Number Representations and Precision in Vector Graphics

Author: Samuel Moore[?]
Partners: David Gow[1]
Supervisors: Prof Tim French, Dr Rowan Davies

THE UNIVERSITY OF
WESTERN AUSTRALIA
Achieve International Excellence

October 26, 2014

Abstract

Early document formats such as PostScript were motivated by a desire to print text and visual information onto a static paper medium. Although documents are increasingly viewed digitally, modern standards including PDF and SVG are still largely based upon this model. Digital document viewers are able to scale a subregion of the document to fit the display. However, coordinates of graphics primitives are typically represented with IEEE-754 floating point numbers. This places limits on the precision with which primitives in the document can be specified and rendered.

We have implemented a minimal SVG viewer, with which we have compared a number of approaches to achieving arbitrary precision document formats. We demonstrate the trade off between performance and precision with alternative number representations including arbitrary precision floats, rationals, and IEEE-754 fixed precision floats. We also consider approaches to increasing the precision that can be attained with IEEE-754 floats.

Keywords: *document formats, precision, floating point, vector images, graphics, OpenGL, SDL2, PostScript, PDF, TEX, SVG, HTML5, Javascript*

Note: This report is best viewed digitally as a PDF. The digital version is available at ⟨http://szmoore.net/ipdf/sam/thesis.pdf⟩

Word Count: 7620 (9335 with appendices)

Samuel Z. Moore
45 Wheyland Street
Willagee, WA, 6156

27th October, 2014

Wintrhop Professor John Dell
Dean
Faculty of Engineering, Computing and Mathematics
University of Western Australia
35 Stirling Highway
Crawley, WA, 6009

Dear Professor Dell,

I am pleased to submit this thesis, entitled "Number Representations and Precision in Vector Graphics", as part of the requirement for the Engineering component of the degree of Bachelor of Science and Engineering.

Yours Sincerely,

Samuel Z. Moore
20503628

Acknowlegments

I would like to acknowledge my supervisors, Prof Tim French and Dr Rowan Davies for their support and feedback during this project. I would also like to thank my colleague David Gow for his contributions to the joint part of the project. Lastly, as a double degree student I need to express a double degree of thanks to my friends and family for their continued patience whilst I completed my second “final year” project in what is technically a penultimate year.

The rest of this space is left intentionally blank, apart from this sentence which informs the reader that the space is left intentionally blank.

Contents

1 Introduction **1**

2 Background **2**

2.1 Raster and Vector Graphics . . . 2

2.2 Rendering Vector Primitives . . . 3

2.2.1 Straight Lines . . . 3

2.2.2 Bézier Splines . . . 4

2.2.3 Fonts . . . 6

2.3 Precision Specified by Document Standards . . . 6

2.3.1 PostScript . . . 6

2.3.2 PDF . . . 6

2.3.3 TeX and METAFONT . . . 7

2.3.4 SVG . . . 7

2.3.5 Javascript . . . 7

2.4 Fixed Point and Integer Number Representations . . . 7

2.4.1 Big Integers . . . 8

2.5 Floating Point Number Representations . . . 8

2.5.1 Visualisation of Floating Point Representation . . . 9

2.6 Arbitrary Precision Floating Point Numbers . . . 10

2.7 Rational Number Representations . . . 11

2.8 Floating Point Operations on the CPU and GPU . . . 11

3 Implementation of an SVG Viewer **13**

3.1 Software Overview . . . 13

3.2 Document Structure . . . 13

3.3 CPU and GPU Renderering . . . 14

3.4 Coordinate Systems and Transformations . . . 14

3.4.1 View Transformations . . . 15

3.5 Interactivity and Obtaining Results . . . 15

3.6 Version Control . . . 16

3.7 Approaches to Arbitrary Precision . . . 16

3.7.1 Naïve Approach . . . 16

3.7.2 Intermediate Coordinate Systems . . . 17

3.7.3 Quadtree Document Division . . . 17

3.8 Libraries Used . . . 17

4 Results and Discussion **18**

4.1 Qualitative Rendering Accuracy . . . 18

4.1.1 Applying the view transformation directly . . . 18

4.1.2 Applying cumulative transformations to all Béziers 18
4.1.3 Applying cumulative transformations to Paths 19
4.2 Quantitative Measurements of Rendering Accuracy 20
4.2.1 Precision for Fixed View 20
4.2.2 Accumulated error after changing the View 21
4.3 Performance Measurements 22
4.3.1 Performance of Static Detail at Different View Locations 22
4.3.2 Performance whilst adding Detail 23
4.4 Video Demonstrations 23

5 Conclusion **24**
5.1 Work Achieved 24
5.2 Limitations and Future Work 24

References **27**

Appendices **28**

A An Overview of Document Standards **29**
A.1 Interpreted Models 29
A.2 The Document Object Model 30
A.2.1 Javascript and the DOM 31
A.3 Compositing 33

List of Figures

2.1 Original Vector and Raster Images 2
2.2 Scaled Vector and Raster Images 3
2.3 Rasterising a Straight Line 4
2.4 Constructing a Spline from two cubic Béziers (a) Showing the Control Points (b) Representations in SVG and PostScript (c) Rendered Spline 5
2.5 a) Vector glyph for the letter Z b) Screenshot showing Bézier control points in Inkscape . . 6
2.6 Positive 8-Bit Number Representations 10
2.7 Difference between successive numbers 10
2.8 CPU and GPU evaluation of $x^2 + y^2 < 1$ (black) at $\approx 10^6$ magnification 12

3.1 Rendering of Figure 2.2 in the IPDF software a) Outline with individual Béziers highlighted in rectangles b) With shading enabled 13

