmod 2\Delta y,$$

$$\hat{c} = \lceil \frac{\Delta x}{2\Delta y} \rceil.$$

Letting $X_i = x_{2i}$ and $\dot{X}_i = \lceil X_i \rceil$, we have

$$X_i \leq \dot{X}_i < X_i + 1, \tag{4.23}$$

$$X_{i+1} \leq \dot{X}_{i+1} < X_{i+1} + 1, \tag{4.24}$$

Subtracting Eq. (4.23) from Eq. (4.24) yields

$$2k - 1 < \dot{X}_{i+1} - \dot{X}_i < 2k + 1,$$

which implies that

$$\dot{X}_{i+1} - \dot{X}_i = C \text{ or } C + 1.$$

A similar derivation establishes

$$\dot{x}_{i+1} - \dot{x}_i = c \text{ or } c + 1.$$

We therefore have

$$\dot{X}_{i+1} = \begin{cases} \dot{X}_i + C & X_{i+1} - \dot{X}_i - C \leq 0 \\ \dot{X}_i + C + 1 & X_{i+1} - \dot{X}_i - C > 0. \end{cases}$$

Defining the discriminator to be

$$D_i = 2\Delta y(X_{i+1} - \dot{X}_i - C) \tag{4.25}$$

we have

$$\dot{X}_{i+1} = \begin{cases} \dot{X}_i + C & D_i \leq 0 \\ \dot{X}_i + C + 1 & D_i > 0. \end{cases} \tag{4.26}$$

Substituting X_{i+1} by $x_s - \frac{\Delta x}{2\Delta y} + 2(i+1)\frac{\Delta x}{\Delta y}$ in Eq. (4.25) we obtain

$$\begin{aligned} D_i &= 2\Delta y(X_{i+1} - \dot{X}_i - C) \\ &= 2\Delta y(x_s - \frac{\Delta x}{2\Delta y} + 2(i+1)\frac{\Delta x}{\Delta y} - \dot{X}_i - C) \\ &= 2\Delta y x_s - \Delta x + 4(i+1)\Delta x - 2\Delta y\dot{X}_i - 2\Delta y C. \end{aligned}$$

D_{i+1} can be represented in terms of D_i:

$$\begin{aligned} D_{i+1} &= 2\Delta y x_s - \Delta x + 4(i+2)\Delta x - 2\Delta y\dot{X}_{i+1} - 2\Delta y C \\ &= D_i + 4\Delta x - 2\Delta y(\dot{X}_{i+1} - \dot{X}_i) \\ &= \begin{cases} D_i + 4\Delta x - 2\Delta y C & D_i \leq 0 \\ D_i + 4\Delta x - 2\Delta y(C+1) & D_i > 0 \end{cases} \\ &= \begin{cases} D_i + 2R & D_i \leq 0 \\ D_i + 2R - 2\Delta y & D_i > 0. \end{cases} \end{aligned} \tag{4.27}$$

The point between $\dot{X}_i$ and $\dot{X}_{i+1}$, i.e., $\dot{x}_{2i+1}$ can be calculated according to the fol-

lowing equation:

$$\dot{x}_{2i+1} = \begin{cases} \dot{X}_i + c & x_{2i+1} - \dot{X}_i - c \leq 0 \\ \dot{X}_i + c + 1 & x_{2i+1} - \dot{X}_i - c > 0. \end{cases}$$

Noting that

$$\begin{aligned} 2\Delta y(x_{2i+1} - \dot{X}_i - c) &= 2\Delta y(x_s - \frac{\Delta x}{2\Delta y} + (2i+1)\frac{\Delta x}{\Delta y} - \dot{X}_i - c) \\ &= 2\Delta y x_s - \Delta x + 2(2i+1)\Delta x - 2\Delta y \dot{X}_i - 2\Delta y c \\ &= D_i - 2\Delta x + 2\Delta y C + 2\Delta y c \\ &= D_i - 2\Delta x + 2(2\Delta x - R) - 2(\Delta x - r) \\ &= D_i - 2(R - r) \end{aligned}$$

it follows that D_i can also be used to decide the value of $\dot{x}_{2i+1}$:

$$\dot{x}_{2i+1} = \begin{cases} \dot{X}_i + c & D_i \leq 2(R - r) \\ \dot{X}_i + c + 1 & D_i > 2(R - r). \end{cases} \tag{4.28}$$

If $\dot{X}_{i+1} - \dot{X}_i = 2\alpha$, where α is integral, we simply have

$$\dot{x}_{2i+1} = \dot{X}_i + \alpha.$$

To start the iteration, the initial values of $\dot{X}_i$ and D_i are:

$$\dot{X}_0 = \lceil x_s - \frac{\Delta x}{2\Delta y} \rceil = x_s - \lfloor \frac{\Delta x}{2\Delta y} \rfloor = x_s - \tilde{c}, \tag{4.29}$$

and

$$\begin{aligned} D_0 &= 2\Delta y x_s - \Delta x + 4\Delta x - 2\Delta y \dot{X}_0 - 2\Delta y C \\ &= 2\Delta y(x_s - \dot{X}_0) - 2\Delta y C + 3\Delta x \\ &= 2\Delta y \tilde{c} - 2\Delta y C + 3\Delta x \\ &= \Delta x - \tilde{r} - 2(2\Delta x - R) + 3\Delta x \\ &= 2R - \tilde{r}. \end{aligned} \tag{4.30}$$

The recurrence formulas for the double-step backward interpolation can be derived in a similar manner. Here we define $x'_0 = x_e + \frac{\Delta x}{2\Delta y}$, $x'_i = x'_0 - i\frac{\Delta x}{\Delta y}$, and $X'_i = x'_{2i}$. The interpolation points are $\dot{x}'_i = \lceil x'_i \rceil$ for $i = 0, 1, \ldots, \Delta y + 1$. To interpolate in a double-step manner we define $X'_i = x'_{2i}$ and $\dot{X}'_i = \lceil X'_i \rceil$. The formulas are listed below:

$$\begin{aligned} \dot{X}'_0 &= \lceil x_e + \frac{\Delta x}{2\Delta y} \rceil = x_e + \lceil \frac{\Delta x}{2\Delta y} \rceil \\ &= x_e + \hat{c} \\ &= \begin{cases} x_e + \tilde{c} & \tilde{r} = 0 \\ x_e + \tilde{c} + 1 & \tilde{r} > 0, \end{cases} \end{aligned} \tag{4.31}$$

$$D'_0 = \begin{cases} 2R - 2\Delta y & \tilde{r} = 0 \\ 2R - \tilde{r} & \tilde{r} > 0, \end{cases} \tag{4.32}$$

$$\dot{X}'_{i+1} = \begin{cases} \dot{X}'_i - C & D'_i < 0 \\ \dot{X}'_i - C - 1 & D'_i \geq 0, \end{cases} \tag{4.33}$$

$$D'_{i+1} = \begin{cases} D'_i + 2R & D'_i < 0 \\ D'_i + 2R - 2\Delta y & D'_i \geq 0, \end{cases} \tag{4.34}$$

$$\dot{x}'_{2i+1} = \begin{cases} \dot{X}'_i - c & D'_i < 2(R-r) \\ \dot{X}'_i - c - 1 & D'_i \geq 2(R-r). \end{cases} \tag{4.35}$$

To explore the possibility of bi-directional computing, we compare the corresponding formulas for the two sets of equations, Eq. (4.26)-(4.30) and Eq. (4.31)-(4.35). Again the recurrence formulas in the two sets of equations are similar except for equality cases. The initial values of the discriminator D and D' may be different or same the depending on the value of $\tilde{r}$. Further investigation reveals that

1. If $R = 0$ and $\tilde{r} = 0$ then $D_0 = 0$ and therefore $D_i = 0$ for all i. Also $D'_0 = -2\Delta y < 0$ and therefore $D'_i = -2\Delta y < 0$ for all i. Hence

$$\dot{X}_{i+1} = \dot{X}_i + C,$$

$$\dot{X}'_{i+1} = \dot{X}'_i - C.$$

   To calculate $\dot{x}_{2i+1}$ and $\dot{x}'_{2i+1}$, we only need to compare $2(R-r)$ with 0 and with $-2\Delta y$ according to Eq. (4.28) and Eq. (4.35).

2. If $\tilde{r} > 0$ then $D_0 = D'_0 = 2R - \tilde{r}$, hence we can use D as the discriminator to perform bi-directional interpolation. Special considerations are necessary when $D_i = D'_i = 0$.

3. If $\tilde{r} = 0$ and $R > 0$ then $D_0 = 2R > 0$ and $D'_0 = 2R - 2\Delta y < 0$. But after a first

double-step advance in both directions, the equality of D and D' is restored:

$$D_1 = 2R + 2R - 2\Delta y = 4R - 2\Delta y,$$

$$D'_1 = 2R - 2\Delta y + 2R = 4R - 2\Delta y = D_1.$$

Therefore bi-directional interpolation can be executed using only one discriminator.

In the same manner as for the bi-directional least-error integral linear interpolation discussed in chapter 3, the only problem with the bi-directional rounding-up integral linear interpolation using only one discriminator is that if $D_i = D'_i = 0$ for some i then D_{i+1} and D'_{i+1} assume different values ($D_{i+1} = 2R > 0$ and $D'_{i+1} = 2R - 2\Delta y < 0$). But again the equality of D and D' is restored for $D_{i+2} = D'_{i+2} = 2R - 2\Delta y$. The double-step and bi-directional calculation for all the $\dot{x}_i$'s for this case is quite similar to that of the least-error integral linear interpolation given in chapter 3 with special treatment for the case of $D_0 \neq D'_0$, which, however, does not significantly complicate the algorithm. If we keep track of the ordinates of runs in both directions in the algorithm and write pixel runs instead of output $\dot{x}_i$'s, we obtain a double-step, bi-directional run-length line scan-conversion algorithm.

We have shown that integral linear interpolation can be used as a unified framework for the derivation of incremental line algorithms. As a rather complicated example, we now present the complete algorithm using a C like pseudo-code for double-step, bi-directional run-length slice line generation based on the mathemati-

cal and logical foundations established so far.

ALGORITHM 4.1: Double-step, bi-directional run-length line generation

$dx = xe - xs;$

$dy = ye - ys;$

$c = dx/dy; \quad r = dx\%dy;$

$C = c << 1; \quad R = r << 1;$

if $(R >= dy)$ {

$\quad C = C + 1; \quad R = R - dy;$

}

$\tilde{c} = c >> 1;$

if (c is odd) $\tilde{r} = dy + r;$

else $\tilde{r} = r;$

$Inc1 = R << 1;$

$Inc2 = Inc1 - (dy << 1);$

$D = Inc1 - \tilde{r};$

$V = (R - r) << 1;$

$N = (dy + 1) >> 2;$

$LeftOver = dy + 1 - (N << 2);$

$X1 = xs - \tilde{c};$

if $(\tilde{r} == 0)$ $X2 = xe + \tilde{c};$

else $X2 = xe + \tilde{c} + 1;$

if $(N == 0)$ **goto** L;

```
if (R == 0 && r̃ == 0) {
    if (V >= 0) c1 = c;
    else c1 = c + 1;
    if (-(dy << 1) < V) c2 = c;
    else c2 = c + 1;
    for (i = 0; i < N; i = i + 1) {
        x1 = X1 + c1;    x2 = X2 - c2;
        X1 = X1 + C;     X2 = X2 - C;
        DrawTwoRuns(ys, xs, x1, X1 - 1);
        DrawTwoRuns(ye - 1, X2, x2, xe);
        ys = ys + 2;     ye = ye - 2;
        xs = X1;     xe = X2 - 1;
    }
    D = 0;
    goto L;
}
if (r̃ == 0 && R > 0) {
    if (Inc1 <= V) x1 = X1 + c;
    else x1 = X1 + c + 1;
    if (Inc1 + Inc2 < V) x2 = X2 - c;
    else x2 = X2 - c - 1;
    X1 = X1 + C + 1;    X2 = X2 - C;
    DrawTwoRuns(ys, xs, x1, X1 - 1);
```

```
        DrawTwoRuns(ye − 1, X2, x2, xe);
        xs = X1;    xe = X2 − 1;
        ys = ys + 2;    ye = ye − 2;
        D = Inc1 + Inc2;
        N = N − 1;
    }
    if (C is even) {
        Ch = C >> 1;
        for (i = 0; i < N; i = i + 1) {
            if (D< 0) {
                x1 = X1 + Ch;    x2 = X2 − Ch;
                X1 = X1 + C;    X2 = X2 − C;
                DrawTwoRuns(ys, xs, x1, X1 − 1);
                DrawTwoRuns(ye − 1, X2, x2, xe);
                xs = X1;    xe = X2 − 1;
                ys = ys + 2;    ye = ye − 2;
                D = D + Inc1;
            }
            else if (D > 0) {
                if (D< V) {
                    x1 = X1 + c;    x2 = X2 − c;
                }
                else if (D > V) {
```

```
            x1 = X1 + c + 1;    x2 = X2 - c - 1;
        }
        else {
            x1 = X1 + c;    x2 = X2 - c - 1;
        }
        X1 = X1 + C + 1;    X2 = X2 - C - 1;
        DrawTwoRuns(ys, xs, x1, X1 - 1);
        DrawTwoRuns(ye - 1, X2, x2, xe);
        xs = X1;    xe = X2 - 1;
        ys = ys + 2;    ye = ye - 2;
        D = D + Inc2;
    }
    else {    /* D = 0 */
        x1 = X1 + Ch;    X1 = X1 + C;
        if (D < V) x2 = X2 - c;
        else x2 = X2 - c - 1;
        X2 = X2 - C - 1;
        DrawTwoRuns(ys, xs, x1, X1 - 1);
        DrawTwoRuns(ye - 1, X2, x2, xe);
        xs = X1;    xe = X2 - 1;
        ys = ys + 2;    ye = ye - 2;
        i = i + 1;
        if (i == N) {
```

```
            D = Incl;    break;
        }
        if (Incl <= V) x1 = X1 + c;    /* D = Incl after D = 0 */
        else x1 = X1 + c + 1;
        X1 = X1 + C + 1;
        x2 = X2 - Ch;    X2 = X2 - C;
        DrawTwoRuns(ys, xs, x1, X1 - 1);
        DrawTwoRuns(ye - 1, X2, x2, xe);
        xs = X1;    xe = X2 - 1;
        ys = ys + 2;    ye = ye - 2;
        D = Incl + Inc2;
    }
}
else {    /* C+1 is even */
    Ch = (C + 1) >> 1;
    for (i = 0; i < N; i = i + 1 {
        if (D > 0) {
            x1 = X1 + Ch;    x2 = X2 - Ch;
            X1 = X1 + C + 1;    X2 = X2 - C - 1;
            DrawTwoRuns(ys, xs, x1, X1 - 1);
            DrawTwoRuns(ye - 1, X2, x2, xe);
            xs = X1;    xe = X2 - 1;
            ys = ys + 2;    ye = ye - 2;
```

```
        D = D + Inc2;
    }
    else if (D < 0) {
        if (D< V) {
            x1 = X1 + c;    x2 = X2 - c;
        }
        else if (D > V) {
            x1 = X1 + c + 1;    x2 = X2 - c - 1;
        }
        else {
            x1 = X1 + c;    x2 = X2 - c - 1;
        }
        X1 = X1 + C;    X2 = X2 - C;
        DrawTwoRuns(ys, xs, x1, X1 - 1);
        DrawTwoRuns(ye - 1, X2, x2, xe);
        xs = X1;    xe = X2 - 1;
        ys = ys + 2;    ye = y2 - 2;
        D = D + Inc1;
    }
    else {    /* D = 0 */
        if (D <= V) x1 = X1 + c;
        else x1 = X1 + c + 1;
        X1 = X1 + C;
```

```
                x2 = X2 − Ch;    X2 = X2 − C − 1;
                DrawTwoRuns(ys, xs, x1, X1 − 1);
                DrawTwoRuns(ye − 1, X2, x2, xe);
                xs = X1;    xe = X2 − 1;
                ys = ys + 2;    ye = ye − 2;
                i = i + 1;
                if (i == N) {
                    D = Inc1;    break;
                }
                x1 = X1 + Ch;    X1 = X1 + C + 1;
                if (Inc2 < V) x2 = X2 − c;    /* D' = Inc1  after  D' = 0 */
                else x2 = X2 − c − 1;
                X2 = X2 − C;
                DrawTwoRuns(ys, xs, x1, X1 − 1);
                DrawTwoRuns(ye − 1, X2, x2, xe);
                xs = X1;    xe = X2 − 1;
                ys = ys + 2;    ye = ye − 2;
                D = Inc1 + Inc2;
            }
        }
    }
L :  /* Handling of leftover runs */
    switch (LeftOver) {
```

```
        case 3:
        case 2:
            if (D <= V) x1 = X1 + c;
            else x1 = X1 + c + 1;
            if (D <= 0) X1 = X1 + C;
            elseX1 = X1 + C + 1;
            DrawTwoRuns(ys, xs, x1, X1 - 1);
            if (LeftOver == 2) break;
            ys = ys + 2;
        case 1:
            DrawRun(ys, X1, X2 - 1);
    }
```

In the above code procedure DrawTwoRuns$(y, x1, x2, x3)$ draws two horizontal pixel runs, one from $x1$ to $x2-1$ with ordinate y, and another from $x2$ to $x3$ with ordinate $y+1$.

4.4.2 Diagonal Run-Length Slice Line Algorithm

We have shown that if the slope of a line is between 0 and 1, its digitized image can be divided into slices of horizontal runs so that multiple pixels can be rendered in each iteration, as shown above. When the slope of the line is greater than 0.5, however, the length of each horizontal run reduces to 1 or 2 since $c = \lfloor \Delta x/\Delta y \rfloor = 1$. In this case, the advantage of horizontal run-length algorithms is greatly reduced.

