A Formal Specification of Line Representations on Graphics Devices

Lynn S. Marshall
Department of Computer Science
University of Manchester
Manchester, UK
M13 9PL

Abstract

To show that a computer graphics system functions properly it is necessary to prove that the images it produces are correct. Most graphical devices are unable to exactly represent an image, or even just a straight line. Thus each device must display an approximation to the ideal. This paper presents a formal specification of the properties any reasonable approximation to a straight line should have. Bresenham's algorithm is shown to satisfy this specification and extensions to the specification are discussed.

1 Introduction

Formal specification is a useful tool in many areas of computer science since it allows the aims of a computer system to be clearly and unambiguously expressed, and statements concerning the system to be formally proven [17]. Formal specification of computer graphics systems is in its infancy. Research in this field has been pioneered by Carson [7], Gnatz [14], and Mallgren [19]. Formal specification is of great potential help in computer graphics.

Graphical data is usually in the form of images composed of various drawing primitives such as points, lines, and text. Most graphical devices are unable to represent drawing primitives exactly and thus must produce an approximation to the ideal. This makes the use of conventional program verification tools, such as a test driver, very difficult. Graphical Kernel System (GKS) is the new draft international standard for two-dimensional interactive computer graphics [13]. Designing test suites for GKS implementations is certainly not straightforward [5], and work on formal specification of GKS is underway [8,21].

A formal description of the approximation to an image that a given computer graphics device should display will be useful in proving that the various devices in a computer graphics system function correctly. The idea of specifying what comprises a valid approximation to some ideal picture on a given graphics device has been deliberately ignored in previous research in the formal specification of graphics area. Mallgren [19] says, "the display system is assumed to provide a recognizable approximation to this representative picture," while Carson [7]

admits, "of course, someone must eventually describe how a line should look but we could treat this as a binding issue, not a specification issue." However, it seems meaningless to maintain that a graphics program is functioning correctly unless it produces recognizable output. Carson [7] notes the following:

> "At one extreme, nothing at all is said about the actual effects on a display surface when an output primitive is drawn. This would enable any vendor to claim that almost any implementation conformed to the standard, since it would be impossible to test implementations. At the other extreme, the ... specification could completely describe the effects of output primitives in terms of parameters such as addressability, color, hardware, text fonts, etc. that apply to typical display devices. Unfortunately, any parameter set considered by the specifiers placed unfair restrictions on manufacturers of certain classes of display devices. Furthermore, fixed parameters would inhibit the degree of technological flexibility available to implementors."

Thus, it is necessary to devise a specification that will permit the display of any one of a range of approximations to a picture so allowing any reasonable output, but only reasonable output.

Section 2 of this paper discusses graphics devices and their capabilities, and section 3 describes lines and their attributes. In section 4 a formal specification of thin solid lines is given, while section 5 describes various line drawing algorithms, and section 6 is a proof that Bresenham's line drawing algorithm satisfies the specification. Section 7 suggests some extensions to the specification, and a discussion of ideas for further research and conclusions follow in sections 8 and 9. Appendix A shows a sample line plotted by various line drawing algorithms. A thick line specification can be found in an extended version of this paper [20].

2 Graphics Devices

The two major graphical display device types are the vector device and the raster device [11]. A picture on a vector device is composed of straight line segments, while on a raster device the picture is made up of picture elements, or pixels, at fixed positions. Vector drawing displays and pen plotters are examples of vector devices. Raster devices include raster displays, laser printers and electrostatic plotters [29]. The graphics device model used is that of a raster device since drawing lines on vector devices is simpler.

A graphics device displays images in a number of colours. It may be capable of depicting thousands of colours, a range from black to white, or possibly just two colours (binary). For simplicity the model of a raster device is limited to two colours: a background colour (OFF) and a foreground colour (ON). The display surface is composed of pixels, each one unit square with its centre having integer coordinates. Each pixel on the screen of the device may be either ON or OFF, and the pixels approximating the line are those to which the foreground colour is assigned.

3 Lines

A straight line to be displayed on a graphical device usually has a number of associated parameters. It must have a start point, an end point, a width, a linetype, and a colour. The line can be any length, have any slope, be thick or thin, and solid or broken. Since the pixels of the raster device lie in a grid formation, the device must produce an approximation to the line to be displayed. Thus, the representation of a line on a raster device is non-trivial. The following specification is for thin solid lines having integral endpoints.

4 Straight Solid Thin Lines with Integral Endpoints on a Two-Colour Raster Device

What properties should the approximation to a line on a raster device have? As stated earlier, the properties given should be specific enough to allow only reasonable approximations but general enough to allow any reasonable approximation. Thus it is inappropriate to specify an exact algorithm since a range of approximations is permitted. Neither is it appropriate for the representation to be entirely implementation dependent as the role of the specification is to limit the implementor.

4.1 Properties

The following are some intuitive ideas concerning the approximation to a straight line on a raster device:

1. If a pixel is ON it must be "close" to the line.
   (i.e., No pixels that are very far from the line should be ON.)
2. If a pixel is "very close" to the line it must be ON.
   (i.e., No pixels very close to the line should be OFF.)
3. If two pixels are adjacent, on the same side of the line and the further of the two from the line is ON then the closer of the two to the line must also be ON.
4. The pixels which are ON form a connected region with no holes or bends.

4.2 Notation

The notation used is adapted from the Vienna Development Method (VDM) [18].

4.2.1 Data Types and Instances

A data type is defined in one of three ways. It may be a basic data type a, a composite data type, or a data type having multiple components. A basic data type is defined as follows:

$$\text{Data-type}_1 = \text{English description of the data type}$$

A composite data type is a collection of another data type. For example:

$$\text{Data-type}_2 = \textbf{set of } \text{Data-type}_1$$

If the data type has multiple components it is defined as follows:

$Data\text{-}type_3$ ::
    $PART_1$: $Data\text{-}type_1$
    $PART_2$: $Data\text{-}type_2$
    :
    $PART_n$: $Data\text{-}type_n$

A data type name is made up of the underscore (_), alphabetic, and numeric characters. The name normally begins with a capital, with the remainder being lower case. Special names using bold letters and special symbols may be used for the basic data types (e.g. $\mathbf{R}^*$ = reals ≥ 0). The components of a data type are given names in upper case, possibly with a subscript. If there are any special restrictions to a data type these are expressed using the **require** keyword. For example:

$Data\text{-}type_4$: **set of** $Data\text{-}type_1$ [**require set of** $Data\text{-}type_1 \neq \{\}$]

4.2.2 Function Definitions

A function is defined as follows:

Function : $Data\text{-}type_1 \times Data\text{-}type_2 \times \ldots \times Data\text{-}type_n \rightarrow Data\text{-}type$
Function($d_1, d_2, \ldots, d_n$) ≙ mathematical function description

The name of the function is the word preceding the colon (:). The function name usually starts with a letter but may begin with a greek character. The input data type is described as the Cartesian product followed by the type of the function. The function is then described in postfix format, using instances of the input data types. Infix format is used for some two-parameter functions. The only unusual constructs used in the function descriptions are the **require** and **where** keywords; **require** is used to restrict the input to a function, similar to restricting a data type as described above, and **where** is used to allow an abbreviation to be used in the function description. For example:

Setfunc : **set of** $\mathbf{R} \rightarrow \mathbf{R}$ [**require set of** $\mathbf{R} \neq \{\}$]
Setfunc(elements) ≙ ... N ...
    where N = Max({ elements })

4.3 Specification

The above mentioned concepts are now formalized.

4.3.1 Data Types

A line on the screen with integral endpoints:

Line :: [**require** $P_1 \neq P_2$]
  P_1 : Pixel
  P_2 : Pixel

A pixel on the screen:

Pixel ::
  X : $\mathbf{Z}_X$
  Y : $\mathbf{Z}_Y$

$\mathbf{Z}_X$: integral x-range of screen $\subset \mathbf{Z}$
$\mathbf{Z}_Y$: integral y-range of screen $\subset \mathbf{Z}$

The set of pixels turned on when approximating a line:
Pixel-set : set of Pixel

A line $\in R^2$*:*
Realline :: [require $P_1 \neq P_2$]
P_1 : Point
P_2 : Point

A point $\in R^2$*:*
Point ::
X : R
Y : R

R : reals
R^* : reals ≥ 0
R^2 : Cartesian plane
Z : integers
B : Booleans

Note that Pixel is treated as a subset of Point. Thus any function accepting a Point as a parameter will also accept a Pixel (but not vice versa).

4.3.2 Make-Functions

Make-functions are used to form instances of a multiple component data type. A compound type written in the form (x, y) is always assumed to be of type Point (or Pixel). To form an instance of any other compound data type a make-function is used. Assume we have a type as follows:

Data-type ::
D_1 : $Type_1$
D_2 : $Type_2$
:
D_n : $Type_n$

Then to form an instance of this data type the following function would be used:

mk-Data-type : $Type_1 \times Type_2 \times \ldots \times Type_n \rightarrow$ Data-type
mk-Data-type($d_1, d_2, \ldots, d_n$) $\triangleq$ ι d $\in$ Data-type.
($D_1(d) = d_1 \wedge D_2(d) = d_2 \wedge \ldots \wedge D_n(d) = d_n$)

4.3.3 Point Operations

Addition:
$+_p$: Point $\times$ Point $\rightarrow$ Point
$p_1 +_p p_2 \triangleq$ ($X(p_1) + X(p_2)$, $Y(p_1) + Y(p_2)$)

Subtraction:
$-_p$: Point $\times$ Point $\rightarrow$ Point
$p_1 -_p p_2 \triangleq$ ($X(p_1) - X(p_2)$, $Y(p_1) - Y(p_2)$)

Multiplication:
$\cdot_p$: R $\times$ Point $\rightarrow$ Point
$c \cdot_p p \triangleq$ ($c \cdot X(p)$, $c \cdot Y(p)$)

$\times_p$: Point $\times$ Point $\rightarrow$ Point
$p_1 \times_p p_2 \triangleq$ ($X(p_1) \cdot X(p_2)$, $Y(p_1) \cdot Y(p_2)$)

Division:

$/_p$: Point × Point → Point
$p_1 /_p p_2 \triangleq (X(p_1) / X(p_2),\ Y(p_1) / Y(p_2))$

Less Than:

$<_p$: Point × Point → B
$p_1 <_p p_2 \triangleq (X(p_1) < X(p_2)) \wedge (Y(p_1) < Y(p_2))$

Less Than or Equal to:

$\leq_p$: Point × Point → B
$p_1 \leq_p p_2 \triangleq (X(p_1) \leq X(p_2)) \wedge (Y(p_1) \leq Y(p_2))$

Summation:

$\sum_{i=1}^{n}{}_p$: Point × Point × ... × Point → Point

$\sum_{i=1}^{n}{}_p\ p_i \triangleq (\sum_{i=1}^{n} X(p_i),\ \sum_{i=1}^{n} Y(p_i))$

4.3.4 Line Operation

Equality:

$=_l$: Realline × Realline → B
$l_1 =_l l_2 \triangleq (P_1(l_1) = P_1(l_2) \wedge P_2(l_1) = P_2(l_2)) \vee$
$(P_1(l_1) = P_2(l_2) \wedge P_2(l_1) = P_1(l_2))$

4.3.5 Function Specification

Is the approximation to the given line valid and within a tolerance of δ?

Validapprox : Pixel-set × Line × R^* → B
Validapprox(pixset, line, δ) ≙
(∀ pix ∈ pixset. Withintol(pix, line, δ))
if a pixel is ON it is "close" to the line
∧ (∀ pix ∈ Pixel. (Nearline(pix, line) ⇒ pix ∈ pixset))
if a pixel is "very near" the line it is ON
∧ Closrptson(pixset, line)
any pixel closer to the line than a pixel that is ON is ON
∧ Validpic(pixset)
the pixel formation is valid

Is the pixel within the given tolerance of the line?

Withintol : Pixel × Line × R^* → B
Withintol(pix, line, δ) ≙ ∃ p ∈ Point.
(Onlineseg(p, line) ∧ Maxdist(pix, p) ≤ δ)

Is the point on the line segment?

Onlineseg : Point × Line → B
Onlineseg(p, line) ≙ ∃ δ ∈ [0,1]. (p = P_1(line) $+_p$ (δ $\cdot_p$ Δ(line)))

What is the difference between the endpoints of the line?

Δ : Line → Point
Δ(line) ≙ P_2(line) $-_p$ P_1(line)

What is the maximum horizontal or vertical distance between the two points?