3.2 Illustration of view transformation (3.1) 15

3.3 The Qt4 Control Panel provides basic interactivity 16

3.4 Commit statistics from the repository at Github (this author is "szmoore") 16

4.1 The vector image from Figure 2.1 under two different scales 18

4.2 The effect of applying cumulative transformations to all Béziers 19

4.3 Effect of cumulative transformations applied to Paths a) Path bounds represented using floats b) Path bounds represented using GMP Rationals 19

4.4 Effect of applying (3.1) to a grid of lines seperated by 1 pixel a) Near origin (denormals) b), c), d) Increasing the exponent of (v_x, v_y) by 1 20

4.5 Loss of precision of the grid 21

4.6 Error in the coordinates of the grid **Note:** Logarithmic Axes 22

4.7 a) Memory used per Path coordinate and b) Time taken to scale 22

4.8 a) Performance including Naïve Implementations b) Excluding `Gmprat` data Legend is in descending order to correspond with the height of the curves 23

4.9 The test SVG used to produce the videos 23

A.1 Vector image and a possible PostScript representation 30

A.2 Vector image and a possible SVG representation 32

A.3 Koch "snowflakes" generated using Javascript to modify an SVG DOM. The interactive HTML5 document can be found at ⟨http://szmoore.net/ipdf/sam/figures/koch.html⟩ 32

1. Introduction

Early electronic document formats such as PostScript were motivated by a need to print documents onto a paper medium. In the PostScript standard, this lead to a model of the document as a program; a series of instructions to be executed by an interpreter which would result in "ink" being placed on "pages" of a fixed size[2].

The ubiquitous Portable Document Format (PDF) standard provides many enhancements to PostScript taking into account desktop publishing requirements [3], but it is still fundamentally based on the same imaging model [4]. This idea of a document as a static "page" has lead to limitations on what could be achieved with a digital document viewers [5].

The emergence of the internet, web browsers, XML/HTML, JavaScript and related technologies has seen a revolution in the ways in which information can be presented digitally, and the PDF standard itself has begun to move beyond static text and figures [5, 6]. However, the popular document formats are still designed with the intention of showing information at either a single, fixed level of detail, or a small range of levels.

As most digital display devices are smaller than physical paper medium, all useful viewers are able to "zoom" to a subset of the document. Vector graphics formats including PostScript, PDF and SVG support rasterisation at different zoom levels [2, 4, 7], but the use of fixed precision floating point numbers causes problems due to imprecision either far from the origin, or at a high level of detail [8, 9].

There are many possible applications for documents in which precision is unlimited. Several areas of use include: visualisation of extremely large or infinite data sets; visualisation of high precision numerical computations; digital artwork; computer aided design; and maps.

The goal of this work is to explore to explore the limitations of floating point arithmetic and possible approaches to achieving arbitrary precision document formats. In collaboration with Gow [1] we have implemented a proof of concept document viewer compatable with a subset of the SVG standard as a starting point for our research.

With the aim of being able to correctly insert and render "detail" (constructed by importing test SVG images) seperated by arbitrary distance, this work explores the limitations in floating point arithmetic and how these may be mitigated

Using the Rational representation of the GNU Multiple Precision (GMP) library [10] we are able to implement correct rendering of SVG test images seperated by extremely large distances. We will present measurements of rendering accuracy and performance for our implementation.

An alternative implementation based on a spatial approach to constructing the document is discussed by Gow [1].

2. Background

2.1 Raster and Vector Graphics

At a fundamental level everything that is seen on a display device is represented as either a vector or raster image. These images can be stored as stand alone documents or embedded within a more complex document format capable of containing many other types of information.

A raster image's structure closely matches it's representation as shown on modern display hardware; the image is represented as a grid of filled square "pixels". Each pixel is considered to be a filled square of the same size and contains information describing its colour. This representation is simple and also well suited to storing images as produced by cameras and scanners. The drawback of raster images is that by their very nature there can only be one level of detail; this is illustrated in Figures 2.1 and 2.2.

A vector image contains information about the positioning and shading of geometric shapes. To display this image on modern display hardware, coordinates are transformed according to the view and then the image is converted into a raster like representation. Whilst the raster image merely appears to contain edges, the vector image actually contains information about these edges, meaning they can be displayed "infinitely sharply" at any level of detail — or they could be if the coordinates are stored with enough precision.

Figures 2.1 and 2.2 illustrate the advantage of vector formats by comparing raster and vector images in a similar way to Worth and Packard[11]. On the right is a raster image which should be recognisable as an animal defined by fairly sharp edges. Figure 2.2 shows how these edges appear jagged when scaled. There is no information in the original image as to what should be displayed at a larger size, so each square shaped pixel is simply increased in size. A blurring effect will probably be visible in most PDF viewers; the software has attempted to make the "edge" appear more realistic using a technique called "antialiasing"[1].

The left side of the Figures are a vector image. When scaled, the edges maintain a smooth appearance which is limited by the resolution of the display rather than the image itself.