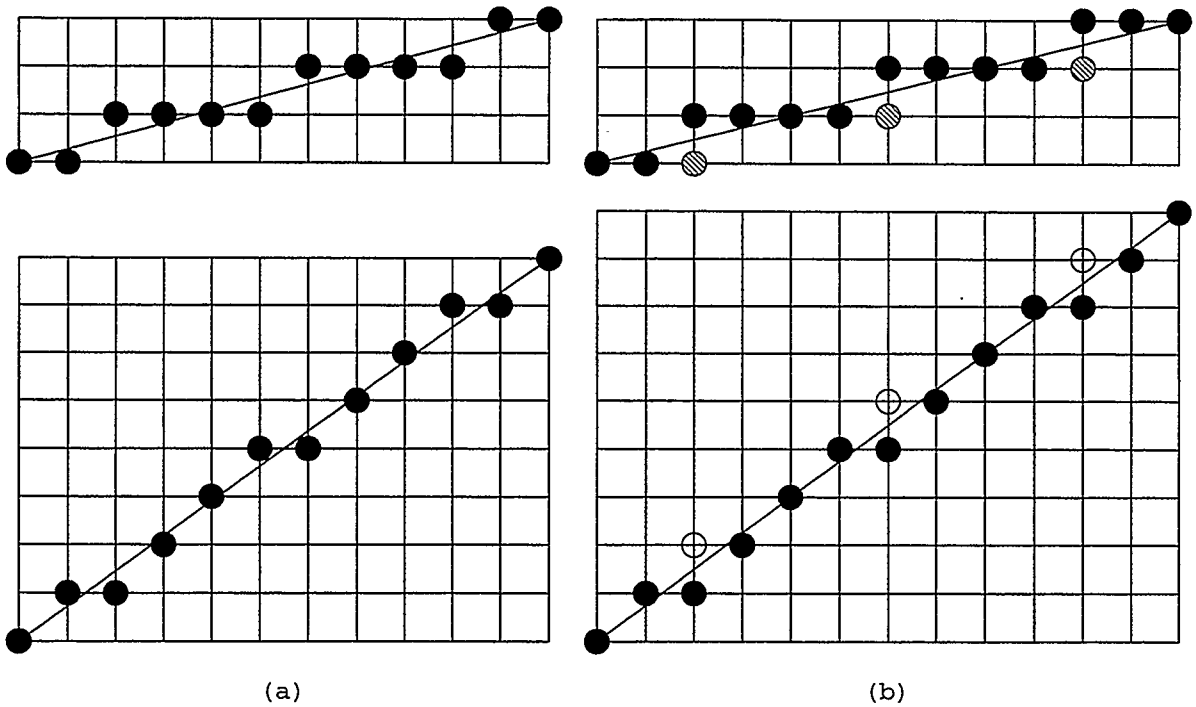


Figure 4.5: Lines (bottom) and their complementary lines (top). Diagonal runs in lines are obtained by calculating horizontal runs of the complementary lines. Empty circles in (b) denote the pixels which should be generated by Bresenham line algorithm. This suggests that we use rounding-down integral linear interpolation to calculate the end positions of horizontal runs of the complementary line. The hatched circles denote this adjustment.

In the extreme case of the slope being 1, the length of each run reduces to 1 and we actually obtain a single-step algorithm. We can, however, divide the line into diagonal runs since the starting of a new horizontal run is equivalent to a diagonal move of the pixel and each horizontal move of a pixel can be viewed as the starting of a new diagonal run.

The diagonal runs of a line can be obtained by calculating the horizontal runs for its complementary line [Bresenham 85]. For a line with $\Delta y > 0.5\Delta x$, i.e., the slope of the line is greater than 0.5, we compute the horizontal runs of its complementary

line starting from (x_s, y_s) and ending at (x_e, y'_e) where $y'_e = \Delta x - y_e$, and change the role of horizontal steps to diagonal steps to obtain the complementary incremental step sequence of the line to be drawn (see Figure 4.5(a)). Interchanging the role of horizontal steps and diagonal steps also interchanges the step choice for equal error instances. So if we use a horizontal run-length slice algorithm to compute the diagonal runs of a line with slope > 0.5, the equal error default is a horizontal move (see Figure 4.5(b)), resulting in a discrepancy between lines generated by Bresenham line algorithm and lines generated in this manner. This discrepancy can be eliminated by obtaining the horizontal runs by calculating the end positions of the horizontal runs of its complementary line using rounding-down integral linear interpolation and then changing horizontal runs to diagonal runs. Thus $\dot{x}_{i-1} = \lfloor x_{i-1} \rfloor$ is the end of the i-th run, and we simply obtain a diagonal run which ends at abscissa $\dot{x}_i$. In the case of drawing a line with slope less than or equal to 0.5, we still use rounding-up integral linear interpolation to calculate the horizontal runs of the line since a digitized line generated in this manner is identical to the digitized line generated by Bresenham's algorithm even if there exist equal error instances. We will not detail the derivation of recurrence formulas for diagonal run-length calculation by means of rounding-down integral linear interpolation since it is quite similar to the calculation horizontal runs by means of rounding-up integral linear interpolation.

4.5 Summary

We have presented a new approach to the design of incremental line algorithms in this chapter which reduces the problem of designing incremental line algorithms to

the problem of integral linear interpolation over the y extent or the x extent of a line. This new treatment unifies a considerable body of literature on incremental line drawing and it has the advantage of simplifying derivations. The variations of the original Bresenham line algorithm become natural extensions of the algorithm in this framework, and the properties of digitized lines upon which the double-step and the run-length slice line algorithms are based are the direct results of the integral linear interpolation. As a complex case study we derived a double-step, bi-directional run-length slice line algorithm.

A double-step run-length line scan-conversion algorithm was presented previously by Fung *et al* [Fung 92]. Bi-directional double-step generation of runs is also suggested in their paper without giving the technical details. We also note that the run-lengths are, in general, not symmetric, and thus copying two runs generated in one direction to another direction usually results in incorrect digitization. This is evidenced by Figure 4.3. The recurrence formulas for both forward and backward interpolation, and the discussions of their relations in this chapter result in the correct generations of runs in both direction simultaneously. Note that our approach is completely different from that of Fung. The integral linear interpolation approach adopted in this chapter avoids the tedious derivation of the run-lengths.

Chapter 5

Applying Integral Linear Interpolation to the Scan-Conversion of Filled Polygons

5.1 Introduction

The application of integral linear interpolation to the design of line scan-conversion algorithms was discussed in chapter 4. In particular, we showed that rounding-up and rounding-down integral linear interpolation can be employed to design run-length slice line scan-conversion algorithms. In this chapter we will apply rounding-up integral linear interpolation to the scan-conversion of yet another important graphics primitive: a filled polygon.

Algorithms for drawing filled polygons have been devised by a number of authors [Gay 85, Gourret 87, Knott 79, Lane 83, Pavlidis 79, Rankin 87, Richards 87]. They can be categorized as either region-filling algorithms or scan-line algorithms. Region-filling algorithms work by first drawing the boundary of the polygon and then setting the pixels inside the polygon. Scan-line algorithms operate by computing spans on scan lines of the raster disply that lie between left and right edges of a polygon using an odd-parity rule, then filling the spans. The span extrema are calculated by an incremental algorithm which takes advantage of edge coherence. The pixels along the polygon edges can be calculated based on incremental linear interpolation with a rounding procedure that does not satisfy the least-error crite-

rion. The reason is that the boundary pixels, i.e., the pixels at both ends of spans should be chosen so that they lie inside the polygon and when two polygons share an edge, no pixels of one polygon will intrude into another polygon. This effectively rounds the fractional intersections to integer grid points by means of rounding-up or rounding-down integral linear interpolation as will be explained in more detail in the next section. The problem of ensuring that the centers of pixels to be set to lie inside the polygon should also be considered when designing polygon filling algorithms using region-filling methods. It is, however, often the case that this is neglected, as we can see in [Gourret 87] where the polygon boundaries are drawn using a line scan-conversion algorithm such as Bresenham's algorithm [Bresenham 65] or symmetrical DDA [Newman 79]. In both cases the problem of polygon overlap is ignored.

The calculation of intersections of scan-lines with polygon edges and the directed rounding of fractional intersections to integral grid coordinates present a significant portion of the computations in scan-line algorithms. It is therefore of interest to explore the possibility of speeding up these computations by means of rounding-up or rounding-down integral linear interpolation. It turns out that this is quite straightforward since the fast computation of integral linear interpolation which was discussed in chapter 2 can be used here. Our discussion will be based on the scan-line algorithm for polygon filling described in [Foley 90]. We will show how this scan-line algorithm can be sped up by incorporating the fast computation of rounding-up integral linear interpolation. In the remainder of this chapter we will first give a brief introduction to the scan-line algorithm for filled polygons in [Foley 90], and then we elaborate on the incorporation of rounding-up integral linear interpolation into the

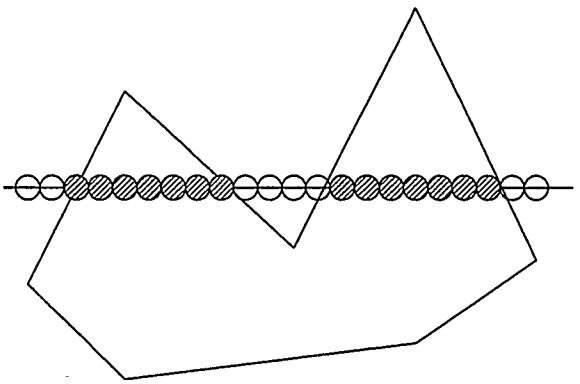


Figure 5.1: Filling the spans inside the polygon for one scan-line.

this algorithm.

5.2 Scan-Line Algorithm for Filled Polygons

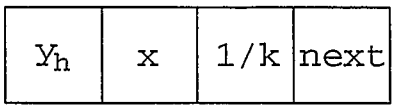
Scan-line methods to scan-convert filled polygons can be found in many text books on computer graphics. We will therefore assume that the reader is familiar with such algorithms and we will only outline the algorithm found in the text book [Foley 90] so that we can refer to it in the development of the improved algorithm.

In the algorithm in [Foley 90] scan-lines are formed horizontally so that non-horizontal edges of the polygon to be scan-converted form span boundaries for the scan-lines. The core of the scan-line algorithm is then to compute the spans of scan-lines that lie inside the polygon and to fill those spans. The following processing steps are performed for each scan line that intersects the polygon to be scan-converted:

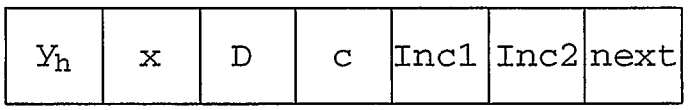
1. Find all intersections of the scan line with the polygon.

2. Sort the intersections by increasing x coordinate.

3. Fill in all pixels between pairs of intersections that lie in the interior of the



(a)



(b)

Figure 5.2: Data structures for active edges when calculating edge-scan-line intersections: (a) commonly used data structure for the DDA method; (b) new data structure for the rounding-up integral linear interpolation method.

polygon using the odd-parity rule.

Figure 5.1 shows the result for one scan-line.

Let us denote the lower endpoint of an edge by (x_l, y_l) and the higher endpoint of the edge by (x_h, y_h) (where low and high is relative to the vertical direction on the raster screen). The intersections of scan lines with polygon edges can be performed in an incremental manner. As we move from one scan line to the next which is immediately above the current scan-line, we can compute the new x intersection of an edge on the basis of the the old x intersection by applying.

$$x_{i+1} = x_i + 1/k,$$

where $k = (y_h - y_l)/(x_h - x_l)$ is the slope of the edge. The increasing order of x intersections of the scan line with polygon edges is maintained by a data structure called the *active edge table*, abbreviated AET. The AET is a linked list with a header

pointing to the first active edge in the list. The x intersection of an edge is initialized to be x_l when the edge is inserted into the list, which happens when the current scan line begins to intersect this edge. The value of x is updated as the scan-line moves upward until it reaches the higher endpoint of the edge. When this happens, the edge is no longer active and it is therefore removed from the AET. The data structure for an active edge in the AET is illustrated in Figure 5.2(a), where *next* is a pointer to the next active edge. The intersections of the scan-line with polygon edges are found by taking the edge items pairwise from the AET. Before we can fill the spans which lie inside the polygon the fractional x intersections must be rounded to integers so that the extreme pixels lie inside the polygon. The rules for choosing extreme pixels that guarantee the above requirement are described in [Foley 90] and they are summarized here.

Referring to Figure 5.3, AB, BC, and DE are left edges, and DC, FE, and AF are right edges. The following rules are suggested for obtaining the left/right extreme pixels: 1) for the left edges of the polygon, each fractional x intersection is rounded to the smallest integer which is greater than x; 2) for the right edges of the polygon each fractional x intersection is rounded to the greatest integer which is less than x; 3) for each integral x intersection on left edges the edge pixel is retained; 4) for each integral x intersection on right edges, the corresponding right extreme is one pixel to the left of the intersection.

If these rules are followed then the result is the filled polygon in Figure 5.3 where the solid circles represent the left or right extrema of spans that lie inside the polygon. The rules guarantee that there is no pixel overlap for two polygons sharing an edge. A similar discussion is made in [Narayanswam 95] about the consistent

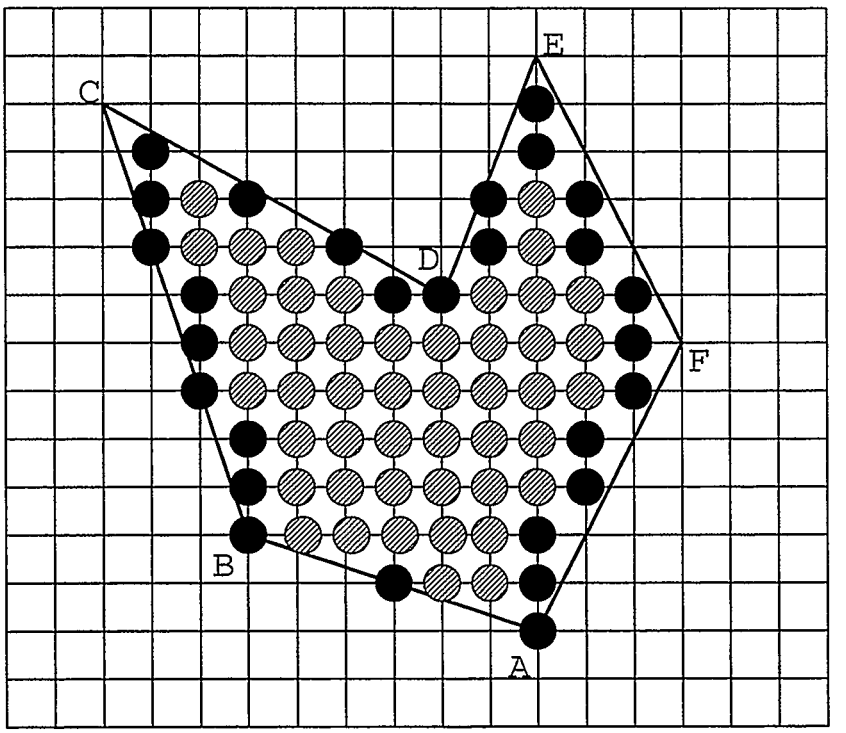


Figure 5.3: A filled polygon where extreme pixels of spans represented by solid circles are chosen following some constrains to ensure that all pixels lie inside the polygon and the the ownership of boundary pixels of two polygons sharing an edge is mutually exclusive.

rule for making ownership of boundary pixels mutually exclusive between adjacent triangles when they are rendered.

The computation of extreme pixels can be performed in a variety of ways. In the most inefficient scheme, the x intersections of an edge with the scan lines are calculated incrementally using floating point arithmetic then rounding the result up or down to integers to yield left or right extreme pixels. Fixed point arithmetic can improve the efficiency in the calculation of the x intersections provided sufficient bits of precision are maintained so that the error accumulation is negligible. An integer algorithm for computing the extreme pixels for a left edge is given in [Foley 90]. The

algorithm is implemented by a procedure named LeftEdgeScan. The x intersection is initialized to be x_l, and the variables *numerator* and *denominator* are assigned values $x_h - x_l$ and $y_h - y_l$ respectively before calculating the extreme pixels. The algorithm uses the variable *increment* to keep track of successive additions of the *numerator* until it "overflows" past the *denominator*; the variable *increment* is then decremented by the *denominator* and x is incremented. This procedure can only handle left edges whose slopes are greater then $+1$. Right edges and other slopes can be handled by similar, but somewhat trickier, arguments.

Let x_i be the x intersection of a left edge with the i-th scan line which intersects the edge. Then

$$x_i = x_l + \frac{x_h - x_l}{y_h - y_l} i, \tag{5.1}$$

and x_i should be rounded to

$$\dot{x}_i = \lceil x_i \rceil, \tag{5.2}$$

which is the abscissa of the left extreme pixel. The use of the ceiling function also correctly obtains the left extreme if x_i is itself integral. For a fractional x intersection of a right edge x'_i, the corresponding right extreme should be

$$\dot{x}'_i = \lfloor x'_i \rfloor.$$

If x'_i is integral then the corresponding right extreme should be

$$\dot{x}'_i = x'_i - 1.$$

This means that the use of floor function to round intersections of scan lines with right edges does not always yield correct abscissas of right extreme pixels. It is obvious that left extreme pixels can be calculated by applying rounding-up integral linear interpolation, and we will show that the computation of right extreme pixels can also be transformed to rounding-up integral linear interpolation. This allows us to develop a unified and efficient approach to the calculation of span extrema for the scan-line algorithm for polygon filling.

5.3 An Integral Linear Interpolation Scan-Line Algorithm

We have seen that the abscissas of the left extreme pixels along a left edge of the polygon are the results of rounding-up integral linear interpolation over the x-extent of that left edge. Therefore, the recurrence formulas for single-step rounding-up integral linear interpolation in chapter 3 can be readily used to calculate the left extreme pixels in the scan-line algorithm for filled polygons. Note that in the recurrence formulas in chapter 3, we assume that $a < b$, and interpolation proceeds from a to b. If we investigate the derivation of those recurrence formulas, we see that the formulas are the same whether $a < b$ or not. This means if $a > b$ the formulas still work as long as c and r are calculated correctly. For example, let $a = 10, b = 0$ and $n = 3$, then $c = \lfloor \frac{-10}{3} \rfloor = -4$, and $r = -10 \ \mathbf{mod}\ 3 = 2$. Assume that the scan-line moves upward one raster unit at a time, then the case of $a > b$ in calculating left extreme pixels happens if a left edge has a negative slope, i.e., $a = x_l > x_h = b$.

Let us now investigate the calculation of right extreme pixels. Let x_i' be the x intersection of an right edge with a scan-line. If x_i' is fractional, then the x coordinate

of the right extreme pixel is

$$\lfloor x_i' \rfloor = \lceil x_i' \rceil - 1.$$

If x_i' is integral the x coordinate of the right extreme pixel is similarly

$$x_i' - 1 = \lceil x_i' \rceil - 1.$$

The right extreme pixels can therefore also be calculated using rounding-up integral linear interpolation. After an integral x value for the scan-line intersection of a right edge in the AET is obtained, it should be decremented by 1. It would seem that we could subtract the x coordinate of the lower endpoint of a right edge by 1 when the edge is inserted into the AET to save the subtraction each time the x value is taken from the AET. But this will cause a right edge to be inserted in the wrong place in the AET. The new data structure for an active edge is therefore illustrated by Figure 5.2(b). Except for *next* which is a pointer to the next active edge, all fields in the new data structure are integral. D is the discriminator whose sign determines if x will be incremented by c or $c+1$ to obtain the next x, and D itself will be incremented by $Inc1$ or $Inc2$ according to its current sign. When a new active edge is inserted into the AET then c, D, $Inc1$, and $Inc2$ are calculated. The following calculations are first performed:

$$c = (x_h - x_l) \textbf{ div } (y_h - y_l),$$

$$r = (x_h - x_l) \textbf{ mod } (y_h - y_l).$$

The values of $Inc1$ and $Inc2$ are r and $r-(y_h-y_l)$ according to Eq. (3.19), and the initial value of D is r according to Eq. (3.20).