Maxdist : Point × Point → R^*
Maxdist(p_1, p_2) ≙ Max({ |X(p_1) − X(p_2)|, |Y(p_1) − Y(p_2)| })

What is the maximum of the set?

```
Max : set of R → R  [ require set of R ≠ {} ]
Max( s ) ≙  ι a ∈ s. ( ∀ b ∈ s. ( a ≥ b ) )
```

Is the pixel very close to the line?

```
Nearline : Pixel × Line → B
Nearline( pix, line ) ≙ Endpt( pix, line ) ∨ Linethru( pix, line )
```

Is the pixel an endpoint of the line?

```
Endpt : Pixel × Line → B
Endpt( pix, line ) ≙ pix = P1(line) ∨ pix = P2(line)
```

Does the line run right through the pixel?

```
Linethru : Pixel × Line → B
Linethru( pix, line ) ≙ ∃ p1, p2 ∈ Point.
   ( Onlineseg( p1, line ) ∧ Onlineseg( p2, line )
   ∧ ~Adjcorn( p1, p2, pix ) ∧
   ∧ ( ( Onreallineseg( p1, Leftbord( pix ) )
      ∧ Onreallineseg( p2, Rightbord( pix ) ) )
   ∨ ( Onreallineseg( p1, Botbord( pix ) )
      ∧ Onreallineseg( p2, Topbord( pix ) ) ) )
```

Are the two points adjacent corners of the pixel?

```
Adjcorn : Point × Point × Pixel → B
Adjcorn( p1, p2, pix ) ≙ p1 ≠ p2 ∧ let rline = mk-Realline( p1, p2 ) in
   ( rline =1 Leftbord( pix ) ∨ rline =1 Rightbord( pix )
   ∨ rline =1 Botbord( pix ) ∨ rline =1 Topbord( pix ) )
```

What is the left border of the pixel?

```
Leftbord : Pixel → Realline
Leftbord( pix ) ≙ mk-Realline( pix +p (-1/2,-1/2), pix +p (-1/2,1/2) )
```

What is the right border of the pixel?

```
Rightbord : Pixel → Realline
Rightbord( pix ) ≙ mk-Realline( pix +p (1/2,-1/2), pix +p (1/2,1/2) )
```

What is the bottom border of the pixel?

```
Botbord : Pixel → Realline
Botbord( pix ) ≙ mk-Realline( pix +p (-1/2,-1/2), pix +p (1/2,-1/2) )
```

What is the top border of the pixel?

```
Topbord : Pixel → Realline
Topbord( pix ) ≙ mk-Realline( pix +p (-1/2,1/2), pix +p (1/2,1/2) )
```

Is the point on the given real line segment?

```
Onreallineseg : Point × Realline → B
Onreallineseg( p, rline ) ≙ ∃ δ ∈ [0,1]. p = P1(rline) +p ( δ ·p Δr( rline ) )
```

What is the difference between the endpoints of the real line?

```
Δr : Realline → Point
Δr( rline ) ≙ P2(rline) -p P1(rline)
```

Are all pixels closer to the line than an ON pixel ON?

```
Closrptson : Pixel-set × Line → B
Closrptson( pixset, line ) ≙ ∀ pix1, pix2 ∈ Pixel.
   ( ( Adjacent( pix1, pix2 ) ∧ ~Oppsides( pix1, pix2, line )
   ∧ Closrl( pix1, pix2, line ) ∧ pix2 ∈ pixset ) ⇒ pix1 ∈ pixset )
```

Are the two pixels adjacent?

```
Adjacent : Pixel × Pixel → B
Adjacent( pix1, pix2 ) ≙ ( Mindist( pix1, pix2 ) = 0 )
```

What is the minimum horizontal or vertical distance between the two points?

Mindist : Point × Point → R^*

Mindist(p_1, p_2) ≙ Min({ |X(p_1) − X(p_2)|, |Y(p_1) − Y(p_2)| })

What is the minimum of the set?

Min : **set of R** → R [**require set of R** ≠ {}]

Min(s) ≙ ι a ∈ s. (∀ b ∈ s. (a ≤ b))

Are the pixels on opposite sides of the line?

Oppsides : Pixel × Pixel × Line → B

Oppsides(pix_1, pix_2, line) ≙ pix_1 ≠ pix_2 ∧ ∃ p ∈ Point.
(Inlineseg(p, mk-Line(pix_1, pix_2)) ∧ Online(p, line))

Is the point a non-endpoint of the line segment?

Inlineseg : Point × Line → B

Inlineseg(p, line) ≙ ∃ δ ∈ (0,1). (p = P_1(line) $+_p$ (δ $\cdot_p$ Δ(line)))

Is the point on the line?

Online : Point × Line → B

Online(p, line) ≙ ∃ δ ∈ R. (p = P_1(line) $+_p$ (δ $\cdot_p$ Δ(line)))

Is the first pixel closer to the line than the second?

Closrl : Pixel × Pixel × Line → B

Closrl(pix_1, pix_2, line) ≙ ∃ δ ∈ R^*.
(Withintol(pix_1, line, δ) ∧ ~Withintol(pix_2, line, δ))

Is the pixel formation valid?

Validpic : Pixel-set → B

Validpic(pixset) ≙ Validrows(pixset) ∧ Validcols(pixset)

Are the rows of the display valid? (i.e. Do only rows in a continuous range contain ON pixels and is each of these rows valid?)

Validrows : Pixel-set → B

Validrows(pixset) ≙ ∃ y_1, y_2 ∈ Z_Y. (y_1 ≤ y_2 ∧
∀ y ∈ (Z_Y − { y_1,...,y_2 }). (∀ x ∈ Z_X. (x,y) ∉ pixset)
∧ ∀ y ∈ { y_1,...,y_2 }. Validrow(pixset, y))

Is this row of the display valid? (i.e. Does this row have only one continuous range of pixels ON?)

Validrow : Pixel-set × Z_Y → B

Validrow(On, y) ≙ ∃ x_1, x_2 ∈ Z_X. (x_1 ≤ x_2 ∧
∀ x ∈ (Z_X − { x_1,...,x_2 }). (x,y) ∉ pixset ∧
∀ x ∈ { x_1,...,x_2 }. (x,y) ∈ pixset)

Are the columns of the display valid? (i.e. Do only columns in a continuous range contain ON pixels and is each of these columns valid?)