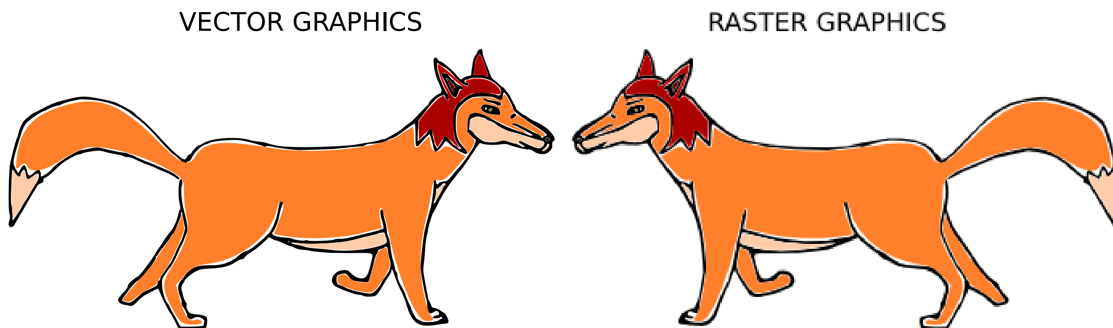


Figure 2.1: Original Vector and Raster Images

[1]We recommend disabling this if your PDF viewer supports it

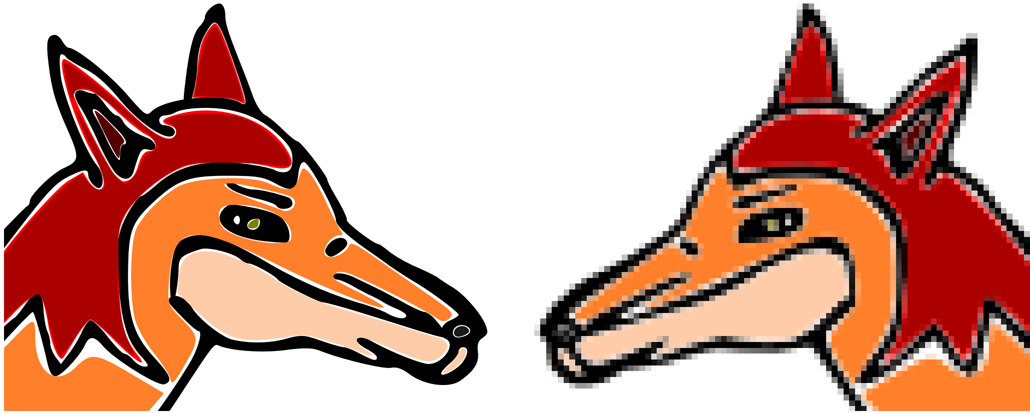


Figure 2.2: Scaled Vector and Raster Images

2.2 Rendering Vector Primitives

It is of some historical significance that vector display devices were popular during the 70s and 80s, and papers oriented towards drawing on these devices can be found[12]. Whilst curves can be drawn at high resolution on vector displays, a major disadvantage was shading[13]; by the early 90s the vast majority of computer displays were raster based[14].

Hearn and Baker's textbook "Computer Graphics" [14] gives a comprehensive overview of graphics from physical display technologies through fundamental drawing algorithms to popular graphics APIs. This section will examine algorithms for drawing two dimensional geometric primitives on raster displays as discussed in "Computer Graphics" and the relevant literature. This section is by no means a comprehensive survey of the literature but intends to provide some idea of the computations which are required to render a document.

We will restrict our focus to drawing the outlines of shapes, as the accuracy of shading a region will depend on the accuracy of the outline.

2.2.1 Straight Lines

It is well known that in cartesian coordinates, a line between points (x_1, y_1) and (x_2, y_2), can be described by:

$$y(x) = mx + c \quad \text{on } x \in [x_1, x_2] \text{ for } m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \text{ and } c = y_1 - mx_1 \tag{2.1}$$

On a raster display, only points (x, y) with integer coordinates can be displayed; however m will generally not be an integer. Thus a straight forward use of Equation 2.1 will require floating point operations and therefore rounding (See Section 2.5). Modifications based on computing steps Δx and Δy eliminate the multiplication but are still less than ideal in terms of performance[14].

It should be noted that algorithms for drawing lines can be based upon sampling $y(x)$ only if

$|m| \leq 1$; otherwise sampling at every integer x coordinate would leave gaps in the line because $\Delta y > 1$. Line drawing algorithms can be trivially adopted to sample $x(y)$ if $|m| > 1$.

Bresenham's Line Algorithm was developed in 1965 with the motivation of controlling a particular mechanical plotter in use at the time[15]. The plotter's motion was confined to move between discrete positions on a grid one cell at a time, horizontally, vertically or diagonally. As a result, the algorithm presented by Bresenham requires only integer addition and subtraction, and it is easily adopted for drawing pixels on a raster display. Because integer operations are exact, only an error in the calculation of the line end points will affect the rendering.

In Figure 2.3 a) and b) we illustrate the rasterisation of a line width a single pixel width. The path followed by Bresenham's algorithm is shown. It can be seen that the pixels which are more than half filled by the line are set by the algorithm. This causes a jagged effect called aliasing which is particularly noticable on low resolution displays. From a signal processing point of view this can be understood as due to the sampling of a continuous signal on a discrete grid[16].

Figure 2.3 c) shows an (idealised) antialiased rendering of the line. The pixel intensity has been set to the average of the line and background colours over that pixel. Such an ideal implementation would be impractically computationally expensive on real devices[17]. In 1991 Wu introduced an algorithm for drawing approximately antialiased lines which, while equivelant in results to existing algorithms by Fujimoto and Iwata, set the state of the art in performance[16][2]. .