The following modifications in the inner loop of the scan-line algorithm are needed to incorporate rounding-up integral linear interpolation into the algorithm:

1. When a new active edge is inserted into the AET, calculate c, D, $Inc1$, and $Inc2$ for this edge.

2. When pairs of x coordinates are taken from the the AET to fill the spans on the scan-line y, the second x coordinate in each pair is decremented by one.

3. When y is incremented by one, update x for each edge in the AET by first testing the sign of D: if $D \leq 0$, x is incremented by c, and D is then incremented by $Inc1$; otherwise, x is incremented by $c+1$, and D is then incremented by $Inc2$.

Note that the lower endpoint of an edge is considered an intersection of the edge with a scan-line in the scan-line algorithm. The rule of always decrementing the integral right x intersection by one to yield a right extreme pixel causes a problem when a right edge share its lower endpoint with a left edge and the corresponding vertex of the polygon is convex. Referring to Figure 5.3, the left edge AB and the right edge AF share the lower endpoint A. For the scan-line passing through A, the left and right x intersections are both integral with the same value. This causes the x coordinate of the right extreme pixel to be 1 less than the x coordinate of the left extreme pixel by rule 4), which may cause the problem of no pixel being painted for that span or one extra pixel to the left of vertex A being painted depending on how

a span is drawn. To avoid this problem we should check if the right extreme is to the left of the left extreme for each span after we get them from the AET. This is not a problem introduced by using the suggested new method of the rounding-up integral linear interpolation however. The original scan-line algorithm would also have to handle this special case. If two right edges share a vertex of the polygon, the pixel centered at that vertex will not be painted because of rule 4), as is the case of vertex F in Figure 5.3.

5.4 Summary

Incorporating rounding-up integral linear interpolation into scan-line algorithms for the scan-conversion of filled polygons is suggested. Two advantages of the new method to calculate the left and right extreme pixels of each span that lie inside the polygon are obvious: 1) It uses integer arithmetic, and except for the integer division operations needed for calculating the discriminator D and incremental amount c for each edge when it is inserted into the AET, all the remaining operations are addition and sign testing. 2) Unlike the method presented in [Foley 90], the new method unifies the treatment of left edges and right edges of the polygon in the incremental computation of left and right extreme pixels, i.e., we use rounding-up integral linear interpolation for both left and right edges. A gain in speed can be expected since the calculation of extreme pixels presents a significant portion of the computations required in scan-line algorithms for filled polygons.

Chapter 6

Run-Length Slice Algorithms for the Scan-Conversion of Ellipses

6.1 Introduction

The ellipse is an important computer graphics primitive. Fast and accurate generation of ellipses on raster displays has therefore received considerable research attention. Several authors have proposed algorithms for the scan-conversion of ellipses [Danielsson 70, Fellner 93, Foley 90, Kappel 85, McIlroy 92, Pitteway 67, Van Aken 84, Van Aken 85, Wu 89]. Most of the algorithms have been designed for ellipses centered at integer coordinates with integer major and minor axes parallel to the coordinate axes since they are the more commonly required ellipses in most graphics systems. Such ellipses are called *canonical ellipses* in this chapter as opposed to ellipses in general position. Most of the published algorithms handle canonical ellipses in a manner similar to that used by Bresenham to scan-convert circles [Bresenham 77], i.e., they are all based on the choice of individual pixels in incremental steps. For example, the midpoint ellipse algorithm proposed in [Van Aken 84] produces one pixel in each incremental step. The double-step ellipse algorithm proposed in [Wu 89] generates two pixels in each incremental step. No other techniques to speed up the scan-conversion of canonical ellipses seem to have been reported. Algorithms to scan-convert ellipses in general position can be found

in [Fellner 93, Foley 90, Pitteway 67].

In this chapter we will investigate the application of the run-length slice methodology to the scan-conversion of canonical ellipses. This is not an essential restriction since the methodology could also be used for general ellipses, however, it would result in a much lengthier and less transparent development. As has been pointed out by McIlroy [McIlroy 92] many published ellipse algorithms have a flaw for certain degenerate ellipses. We will therefore give careful treatment for the degenerate cases. We will first formulate the problem and then delve into the design of run-length slice algorithms for the scan-conversion of canonical ellipses. Complexity analyses are then performed and numerical results are presented.

6.2 Basics of Scan-Converting Canonical Ellipses

We are concerned in this chapter with the scan-conversion of an ellipse in the standard position described by the equation

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1, \tag{6.1}$$

where a, b are positive integers. The ellipse described in this equation is centered at the origin with its major and minor axes coinciding with the coordinate axes. Because of the four-way symmetry of a canonical ellipse, only the scan-conversion of the first quadrant of the ellipse needs to be calculated. A canonical ellipse that is not centered at the origin (but with axes parallel to the coordinate axes) can be drawn by a simple coordinate transformation.

In the sequel we use the term move to denote the step from one pixel to the

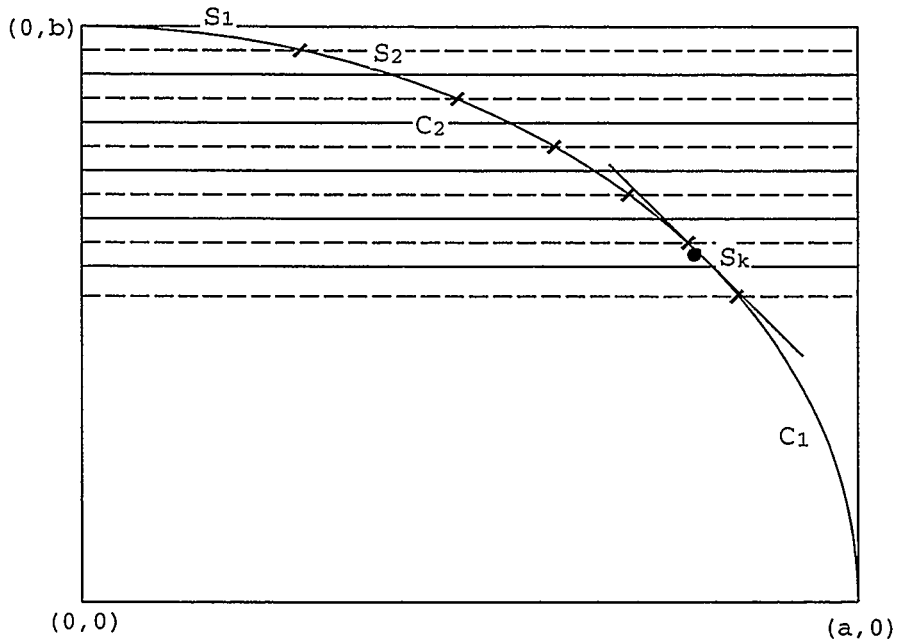


Figure 6.1: The first quadrant of an ellipse is divided into two octants C_1 and C_2. The black dot denotes the juncture of the two octants where the slope of the ellipse is -1. Stippled lines are mid-lines which partition the second octant ellipse arc into segments. The last segment S_k contains the juncture. This segment also forms part of the C_1 octant and it is called the transition segment.

next. We also sometimes use compass directions to denote the direction of a move. In those cases the positive y-axis is north and the positive x-axis is east.

As has been stated in chapter 2, there are generally three different schemes for choosing pixels optimally for the scan-conversion of a 2-D curve. In this thesis we choose the grid-intersection method, which, we will see, is suitable for the design of run-length slice algorithms for the scan-conversion of ellipses.

Most published algorithms for drawing canonical ellipses split the arc in the first quadrant into two octants (Figure. 6.1). The *juncture* of the octants is defined to be the point where the slope of the ellipse is -1. The slope of the ellipse is bounded to

the range $[-\infty, -1]$ in the first octant C_1 and to $[-1, 0]$ in the second octant C_2 .

The choice of approximating grid points can be simplified because of the slope bounds of the octants and the monotonicity of the ellipse in these two octants. The following facts are obvious: If C_1 intersects with a horizontal grid line and then a vertical grid line, the horizontal displacement is always less than the vertical displacement with respect to the intersection of the two grid lines, and vice versa. If C_2 intersects with a vertical grid line and then a horizontal grid line, the vertical displacement is always less than the horizontal displacement with respect to the intersection of the two grid lines, and vice versa. Recall that in chapter 2 we used the abbreviations MHD and MVD to denote minimum-horizontal-displacement and minimum-vertical-displacement respectively. The above observations allow us to only consider MVD points for C_2 and only MHD points for C_1 except that a special treatment has to be performed near the juncture. If, starting from $(0, b)$ and moving along C_2 in a clockwise direction, the last grid line crossed by C_2 is a horizontal grid line, and this crossing causes the selection of an MHD point (Figure 6.2 (a)), then this grid point lies to the northeast of the juncture and is called an *outside point* in [McIlroy 92]. We will call it a *transition point* here. Similarly, starting from $(a, 0)$ and moving along C_1 in a counter-clockwise direction, a transition point may be an MVD point caused by the last intersection of C_1 with a vertical grid line (Figure 6.2 (b)). There may be situations where a transition point is both a MVD point and a MHD point (Figure 6.2 (c)). The transition point does not necessarily exist in a digitized ellipse, and there is only one transition point in the first quadrant if it does exist. The following fact is stated in [McIlroy 92] as a lemma: Any MHD point for the second octant is also an MVD point for that octant, unless the approximating

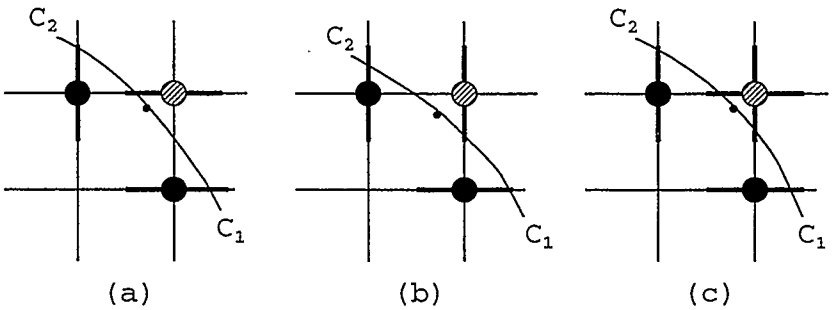


Figure 6.2: Three situations that cause a transition point (denoted by a hatched circle). The slope of the ellipse is -1 at the black dot.

point is a transition (outside) point. A similar statement, with the roles of horizontal and vertical interchanged, holds for the first octant.

6.3 Run-Length Slice Ellipse Algorithm

The run-length slice ellipse algorithm calculates horizontal pixel runs in the second octant and vertical pixel runs in the first octant. If there exists a transition point, it will be caught either as a pixel in a horizontal run, or as a pixel in a vertical run, as will be illustrated in the design of the algorithm. We will first discuss the calculation of horizontal pixel runs and vertical pixel runs, then we discuss the joint of the two octants.

6.3.1 Run-Length Calculation

The run-length calculations for the two octants of the elliptical arc in the first quadrant is based on the discussion on the scan-conversion of a monotonic, x-dominant or y-dominant 2-D curve segment in chapter 2. A set of mid-lines $l_i: \ y = b - i + 0.5$, $i = 1, 2, \cdots, k$ are imposed on the graph of the ellipse midway between the horizontal

grid lines. The i-th mid-line intersects with the second octant ellipse arc at (x_i, y_i) where $y_i = b - i + 0.5$ (see Figure 1.1). Let $(x_0, y_0) = (0, b)$. The mid-lines subdivide the ellipse arc into a series of arc segments, $S_1, S_2, \cdots, S_k$, where segment S_i starts from (x_{i-1}, y_{i-1}) and ends at (x_i, y_i). The segments $S_2, \cdots, S_{k-1}$ are unit segments according to the discussion in chapter 2. Although S_1 is not a unit segment because one of its endpoints is not the intersection of the elliptical arc with a midline, its digitized image is obviously a horizontal run of pixels with ordinate b. The juncture of the first and the second octant falls in the last segment S_k. This segment is called the *transition segment*. Because S_2 through S_{k-1} are unit segments where the ellipse is x-dominant and monotonically decreasing, their digitized images are horizontal pixel runs with abscissa starting from $\lfloor x_{i-1} \rfloor + 1$ to $\lfloor x_i \rfloor$, $i = 2, \cdots, k-1$, according to the discussion in chapter 2. The digitized image of S_1 is also a horizontal run of pixels with abscissa starting from 0 to $\lfloor x_1 \rfloor$. We will delay the discussion of the transition segment until the joint of the two octants is discussed.

Letting $\dot{x}_i = \lfloor x_i \rfloor$ and $\dot{y}_i = b - i + 1$, it follows that the i-th horizontal run starts from $(\dot{x}_{i-1} + 1, \dot{y}_i)$ and ends at $(\dot{x}_i, \dot{y}_i)$. The length of the i-th horizontal run is therefore $\dot{x}_i - \dot{x}_{i-1}$ for $i > 1$. We show that the calculation of $\dot{x}_i$ $(i > 1)$ can be performed in an incremental manner.

From Eq. (1.1) we have

$$b^2 x_i^2 = a^2 b^2 - a^2 (b - i + 0.5)^2, \tag{1.2}$$

and

$$b^2 x_{i+1}^2 = a^2 b^2 - a^2 (b - i - 0.5)^2. \tag{1.3}$$

Subtracting (6.2) from (6.3) and rearranging yields

$$b^2 x_{i+1}^2 = b^2 x_i^2 + 2a^2 b - 2a^2 i. \tag{6.4}$$

From Eq. (6.2) we have $b^2 x_1^2 = a^2 b - \frac{a^2}{4}$. Let $X_i = b^2 x_i^2$, it follows from Eq. (6.4) that

$$\Delta X_i = X_{i+1} - X_i = 2a^2 b - 2a^2 i.$$

We therefore have the recurrence formulas

$$X_1 = a^2 b - \frac{a^2}{4}, \tag{6.5}$$

$$X_{i+1} = X_i + \Delta X_i = X_i + 2a^2 b - 2a^2 i. \tag{6.6}$$

Hence, instead of calculating x_i^2 directly, we calculate X_i in the iterations. To obtain $\dot{x}_i$ without invoking the square root function, forward differencing is employed. Suppose that $\dot{x}_{i-1} = \lfloor \sqrt{X_{i-1}/b_2} \rfloor$ has been evaluated, and the variables x and X hold the value of $\dot{x}_{i-1}$ and $b^2 \dot{x}_{i-1}^2$ respectively. Noting the equation

$$b^2(x+1)^2 - b^2 x^2 = 2b^2 x + b^2,$$

we obtain $\dot{x}_i$ by repeating the iteration

$$X = X + 2b^2 x + b^2$$

$$x = x + 1$$

until $X \geq X_i$. In the case of $X = X_i$, the value of x is $\dot{x}_i$, otherwise we decrease x by one to obtain $\dot{x}_i$. The process of searching for $\dot{x}_i$ can be improved further. Let the value of variables D and x be $X_i - b^2\dot{x}_{i-1}^2$ and $\dot{x}_{i-1}$ respectively before entering the repetitive loop, then after performing the following iterations, the value of x will reach $\dot{x}_i$. We use the pseudo code to describe the algorithm segment and obtain:

while $(D >= 2b^2x + b^2)$ {

$D = D - 2b^2x - b^2;$

$x = x + 1;$

}

Setting $Delta1 = 2b^2\dot{x}_{i-1} + b^2$ before entering the loop, we can further speed up the above process by using the following code segment:

while $(D >= Delta1)$ {

$D = D - Delta1;$

$Delta1 = Delta1 + 2b^2;$

$x = x + 1;$

}

Noting that $\dot{x}_0 = 0$, we have the following initial values:

$$Delta1 = b^2,$$

$$D = X_1 = a^2b - \frac{a^2}{4},$$

$$DeltaX = \Delta X_1 = 2a^2b - 2a^2.$$

Since we always test if D is greater than or equal to a positive integer, it can be truncated without affecting the result. We therefore assign the initial value of D to

$$D = a^2b - a^2/4,$$

where the operator / represents integer division which operates on two integers and gives the integer part of the quotient as result. This is, for example, an operation used in the C programming language for integers. We name the process for calculating all the horizontal runs the *H-Pass*. The condition for terminating the H-Pass process will be discussed in the sequel. The pseudo code below implements the H-Pass leaving the termination condition unspecified.

ALGORITHM 6.1: H-Pass

```
y = b;
x = x0 = 0;
D = a * a * b - a * a/4;
Delta1 = b * b;
DeltaX = 2 * a * a * b - 2 * a * a;
repeat {
        while (D >= Delta1) {
            D = D - Delta1;
            Delta1 = Delta1 + 2 * b * b;
            x = x + 1;
```

}

$HorizontalRun(x0, x, y)$;

$y = y - 1$;

$x0 = x + 1$;

$D = D + DeltaX$;

$DeltaX = DeltaX - 2 * a * a$;

} **until** (terminating condition is true);

Symmetrically, we use a set of vertical mid-lines l'_i : $x = a - i + 0.5, i = 1, 2, \cdots, k'$ to intersect the first quadrant of the ellipse to yield segments $S'_1, S'_2, \cdots, S'_{k'}$ such that the juncture lies in segment $S'_{k'}$. The calculation of the vertical pixel runs of C_1 can be done in a similar manner: exchange the role of a and b, as well as x and y in Algorithm H-Pass. We call this process the *V-Pass*. The pseudo code is again given without a termination condition.

ALGORITHM 6.2: V-Pass

$x = a$;

$y = y0 = 0$;

$D = b * b * a - b * b/4$;

$Delta1 = b * b$;

$DeltaY = 2 * b * b * a - 2 * b * b$;

repeat {

while ($D >= Delta1$) {

$D = D - Delta1$;

$Delta1 = Delta1 + 2 * a * a;$

$y = y + 1;$

}

$VerticalRun(y0, y, x);$

$x = x - 1;$

$y0 = y + 1;$

$D = D + DeltaY;$

$DeltaY = DeltaY - 2 * b * b;$

} **until** (terminating condition is true);

The next step is to determine the termination conditions for both the H-Pass and the V-Pass and to make sure that the two octants are connected correctly. If we begin with the H-Pass, then its termination condition determines when we switch to the V-Pass. The termination condition of the V-Pass determines when the algorithm ends. To make the correct connection between the two parts of the algorithm, we first discuss the possible pixel configurations at the transition of two octants.