Validcols : Pixel-set → B

Validcols(pixset) ≙ ∃ x_1, x_2 ∈ Z_X. (x_1 ≤ x_2 ∧
∀ x ∈ (Z_X − { x_1,...,x_2 }). (∀ y ∈ Z_Y. (x,y) ∉ pixset)
∧ ∀ x ∈ { x_1,...,x_2 }. Validcol(pixset, x))

Is this column of the display valid? (i.e. Does this column have only one continuous range of pixels ON?)

Validcol : Pixel-set × Z_X → B

Validcol(pixset, x) ≙ ∃ y_1, y_2 ∈ Z_Y. (y_1 ≤ y_2 ∧
∀ y ∈ (Z_Y − { y_1,...,y_2 }). (x,y) ∉ pixset ∧
∀ y ∈ { y_1,...,y_2 }. (x,y) ∈ pixset)

5 Thin Line Drawing Algorithms

If the specification is reasonable any of the common line drawing algorithms will satisfy it. Also, it should be easily extendable. A summary of line drawing algorithm references has been compiled by Earnshaw [9]. Sproull also discusses line drawing algorithms [27]. An outline of a variety of thin line drawing algorithms follows. Each of these algorithms satisfies the above specification. The pixel set for each algorithm and the appropriate tolerance are given.

5.1 Bresenham's, Simple Digital Differential Analysis (DDA), and Chain Code Algorithms

For any given line these three algorithms produce the same approximation by sampling the line once per row or column and turning on the closest pixel to the sampled point. Whether the line is sampled by row or by column is based on the slope of the line and selected so that the maximum number of points will be sampled. The Simple DDA algorithm [23], is the most straightforward. Bresenham's algorithm [4] is optimized to use only integer arithmetic, and the Chain Code algorithm [26] stores the resulting line as a series of integers modulo 7, representing the eight different directions one can proceed to an adjacent pixel.

The line is related to the pixel set by:

$$\begin{aligned}&\text{pix} \in \text{pixset} \;\#\\ &\quad \exists\, n \in \{0,\ldots,N\}.\ (\ \text{pix} = P_1(\text{line}) +_p \text{Round}_p(\ n/N \cdot_p \Delta(\ \text{line}\ )\ )\ )\\ &\quad \textbf{where } N = \text{Maxdist}(\ P_1(\text{line}),\ P_2(\text{line})\ )\end{aligned}$$

$$\begin{aligned}&\text{Round}_p : \text{Point} \to \text{Pixel}\\ &\text{Round}_p(\ p\ ) \triangleq (\ \text{Round}(\ X(p)\ ),\ \text{Round}(\ Y(p)\ )\ )\end{aligned}$$

$$\begin{aligned}&\text{Round} : \mathbf{R} \to \mathbf{Z}\\ &\text{Round}(\ r\ ) \triangleq \iota\, i \in \mathbf{Z}.\ (\ r - 1/2 < i \leq r + 1/2\ )\end{aligned}$$

These algorithms always turn on pixels which the line at least touches, and thus have a tolerance of 1/2.

5.2 Symmetric DDA Algorithm

The Symmetric DDA algorithm [23] is similar to the Simple DDA algorithm, but samples the line more frequently. The length of the line determines the number of times the line is sampled. To make the notation simpler the following abbreviations are used:

Δx for $X(\Delta(\ \text{line}\ ))$, and

Δy for $Y(\Delta(\ \text{line}\ ))$.

The length of the line is usually approximated by:

$$\text{Max}(\ \{\ |\Delta x|,\ |\Delta y|\ \}\ ) + 1/2 \cdot \text{Min}(\ \{\ |\Delta x|,\ |\Delta y|\ \}\ )$$

since $\sqrt{(\ \Delta x^2 + \Delta y^2\ )}$ is expensive to compute. Also, for efficiency reasons, the number of steps is chosen to be a power of two. Thus the number of sampled points is $2^n + 1$, where n is the smallest n such that $2^n > \text{Max}(\ \{\ |\Delta x|,\ |\Delta y|\ \}\ ) + 1/2 \cdot$

Min({ |Δx|, |Δy| }). The Symmetric DDA algorithm gives a more equal density to approximations to lines of different slopes than the Simple DDA.

The line is related to the pixel set by:

$$\begin{aligned}&\text{pix} \in \text{pixset} \Leftrightarrow \\ &\quad \exists\, n \in \{0,\ldots,N\}.\ (\ \text{pix} = P_1(\text{line}) +_p \text{Round}_p(\ n/N \cdot_p \Delta(\ \text{line}\ )\ )\ ) \\ &\quad \textbf{where } N = \text{Minvalidn}(\ \text{line}\ )\end{aligned}$$

$$\begin{aligned}&\text{Minvalidn} : \text{Line} \to \mathbb{N} \\ &\text{Minvalidn}(\ \text{line}\ ) \triangleq n \in \mathbb{N}.\ (\ \text{Validn}(\ n, \text{line}\ ) \wedge \\ &\quad \forall\, m \in \mathbb{N}.\ (\ \text{Validn}(\ m, \text{line}\ ) \Rightarrow n \leq m\ )\ )\end{aligned}$$

$$\begin{aligned}&\text{Validn} : \mathbb{N} \times \text{Line} \to \mathbb{B} \\ &\text{Validn}(\ n, \text{line}\ ) \triangleq \exists\, k \in \mathbb{N}.\ (\ n = 2^k\ ) \wedge n > \\ &\quad \text{Maxdist}(\ P_1(\text{line}), P_2(\text{line})\ ) + 1/2 \cdot \text{Mindist}(\ P_1(\text{line}), P_2(\text{line})\ )\end{aligned}$$

The symmetric DDA algorithm always turns on pixels touched by the line and thus has a tolerance of 1/2.

5.3 All Pixels Touched Algorithm

It is easy, theoretically, to imagine a line drawing algorithm which samples the line "everywhere" thus turning on all pixels touched by the line. Of course, this could only be implemented approximately and would be inefficient.

The line is related to the pixel set by:

$$\begin{aligned}&\text{pix} \in \text{pixset} \Leftrightarrow \\ &\quad \exists\, p \in \text{Point}.\ (\ \text{Onlineseg}(\ p, \text{line}\ ) \wedge \text{pix} = \text{Round}_p(\ p\ )\ )\end{aligned}$$

This algorithm also has a tolerance of 1/2.