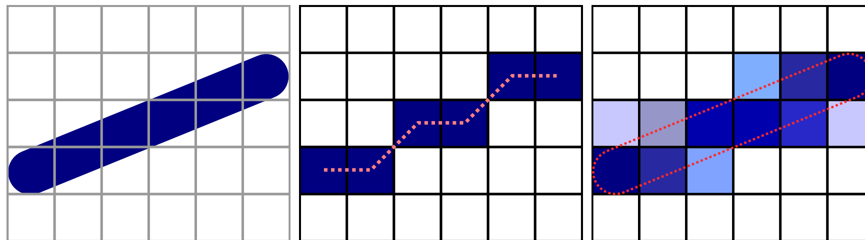


Figure 2.3: Rasterising a Straight Line

a) Before Rasterisation b) Bresenham's Algorithm c) Anti-aliased Line (Idealised)

2.2.2 Bézier Splines

Splines are continuous curves formed from piecewise polynomial segments. A polynomial of nth degree is defined by n constants $\{a_0, a_1, ...a_n\}$ and:

$$y(x) = \sum_{k=0}^{n} a_k x^k \tag{2.2}$$

Cubic and Quadratic Bézier Splines are used to define curved paths in the PostScript[2], PDF[4] and SVG[7] standards. Cubic Béziers are also used to define vector fonts for rendering text in these standards and the TEX typesetting language [18, 19]. Although he did not derive the mathematics, the usefulness of Bézier curves was realised by Pierre Bézier who used them in the 1960s for the

[2] Techniques for antialiasing primitives other than straight lines are discussed in some detail in Chapter 4 of "Computer Graphics" [14]

computer aided design of automobile bodies[20].

A Bézier Curve of degree n is defined by n "control points" $\{P_0, ...P_n\}$. Points $P(t) = (x(t), y(t))$ along the curve are defined by:

$$P(t) = \sum_{j=0}^{n} B_j^n(t) P_j \tag{2.3}$$

Where $t\epsilon[0, 1]$ is a control parameter. The polynomials $B_j^n(t)$ are Bernstein Basis Polynomials which are defined as:

$$B_j^n(t) = \tbinom{n}{j}\, t^j\, (1-t)^{n-j} \qquad j = 0, 1, ..., n \tag{2.4}$$

$$\text{Where } \tbinom{n}{j} = \frac{n!}{n!(n-j)!} \qquad \text{(The Binomial Coefficients)} \tag{2.5}$$

From these definitions it should be apparent that in all cases, $P(0) = P_0$ and $P(1) = P_n$. An $n = 1$ Bézier Curve is a straight line.

Algorithms for rendering Bézier's may simply sample $P(t)$ for suffiently many values of t — enough so that the spacing between successive points is always less than one pixel distance. Alternately, a smaller number of points may be sampled with the resulting points connected by straight lines using one of the algorithms discussed in Section 2.2.1.

De Casteljau's algorithm of 1959 is often used for decomposing Béziers into line segments[14, 18]. This algorithm subdivides the original curve with n control points $\{P_0, ...P_n\}$ into 2 halves, each with n control points: $\{Q_0, ...Q_n\}$ and $\{R_0, ...R_n\}$; when iterated, the produced points will converge to $P(t)$. As a tensor equation this subdivision can be expressed as[21]:

$$Q_i = \left(\frac{\tbinom{n}{j}}{2^j}\right) P_i \text{ and } R_i = \left(\frac{\tbinom{n-j}{n-k}}{2^{n-j}}\right) P_i \tag{2.6}$$

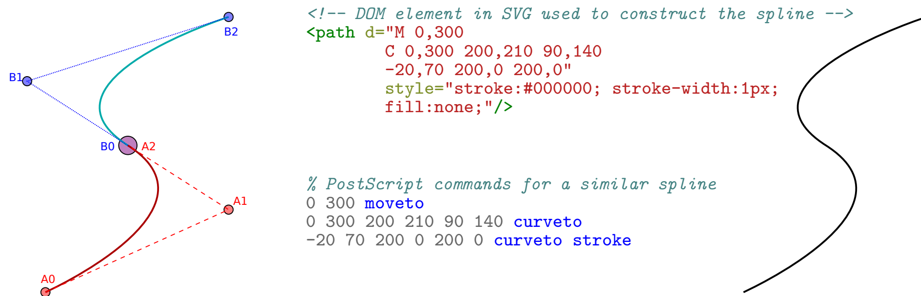


Figure 2.4: Constructing a Spline from two cubic Béziers
(a) Showing the Control Points (b) Representations in SVG and PostScript (c) Rendered Spline

2.2.3 Fonts

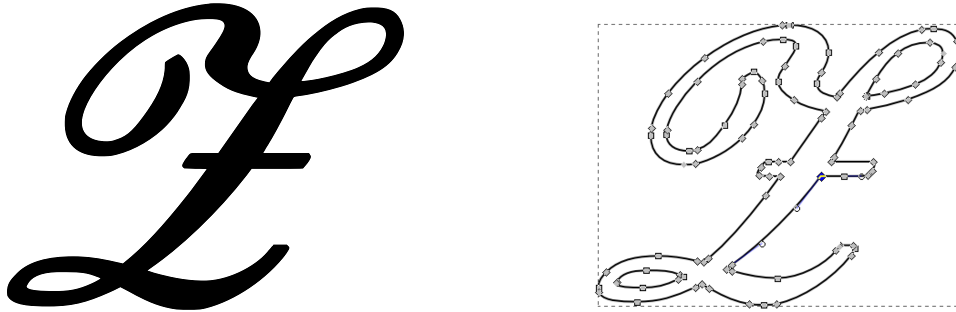


Figure 2.5: a) Vector glyph for the letter Z b) Screenshot showing Bézier control points in Inkscape

A the term "font" refers to a set of images used to represent text on a graphical display. In 1983, Donald Knuth published "The METAFONT Book" which described a vector approach to specifying fonts and a program for creating these fonts[18]. Previously, only rasterised font images (glyphs) were popular; as can be seen from the zooming in Figure 2.2 this can be problematic given the prevelance of textual information at different scales and on different resolution displays.