6.3.2 Pixel Configurations at Octant Transition

The first quadrant is traversed in a counter-clockwise direction starting from point $(0, b)$ of the second octant C_2. Let P be the last MVD point of C_2, T be the transition point, and Q be the first MHD point of C_1 immediately following T. Using subscripts

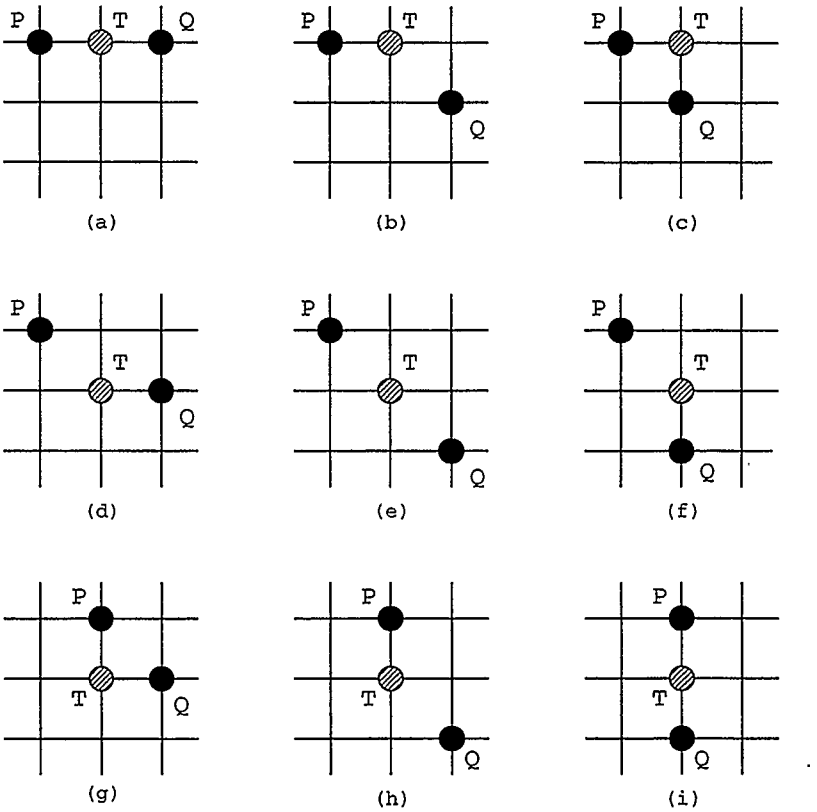


Figure 6.3: Configurations for P, T, and Q. (a), (d), (g), (h), and (i) are impossible.

x and y to denote the x and y coordinates of a point respectively, we have

$$P_x \leq T_x \leq Q_x,$$

$$P_y \geq T_y \geq Q_y.$$

According to these inequalities, there are nine possible configurations for P, T, and Q as illustrated in Figure 6.3 (a) through (i). Not all the configurations in Figure 6.3 are possible for digitized ellipses. We first prove the following lemmas.

Lemma 2 *Q and T cannot lie on the same horizontal grid line.*

Proof: We prove this by contradiction. Assume that Q and T lie on the same horizontal grid line, then

1. T cannot be a MHD point, because the first quadrant ellipse can only intersect a horizontal grid line once and therefore there is only one MHD point on each horizontal grid line.

2. T cannot be a MVD point either. If it is, then it is the map of the intersection point of C_1 with a vertical grid line, and thus the intersection point lies no farther than 0.5 above grid point T (Figure 6.4). This implies from the mean value theorem that there exists a point on C_1 such that the slope of the ellipse at this point is greater than -1.

Since T is a transition point, it must be either a MVD point, or a MHD point, or both. The resulting contradiction eliminates the possibility of Q and T lying on the same horizontal grid line. □

Lemma 3 *P and T cannot lie on the same vertical grid line.*

Proof: If P and T lie on the same vertical grid line, then by rotating the ellipse 90 degrees around the origin we get an ellipse such that Q and T lie on the same horizontal grid line, a contradiction to lemma 2. □

Lemma 2 and 3 lead directly to the following theorem:

Theorem 1 *The transition pixel configurations (a), (d), (g), (h), and (i) are impossible.*

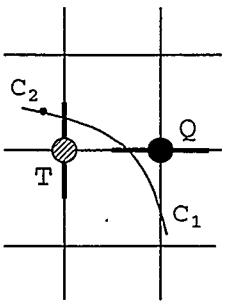


Figure 6.4: Impossible configurations for Q and T. The small dot on the curve stands for the juncture of C_1 and C_2.

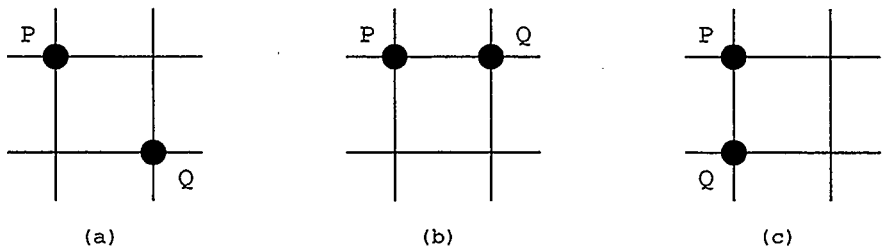


Figure 6.5: Configurations for P and Q without the transition point T.

The four transition configurations (b), (c), (e), and (f) are exemplified in the inventory of configurations at the juncture given by McIlroy [McIlroy 92]. Configuration (b) corresponds to configuration 5 in the inventory; configuration (c) corresponds to configuration 7, 6, and 10; configuration (e) corresponds to configuration 1; and configuration (f) corresponds to configuration 2.

If there is no transition point T, the point P of C_2 and the point Q of C_1 can have three configurations as illustrated in Figure 6.5.

6.3.3 Condition of Octant Change

The slope of an ellipse at a point (x, y) in the first quadrant is given by

$$\frac{dy}{dx} = -\frac{b^2 x}{a^2 y}.$$

In the first octant the slope is less than -1, and therefore $b^2x > a^2y$; in the second octant the slope is greater than -1, and therefore $b^2x < a^2y$. The slope can be approximated by calculating b^2x and a^2y at the grid points approximating the ellipse as was suggested by Kappel [Kappel 85]. Suppose we first do the H-Pass. If at a tentative beginning of a new horizontal run the condition $b^2x < a^2y$ is false, then we assume that the H-Pass is completed. A problem will occur if $b^2x > a^2y$ at a MVD point. This may happen when the y coordinate of the intersection point of C_2 with a vertical grid line is rounded down to the nearest integer, i.e., there may be an integral $x > 0$ and a real $y > 0$ satisfying the following conditions simultaneously:

$$\begin{aligned} b^2x^2 + a^2y^2 &= a^2b^2, \\ b^2x &< a^2y, \\ b^2x &> a^2\lfloor y + 0.5 \rfloor. \end{aligned}$$

If this happens then the H-Pass will be terminated prematurely. A similar problem may occur in the V-Pass such that the V-Pass, and hence the whole algorithm, is terminated prematurely. Let $\mathcal{L}$ be the line from the origin to the juncture of C_1 and C_2. If the first pixel in a horizontal run, which we call the *horizontal leading pixel*, lies below $\mathcal{L}$, then the whole run will be lost, as will the subsequent horizontal runs

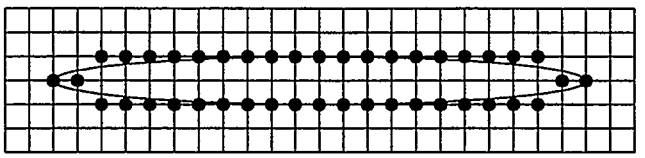


Figure 6.6: "Tails" in a digitized ellipse.

if there are any. One situation that evidently causes the loss of a horizontal run is that the y coordinate of a leading pixel is zero. In this scenario, all pixels in this horizontal run lie below line $\mathcal{L}$. These pixels, plus one pixel at $(b, 0)$, forms a "tail" in the digitized first quadrant ellipse (Figure 6.6). McIlroy has proved that the tail happens if and only if $a \geq 8b^2$ [McIlroy 92]. Symmetrically, we term the first pixel in a vertical run (the pixel with the least ordinate in the vertical run) in the first octant the *vertical leading pixel.* If a vertical leading pixel lies above $\mathcal{L}$, the whole run will be lost, and so do the subsequent vertical runs if there are any. One can easily see that a tail appears on the y-axis if and only if $b \geq 8a^2$ by symmetry.

It is obvious that if there is neither a horizontal leading pixel lying below $\mathcal{L}$, nor a vertical leading pixel lying above $\mathcal{L}$, then the H-Pass will capture all the horizontal runs of C_2 and the V-Pass will capture all the vertical runs of C_1. We will prove that if there is a transition point, then it will be captured by either the H-Pass or the V-Pass, or by both. In the following theorem, P, T, and Q refer to the pixels in Figure 6.3 $(b), (c), (e)$ and (f).

Theorem 2 *If pixels P and Q are captured by the H-Pass and V-Pass respectively, then the transition point T will also be captured by either the H-Pass, or the V-Pass, or both.*

Proof: We prove that for each of the four possible transition configurations with

transition point T, the transition point will be captured.

1. In Figure 6.3 (b), T must be a MVD point of C_1, otherwise there will be a point on C_1 such that the slope of the ellipse at that point is greater than -1. Therefore T will be captured by the H-Pass as the last pixel in the digitized image of S_k.

2. In Figure 6.3 (c), T must be either a MVD point of C_1, or a MHD point of C_2, or both. If T is a MVD point of C_1, then it will be captured by the H-Pass as the last pixel in the digitized image of S_k. If T is a MHD point of C_2, then it can be captured by the V-Pass as the last pixel in the digitized image of $S'_{k'}$.

3. In Figure 6.3 (e), T must be both a MVD point of C_1 and a MHD point of C_2. If T is a MVD point of C_1, then C_1 must cross the grid line $x = T_x$ below T, otherwise C_1 crosses the grid line $y = T_y$ before grid line $x = T_x$, which makes T a MHD point of C_1, implying that T is not a transition point. Therefore C_2 must cross the grid line $y = T_y$ to the left of T and no farther than 0.5 from T, otherwise there exists a point on C_2 such that the slope of the ellipse at that point is less than -1. The above argument establishes that T is also a MHD point of C_2. Similarly we can prove that if T is a MHD point of C_2, it is also a MVD point of C_1. Therefore T must be both a MVD point of C_1 and a MHD point of C_2. Since T lies in a different row from P and in a different column from Q, it will be captured by either the H-Pass or the V-pass depending on which side of $\mathcal{L}$ it lies.

4. In Figure 6.3 (f), T must be a MHD point of C_2, otherwise there exists a point

on C_2 such that the slope of the ellipse at that point is less than -1. Therefore T will be captured by the V-Pass as the last pixel in the digitized image of $S'_{k'}$.

If there is no transition point, then P will be captured by the H-Pass and Q will be captured by the V-Pass. Referring to Figure 6.5, Q may also be captured by the H-Pass in Figure 6.5 (b) if C_1 crosses the vertical grid line $x = Q_x$ less than 0.5 away from Q; and P may also be captured by the V-Pass in Figure 6.5(c) if C_2 crosses the horizontal grid line $y = P_y$ less than 0.5 away from P. We will present the method for avoiding setting pixels P or Q twice in the sequel. □

6.3.4 Run-Length Slice Algorithm

The run-length slice ellipse algorithm combines the H-Pass and the V-Pass. Suppose that the H-Pass is performed first. The H-Pass keeps track of the values of b^2x and a^2y, and compares them at each tentative horizontal leading pixel to determine if the H-Pass should terminate. The values of b^2x and a^2y can be calculated in an incremental manner. At the leading pixel of the first horizontal run, we have $b^2x = 0$, and $a^2y = a^2b$. To obtain the values of b^2x and a^2y at the subsequent horizontal leading pixels in the H-Pass, the value of b^2x is initialized to be b^2 and incremented by b^2 with each horizontal move to the next pixel so that when a horizontal run ends which causes y to be decremented by 1 and a^2y to be decremented by a^2, the new x, y values are the coordinates of the next horizontal leading pixel, and the values of b^2x and a^2y can be used to test if the H-Pass should continue. Let x' be the abscissa of the last pixel in the last horizontal run. Then we can calculate vertical runs from b to $x' + 1$. The termination condition for the V-Pass is therefore that the abscissa of a vertical run reaches $x' + 1$. This saves time in the V-Pass for keeping track

of the values of b^2x and a^2y, and it also avoids a pixel being drawn twice. One or more pixels will be missing if there is a pixel or pixels immediately below the last pixel of the last horizontal run. This may for example happen in configuration (c) in Figure 6.3, where pixel T is captured by the H-Pass as the last horizontal pixel in the last horizontal run. Pixel Q, immediately below T, will be missing. Another scenario that causes missing pixels is the existence of a vertical "tail". In this case, if the tail is n pixels long, then $n-1$ pixels will be missing. To avoid missing pixels, we can check the last pixel of the last horizontal run and the last pixel of the last vertical run. If the difference of the ordinates of the two pixels is greater than one, we insert one or more pixels to fill the gap.

As has been pointed out, the H-Pass will terminate prematurely if a horizontal leading pixel lies below $\mathcal{L}$, i.e., if $b^2x \geq a^2y$ at that point. We know that one condition for this to happen is that tails exist, i.e., $a > 8b^2$. If this is the case, one horizontal leading pixel will lie on the x-axis, which can be easily detected, and the algorithm can then switch to draw the tails. It is not known if there are other conditions under which the inequality $b^2x > a^2y$ holds at a horizontal leading pixel. But according to our extensive test with a and b varying from 1 to 1024, no such situation happens except when $a \geq 8b^2$.

Caution must also be given to test if a vertical "tail" exists. In that case x' must be 0.

The following is a complete run-length slice ellipse algorithm combining the H-Pass and the V-pass and giving the termination conditions for both the H-Pass and the V-Pass. The H-Pass is performed first. The operators << and >> used in the algorithm represents binary left and right shift respectively to implement the

multiplication and the division by a power of two for the purpose of efficiency.

ALGORITHM 6.3: Run-length slice ellipse algorithm 1 (RLS-Ellipse1)

$y = b;$

$x = x0 = 0;$

$aa = a * a; \quad bb = b * b; \quad aab = aa * b;$

$aa2 = (aa << 1); \quad bb2 = (bb << 1);$

$c1 = bb; \quad c2 = aab;$

$D = aab - aa >> 2;$

$Delta1 = bb;$

$DeltaX = (aab - aa) << 1;$

repeat {

while $(D >= Delta1)$ {

$D = D - Delta1;$

$Delta1 = Delta1 + bb2;$

$x = x + 1;$

$c1 = c1 + bb;$

}

$HorizontalRun(x0, x, y);$

$y = y - 1;$

$c2 = c2 - aa;$

$x0 = x + 1;$

$D = D + DeltaX;$

$DeltaX = DeltaX - aa2;$

```
} until (c1 >= c2);

if (y == 0) {
        HorizontalRun(x0, a, 0);
        return;
}

y1 = y; /* Saves the ordinate of the last pixel in the last horizontal run */
x = a;
y = y0 = 0;
D = bba - bb >> 2;
Delta1 = aa;
DeltaY = (bba - bb) << 1;
while (x >= x0) {
        while (D >= Delta1) {
                D = D - Delta1;
                Delta1 = Delta1 + aa2;
                y = y + 1;
        }
        VerticalRun(y0, y, x);
        x = x - 1;
        y0 = y + 1;
        D = D + DeltaY;
```

$DeltaY = DeltaY - bb2;$

}

if $(y0 < y1)$ $VerticalRun(y0, y_1 - 1, x);$

if $(x == 0)$ $VerticalRun(y0, b - 1, 0);$

6.4 Further Improvement

The run-length slice ellipse algorithm given in section 3 can be improved by speeding up the calculation of pixel runs. Consider an ellipse represented by Eq. (6.1). If $a > b$, then it has longer horizontal pixel runs than vertical pixel runs. This becomes more apparent as the ratio of a to b increases. In the extreme case, "tails" appear on the x-axis as in Figure 6.6, and there is only one vertical run of length one obtained in the V-Pass. Since we use forward differencing to calculate the run-lengths, it follows that longer incremental step can be used for those long runs. To avoid complicating the algorithm too much, a simple scheme can be adopted to calculate horizontal runs in the H-Pass: use double-step forward differencing for the the part where the run lengths are at least two and then switch to single-step forward differencing for the part where the run lengths are at most two. Let the point $P(x, y)$ on C_2 satisfy

$$2b^2x = a^2y.$$

Point P partitions C_2 into two segments C_{21} and C_{22}. Each point (x, y) on C_{21} satisfies

$$2b^2x < a^2y,$$

and each point (x, y) on C_{22} satisfies

$$2b^2x > a^2y.$$

The following theorems give the hint of how to switch from double-step forward differencing to single-step forward differencing in the H-Pass.