5.4 Brons' Chain Code Algorithm

The Chain Code algorithm presented by Brons [6] produces a line similar, but not identical to the Chain Code algorithm discussed earlier. The chain code is produced in a recursive manner, giving successive approximations to the line until the "best" approximation is achieved.

It has not been possible to find a simple non-recursive description of this algorithm! Brons' Chain Code algorithm is often identical to the standard Chain Code Algorithm. However, in cases with $|\Delta x| = n$, and $|\Delta y| = 1$, it gives approximations with a tolerance approaching 1.

5.5 Binary Rate Multiplier (BRM) Algorithm

The BRM Algorithm [22] was once a popular line drawing algorithm due to its speed. However, it tends to produce rather inaccurate approximations and, with the advent of more accurate quick algorithms, it is rarely used. It is based on binary arithmetic. Both $|\Delta x|$ and $|\Delta y|$ are expressed in binary notation using n bits. The point (x_1,y_1) is turned on and a binary clock then counts from 0 to $2^n - 1$. At each stage, x is incremented if and only if the bit changing from 0 to 1 in the counter is 1 in the binary representation of $|\Delta x|$. The same applies to y.

The line is related to the pixel set by:

$$\begin{aligned}
&\mathrm{pix} \in \mathrm{pixset} \Leftrightarrow \\
&\quad \exists\, d \in \{0,\ldots,2^n - 1\}.\ \mathrm{pix} = P_1(\mathrm{line}) +_p \\
&\quad \mathrm{Sign}_p(\ \mathrm{line}\ ) \times_p \sum_{i=1}^{n}{}_p(\ \mathrm{Round}_p(\ d \,/\, 2^{n+1-i} \cdot_p c_i\ )\ ) \\
&\quad \textbf{where } n = \mathrm{Minvalidn}(\ \mathrm{line}\ ), \\
&\qquad \forall\, i \in \{1,\ldots,n\}.\ c_i \in \{(0,0),(0,1),(1,0),(1,1)\}. \\
&\qquad (\ |\ \Delta(\ \mathrm{line}\ )\ | = \sum_{i=1}^{n}{}_p\ 2^{i-1} \cdot_p c_i\ )
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Validn} : \mathbb{N} \times \mathrm{Line} \to \mathbb{B} \\
&\mathrm{Validn}(\ n,\ \mathrm{line}\ ) \triangleq 2^n > \mathrm{Maxdist}(\ P_1(\mathrm{line}),\ P_2(\mathrm{line})\ )
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Sign}_p : \mathrm{Line} \to \mathrm{Point} \\
&\mathrm{Sign}_p(\ \mathrm{line}\ ) \triangleq (\ \mathrm{Sign}(\ X(\Delta(\ \mathrm{line}\ ))),\ \mathrm{Sign}(\ Y(\Delta(\ \mathrm{line}\ ))\ )
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Sign} : \mathbb{Z} \to \mathbb{Z} \\
&\mathrm{Sign}(\ a\ ) \triangleq \textbf{if } a = |a| \textbf{ then } 1 \textbf{ else } -1
\end{aligned}$$

The BRM algorithm can be very inaccurate, especially for lines with $|\Delta x|$ equal to the reflection of $|\Delta y|$, in binary notation. The tolerance for this algorithm is approximately 2.

See appendix A for a sample line and the approximations produced by these line drawing algorithms.

6 Proof for Bresenham's Algorithm

Throughout this section the following abbreviations are used:

P_1	for P_1(line)
X_1	for $X(P_1(\mathrm{line}))$
Y_1	for $Y(P_1(\mathrm{line}))$
Δ	for Δ(line)
Δx	for $X(\Delta(\ \mathrm{line}\ ))$
Δy	for $Y(\Delta(\ \mathrm{line}\ ))$
R	for Round
R_p	for Round_p

6.1 ∀ pix ∈ pixset. Withintol(pix, line, 1/2)

$$\mathrm{pix} \in \mathrm{pixset} \Leftrightarrow \mathrm{pix} = P_1 +_p R_p(\ n/N \cdot_p \Delta\ ).$$

Now, $p = P_1 +_p (\ n/N \cdot_p \Delta\ )$ is on the line segment, since $0 \leq n/N \leq 1$. And, either $\Delta x/N$, or $\Delta y/N$ is an integer, as $N = |\Delta x|$ or $|\Delta y|$. Thus, Maxdist(pix, p) = $|R(\ X(p)\ ) - X(p)|$ or $|R(\ Y(p)\ ) - Y(p)|$, and so Maxdist(pix, p) $\leq 1/2$, and the ON pixel is within 1/2 of the line as desired.

6.2 ∀ pix ∈ Pixel. (Nearline(pix, line) ⇒ pix ∈ pixset)

Nearline(pix, line) ⇔ Endpt(pix, line) ∨ Linethru(pix, line).

Now if the pixel is an endpoint of the line, it will be ON (cases n = 0 and n = N). If the line runs right through the pixel, there are two cases:

Case 1: If N = |Δx| then the line runs through the pixel in a horizontal direction, and we have that the point (X(pix), Y(pix) + δ), for δ ∈ (-1/2,1/2), is on the line. Since N = |Δx| this column will be sampled, and this pixel will be turned ON since R(Y(pix) + δ) = Y(pix) + R(δ) = Y(pix).

Case 2: On the other hand, if N = |Δy| then the line runs through the pixel in a vertical direction, and the point (X(pix) + δ, Y(pix)), for δ ∈ (-1/2,1/2), is on the line. This row will be sampled, and since R(X(pix) + δ) = X(pix), this pixel will be turned ON.

6.3 Closrptson(pixset, line)

Closrptson(pixset, line) ⇔ ∄ pix_1, pix_2 ∈ Pixel.
(Adjacent(pix_1, pix_2) ∧ ~Oppsides(pix_1, pix_2, line) ∧
Closrl(pix_1, pix_2, line) ∧ pix_2 ∈ pixset ∧ pix_1 ∉ pixset).