Knuth used Bézier Cubic Splines to define "pleasing" curves in METAFONT, and this approach is still used in modern vector fonts. Since the paths used to render an individual glyph are used far more commonly than general curves, document formats do not require such curves to be specified in situ, but allow for a choice between a number of internal fonts or externally specified fonts. In the case of Knuth's typesetting language TEX, fonts were intended to be created using METAFONT[18]. Figure 2.5 shows a $\mathscr{Z}$ (script Z) produced by LATEX with Bézier cubics identified.

2.3 Precision Specified by Document Standards

In this section we will overview the statements made about the precision with which an object can be stored by various vector graphics and document standards. A more detailed description of the standards discussed here can be found in Appendix A.

2.3.1 PostScript

The PostScript reference describes a "Real" object for representing coordinates and values as follows: "Real objects approximate mathematical real numbers within a much larger interval, but with limited precision; they are implemented as floating-point numbers"[2]. There is no reference to the precision of mathematical operations, but the implementation limits *suggest* a range of $\pm 10^{38}$ "approximate" and the smallest values not rounded to zero are $\pm 10^{-38}$ "approximate".

2.3.2 PDF

PDF defines "Real" objects in a similar way to PostScript, but suggests a range of $\pm 3.403 \times 10^{38}$ and smallest non-zero values of $\pm 1.175 \times 10^{38}$[4]. A note in the PDF 1.7 manual mentions that

Acrobat 6 now uses IEEE-754 single precision floats, but "previous versions used 32-bit fixed point numbers" and "... Acrobat 6 still converts floating-point numbers to fixed point for some components".

2.3.3 TEX and METAFONT

In "The METAFONT book" Knuth appears to describe coordinates as fixed point numbers: "The computer works internally with coordinates that are integer multiples of $\frac{1}{65536} \approx 0.00002$ of the width of a pixel"[18]. [3] There is no mention of precision in "The TEX book". In 2007 Beebe claimed that TEX uses a 14.16 fixed point encoding, and that this was due to the lack of standardised floating point arithmetic on computers at the time; a problem that the IEEE-754 was designed to solve[22]. Beebe also suggested that TEX and METAFONT could now be modified to use IEEE-754 arithmetic.

2.3.4 SVG

The SVG standard specifies a minimum precision equivelant to that of "single precision floats" (presumably referring to IEEE-754) with a range of `-3.4e+38F` to `+3.4e+38F`, and states "It is recommended that higher precision floating point storage and computation be performed on operations such as coordinate system transformations to provide the best possible precision and to prevent round-off errors."[7] An SVG Viewer may refer to itself as "High Quality" if it uses a minimum of "double precision" floats for view transformations.

2.3.5 Javascript

We include Javascript here due to its relation with the SVG, HTML5 and PDF standards. According to the EMCA-262 standard, "The Number type has exactly 18437736874454810627 (that is, $2^{6}4 -^{5} 3 + 3$) values, representing the double-precision 64-bit format IEEE 754 values as specified in the IEEE Standard for Binary Floating-Point Arithmetic"[23]. The Number type does differ slightly from IEEE-754 in that there is only a single valid representation of "Not a Number" (NaN). The EMCA-262 does not define an "integer" representation.

2.4 Fixed Point and Integer Number Representations

A positive real number z may be written as the sum of smaller integers "digits" d_i multiplied by powers of a base β.

$$z = ... \ + d_{-1}\beta^{-1} + d_0\beta^0 + d_1\beta^1 + \ ... = \sum_{i=-\infty}^{\infty} d_i\beta^i \tag{2.7}$$

Where each digit $d_i < \beta$. A set of β unique symbols are used to represent values of d_i. A seperate sign '-' can be used to represent negative reals using equation (2.7).

To express a real number using equation (2.7) in practice we are limited to a finite number of terms between $i = -m$ and $i = n$. Fixed point representations are capable of representing a

[3]This corresponds to using 16 bits for the fractional component of a fixed point representation

discrete set of numbers $0 \leq |z| \leq \beta^{n+1} - \beta^{-m}$ seperated by $\Delta z = \beta^{-m} \leq 1$. In the case $m = 0$, only integers can be represented.

Example integer representation in base 10 (decimal) and base 2 (binary):

$$5682_{10} = 5 \times 10^3 + 6 \times 10^2 + 8 \times 10^1 + 2 \times 10^0$$
$$1011000110010_2 = 1 \times 2^{12} + 0 \times 2^{11} + \ ... + 0 \times 2^0$$

2.4.1 Big Integers

Computer hardware implements operations for fixed size integers. The base is $\beta = 2$ and the digits are 0, 1. The most significant bit can be reserved for the sign instead of a digit. We can construct larger size integers by considering some sequence of fixed size integers to be individual digits. In practice we will still be limited by the memory and processing time required 1for "big" integers.

For example, we can represent 5682_{10} as a single 16 bit digit or as the sum of two 8 bit digits. Each digit is being written in base 2 or 10 because there is not a universal base with $\geq 2^8$ unique symbols.

$$5682_{10} = 1011000110010_2 = 10110_2 \times 2^8 + 110010_2 \times 2^0$$

When performing an operation involving two m digit integers, the result will in general require at most $2m$ digits. A straight forward big integer implementation merely needs to allocate memory for leading zeroes

Big Integers are implemented on the CPU as part of the standard for several languages including Python[24] and Java[25]. Most implementations are based on the GNU Multiple Precision library (GMP) [?]. There have also been implementations of Big Integer arithmetic for GPUs[26].

During this project a custom Big Integer type was implemented, but was found to be vastly inferior to the GMP implementation[27].