Theorem 3 *If the segment S_i lies entirely in C_{21} then the digitized image of S_i is a horizontal run of pixels of length at least two.*

Proof: The explicit form of the equation for the first quadrant ellipse is

$$y = f(x) = \frac{b}{a}\sqrt{a^2 - x^2}.$$

The segment $S_i, i = 1, \cdots, k-1$ begins at (x_{i-1}, y_{i-1}) and ends at (x_i, y_i). For $i > 1$ the length of the i-th run is $\gamma_i = \lfloor x_i \rfloor - \lfloor x_{i-1} \rfloor$. The length of the first run is $\gamma_1 = \lfloor x_1 \rfloor + 1 > \lfloor x_1 \rfloor - \lfloor x_0 \rfloor$ since $x_0 = 0$. If segment S_i lies entirely in C_{21} then according to the mean value theorem, there exists $\xi, x_{i-1} < \xi < x_i$ such that

$$y_i - y_{i-1} = f'(\xi)(x_i - x_{i-1}),$$

where $|f'(\xi)| < 1/2$. Noting that $|y_i - y_{i-1}| = 1$, we have

$$x_i - x_{i-1} = |x_i - x_{i-1}| = |y_i - y_{i-1}|/|f'(\xi)| > 2|y_i - y_{i-1}| = 2.$$

It follows from the inequalities

$$\lfloor x_{i-1} \rfloor \leq x_{i-1},$$

$$x_i - 1 < \lfloor x_i \rfloor$$

that

$$\lfloor x_i \rfloor - \lfloor x_{i-1} \rfloor > x_i - 1 - x_{i-1} > 1.$$

i.e.,

$$\lfloor x_i \rfloor - \lfloor x_{i-1} \rfloor \geq 2.$$

Therefore $\gamma_i \geq 2$ for $i = 1, \cdots, k-1$. □

Theorem 4 *If the segment S_i lies entirely in C_{22} then it follows that the digitized image of S_i is a horizontal run of pixels of length at most two.*

Proof: The proof is similar to the proof of Theorem 3. In this case we have

$$x_i - x_{i-1} = |x_i - x_{i-1}| = |y_i - y_{i-1}|/|f'(\xi)| < 2|y_i - y_{i-1}| = 2,$$

where $x_{i-1} < \xi < x_i$, and $|f'(\xi)| > 1/2$. It follows from the inequalities

$$x_{i-1} - 1 < \lfloor x_{i-1} \rfloor,$$

$$\lfloor x_i \rfloor \leq x_i$$

that

$$\lfloor x_i \rfloor - \lfloor x_{i-1} \rfloor < x_i - x_{i-1} + 1 < 3,$$

i.e.,

$$\gamma_i = \lfloor x_i \rfloor - \lfloor x_{i-1} \rfloor \leq 2. \qquad \square$$

Recall that in ALGORITHM 6.3 H-Pass keeps track of the values of b^2x and a^2y at each pixel, and compares them at each tentative horizontal leading pixel to determine if the H-Pass should terminate. We can divide the H-Pass into two phases. In the first phase, we calculate the run lengths using double-step forward differencing. We check if $2b^2x > a^2y$ at each horizontal leading pixel. If the run length is odd, we must advance one single step after we have made enough double-step advances. If the condition $2b^2x > a^2y$ is true at a horizontal leading pixel, we switch to the second phase to calculate the run lengths using single-step forward differencing while checking if $b^2x < a^2y$. If this condition is true we terminate the H-Pass and switch to the V-Pass. In ALGORITHM 6.3 D is decremented by $Delta1$ in the inner loop if it is still greater than or equal to $Delta1$, and $Delta1$ is then incremented by $2b^2$. For double-step forward differencing, we use the variable $Delta2$. D is decremented by $Delta2$ in the inner loop if it is still greater than or equal to $Delta2$, and $Delta2$ is then incremented by $8b^2$. The initial value of $Delta2$ is $4b^2$. When we switch to the second phase, D is decremented by $Delta1$ in the inner loop as it is in ALGORITHM 6.3. The value of $Delta1$ before entering the second phase can be easily obtained by investigating the value of variable $c1$ which holds the value of b^2x for each leading pixel generated so far. On entering the second phase, the value of $c1$ is $b^2\dot{x}_i$ where $\dot{x}_i$ is the x-coordinate of the leading pixel of the next horizontal run. Note that the

value of $Delta1$ on entering the second phase is $2b^2\dot{x}_{i-1} + b^2$ where $\dot{x}_{i-1} = \dot{x}_i - 1$. The value of $Delta1$ on entering the second phase is therefore $2c1 - b^2$. The value of $Delta1$ is also required in the first phase after enough double-step advances have been made when we want to determine if an extra single-step advance is necessary to form a run of odd length. Below is the adaptive forward differencing version of the H-Pass.

ALGORITHM 6.4: H-Pass2

$y = b;$

$x = x0 = 0;$

$c1 = b * b; \quad c2 = a * a * b;$

$D = a * a * b - a * a/4;$

$Delta2 = 4 * b * b;$

$DeltaX = 2 * a * a * b - 2 * a * a;$

repeat {

while $(D >= Delta2)$ {

$D = D - Delta2;$

$Delta2 = Delta2 + 8 * b * b;$

$x = x + 2;$

$c1 = c1 + 2 * b * b;$

}

$Delta1 = 2 * c1 - b * b;$

if $(D >= Delta1)$ {

$D = D - Delta1;$

```
            Delta2 = Delta2 + 4 * b * b;
            x = x + 1;
            c1 = c1 + b * b;
        }
        HorizontalRun(x0, x, y);
        y = y - 1;
        x0 = x + 1;
        c2 = c2 - a * a;
        D = D + DeltaX;
        DeltaX = DeltaX - 2 * a * a;
} until (2 * c1 >= c2);

Delta1 = 2 * c1 - b * b;
repeat {
        while (D >= Delta1) {
            D = D - Delta1;
            Delta1 = Delta1 + 2 * b * b;
            x = x + 1;
            c1 = c1 + b * b;
        }
        HorizontalRun(x0, x, y);
        y = y - 1;
        x0 = x + 1;
```

$c2 = c2 - a * a;$

$D = D + DeltaX;$

$DeltaX = DeltaX - 2 * a * a;$

} **until** $(c1 >= c2)$;

Because of the symmetry of the H-Pass and the V-Pass, the derivation of the adaptive forward differencing version of the V-Pass is quite straightforward. This results a new run-length slice ellipse scan-conversion algorithm. The new algorithm is the the result of combining the the H-Pass2 and the V-Pass2 with minor modifications in the V-Pass regarding the termination condition.

ALGORITHM 6.5: Run-length slice ellipse algorithm 2 (RLS-Ellipse2)

$y = b;$

$x = x0 = 0;$

$aa = a * a; \quad bb = b * b; \quad aab = aa * b; \quad bba = bb * a$

$aa2 = aa << 1; \quad aa4 = aa2 << 1; \quad aa8 = aa4 << 1;$

$bb2 = bb << 1; \quad bb4 = bb2 << 1; \quad bb8 = bb4 << 1;$

$c1 = bb; \quad c2 = aab;$

$D = aab - aa4;$

$Delta2 = bb4;$

$DeltaX = (aab - aa) << 1;$

L_1: **repeat** {

while $(D >= Delta2)$ {

```
L2:             D = D - Delta2;
                Delta2 = Delta2 + bb8;
                x = x + 2;
                c1 = c1 + bb2;
            }
            Delta1 = (c1 << 1) - bb;
            if (D >= Delta1) {
L3:             D = D - Delta1;
                Delta2 = Delta2 + bb4;
                x = x + 1;
                c1 = c1 + bb;
            }
            HorizontalRun(x0, x, y);
            y = y - 1;
            x0 = x + 1;
            c2 = c2 - aa;
            D = D + DeltaX;
            DeltaX = DeltaX - aa2;
    } until ((c1 << 1) < c2);

    Delta1 = (c1 << 1) - bb;
L4: repeat {
            while (D >= Delta1) {
```

```
L5:             D = D - Delta1;
                Delta1 = Delta1 + bb2;
                x = x + 1;
                c1 = c1 + bb;
            }
            HorizontalRun(x0, x, y);
            y = y - 1;
            x0 = x + 1;
            c2 = c2 - aa;
            D = D + DeltaX;
            DeltaX = DeltaX - aa2;
    } until (c1 >= c2);

    if (y == 0) {
            HorizontalRun(x0, a, 0);
            return;
    }
    y1 = y;
    y = y0 = 0;    x = a;
    c1 = bba;    c2 = aa;
    D = bba - bb4;
    Delta2 = aa4;
    DeltaY = (bba - bb) << 1;
```

```
L6: repeat {
L7:         while (D >= Delta2) {
                D = D - Delta2;
                Delta2 = Delta2 + aa8;
                y = y + 2;
                c2 = c2 + aa2;
            }
            Delta1 = (c2 << 1) - aa;
            if (D >= Delta1) {
L8:             D = D - delta1;
                Delta2 = Delta2 + aa4;
                y = y + 1;
                c2 = c2 + aa;
            }
            VerticalRun(y0, y, x);
            x = x - 1;
            y0 = y + 1;
            c1 = c1 - bb;
            D = D + DeltaY;
            DeltaY = DeltaY - bb2;
    } until ((c2 << 1) >= c1);
    Delta1 = (c2 << 1) - aa;
```

```
L9: while (x >= x0) {
L10:            while (D >= Delta1) {
                        D = D - Delta1;
                        Delta1 = Delta1 + aa2;
                        y = y + 1;
                }
                VerticalRun(y0, y, x);
                x = x - 1;
                y0 = y + 1;
                D = D + DeltaY;
                DeltaY = DeltaY - bb2;
    }
    if (y1 > y0)  VerticalRun(y0, y1 - 1, x);
    if (x == 0)  VerticalRun(y0, b - 1, 0);
```

6.5 Complexity Analysis and Numerical Results

In this section we perform complexity analyses for ALGORITHM 6.5 (RLS-Ellipse2) and the well-known midpoint ellipse algorithm of Van Aken [Van Aken 84] because of its simplicity. Another comparison choice could have been the algorithm by McIlroy [McIlroy 92] which works in all cases, but is much more complicated. The result of the theoretical analyses will then be tested by comparing the actual running times

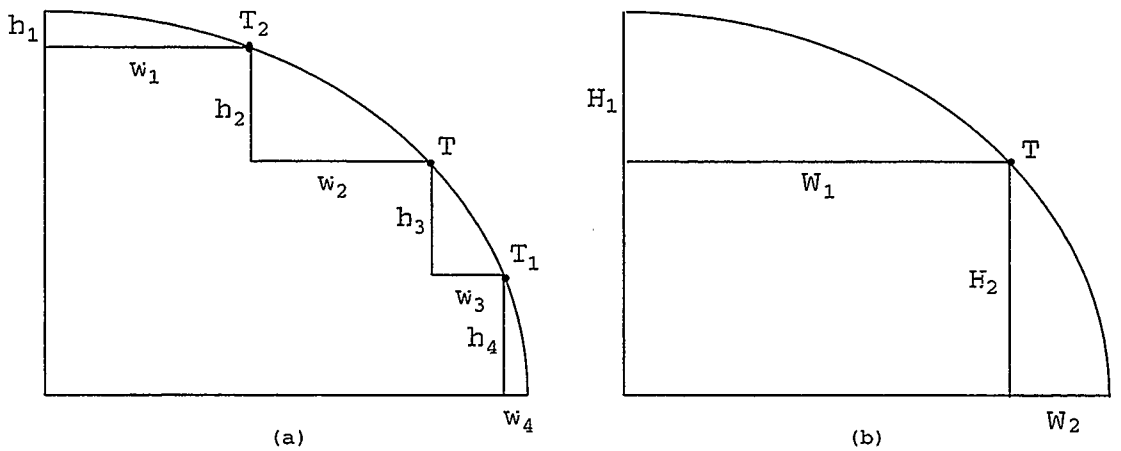


Figure 6.7: Counting arithmetic operations for algorithm RLS-Ellipse2 and the midpoint ellipse algorithm.

of the algorithms.

Referring to ALGORITHM 6.5, we count only those operations in the iteration loops which form the main body of the algorithm, i.e., the **repeat** loop labeled L_1, L_4 which draws the second octant with horizontal runs, the **repeat** loop labeled L_6 and the **while** loop labeled L_9 which draw the first octant with vertical runs. Referring to Figure 6.7(a), T is the juncture of the first and the second octant, T_1 is on the first octant elliptical arc whose coordinates satisfy $b^2x = 2a^2y$, and T_2 is on the second octant elliptical arc whose coordinates satisfy $2b^2x = a^2y$. A trivial geometric computation gives the coordinates of T, T_1 and T_2 as follows:

$$T: \quad x = \frac{a^2}{\sqrt{a^2+b^2}}, \quad y = \frac{b^2}{\sqrt{a^2+b^2}},$$

$$T_1: \quad x = \frac{2a^2}{\sqrt{4a^2+b^2}}, \quad y = \frac{b^2}{\sqrt{4a^2+b^2}},$$

$$T_2: \quad x = \frac{a^2}{\sqrt{a^2+4b^2}}, \quad y = \frac{2b^2}{\sqrt{a^2+4b^2}}.$$

We thus have in Figure 6.7(a):

$$\begin{aligned}
h_1 &= b - \frac{2b^2}{\sqrt{a^2+4b^2}}, \\
w_1 &= \frac{a^2}{\sqrt{a^2+4b^2}}, \\
h_2 &= \frac{2b^2}{\sqrt{a^2+4b^2}} - \frac{b^2}{\sqrt{a^2+b^2}}, \\
w_2 &= \frac{a^2}{\sqrt{a^2+b^2}} - \frac{a^2}{\sqrt{a^2+4b^2}}, \\
h_3 &= \frac{b^2}{\sqrt{a^2+b^2}} - \frac{b^2}{\sqrt{4a^2+b^2}}, \\
w_3 &= \frac{2a^2}{\sqrt{4a^2+b^2}} - \frac{a^2}{\sqrt{a^2+b^2}}, \\
h_4 &= \frac{b^2}{\sqrt{4a^2+b^2}}, \\
w_4 &= a - \frac{2a^2}{\sqrt{4a^2+b^2}}.
\end{aligned}$$

The number of arithmetic operations in loops labeled L_1, L_4, L_6 and L_9 respectively can be approximated in terms of h_i and w_i, $i = 1, 2, 3, 4$. Letting N_{ai}, N_{si} and N_{ci}, $i = 1, 4, 6, 9$, denote the number of additions, shifts and comparisons in L_i loop, $i = 1, 4, 6, 9$, respectively, we can derive formulas to approximate them as follows:

1. The **repeat** loop labeled L_1 is executed about h_1 times. It contains six additions, two shifts and two comparisons excluding operations in the conditionally executed bodies of statements labeled L_2 and L_3. The number of times that the body of the nested **while** loop labeled L_2 is executed varies each time this

statement is executed, however, after the outmost **repeat** loop (L_1 loop) is finished, the total number of times this **while** loop (L_2 loop) is executed is approximately $(w_1 - h_1/2)/2$, assuming that the numbers of runs of even length and odd length are on the average equal. There are four additions and one comparison in this loop. The compound statement in the **if** statement labeled L_3 in the body of L_1 loop involves four additions. These operations are executed when the length of a run is odd. Again, these operations are executed $h_1/2$ times assuming that on the average half of the runs have odd lengths. We therefore have

$$\begin{aligned} N_{a1} &= 6h_1 + 4(\frac{1}{2}w_1 - \frac{1}{4}h_1) + 4 \cdot \frac{1}{2}h_1 = 7h_1 + 2w_1, \\ N_{s1} &= 2h_1, \\ N_{c1} &= 2h_1 + \frac{1}{2}w_1 - \frac{1}{4}h_1 = \frac{7}{4}h_1 + \frac{1}{2}w_1 \end{aligned}$$

2. The **repeat** loop labeled L_4 is executed about h_2 times. It contains five additions and one comparison excluding operations in the conditionally executed bodies of statements labeled L_5. The number of times that the body of the nested **while** loop labeled L_5 is executed varies each time this statement is executed, however, after the outmost **repeat** loop (L_4 loop) is finished, the total number of times this **while** loop (L_5 loop) is executed is approximately w_2. Considering that there are four additions and one comparison in this **while** loop, the total numbers of arithmetic operations performed when the L_4 loop

is executed are therefore

$$\begin{aligned} N_{a4} &= 4w_2 + 5h_2, \\ N_{s4} &= 0, \\ N_{c4} &= w_2 + h_2. \end{aligned}$$

3. In a similar manner we have the following formulas giving the numbers arithmetic operations when the loops labeled L_6 and L_9 are executed:

$$\begin{aligned} N_{a6} &= 2h_4 + 7_{w4}, \\ N_{s6} &= 2w_4, \\ N_{c6} &= \frac{1}{2}h_4 + \frac{7}{4}w_4, \\ N_{a9} &= 3h_3 + 4w_3, \\ N_{s9} &= 0, \\ N_{c9} &= h_3 + w_3. \end{aligned}$$

The total numbers of arithmetic operations when the above four loops are executed are therefore

$$\begin{aligned} N_a &= 7h_1 + 5h_2 + 3h_3 + 2h_4 + 2w_1 + 4w_2 + 4w_3 + 7w_4, \\ N_s &= 2h_1 + 2w_4, \end{aligned}$$

$$N_c = \frac{7}{4}h_1 + h_2 + h_3 + \frac{1}{2}h_4 + \frac{1}{2}w_1 + w_2 + w_3 + \frac{7}{4}w_4.$$

To compare ALGORITHM 6.5 with the well-known midpoint ellipse algorithm by Van Aken, we list the midpoint algorithm below. Since it was originally presented in [Van Aken 85] using PASCAL-like pseudo code, we rewrote it using C-like pseudo code.

ALGORITHM 6.6: Midpoint ellipse algorithm by Van Aken

```
    x = a;   y = 0;
    t1 = a * a;   t2 = t1 << 1;   t3 = t2 << 1;
    t4 = b * b;   t5 = t4 << 1;   t6 = t5 << 1;
    t7 = a * t5;   t8 = t7 << 1;   t9 = 0;
    d1 = t2 - t7 + t4;
    d2 = t1 - t8 + t5;
L1: while (d2 < 0) {
        SetPixel(x, y);
        y = y + 1;
        t9 = t9 + t3;
        if (d1 < 0) {
L2:         d1 = d1 + t9 + t2;
            d2 = d2 + t9;
        }
        else {
```

```
L3:          x = x − 1;
             t8 = t8 − t6;
             d1 = d1 − t8 + t9 + t2;
             d2 = d2 − t8 + t5 + t9;
         }
    }
L4: repeat {
         SetPixel(x, y);
         x = x − 1;
         t8 = t8 − t6;
         if (d2 < 0){
L5:          y = y + 1;
             t9 = t9 + t3;
             d2 = d2 − t8 + t5 + t9;
         }
         else
L6:          d2 = d2 + t5 − t8;
    } until (x < 0);
```

We now count the arithmetic operations for the midpoint ellipse algorithm. Again we count only those operations in the iteration loops which constitute the main body of the algorithm, i.e., the **while** loop labeled L_1, which draws the first octant elliptical arc, and the **repeat** loop labeled L_4 which draws the second octant elliptical arc.

Referring to Figure 6.7(b), T is the juncture of the first and the second octant. We have

$$\begin{aligned} H_1 &= b - \frac{b^2}{\sqrt{a^2+b^2}}, \\ W_1 &= \frac{a^2}{\sqrt{a^2+b^2}}, \\ H_2 &= \frac{b^2}{\sqrt{a^2+b^2}}, \\ W_2 &= a - \frac{a^2}{\sqrt{a^2+b^2}}. \end{aligned}$$

Denote by N'_{ai}, N'_{ti}, $i = 1, 4$, the number of additions and the number of sign tests when loops L_1 and L_4 are executed. We have the following derivations.