Case 1: N = |Δx|, pix_1 and pix_2 are horizontally adjacent
Without loss of generality, assume Δx is positive. Then since pix_2 is ON, X(pix_2) = X_1 + n and Y(pix_2) = Y_1 + R(n·Δy/Δx). And thus, X(pix_1) = X_1 + n + 1 and Y(pix_1) = Y_1 + R(n·Δy/Δx). Now, pix_1 and pix_2 are on the same side of the line, and pix_1 is closer to the line than pix_2. So, the line must cross the line x = X(pix_1) between Y(pix_1) and Y(pix_1) - 1/2, or Y(pix_1) and Y(pix_1) - 1/2. Thus R((n+1)·Δy/Δx) = R(n·Δy/Δx) and thus pix_1 will be ON. Thus it is true that no such pix_1 and pix_2 exist, and the above is satisfied.

Case 2: N = |Δx|, pix_1 and pix_2 are vertically adjacent
Since pix_2 is ON, it is within 1/2 of the line. However, pix_1 is closer to the line so the points must be on opposite sides of the line, so again the above is satisfied.

Case 3: N = |Δy|, pix_1 and pix_2 are horizontally adjacent
Similar to case 2.

Case 4: N = |Δy|, pix_1 and pix_2 are vertically adjacent
Similar to case 1.

6.4 Validpic(pixset)

Validpic(pixset) ⇔ Validrows(pixset) ∧ Validcols(pixset). Bresenham's algorithm only turns on pixels in rows and columns between p_1 and p_2, and it turns on at least one pixel in each of these, due to the choice of N. Thus, it is necessary only to check that each of these rows and columns is valid.

Case 1: N = |Δx|
Only one pixel will be turned on in each column, so the columns are valid. Assume we have an invalid row, ie. two pixels in a row are ON, but one in between them is

off. So $\exists$ n,m $\in$ **N**. $p_1 = P_1 +_p R_p(\ n/|\Delta x| \cdot_p \Delta line\ )$, and $p_2 = P_1 +_p R_p(\ (n+m)/|\Delta x| \cdot_p \Delta line\ )$. Since p_1 and p_2 are in the same row $R(\ n \cdot \Delta y/|\Delta x|\ ) = R(\ (n+m) \cdot \Delta y/|\Delta x|\ )$, and thus $R(\ (n+i) \cdot \Delta y/|\Delta x|\ )$, for $i \in \{0,\ldots,m\} = R(\ n \cdot \Delta y/|\Delta x|\ )$. So all pixels in the row between p_1 and p_2 will be ON, and the row must be valid.

Case 2: $N = |\Delta y|$
The argument is the same as in Case 1, with the roles of the rows and columns reversed.

Thus Bresenham's algorithm satisfies the thin solid line specification.

7 Extensions to the Specification

7.1 Vector Devices

Although the drawing primitive on a vector device is a line, a vector device is still not able to reproduce all lines exactly. The limitation is the addressing resolution of the device. Thus, if the pixel size is set equal to the resolution of the vector device the model presented will also be appropriate for vector devices. There may be some parts of the specification that are redundant for a vector device. For example, Closrptson should always be true. But the specification will still suffice.

7.2 Lines With Non-Integral Endpoints

The specification can easily be changed to allow for lines with non-integral endpoints by using Realline everywhere instead of Line. It might be desirable to impose an additional condition on Validapprox to ensure that the pixels containing the endpoints are turned on under certain conditions, but this is probably unnecessary.

7.3 Thick Lines

It is quite easy to extend the thin solid line specification to one for solid lines of thickness "t." One question that arises is how the endpoints of the thick line should be treated, as both round-end and square-end models for thick lines exist. Another requirement that should be added to the specification is that any pixel entirely covered by the thick line should be on.

A specification including these extensions is included in the extended version of this paper [20].

8 Ideas for Further Research

8.1 Related Research

Although none of the recent formal specification of computer graphics systems research has discussed the properties of the approximation to a line on a graphics

device, work was carried out in the 1960's and 1970's concerning the representation of solid thin lines on raster or incremental plotter devices [1,2,12,25]. The model used to describe a line is to number the eight pixels adjacent to a given pixel from 0 to 7 in a counter-clockwise direction starting with the pixel on the right. An approximation to a thin line, called the chain code, is then given by a sequence of numbers indicating the direction to proceed from each pixel of the approximation.

Freeman [12] notes:

> All chains of straight lines must possess the following three specific properties:

1. the code is made up of at most 2 elements differing by 1 modulo 8
2. one of the two elements always appears singly
3. the occurrences of the singly occuring element are as uniformly spaced as possible

Rosenfeld [25] proves that the above is satisfied if and only if the chain code has the chord property. That is, if and only if for every point, p, of a line segment between two pixels which are ON, there is an ON pixel, pix, such that Maxdist(p, pix) < 1. No extensions are given for thick lines.

While this area has been ignored for some time, raster displays and operations on them are again being researched. Guibas and Stolfi [15] explain that it has been believed that "the graphics programmer should be spared the pain" of dealing with raster images, but it is now being realized that raster images "should be given full citizenship in the world of computer science." They discuss a function, LINE[p_1, p_2, w], which draws a line of thickness w from p_1 to p_2, but note that, "the exact definition of this shape, particularly at the two endpoints, is ... application-dependent."

8.2 Alternate Approaches

The work presented is all based on the model introduced in Section 2. If a different model to that of the square pixel is used new insight into the properties of output primitives on graphics devices might be obtained. One idea is to look at different tesselations of the Cartesian plane. What would the specification look like if hexagonal pixels, for example, were used? The concepts of rows, columns and adjacent pixels would need to be examined.

Another approach might involve the splitting of the specification into two parts: the local and global properties of the line. Local and global properties are discussed by Guibas and Stolfi [15]. A local property is one that can be checked for each pixel or small piece of the approximation. Such as:

> If a pixel is ON it is "close" to the line.

On the other hand, a global property is one requiring the entire approximation to be considered as a whole. For example:

> The line "looks" straight.

Examining the specification in this way may present new ideas.

The choice of distance function can also influence the specification. Although the maximum horizontal or vertical distance between two points conforms to the square pixel model, the Euclidean distance function is introduced when thick lines are discussed [20]. A different choice of distance function may simplify the specification or suggest a new model.

8.3 Further Properties of Solid Straight Lines

There are many additional properties of a solid straight line that could supplement or replace some of those given in the specification. It is desirable to come up with a simple specification and, at the same time, keep it both specific and general enough to encompass all reasonable approximations. One property the approximation should have is that the line "looks straight." This idea is incorporated in the Validpic portion of the specification. However, perhaps a better formulation of this notion can be given. For example, for a device with a very high precision, it may not be necessary to require that there are no "holes" in the approximation, as a small hole would be undetectable.