2.5 Floating Point Number Representations

The use of floating point arithmetic in computer systems was pioneered by Knuth, Goldberg[28], Dekker, and others[29], but modern systems are largely compatable with the IEEE-754 standard pioneered by William Kahan in 1985 [30] and revised (also with contributions from Kahan) in 2008[31]. Recently, the "Handbook of Floating Point Arithmetic"[29] by Muller et al (2010) provides a detailed overview of IEEE-754 floating point arithmetic.

Whilst a Fixed Point representation keeps the "point" (the location considered to be $i = 0$ in (2.7)) at the same position in a string of bits, Floating point representations can be thought of as scientific notation; an "exponent" and fixed point value are encoded, with multiplication by the exponent moving the position of the point.

A floating point number x is commonly represented by a tuple of values (s, e, m) in base B as[29, 32]: $x = (-1)^s \times m \times B^e$

Where s is the sign and may be zero or one, m is commonly called the "mantissa" and e is the exponent. Whilst e is an integer in some range $\pm e_m ax$, the mantissa m is a fixed point value in the range $0 < m < B$. The choice of base $B = 2$ in the original IEEE-754 standard matches the nature of modern hardware. It has also been found that this base in general gives the smallest rounding errors[29].

The IEEE-754 encoding of s, e and m requires a fixed number of continuous bits dedicated to each value. Originally two encodings were defined: binary32 and binary64. s is always encoded in a single leading bit, whilst (8,23) and (11,53) bits are used for the (exponent, mantissa) encodings respectively.

The encoding of m in the IEEE-754 standard is not exactly equivelant to a fixed point value. By assuming an implicit leading bit (ie: restricting $1 \leq m < 2$) except for when $e = 0$, floating point values are gauranteed to have a unique representations; these representations are said to be "normalised". When $e = 0$ the leading bit is not implied; these representations are called "denormals" because multiple representations may map to the same real value. The idea of using an implicit bit appears to have been considered by Goldberg as early as 1967[28], and it leads to an increase of precision near the origin.

2.5.1 Visualisation of Floating Point Representation

To assist with understanding the limitations of floating point representations, we have produced a plot of the positive real numbers which can be represented exactly by an 8 bit floating point number encoded in the IEEE-754 format. We could not find any similar visualisations in the literature.

In Figure 2.6 we show two encodings using (1,2,5) and (1,3,4) bits to encode (sign, exponent, mantissa) respectively. For each distinct value of the exponent, the successive floating point representations lie on a straight line with constant slope. As the exponent increases, larger values are represented, but the distance between successive values increases; this can be seen in Figure 2.7. The marked single point discontinuity at `0x10` and `0x20` occur when e leaves the denormalised region and the encoding of m changes. We have also plotted a fixed point representation for comparison; fixed point and integer representations appear on straight lines.

Real values which cannot be represented exactly in a floating point representation must be rounded to the nearest floating point value. The results of a floating point operation will in general be such values and thus there is a rounding error possible in any floating point operation[29, 31, 8].

Referring to Figure 2.6 it can be seen that the largest possible rounding error is half the distance between successive floats; this means that rounding errors increase as the value to be represented increases. For the result of a particular operation, the maximum possible rounding error can be determined and is commonly expressed in "units in the last place" (ulp), with 1 ulp equivelant to half the distance between successive floats[8].