1. The **while** loop labeled L_1 is executed about H_2 times. It contains two additions and two sign tests, excluding operations in the conditionally executed **if** ... **else** statement. The compound statement labeled L_2 is executed when condition $d_1 < 0$ is true, and the compound statement labeled L_3 is executed when condition $d_1 < 0$ is false. Note that the digitized first octant elliptical arc consists of vertical pixel runs, and there are a total of H_2 pixels in these vertical runs. Condition $d_1 < 0$ being false corresponds to a pixel move in northwest direction, i.e., the beginning of a new vertical run. Since there are approximately W_2 vertical runs in the first octant, the compound statement labeled L_3, which involves eight additions, is therefore executed approximately W_2 times, and thus the compound statement labeled L_2, which involves three additions, is executed approximately $H_2 - W_2$ times. The total numbers of

arithmetic operations when L_1 the loop is executed are therefore

$$\begin{aligned} N'_{a1} &= 2H_2 + 3(H_2 - W_2) + 8W_2 = 5H_2 + 5W_2, \\ N'_{t1} &= 2H_2. \end{aligned}$$

2. The **repeat** loop labeled L_4 is executed about W_1 times. It contains two additions and two sign tests, excluding operations in the conditionally executed compound statements in the **if** ... **else** statement. The compound statement labeled L_5 is executed when condition $d_2 < 0$ is true, and the statement labeled L_6 is executed when condition $d_2 < 0$ is false. Note that the digitized second octant elliptical arc consists of horizontal pixel runs, and there are a total of W_1 pixels in the horizontal runs. Condition $d_2 < 0$ being true corresponds to a pixel move in northwest direction, i.e., the beginning of a new horizontal pixel run. Since there are approximately H_1 horizontal runs in the second octant, the compound statement labeled L_5, which involves five additions, is therefore executed approximately H_1 times. The statement labeled L_6, which involves two additions, is therefore executed approximately $W_1 - H_1$ times.

The total numbers of arithmetic operations when L_4 loop is executed is therefore

$$\begin{aligned} N'_{a4} &= 2W_1 + 2(W_1 - H_1) + 5H_1 = 4W_1 + 3H_1, \\ N'_{t4} &= 2H_1 + 2W_1. \end{aligned}$$

We thus have the total numbers of arithmetic operations in the main body of the

midpoint ellipse algorithm as follows:

$$\begin{aligned} N'_a &= 3H_1 + 5H_2 + 4W_1 + 5W_2, \\ N'_t &= 2W_1. \end{aligned}$$

It is still not clear which algorithm has fewer arithmetic operations since it is related to the values of a and b. Therefore in order to get some concrete results, we let $a = b$, $a = 2b$, and $a = 10b$. In all these cases, the numbers of the various arithmetic operations in the two algorithms can be represented in terms of b. For $a = 2b$, we have

$$\begin{aligned} h_1 &= (1 - \frac{1}{\sqrt{2}})b, & h_2 &= (\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{5}})b, \\ h_3 &= (\frac{1}{\sqrt{5}} - \frac{1}{\sqrt{17}})b, & h_4 &= \frac{1}{\sqrt{17}}b, \\ w_1 &= \frac{2}{\sqrt{2}}b, & w_2 &= (\frac{4}{\sqrt{5}} - \frac{2}{\sqrt{2}})b, \\ w_3 &= (\frac{8}{\sqrt{17}} - \frac{4}{\sqrt{5}})b, & w_4 &= (2 - \frac{8}{\sqrt{17}})b, \\ H_1 &= (1 - \frac{1}{\sqrt{5}})b, & H_2 &= \frac{1}{\sqrt{5}}b, \\ W_1 &= \frac{4}{\sqrt{5}}b, & W_2 &= (2 - \frac{4}{\sqrt{5}})b. \end{aligned}$$

We therefore have

$$N_a \approx 9.80b, \quad N_s \approx 0.71b, \quad N_c \approx 2.44b,$$

and

$$N'_a \approx 12.11b, \quad N'_t \approx 4.47b.$$

Using the same method as above we calculate the number of arithmetic operations

	N_a	N_s	N_c	N'_a	N'_t
$a = b$	$6.77b$	$0.42b$	$1.71b$	$8.71b$	$2.83b$
$a = 2b$	$9.80b$	$0.71b$	$2.44b$	$12.11b$	$4.47b$
$a = 10b$	$26.78b$	$1.63b$	$6.68b$	$43.25b$	$20.10b$

Table 6.1: Operation counts for RLS-Ellipse2 and Midpoint algorithms.

for both algorithms for $a = b$ and $a = 10b$. The results, together with the results for $a = 2b$ are tabulated in Table 6.1.

Table 6.1 shows that ALGORITHM 6.5 runs slightly faster than the midpoint algorithm when $a = b$, but the gain in speed increases as the ratio of a over b increases. When $a = 2b$ the running time of algorithm RLS-Ellipse2 is about 3/4 of the running time of the midpoint algorithm. It runs twice as fast as the midpoint algorithm when $a = 10b$, when the ellipse is very flat. The theoretical results agree with our intuitive guess that as an ellipse becomes elongated, there are more long runs in the digitized ellipse, and the run-length slice method becomes more favorable. If the ellipse is close to symmetric ($a \approx b$) then the overhead in the run-length approach in the ellipse case is too large to make the algorithms presented much more efficient than previously published algorithms. For the case of the circle ($a = b$) specialized run-length algorithms can again be made competitive as will be discussed in the next chapter.

ALGORITHM 6.3 (RLS-Ellipse1), ALGORITHM 6.5 (RLS-Ellipse2), as well as ALGORITHM 6.6 (Van Aken's midpoint ellipse algorithm) were implemented in C and tested extensively on an SGI INDIGO2. We let the output functions $SetPixel()$, $HorizontalRun()$ and $VerticalRun()$ be dummy functions in our implementations,

i.e., the output is actually not implemented. The purpose is to compare only the time used in arithmetic operation. Table 6.2 and Table 6.3 give the comparisons of the efficiencies of the three algorithms based on the accumulated running times in seconds. For a specific pair of a and b, each algorithm was run 3000 times. Column MIDPOINT, RLS-Ellipse1 and RLS-Ellipse2 in both tables list the running times of the three algorithms for different pairs of a and b. The ratios of running time of ALGORITHM 6.5 (RLS-Ellipse2) to the running time of ALGORITHM 6.6 (Van Aken's midpoint ellipse algorithm) are graphed in Figure 6.8. The running times were obtained by using UNIX command *time*. In Table 6.2, a is fixed to 800, and b varies between 640 and 80 in steps of 80, i.e., we let b be $\frac{8}{10}a, \frac{7}{10}a, \cdots, \frac{1}{10}a$. In Table 6.3, a is fixed to 200, and b varies from a to $\frac{1}{10}a$ in steps of 20. The pairs of (800, 800), (800, 720) were not used in Table 6.2. This is because the midpoint algorithm overflows for large a and b pair such as (800, 800) and (800, 720) in our implementation where integers are four bytes signed integers. The reason for the overflow can be explained as follows. Referring to the code of the midpoint ellipse algorithm, once the algorithm enters the **while** loop labeled L_1, the pixels in the first vertical run in the first octant are set while the condition $d_1 < 0$ is true. Once $d_1 \geq 0$, x is decreased by one, which means the drawing of the second vertical run will begin. Note that d_2 is also updated by the statement $d_2 = d_2 - t_8 + t_5 + t_9$, which is in effect equivalent to $d_2 = d_2 - 4 * b * x * x + 2 * b * b + 4 * a * a * y$. If there is no overflow, d_2 remains negative, and the algorithm begins to draw the second vertical run. When a and b are very large and $a \approx b$, overflow may occur since after the first vertical run is drawn, $x = a - 1$ is far greater than y, and the absolute value of d_2 becomes very large after it is updated by executing the above statement. If d_2

(a, b)	MIDPOINT	RLS-Ellipse1	RLS-Ellipse2
(800, 640)	2.278	1.920	1.816
(800, 560)	2.133	1.816	1.716
(800, 480)	2.011	1.709	1.562
(800, 400)	1.893	1.594	1.429
(800, 320)	1.792	1.469	1.281
(800, 240)	1.679	1.336	1.120
(800, 160)	1.596	1.189	0.936
(800, 80)	1.520	1.032	0.730

Table 6.2: Runtime comparison of three ellipse algorithms for $a = 800$.

changes its sign to positive because of overflow, the **while** loop labeled L_1 terminates prematurely, and the **repeat** loop is entered incorrectly to draw the second octant. The overflow problem therefore occurs much sooner for the midpoint algorithm then the run-length slice algorithm.

6.6 Summary

We discussed the use of run-length slice methodology to the scan-conversion of canonical ellipses. The first quadrant of an ellipse was partitioned into two octants. Horizontal and vertical pixel runs were then calculated for the two octants using forward differencing with integer arithmetic. The correct octant transition was investigated. Two run-length slice ellipse algorithms, ALGORITHM 6.3 and ALGORITHM 6.5 were presented. The new algorithms involve fewer arithmetic operations as compared with the well-known midpoint ellipse algorithm and thus yield higher efficiency, especially for ALGORITHM 6.5 which takes advantage of adaptive forward differencing

(a, b)	MIDPOINT	RLS-Ellipse1	RLS-Ellipse2
(200, 200)	0.648	0.533	0.511
(200, 180)	0.615	0.511	0.486
(200, 160)	0.578	0.486	0.462
(200, 140)	0.544	0.461	0.431
(200, 120)	0.512	0.434	0.401
(200, 100)	0.479	0.405	0.367
(200, 80)	0.453	0.374	0.331
(200, 60)	0.427	0.340	0.288
(200, 40)	0.405	0.304	0.243
(200, 20)	0.387	0.262	0.191

Table 6.3: Runtime comparison of three ellipse algorithms for $a = 200$.

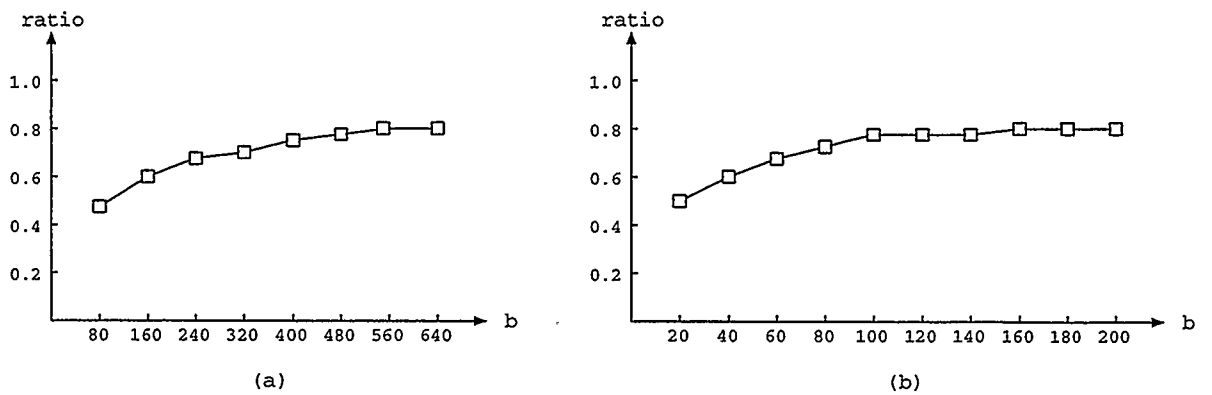


Figure 6.8: Ratios of the running time of algorithm RLS-Ellipse2 to the running times of the midpoint ellipse algorithm. In (a) $a = 800$, b varies from 80 to 640; in (b) $a = 200$, b varies from 20 to 200.

in calculating the lengths of runs. A numerical comparison supports the theoretical results.

Chapter 7

Hybrid Scan-Conversion of Circles

7.1 Introduction

This chapter is devoted to the application of the run-length slice methodology to scan-conversion of circles. The run-length slice algorithms for canonical ellipses developed in chapter 6 can be used for the scan-conversion of circles since circles are special ellipses where the lengths of major axis and minor axis are equal. It was, however, pointed out in chapter 6 that the overhead in the run-length ellipse algorithms is so large that the algorithms turns out to be less efficient than previously published circle algorithms. Specialized run-length algorithms for circles can, however, be made competitive as will be shown in this chapter. It is assumed that the circle has integer radius and is centered at the origin. The equation of the circle is therefore

$$x^2 + y^2 = r^2 \tag{7.1}$$

where r is an integer. Because of the 8-way symmetry of a scan-converted circle only a 45° circular arc, usually in the first or second octant, has to be scan-converted. The remainder of the circle can be obtained by symmetry. Other circles with integer radii and centers can be obtained by translation.

Scan-conversion of circles has received considerable research attention since the early 1960s [Bresenham 77, Foley 82, Foley 90, Horn 76, Hsu 93, McIlroy 83, Wright 90,

Wu 87]. Conventional circle scan-conversion algorithms generate one pixel in each iteration loop and we therefore call them *pixel-based* circle algorithms. One pixel is chosen from two candidate pixels according to a specific optimization criterion (refer to chapter 2 for the three optimization criteria normally used). This is implemented logically by testing the sign of a discriminator in the algorithm. The discriminator is then updated using simple arithmetic operations so that the algorithm can proceed to select the next pixel. This simple logic allows simple implementation and relatively fast speed of scan-conversion. Pixel-based circle algorithms are represented by Bresenham's circle algorithm [Bresenham 77] and the midpoint circle algorithm [Foley 90].

Methods for speeding up the scan-conversion of circles involve generating multiple pixels in each iteration instead of scan converting one pixel at a time. Wu and Rokne [Wu 87] for example developed an algorithm which draws two pixels in each incremental step.

The use of run-length slice methodology for the scan-conversion of circles was first proposed by Hsu, Chow and Liu [Hsu 93]. The algorithms presented in [Hsu 93] are called *short line segment incremental algorithms* where a short line segment effectively refers to a horizontal run of pixels. All of the horizontal runs that approximate a 45° circular arc in the second octant are calculated incrementally with descending y. The authors noted that their new algorithms are, on the average, at least 1.36 times faster than the other existing circle algorithms such as Bresenham's algorithm and the midpoint algorithm.

In this chapter we will present new run-length slice circle algorithms that further speed up the scan-conversion of circles. The improvements are due to the faster

calculation of run-lengths and the use of a hybrid approach that combines the advantages of the run-length method and the pixel-based method for the scan-conversion of circles. The hybrid approach is introduced based on the observation that while there are long runs of pixels in a 45° circular circle, there are also short runs of length one. For example, when the radius is 128, the sequence of run lengths are 12,8,6,4,4,4,3,3,2,3,2,3,2,2,2,2,1,2,2, 1,2,2,1,1,2,1,1, 2,1,1,1,2,1,1,1,1,1,1. The further (maybe obvious) observation is that it costs more to calculate a run of length one than to generate a single pixel according to the sign of a discriminator. Calling a function to draw a pixel run of of length one also costs more than simply setting a pixel at position (x, y). Our suggestion is therefore to switch to the pixel-based approach when the lengths of the remaining runs are at most two (and most often they are of length one).

The techniques used to speed up the run-length calculation have been used in chapter 6 for the calculation of the run-lengths of elliptical arc. In this chapter we will again use double-step forward differencing for the run-length calculation for the portion of a 45° circular arc where the run-lengths are at least 2. The incremental computation of run-lengths are based on the circle equation Eq. (7.1). For illustrative purpose we first introduce a new version of the run-length 45° circular arc algorithm which contains some improvements on algorithm 1 of [Hsu 93] (abbreviated *Algorithm HCL* in the sequel). Single-step forward differencing is used in the run-length calculation in this algorithm. The techniques mentioned above are then used to devise further improved algorithms. Before we introduce new algorithms, it is of interest to first investigate the run-length properties of a 45° circular arc in the second octant.

7.2 Run-Length Properties of 45° Circular Arc

As we did for deriving the run-length slice ellipse algorithms, we impose a set of mid-lines $y = r - i + 0.5$, $i = 1, 2, \cdots$. The i-th mid-line intersects the second octant circular arc at (x_i, y_i) where $y_i = r - i + 0.5$(see Figure 7.1). Similar to subdividing elliptical arc by a set of midlines we immediately have that the sequence of pixels in the i-th ($i > 1$) horizontal run is

$$(\lfloor x_{i-1} \rfloor + 1, r - i + 1), \cdots, (\lfloor x_i \rfloor, r - i + 1),$$

and the sequence of pixels in the first horizontal run is

$$(0, r), \cdots, (\lfloor x_1 \rfloor, r),$$

where r is the radius of the circle. Letting $\dot{x}_i = \lfloor x_i \rfloor$ and $\dot{y}_i = r - i + 1$, it follows that the i-th ($i > 1$) horizontal run starts from $(\dot{x}_{i-1} + 1, \dot{y}_i)$ and ends at $(\dot{x}_i, \dot{y}_i)$. The length of the i-th horizontal run is therefore $\dot{x}_i - \dot{x}_{i-1}$ for $i > 1$.

Let $\gamma_i, i = 1, 2, \cdots$, be the length of the i-th horizontal pixel run.

Theorem 5 *One of the following relations hold for γ_i and γ_{i+1} for $i \geq 2$:*

$$1: \quad \gamma_i > \gamma_{i+1},$$

$$2: \quad \gamma_i = \gamma_{i+1},$$

$$3: \quad \gamma_i + 1 = \gamma_{i+1}.$$

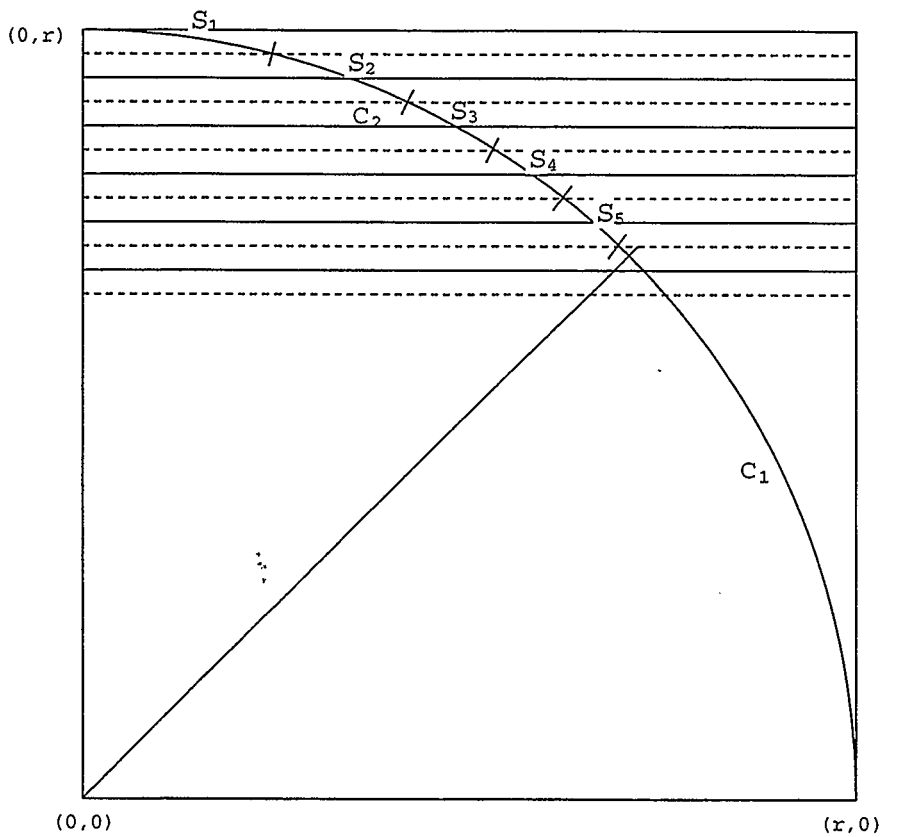


Figure 7.1: A set of midlines (stippled) $y = r - i + 0.5$ divide a 45° circular arc into segments.