Other properties which are desirable in line drawing algorithms are:

1. A line produced has constant density.
2. All lines produced have the same density.
3. The line from p_1 to p_2 is identical to the line from p_2 to p_1.

However, these properties are not possessed by some of the commonly used algorithms. A line produced by the BRM algorithm may not be of constant density. For Bresenham's algorithm, the density of the line depends on its slope [11], and, unless the algorithm is adjusted slightly [3], lines drawn in opposite directions may differ. It may be desirable to try to incorporate relaxations of these conditions into the specification. For example:

1. A line produced has "nearly" constant density.
2. All lines produced have "approximately" the same density.
3. The line from p_1 to p_2 is "close to" the line from p_2 to p_1.

8.4 Further Extensions to the Specification

It would be interesting to give a specification for dashed lines. Dashed lines are usually defined as sections of ink and space [28]. One approach would be to split the line up into a collection of short lines, each specified as a solid line. However, as the point within the ink-space pattern to start with may be implementation dependent, this becomes quite complicated.

Another extension would include the specification of grey-scale lines on a grey-scale or multicolour device. In a grey-scale algorithm [24], each pixel is set to an appropriate shade depending on the portion of it covered by the line. Anti-aliasing [16] is even more complicated as a filtering pattern is used, along

with a selection of colours, to smooth the edges of the line preventing them from appearing to be jagged.

Once the specification of a line on a graphics device is complete there are many other drawing primitives to consider, including marker, filled area, and text. And since a picture is rarely composed of a single primitive it is necessary to look at all the primitives within a picture, and decide how to deal with those that overlap, especially on a device with many colours. This problem is discussed by Carson [7], Fiume and Fournier [10], and Mallgren [19]. These so called combining functions should be specified in a formal description of the properties of a graphics device, thus giving an allowable range for the appearance of the final picture, as well as for each primitive within the picture.

Another area for research is the formal specification of the behaviour of graphics input devices.

9 Conclusions

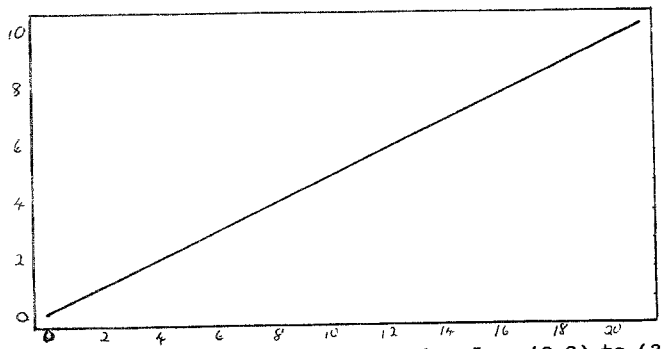
When a new graphics device is produced, it is necessary to be certain that it functions correctly. Although the formal specification presented here is only the tip of the iceberg with regards to the specification of a complete graphics device, it is encouraging that such specifications can be produced, and actually used to prove that algorithms for drawing graphical primitives produce reasonable results.

Acknowlegements

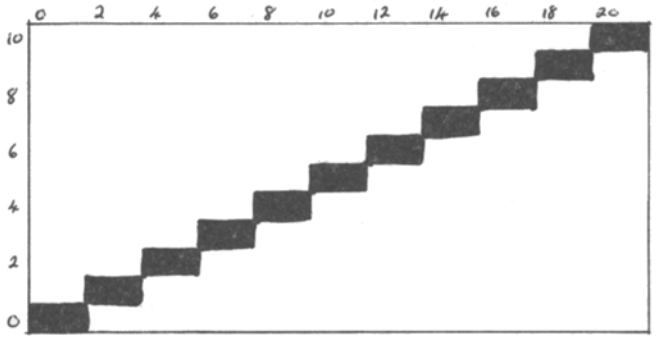
Many thanks to Professor Cliff Jones for his assistance and encouragement, and to Steve Carson, David Duce, Elizabeth Fielding, and my colleagues at the University of Manchester for their valuable ideas and suggestions.

Appendix A - Sample Line

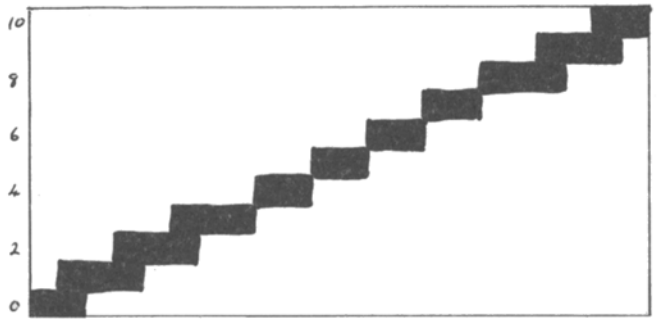
The following diagrams show how the various thin line drawing algorithms discussed in section 5 approximate the line from (0,0) to (21,10). This line was chosen as it illustrates the differences between the line drawing algorithms.



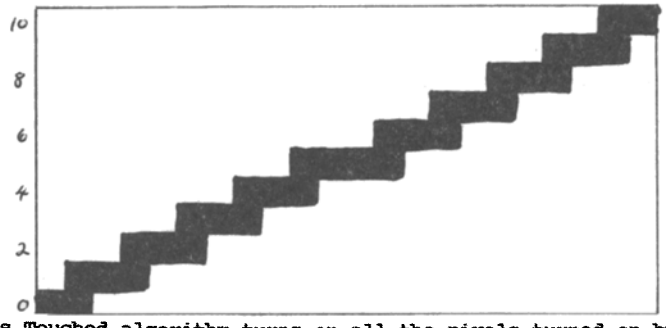
The line to be approximated, running from (0,0) to (21,10).



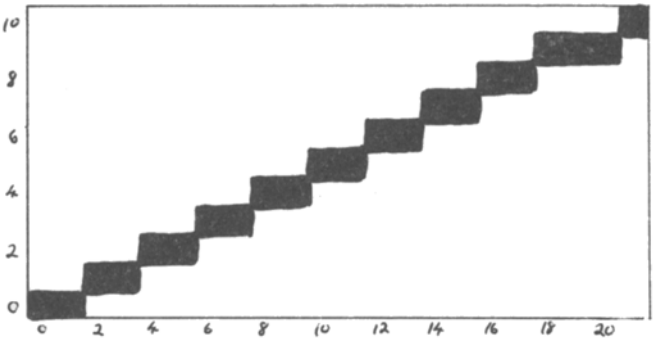
The shaded pixels indicate the approximation produced by Bresenham's, the Simple DDA, and Chain Code algorithms.