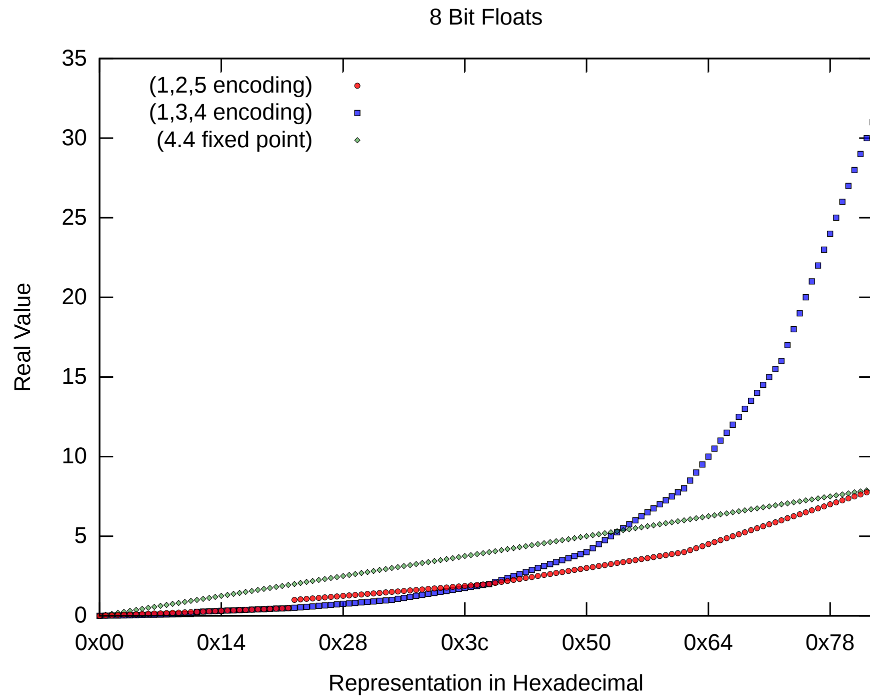


Figure 2.6: Positive 8-Bit Number Representations

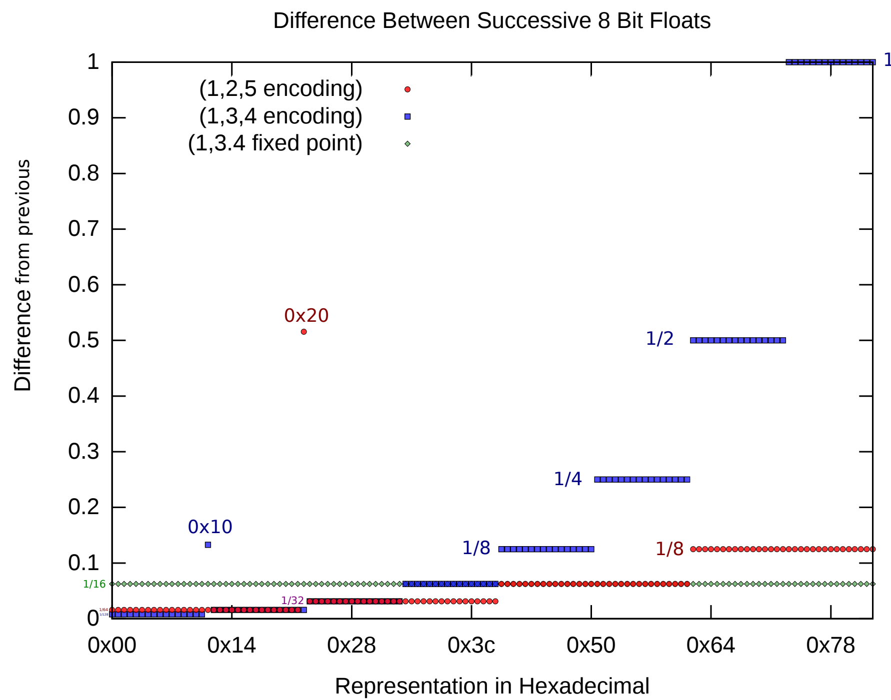


Figure 2.7: Difference between successive numbers

2.6 Arbitrary Precision Floating Point Numbers

Arbitrary precision floating point numbers are implemented in a variety of software libraries which will allocate extra bits for the exponent or mantissa as required. An example is the GNU

MPFR library discussed by Fousse in 2007[33]. Although many arbitrary precision libraries already existed, MPFR intends to be fully compliant with some of the more obscure IEEE-754 requirements such as rounding rules and exceptions.

It is trivial to find real numbers that would require an infinite number of bits to represent exactly (for example, $\frac{1}{3} = 0.333333...$). The GMP and MPFR libraries require a fixed but arbitrarily large precision (size of the mantissa) be set; although it is possible to increase or decrease the precision of individual numbers as desired.

2.7 Rational Number Representations

A rational number Q may be represented by two integers N the numerator and D the denominator.

$$Q = \frac{N}{D} \tag{2.8}$$

Compared to floating point arithmetic which is generally inexact, rational arithmetic including the division operation is always exactly representable as another rational number. However, a *fixed size* rational representation is of rather limited use as D will always grow after repeated operations and overflow. Use of arbitrary sized integers as described in section 2.4.1 and implemented by GMP[10] overcomes this issue; however as we will see in Chapter 4 there can be a significant performance cost associated with Rationals.

$$N = \sum_{i=0}^{S} n_i \beta^i \text{ and } D = \sum_{i=0}^{S} d_i \beta^i \text{ where } S \text{ grows as needed} \tag{2.9}$$

2.8 Floating Point Operations on the CPU and GPU

Traditionally, vector images have been rasterized by the CPU before being sent to a specialised Graphics Processing Unit (GPU) for drawing[14]. Rasterisation of simple primitives such as lines and triangles have been supported directly by GPUs for some time through the OpenGL standard[34]. However complex shapes (including those based on Bézier curves such as font glyphs) must either be rasterised entirely by the CPU or decomposed into simpler primitives that the GPU itself can directly rasterise. There is a significant body of research devoted to improving the performance of rendering such primitives using the latter approach, mostly based around the OpenGL[34] API[35, 36, 37, 38, 39, 40]. Recently Mark Kilgard of the NVIDIA Corporation described an extension to OpenGL for NVIDIA GPUs capable of drawing and shading vector paths[41, 42]. From this development it seems that rasterization of vector graphics may eventually become possible upon the GPU.

It is not entirely clear how well supported the IEEE-754 standard for floating point computation is amongst GPUs[4]. Although the OpenGL API does use IEEE-754 number representations,

[4] Informal technical articles are abundant on the internet — Eg: Regarding the Dolphin Wii GPU Emulator: ⟨https://dolphin-emu.org/blog⟩ (accessed 2014-05-22)

research by Hillesland and Lastra in 2004 suggested that many GPUs were not internally compliant with the standard[43].

To test this assertion, Figure 2.8 was produced with an early version of the IPDF software which will be discussed in Chapter 3. The Figure was created jointly with Gow and is also discussed in their work [1].