Proof: According to the circle equation Eq. (7.1) we have for $i \geq 1$ that

$$x_i^2 = r^2 - (r - (i - 0.5))^2 \tag{7.2}$$

and

$$x_{i+1}^2 = r^2 - (r - (i + 0.5))^2. \tag{7.3}$$

Subtracting (7.2) from (7.3) and rearranging yields

$$x_{i+1}^2 = x_i^2 + 2(r - i), \quad i \geq 1. \tag{7.4}$$

Let $l_i = x_i - x_{i-1}, i \geq 2$. It follows from Eq. (7.4) that

$$x_i^2 - x_{i-1}^2 > x_{i+1}^2 - x_i^2.$$

Hence,

$$(x_i - x_{i-1})(x_i + x_{i-1}) > (x_{i+1} - x_i)(x_{i+1} + x_i),$$

$$\frac{l_i}{l_{i+1}} = \frac{x_i - x_{i-1}}{x_{i+1} - x_i} > \frac{x_{i+1} + x_i}{x_i + x_{i-1}} > 1.$$

We therefore have $l_i > l_{i+1}$. The conclusion of the theorem becomes straightforward by noting that

$$\gamma_i = \lfloor l_i \rfloor \text{ or } \lfloor l_i \rfloor + 1,$$

$$\gamma_{i+1} = \lfloor l_{i+1} \rfloor \text{ or } \lfloor l_{i+1} \rfloor + 1,$$

and $l_i > l_{i+1}$ implies

$$\lfloor l_i \rfloor \geq \lfloor l_{i+1} \rfloor. \qquad \square$$

Before we present the next theorem, we prove that the length of the first horizontal run is $\lfloor \sqrt{r-1} \rfloor + 1$. We first prove a small lemma:

Lemma 4 *Let $s \geq 0$ be an integer. Then*

$$\lfloor \sqrt{s} \rfloor = \lfloor \sqrt{s+\epsilon} \rfloor, \ \ 0 \leq \epsilon < 1.$$

Proof. Suppose $a^2 \leq s < (a+1)^2$, where $a \geq 0$ is an integer. Then $s = a^2 + b$, where b is an integer and where $0 \leq b < 2a+1$.

It is obvious that

$$b + \epsilon < 2a + 1.$$

Therefore

$$s + \epsilon = a^2 + b + \epsilon < a^2 + 2a + 1 = (a+1)^2.$$

We then have

$$a \leq \lfloor \sqrt{s} \rfloor \leq \lfloor \sqrt{s+\epsilon} \rfloor = \sqrt{a^2 + r + \epsilon} < a + 1,$$

which amounts to

$$\lfloor \sqrt{s} \rfloor = \lfloor \sqrt{s+\epsilon} \rfloor = a.$$

This proves the lemma. $\square$

Let $i = 1$ in Eq. (7.2) we have $x_1^2 = r - 1/4$. Therefore

$$\dot{x}_1 = \lfloor \sqrt{r - 1/4} \rfloor = \lfloor \sqrt{r - 1 + 3/4} \rfloor = \lfloor \sqrt{r-1} \rfloor$$

using the above lemma. The length of the first horizontal run is thus $\lfloor\sqrt{r-1}\rfloor + 1$.

In the next theorem we assume that there is more than one run.

Theorem 6 *The length of the first horizontal run γ_1 is greater or equal to the length of the second horizontal run γ_2. Equality only holds for $r = 4$.*

Proof: Note that if a 45° discrete circular arc has more than one horizontal run then this implies that the radius of the circle is $r \geq 3$. We already know that

$$\gamma_1 = \lfloor\sqrt{r-1}\rfloor + 1,$$

$$\gamma_2 = \dot{x}_2 - \dot{x}_1 = \lfloor\sqrt{3(r-1)}\rfloor - \lfloor\sqrt{r-1}\rfloor.$$

It is easy to verify that $\gamma_1 > \gamma_2$ for $r = 3$, and $\gamma_1 = \gamma_2 = 2$ for $r = 4$. To prove that $\gamma_1 > \gamma_2$ for $r > 4$, we first give the following inequality:

$$\sqrt{3(k^2+2k)} - k < k + 1 \tag{7.5}$$

where $k \geq 2$ is an integer. This inequality can be proved by noting that

$$(k-1)^2 = k^2 - 2k + 1 > 0$$

for $k \geq 2$. We thus have

$$3k^2 + 6k < 4k^2 + 4k + 1.$$

Therefore

$$\sqrt{3(k^2+2k)} < 2k + 1,$$

or

$$\sqrt{3(k^2+2k)} - k < k+1.$$

Now let $\lfloor\sqrt{r-1}\rfloor = k$, then $r-1 \leq k^2+2k$. Since $r > 4$, we have $k \geq 2$. Therefore

$$\begin{aligned}
\gamma_2 &= \lfloor\sqrt{3(r-1)}\rfloor - \lfloor\sqrt{r-1}\rfloor \\
&\leq \sqrt{3(k^2+2k)} - k \\
&< k+1 \\
&= \lfloor\sqrt{r-1}\rfloor + 1 \\
&= \gamma_1. \qquad \square
\end{aligned}$$

Theorems 5 and 6 yield the following corollary directly. We omit the proof since it is quite straightforward.

Corollary 1 *The horizontal runs of a* 45° *discrete circular arc satisfy* $\gamma_1 \geq \gamma_i$ *for* $i > 1$.

7.3 Run-Length Slice Circle Algorithm

The techniques for calculating run-lengths of a 45° circular arc similar to those used for calculating the run-lengths of the first quadrant elliptical arc in chapter 6. As was stated in the last section, the i-th horizontal run starts from $(\dot{x}_{i-1}+1, \dot{y}_i)$ and ends at $(\dot{x}_i, \dot{y}_i)$ with $\dot{x}_0 = 0$ and $\dot{y}_0 = r$. The calculation of $\dot{x}_i$ $(i > 1)$ can be performed in an incremental manner based on Eq. (7.4).

Letting $X_1 = x_1^2 - 3/4 = r - 1$ and $X_{i+1} = X_i + 2(r - i)$ for $i = 1, 2, \cdots$, we have

$$\dot{x}_i = \lfloor \sqrt{x_i^2} \rfloor = \lfloor \sqrt{X_i} \rfloor.$$

Therefore, instead of calculating x_i^2, we calculate X_i in the iterations, which eliminates floating point operations. Forward differencing is then employed as follows to calculate $\dot{x}_i$ without invoking the square root function. Suppose that $\dot{x}_{i-1} = \lfloor \sqrt{X_{i-1}} \rfloor$ has been evaluated, and the variables x and X hold the values of $\dot{x}_{i-1}$ and $\dot{x}_{i-1}^2$ respectively. Noting that

$$(x + 1)^2 - x^2 = 2x + 1$$

we obtain $\dot{x}_i$ by repeating the iteration

$$X = X + 2x + 1$$

$$x = x + 1$$

until $X \geq X_i$. If $X = X_i$ then the value of x is $\dot{x}_i$, otherwise we decrease x by one to obtain $\dot{x}_i$. The value of $\dot{x}_i$ can be found more efficiently as follows. Let the values of the variables D and x be $X_i - \dot{x}_{i-1}^2$ and $\dot{x}_{i-1}$ respectively before entering the repetitive loop. After the following iterations, the value of x will reach $\dot{x}_i$:

while $(D >= 2x + 1)$ {

$D = D - 2x - 1;$

$x = x + 1;$

}

Setting $Delta1 = 2x_{i-1} + 1$ before entering the loop, the efficiency of the above code segment can be improved by incorporating second order differencing of the square function:

```
while (D >= Delta1) {
        D = D - Delta1;
        Delta1 = Delta1 + 2;
        x = x + 1;
}
```

We next discuss when to terminate the run-length calculation for the scan-conversion of the 45° circular arc in the second octant. This problem relates to the pixel configurations at the octant change for a digitized circular arc. It is readily understood that because the digitized image of a circular arc in the first quadrant is symmetric with respect to the line $y = x$, the pixel configuration at octant change can only be configuration (c) or (e) in Figure 6.1 if there exists a transition point, or configuration (a) in Figure 6.5 if there is not a transition point. Therefore once we start a tentative horizontal run and find that $x > y$ for the leading pixel of that run, we know that the scan-conversion of the second octant circular arc has been finished and we therefore stop the run-length calculation.

Below is the run-length slice algorithm for the scan-conversion of the second octant circular arc. Note that we use single-step forward differencing in the run-length calculation. Incorporation of double-step forward differencing, as we did for

the ellipse algorithm, can further speed up the computation. This will be done in the next section where we also introduce the hybrid approach.

ALGORITHM: Run-Length Slice Circle Algorithm (RLS-Circle)

$y = r;$

$x = x0 = 0;$

$D = r - 1;$

$Delta1 = 1;$

$DeltaX = 2 * r;$

while $(x0 < y)$ {

    while $(D >= Delta1)$ {

        $D = D - Delta1;$

        $Delta1 = Delta1 + 2;$

        $x = x + 1;$

    }

    $DrawRun(x0, x, y);$

    $y = y - 1;$

    $x0 = x + 1;$

    $D = D + DeltaX;$

    $DeltaX = DeltaX - 2;$

}

7.4 Hybrid Scan-Conversion of Circles

We now discuss how to improve the efficiency of the run-length slice circle algorithm RLS-Circle.

Using the same notation as we did in Figure 6.1, we call the 45° circular arc in the second octant C_2. The point on C_2 where the slope of the circle is -1/2 furthermore divides C_2 into C_{21} and C_{22}. Similar to what we have done for the calculation of run-length for elliptical arcs the calculation of run-lengths of the 45° circular arc can be further sped up by using double-step forward differencing in the portion of the arc where the run-lengths are at least two. The point $P(x, y)$ that divides the 45° circular arc C_2 into segments C_{21} and C_{22} satisfies $2x = y$. So we start with double-step forward differencing to calculate the run-lengths until we enter the segment of C_{22}. Then, instead of switching to single-step forward differencing to calculate run-lengths, we switch to to a pixel-based method to scan-convert the remainder of the arc. Since a pixel-based algorithm, such as the midpoint circle algorithm, uses a discriminator to choose one of the two candidate pixels in each iteration, the problem imposed on the hybrid method is to obtain the value of the discriminator with less computational cost at the point where we start pixel-based method. It turns out that this computation is relatively easy and cheap for the discriminator used in the pixel-based circle algorithms. In our implementation we use the midpoint method.

The following theorem provides a condition for switching from the run-length method to the pixel-based method.

Theorem 7 *Let the last pixel of the* $(i-1)$*-th run be* $(\dot{x}_{i-1}, \dot{y}_{i-1})$*. If* $2\dot{x}_{i-1} > \dot{y}_{i-1}$ *then the segment* S_i *starts from a point in* C_{22}*, i.e.,* S_i *is not in* C_{21}*.*

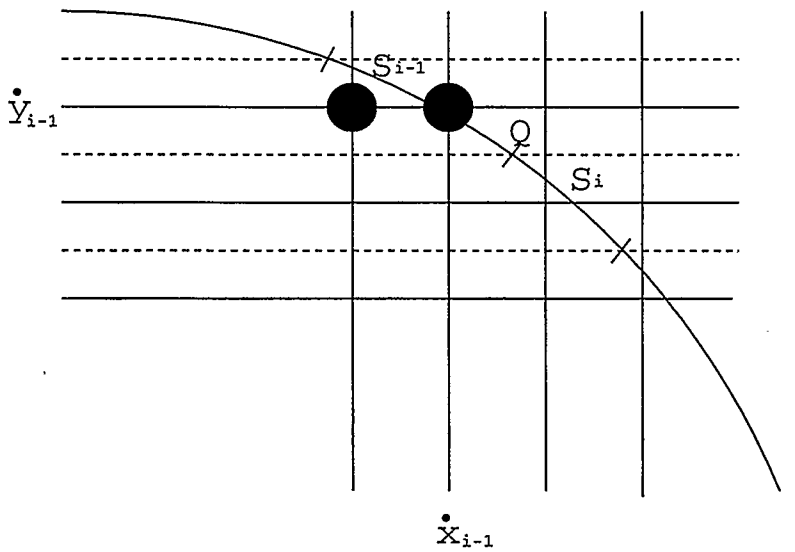


Figure 7.2: Transition from C_{21} to C_{22}.

Proof: Referring to Figure 7.2, segment S_i starts from

$$Q = (Q_x, Q_y) = (x_{i-1}, \dot{y}_{i-1} - 0.5).$$

It follows from $2\dot{x}_{i-1} > \dot{y}_{i-1}$ that

$$2(\dot{x}_{i-1} + 1) > \dot{y}_{i-1} + 2,$$

$$2(x_{i-1} + 1) > 2(\dot{x}_{i-1} + 1) > \dot{y}_{i-1} + 2,$$

$$2x_{i-1} > \dot{y}_{i-1} > \dot{y}_{i-1} - 0.5,$$

$$2Q_x > Q_y.$$

Hence the starting point of the segment S_i is in C_{22}. □

Suppose that we switch to the midpoint algorithm to generate the remaining

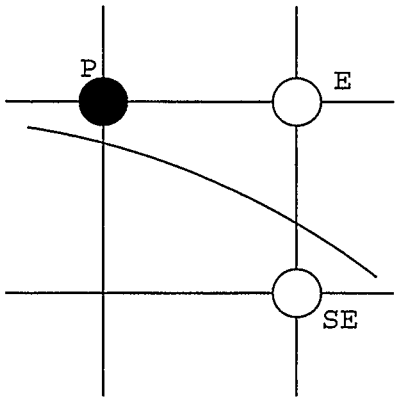


Figure 7.3: Pixel selection in the midpoint circle algorithm.

pixels for the digitized 45° circular arc, and that before we switch to the midpoint algorithm, the last pixel of the last run is (x', y') where $2x' > y'$. Then the remaining pixels start from $x = x' + 1, y = y' - 1$. We need to initialize the discriminator at this point so that we can choose subsequent pixels using the midpoint circle algorithm. In the following we derive the formula to calculate the value of the discriminator at that point.

The initial value of the discriminator in the midpoint circle algorithm is

$$d = 1 - r.$$

Referring to Figure 7.3, suppose the previously selected pixel is $P(x_p, y_p)$, then the algorithm selects the next pixel between E (corresponding to a horizontal move) and SE (corresponding to a diagonal move) according to the sign of d and updates d accordingly:

- If $d < 0$ then choose E, $d = d + \Delta E = d + 2x_p + 3$.
- If $d \geq 0$ then choose SE, $d = d + \Delta SE = d + 2(x_p - y_p) + 5$.

Note that each diagonal move can be decomposed into a horizontal move and a vertical move. The change of d when a diagonal move occurs can therefore be represented as the sum of the change of d for a horizontal move and a vertical move:

$$\Delta SE = 2(x_p - y_p) + 5 = (2x_p + 3) + 2(1 - y_p) = \Delta E + \Delta S,$$

where $\Delta S = 2(1 - y_p)$. There are x horizontal moves and $r - y$ vertical moves from the initial pixel position $(0, r)$ to (x, y). The value of d therefore changes to

$$\begin{aligned} d &= 1 - r + [2 \cdot 0 + 3] + \cdots + [2(x-1) + 3] \\ &\quad +2(1 - r) + 2[1 - (r-1)] + \cdots + 2[1 - (y+1)] \\ &= 1 - r + 2[1 + 2 + \cdots + (x-1)] + 3x \\ &\quad +2(r - y) - 2[r + (r-1) + \cdots + (y+1)] \\ &= 1 - r + x(x-1) + 3x + 2(r-y) - (r + y + 1)(r - y) \\ &= 1 - r + x^2 + 2x - (r - y)(r + y - 1). \end{aligned}$$

Since we know how to calculate the value of d, we can switch to the midpoint circle algorithm for the remaining pixels. Once a horizontal pixel move occurs, the next pixel move must be a diagonal move since the maximum horizontal run length is two. We can combine these two steps into one, i.e., move from (x, y) to $(x+2, y-1)$ adding the intermediary pixel $(x + 1, y)$. The change of the discriminator d is

$$\Delta E - \Delta SE = 2x + 3 + 2(x + 1) - 2y + 5 = 4x - 2y + 10.$$

Now, starting the run-length calculation using double-step forward differencing, switching to the midpoint method once the switching condition is satisfied, we have the following hybrid algorithm for the scan-conversion of 45° circular arc in the second octant.

ALGORITHM: Hybrid Circle Algorithm (Hybrid-Circle)

```
    y = r;
    x = x0 = 0;
    D = r - 1;
    Delta2 = 4;
    DeltaX = 2 * r;
L1: while (2 * x < y) {
L2:     while (D >= Delta2) {     /* double-step forward */
            D = D - Delta2;
            Delta2 = Delta2 + 8;
            x = x + 2;
        }
        Delta1 = 2 * x + 1;
L3:     if (D >= Delta1) {     /* additional single-step forward */
            D = D - Delta1;
            Delta2 = Delta2 + 4;
            x = x + 1;     /* one step adjustment */
        }
        DrawRun(x0, x, y);
```

```
        y = y − 1;
        x0 = x + 1;
        D = D + DeltaX;
        DeltaX = DeltaX − 2;
    }
    x = x + 1;
    d = 1 − r + x ∗ x + 2 ∗ x − (r − y) ∗ (r + y − 1);
L4: while(x <= y) {
        SetPixel(x, y);
L5:     if (d < 0) {
          d = d + 4 ∗ x − 2 ∗ y + 10;
          x = x + 1;
          SetPixel(x, y);
          x = x + 1;  y = y − 1;
        }
        else {
            d = d + 2 ∗ (x − y) + 5;
            x = x + 1;  y = y − 1;
        }
    }
```

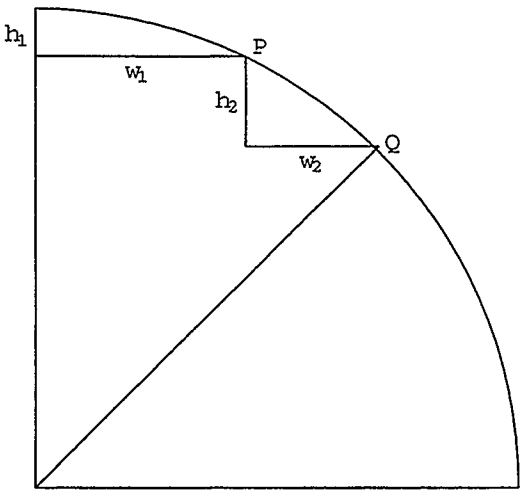


Figure 7.4: Counting arithmetic operations for algorithm Hybrid-Circle.