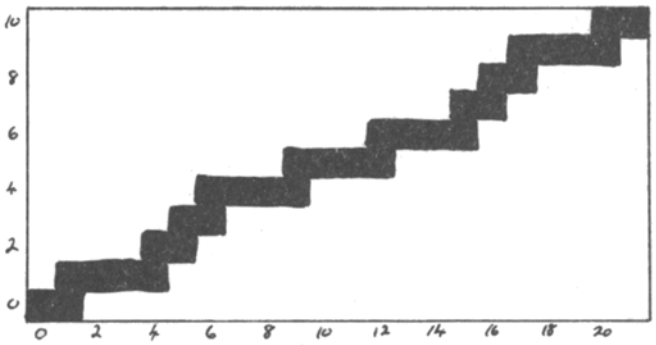
The Symmetric DDA turns on all the pixels turned on by the Simple DDA algorithm, and some additional ones.



The All Pixels Touched algorithm turns on all the pixels turned on by the Symmetric DDA algorithm, and more.



Brons' Chain Code algorithm is identical to the Chain Code algorithm except in column 20.



The BRM algorithm is quite inaccurate when approximating this line, since in binary form Δx is the reflection of Δy.

References

1. C. Arcelli and A. Massarotti, "Regular Arcs in Digital Contours," Computer Graphics and Image Processing Vol. 4 pp. 339-360 (1975).
2. G. Bongiovanni, F. Luccio, and A. Zorat, "The Discrete Equation of a Straight Line," IEEE Transactions on Computers Vol. C-24(3) pp. 310-313 (March 1975).
3. J. Boothroyd and P. A. Hamilton, "Exactly Reversible Plotter Paths," Australian Computer Journal Vol. 2(1) pp. 20-21 (1970).
4. J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," IBM Systems Journal Vol. 4(1) pp. 25-30 (1965).
5. K. W. Brodlie, M. C. Maguire, and G. E. Pfaff, "A Practical Strategy for Certifying GKS Implementations," in EUROGRAPHICS 82 International Conference and Exhibition UMIST 8-10 Sept 1982, eds. D. S. Greenaway and E. A. Warman, North-Holland (1982).
6. R. Brons, "Linguistic Methods for the Description of a Straight Line on a Grid," Computer Graphics and Image Processing Vol. 3 pp. 48-62 (1974).
7. George S. Carson, "The Specification of Computer Graphics Systems," IEEE Computer Graphics and Applications Vol. 3(6) pp. 27-41 (1974).
8. D. A. Duce, E. V. C. Fielding, and L. S. Marshall, Formal Specifiation and Graphics Software, Rutherford Appleton Laboratory Report RAL-84-068, August 1984.
9. R. A. Earnshaw, Display Algorithms - History, Developments and Applications, University Computing Service, University of Leeds.
10. Eugene Fiume and Alain Fournier, A Programme for the Development of a Mathematical Theory of Computer Graphics, Computer Systems Research Group, Department of Computer Science, University of Toronto, Toronto, Ontario (1984).
11. J. D. Foley and A. van Dam, Fundamentals of Interactive Computer Graphics, Addison-Wesley Publishing Company (1982).
12. Herbert Freeman, "Boundary Encoding and Processing," pp. 241-266 in Picture Processing and Psychopictorics, eds. Bernice Sacks Lipkin and Azriel Rosenfeld, Academic Press, New York-London (1970).
13. Graphical Kernel System (GKS) 7.2 Functional Description, Draft International Standard ISO/DIS 7942 (November 14th, 1982).
14. R. Gnatz, Approaching a Formal Framework for Graphics Software Standards, Technical University of Munich.
15. Leo J. Guibas and Jorge Stolfi, "A Language for Bitmap Manipulation," ACM Transactions on Graphics Vol. 1(3) pp. 191-214 (July 1982).

16. Satish Gupta and Robert F. Sproull, "Filtering Edges for Grey-Scale Displays," ACM Computer Graphics Vol. 15(3) pp. 1-5 (August 1981).
17. John Guttag and James J. Horning, Formal Specification as a Design Tool, XEROX PARC Technical Report CSL-80-1, Palo Alto, CA (June 1982).
18. C. B. Jones, Software Development: A Rigorous Approach, Prentice-Hall, Englewood Cliffs, NJ (1980).
19. William R. Mallgren, Formal Specification of Interactive Graphics Programming Languages, ACM Distinguished Dissertation 1982, MIT Press (1983).
20. Lynn S. Marshall, A Formal Specification of Line Representations on Graphics Devices, University of Manchester Transfer Report, September 1984.
21. Lynn S. Marshall, GKS Workstations: Formal Specification and Proofs of Correctness for Specific Devices, University of Manchester Transfer Report, September 1984.
22. William M. Newman and Robert F. Sproull, Principles of Interactive Computer Graphics, McGraw Hill Kogakuska Limited (1973).
23. William M. Newman and Robert F. Sproull, Principles of Interactive Computer Graphics, Second Edition, McGraw Hill International Book Company (1981).
24. M. L. V. Pitteway and D. J. Watkinson, "Bresenham's Algorithm with Grey Scale," Communications of the ACM Vol. 23(11) pp. 625-626 (1980).
25. Azriel Rosenfeld, "Digital Straight Line Segments," IEEE Transactions on Computers Vol. C-23(12) pp. 1264-1269 (December 1974).
26. Jerome Rothstein and Carl Weiman, "Parallel and Sequential Specification of a Context Sensitive Language for Straight Lines on Grids," Computer Graphics and Image Processing Vol. 5 pp. 106-124 (1976).
27. Robert F. Sproull, "Using Program Transformations to Derive Line Drawing Algorithms," ACM Transactions on Graphics Vol. 1(4) pp. 259-273 (October 1982).
28. University of Manchester Computer Graphics Unit, Interactive Computer Graphics Course Notes (March 1984).
29. John Warnock and Douglas K. Wyatt, "A Device Independent Graphics Imaging Model for Use with Raster Devices," ACM Computer Graphics Vol. 16(3) pp. 313-319 (July 1982).