7.5 Complexity Analysis and Numerical Results

In this section we will analyze the complexity of Algorithm Hybrid-Circle to show that it involves less arithmetic operations than conventional pixel based circle algorithms. We then tabulate numerical results for some tests of Algorithm RLS-Circle and Hybrid-Circle as well as Algorithm HCL and Bresenham's circle algorithm that supports the the conclusion based on the theoretical analysis.

The arithmetic operations involved in algorithm Hybrid-Circle are counted as follows. We count only those operations in two iteration loops which constitute the main body of the algorithm, i.e., the *while* loops labeled L_1 which draws part of the 45° circular arc run by run, and the *while* loop labeled L_4 which draws the remainder of the circular arc pixel(s) by pixel(s). The multiplications by powers of 2 in the algorithm are implemented by binary shifts. The quick ± 1 operations are neglected in our analysis. Referring to Figure 7.4, P is the point on the 45° circular arc in the second octant whose coordinates satisfy $2x = y$. A trivial geometric computation

gives the coordinates of $P : x = \frac{1}{\sqrt{5}}r, y = \frac{2}{\sqrt{5}}r$. We thus have in Figure 7.4:

$$h_1 = (1 - \frac{2}{\sqrt{5}})r,$$

$$w_1 = \frac{1}{\sqrt{5}}r,$$

$$h_2 = (\frac{2}{\sqrt{5}} - \frac{1}{\sqrt{2}})r,$$

$$w_2 = (\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{5}})r.$$

The **while** loop at L_1 is executed about h_1 times. It contains three additions, one shift and one comparison excluding operations in the conditionally executed bodies of statements L_2 and L_3. The **while** loop labeled L_4 is executed approximately h_2 times, each of which contains one comparison and one sign test excluding operations in the conditionally executed bodies of statement L_5. The number of times that the nested *while* loop labeled L_2 is executed varies each time this statement is executed, however, after the outmost *while* loop (L_1 loop) is finished, the total number of times this **while** loop (L_2 loop) is executed is approximately $w_1/2 - h_1/4$. There are three additions and one comparisons in this loop. The **if** statement labeled L_3 in the body of L_1 loop involves two additions and one comparison. This statement is executed when the length of a run is odd, and we can assume that on the average case it it is executed half the number of times the L_1 loop does, i.e., $h_1/2$ times. Now let us investigate the **if** statement labeled L_5. The condition $d < 0$ in this **if** statement is true when a horizontal pixel move occurs, i.e., a run of length two occurs. The number of this occurrence is $w_2 - h_2$. So the compound statement after

$if(d < 0)$, which contains three additions and two shifts, is executed $w_2 - h_2$ times; and the compound statement after *else*, which contains three additions and one shift, is executed $2h_2 - w_2$ times. Denoting by N_a, N_s, N_t, and N_c the number of additions, shifts, sign tests, and comparisons respectively in Algorithm Hybrid-Circle, we have

$$\begin{aligned}
N_a &= 3h_1 + 3(\frac{w_1}{2} - \frac{h_1}{2}) + h_1 + 3(w_2 - h_2) + 3(2h_2 - w_2) \\
&= \frac{13}{4}h_1 + \frac{3}{2}w_1 + 3h_2 \\
&\approx 1.58r, \\
N_s &= h_1 + 2(w_2 - h_2) + 2h_2 - w_2 = h_1 + w_2 \\
&\approx 0.47r, \\
N_t &= h_2 \approx 0.19r, \\
N_c &= h_1 + \frac{w_1}{2} - \frac{h_1}{4} + \frac{h_1}{2} + h_2 \\
&= \frac{5}{4}h_1 + \frac{w_1}{2} + h_2 \\
&\approx 0.54r.
\end{aligned} \tag{7.6}$$

These figures illustrate that Algorithm hybrid-Circle involves fewer arithmetic operations than the pixel-based circle algorithms, among which an algorithm attributed to Michener by [Foley 82], for example, has $N_a \approx 1.71r, N_s = r, N_t = N_c \approx 0.71r$.

Algorithm RLS-Circle, Hybrid-Circle as well as Algorithm HCL (Alg. 1 of [Hsu 93]) and Bresenham's algorithm (from [Bresenham 77], abbreviated Bres in Tables 7.1 and 7.2) were implemented in C and tested extensively on a PC-80486 running at 66MHz with a numeric coprocessor. The multiplications by powers of

two in the algorithms are implemented by binary shifts. Algorithm HCL invokes the square root function once to calculate the first horizontal run. Actually only the integer part of the square root is required in this particular problem. In our implementation we just call the function *sqrt*() in the math library since we do not know if they call a specially designed square root function. We let the output functions *SetPixel*() and *DrawRun*() be dummy functions in our implementations, i.e., the output is actually not implemented. The purpose is to compare only the time used in arithmetic operations in these algorithms. It is not surprising that Algorithm Hybrid-Circle runs much faster than Algorithm HCL since it involve fewer arithmetic operations. This fact implies that Hybrid-Circle runs much faster than Bresenham's circle algorithm and the midpoint circle algorithm.

Table 7.1 gives the comparisons of the efficiencies of four algorithms based on the accumulated running times in machine clock ticks (55ms/tick). The testing is done for small radius spans. For each radius span, the radius of the circular arc is changed incrementally from the low end to the high end, and each execution of an algorithm is repeated 3000 times. The ratios of running time of Algorithm HCL to Algorithm RLS-Circle and Hybrid-Circle for each radius span are bracketed. In Table 7.2 the same calculations are repeated for large radius spans. Here the calculations were repeated 300 times.

The figures in the table show the consistent increasing of efficiency of Algorithm RLS-Circle and Hybrid-Circle as compared with Algorithm HCL for the circles of normal sizes. Note that the times listed in Table 1 does not reflect the real running times since no pixels are written. If the the speedups in writing pixels to the frame buffer are considered then the results of the comparisons of Algorithm RLS-Circle

Radius	HCL	RLS-Circle	Hybrid-Circle	Bres
1 to 32	24	13(1.85)	12(2.00)	15
33 to 64	57	32(1.78)	28(2.01)	40
65 to 128	212	124(1.71)	105(2.02)	155

Table 7.1: Runtime comparison of four circle algorithms for small radii.

Radius	HCL	RLS-Circle	Hybrid-Circle	Bres
129 to 256	83	49(1.69)	40(2.18)	62
257 to 384	136	80(1.70)	66(2.06)	101
385 to 512	191	113(1.69)	92(2.08)	142
512 to 1024	1313	767(1.71)	627(2.09)	852

Table 7.2: Runtime comparison of four circle algorithms for large radii.

and Hybrid-Circle with Bresenham's algorithm, should be even better.

7.6 Summary

The run-length properties of a 45° circular arc were discussed. A hybrid approach for scan-conversion of circles was presented based on these properties. The new algorithm Hybrid-Circle combines the run-length slice method and the pixel-based method to yield higher efficiency than both the pixel-based algorithms and the run-length slice algorithms for the scan-conversion of circles. A complexity analysis was performed for the new algorithm Hybrid-Circle. A numerical comparison verifies the theoretical results.

Chapter 8

Final Remarks

This research has focused on scan-conversion of important graphics primitives. This is an area in computer graphics that has a long history and has received considerable research efforts. Our emphasis has been on the further development of some of the ideas presented in previous papers on scan-conversion of lines, circles, ellipses and filled polygons. This research enables a deeper insight into the scan-conversion of these important graphics primitives and results in new and faster scan-conversion algorithms.

Linear interpolation is widely used in computer graphics algorithms. Linear interpolation in a discrete context has been investigated by several researches including the author of this thesis. Our new contributions to linear interpolation in this thesis are the definition and the fast computation of three the types of integral linear interpolation. Rounding-up and rounding down integral linear interpolation are new concepts first investigated by the author of this thesis. The algorithms for the fast computation of three types of integral linear interpolation include double-step, bi-directional algorithms where the method of bi-directional interpolation is also first investigated by the author of this thesis.

The application of the three types of integral linear interpolation to the scan-conversion of lines gives new insight into the the problem of line scan-conversion. The integral linear interpolation approach provides a unified framework to the design of incremental line scan-conversion algorithms, including multi-step algorithms, run-

length slice algorithms, bi-directional algorithms and the combination of the above algorithms. The application of rounding-up integral linear interpolation to the scan-conversion of filled polygons suggests a new method to speed up the existing scan-line algorithms for filled polygons.

The application of run-length slice methodology to the scan-conversion of ellipses is first introduced in this thesis. Run-length slice circle scan-conversion algorithms have been previously presented by other authors. We give a more in-depth investigation of the problem and suggest a hybrid approach which combines run-length slice approach and midpoint approach to further speed up the scan-conversion of circles. The run-length slice circle and ellipse scan-conversion algorithms prove to run faster than previously published circle and ellipse algorithms even though their code is more complex. This is due to the fast computation of run-lengths by using adaptive forward differencing.

We foresee some further research that can be performed based on what we have achieved in this thesis. One topic will be the incorporation of antialiasing to the run-length slice algorithms for the scan-conversion of lines, circles, and ellipses. This will result in fast algorithms for generating antialiased lines, circle, and ellipses. Another possible problem to attack is based on the techniques of run-length computation for digitized circular and elliptical arcs to scan-convert filled circles, ellipses, and rings. The application of run-length slice methodology to the scan-conversion of general curves imposes a more difficult problem. We are not optimistic that more efficient algorithms can be obtained in this case although double-step and /or sophisticated differencing strategies might possibly lead to improvements in the algorithms.

Bibliography

[Bao 89] P. Bao and J. Rokne. Quadruple-step line generation. *Comput. and Graph.*, 13(4):461–469, 1989.

[Bresenham 65] J. E. Bresenham. Algorithm for computer control of digiter plotter. *IBM Syst. J.*, 4(1):25–30, 1965.

[Bresenham 77] J. Bresenham. A linear algorithm for incremental digital display of circular arcs. *Communications of the ACM*, 20(2):100–106, 1977.

[Bresenham 82] J. E. Bresenham. Incremental line compaction. *Comput. J.*, (25):116–120, January 1982.

[Bresenham 85] J. E. Bresenham. Run length slice algorithms for incremental lines. In R. A. Earnshaw, editor, *Fundamental algorithms for computer graphics, NATO Computer and Systems Series, Vol 17*, pages 59–104, New York, 1985. Springer Verlag.

[Brons 74] R. Brons. Linguistic methods for the discription of a straight line on a grid. *Comput. Graph. Image Process.*, 3(1):183–195, 1974.

[Castle 87] C. M. A. Castle and M. L. V. Pitteway. An efficient structural technique for encoding 'best-fit' straight lines. *Comput. J.*, 30(2):168–175, 1987.

[Cederberg 79] R. L. T. Cederberg. A new method for vector generation. *Comput. Graph. Image Process.*, 9(2):183–195, 1979.

[Danielsson 70] P. E. Danielsson. Incremental curve generation. *IEEE Trans. Computers*, 19(9):783–793, September 1970.

[Dorst 85] L. Dorst. The accuracy of the digital representation of a straight line. In R. A. Earnshaw, editor, *Fundamental algorithms for computer graphics, NATO Computer and Systems Series, Vol 17*, pages 141–152, New York, 1985. Springer Verlag.

[Fellner 93] D. W. Fellner. Robust rendering of general ellipses and elliptical arcs. *ACM Trans. on Graphics*, 12(3):251–276, July 1993.

[Field 85] D. Field. Incremental linear interpolation. *ACM Trans. on Graph.*, 4:1–11, January 1985.

[Foley 82] J. D. Foley and A. Van Dam. *Fundamentals of interactive computer graphics.* Addison-Wesley, Reading, Mass., 1982.

[Foley 90] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice.* Addison-Wesley, Reading, Mass., 1990.

[Freeman 61] H. Freeman. On the encoding of arbitrary geometric configurations. *IRE Trans.*, (EC-102):260–268, 1961.

[Freeman 70] H. Freeman. Boundary encoding and processing. In B. S. Lipkin and A. Rosenfeld, editors, *Picture Processing and Psychopictories.*, pages 241–266, New York, 1970. Academic Press.

[Freeman 74] H. Freeman. Computer processing of line-drawing images. *ACM Comput. Surv.*, (6), 1974.

[Fung 92] K. Y. Fung, T. M. Nicholl, and A. K. Dewdney. A run-length slice line drawing algorithm without division operations. *Comput. Graph. Forum*, (3):267–277, 1992.

[Gardner 92] P. L. Gardner. Modification of Bresenham's algorithm for displays. *IBM Technical Disclosure Bulletin*, pages 1595–1596, Oct. 1975.

[Gay 85] A. C. Gay. Experience in practical implementation of boundary-defined area fill. In R. A. Earnshaw, editor, *Fundamental algorithms for computer graphics, NATO Computer and Systems Series, Vol 17*, pages 153–160, New York, 1985. Springer Verlag.

[Gill 94] G. W. Gill. N-step incremental straight-linear algorithms. *IEEE Computer Graphics and Applications*, pages 66–72, May 1994.

[Gouraud 71] H. Gouraud. Continuous shading of curved surfaces. *IEEE Trans. Comput.*, C-20(6):623–629, 1971.

[Gourret 87] J. P. Gourret and J. Paille. Irregular polygon fill using contour encoding. *Computer Graphics Forum*, 6(4):317–325, 1987.

[Graham 93] P. Graham and S. Iyengar. Double-and triple-step incremental generation of lines. In *Proc. 1993 ACM Computer Science Conf.*, New York, 1993.

[Graham 94] P. Graham and S. Iyengar. Double- and triple-step incremental linear interpolation. *IEEE Computer Graphics and Applications*, pages 49–53, May 1994.

[Hobby 90] J. D. Hobby. Rasterization of nonparametric curves. *ACM Trans. on Graphics*, 9(3):262–277, July 1990.

[Horn 76] B. K. P. Horn. Circle generator for display devices. *Computer Graphics and Image Processing*, (5):280–288, 1976.

[Hsu 93] S. Y. Hsu, L. R. Chow, and C. H. Liu. A new approach for the generation of circles. *Computer Graphics Forum*, 12(2):105–109, 1993.

[Kappel 85] M. R. Kappel. An ellipse-drawing algorithm for raster displays. In R. A. Earnshaw, editor, *Fundamental algorithms for computer graphics, NATO Computer and Systems Series, Vol 17*, pages 257–281, New York, 1985. Springer Verlag.

[Knott 79] G. D. Knott. Computing polygon fill-lines. *Comput. & Graphics*, 11(1):21–25, 1979.

[Lane 83] J. M. Lane. Note: an algorithm for filling regions on graphics display devices. *ACM Trans. on Graphics*, 2(3):192–196, 1983.

[McIlroy 83] M. D. McIlroy. Best approximate circles on integer grids. *ACM Trans. on Graphics*, 2(4):237–263, October 1983.

[McIlroy 84] M. D. McIlroy. A note on discrete representation of lines. *AT& T Technical Journal*, 64(2):481–490, February 1984.

[McIlroy 92] M. D. McIlroy. Getting raster ellipse right. *ACM Trans. on Graphics*, 11(3):259–275, July 1992.

[Narayanswam 95] C. Narayanswami. Efficient parallel Gouraud shading and linear interpolation over triangles. *Computer Graphics Forum*, 14(1):17–24, 1995.

[Newman 79] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, New York, 1979.

[Pavlidis 79] T. Pavlidis. Filling algorithms for raster graphics. *Computer Graphics and Image Processing*, 10:126–141, 1979.

[Phong 75] B. T. Phong. Illuminating for computer generated pictures. *Commun. ACM*, (18):311–317, June 1975.

[Pitteway 67] M. L. V. Pitteway. Algorithm for drawing ellipse or hyperbolae with a digital plotter. *The Computer J.*, 10(3):282–289, November 1967.

[Rankin 87] J. R. Rankin. A note on multi-polygon area filling. *Comput. & Graphics*, 11(4):445–447, 1987.

[Reggiori 72] G. B. Reggiori. Digital computer transformations for irregular line drawings. Technical Report 403-22, Department of Electrical Engineering and Computer Science, New York Univ., April 1972.

[Richards 87] J. E. Richards. Filling complex polygons by region-fill methods on raster graphics terminals. *Computer Graphics Forum*, 6:49–54, 1987.

[Rokne 90] J. Rokne, B. Wyvill, and X. Wu. Fast line scan-conversion. *ACM Trans. on Graph.*, 9(4):376–388, October 1990.

[Rokne 92] J. Rokne and C. Yao. Double-step incremental linear interpolation. *ACM Trans. on Graph.*, 11(2):183–192, April 1992.

[Rosenfeld 74] A. Rosenfeld. Digital straight line segments. *IEEE Trans. Comput.*, C-23:1264–1269, 1974.

[S-L. Chang 89] M. Shantz S-L. Chang and R. Rocchetti. Rendering cubic curves and surfaces with integer adaptive forward differencing. *Computer Graphics*, 23(3):157–166, July 1989.

[Sproull 82] R. F. Sproull. Using program transformations to derive line drawing algorithms. *ACM TOG*, 1(4):259–273, October 1982.

[Van Aken 84] J. R. Van Aken. An efficient ellipse-drawing algorithm. *CG&A*, 4(9):24–35, September 1984.

[Van Aken 85] J. R. Van Aken and M. Novak. Curve-drawing algorithms for raster display. *ACM Trans. Graphics*, 4(2):147–169, April 1985.

[Wright 90] W. E. Wright. Parallelization on bresenham's line and circle algorithms. *IEEE Computer Graphics and Applications*, pages 60–67, September 1990.

[Wu 87] X. Wu and J Rokne. Double-step generation of lines and circles. *CVGIP*, (37):331–344, 1987.

[Wu 89] X. Wu and J. G. Rokne. Double-step generation of ellipse. *CG&A*, 9(5):376–388, May 1989.