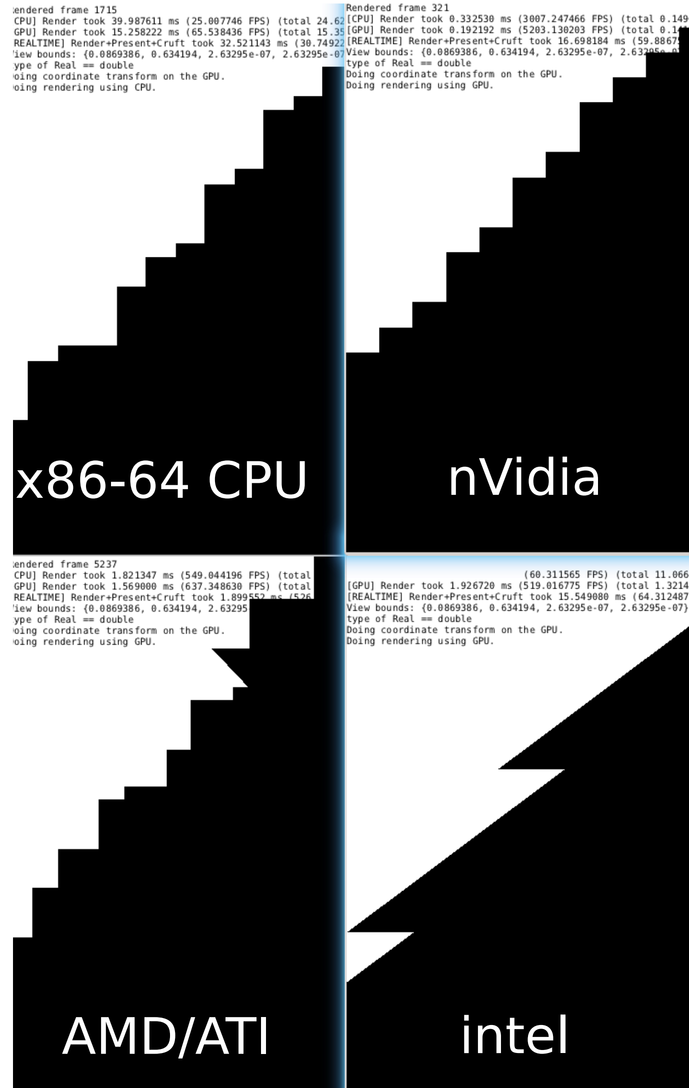


Figure 2.8: CPU and GPU evaluation of $x^2 + y^2 < 1$ (black) at $\approx 10^6$ magnification

Figure 2.8 shows the rendering of a the edge of circle through evaluation of $x^2 + y^2 < 1$ using an x86-64 CPU and various GPU models. If we assume the x86-64 is IEEE-754 compliant performing the default rounding behaviour (to nearest) the GPUs are using different rounding behaviours which may not be IEEE-754 compliant. Whilst outside the scope of this project, consistency of floating point arithmetic on GPUs could be an interesting area for further investigation, particularly given the recent interest in use of GPUs for parallelisable numerical computing.

3. Implementation of an SVG Viewer

To better understand the calculations required to represent and render a vector document, whilst allowing maximum flexibility in approaches to arbitrary precision, a custom vector graphics viewer called IPDF[1] was implemented for this project in collaboration with Gow [1]. This chapter gives a brief overview of the features and limitations of this software.

3.1 Software Overview

The IPDF software has been written using the C++ programming language for x86-64 Debian GNU/Linux machines. The use of C++ offers low level control over CPU and (through the OpenGL API) GPU memory whilst allowing an Object Orientated approach. The choice of C++ was agreed on with Gow [1].

IPDF has been tested on a set of SVG images[2] prepared by the author. Figure 3.1 shows the rendering of the same vector image used in Figure 2.2 in the IPDF software.

The software is capable of importing SVG images scaled to the current view location, and stores a DOM like representation of the document (for discussion of the Document Object Model (DOM) compared to the PostScript style Interpreted Model, refer to Appendix A).

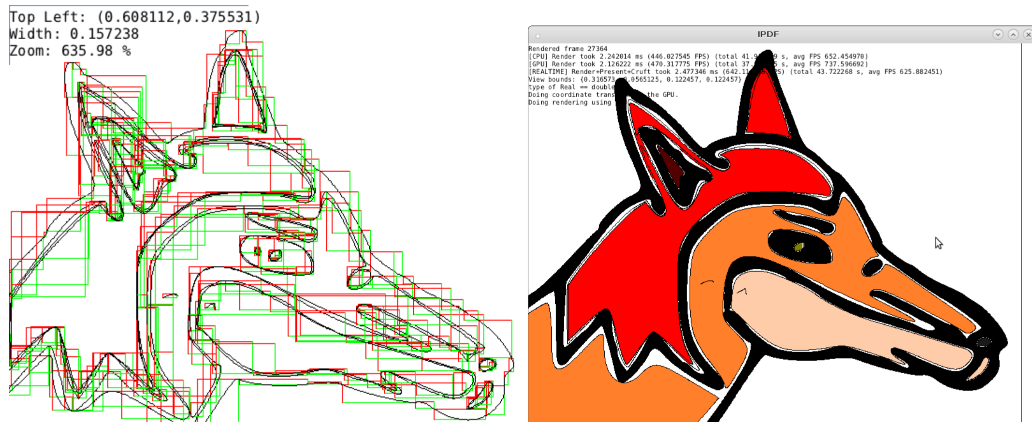


Figure 3.1: Rendering of Figure 2.2 in the IPDF software
a) Outline with individual Béziers highlighted in rectangles b) With shading enabled

3.2 Document Structure

IPDF is built around Objects which are internally represented by bounding rectangles, a type, and any additional coordinates and other data required for rendering the object.

Initially only very simple shapes (Rectangles and Circles) were supported, but in order to produce a meaningful demonstration of arbitrary precision viewing, Paths formed from Quadratic or Cubic Béziers as specified by the SVG standard were added. Shading of paths is partially implemented but detailed discussion is beyond the scope of this report.

[1] The original name "Infinite Precision Document Format" stuck, although the use of the word "infinite" is highly misleading

[2] These can be found at ⟨http://szmoore.net/ipdf/code/src/svg-tests⟩

3.3 CPU and GPU Renderering

As discussed in Section 2.8 it is not clear to what extend GPUs comply with the IEEE-754 standard. In addition, arbitrary precision arithmetic is most easily implemented on the CPU and well supported through libraries such as GMP. For these reasons both a CPU and GPU renderer were implemented.

To render an object on the GPU its bounding rectangle and additional data are provided to a series of OpenGL shader programs. In the case of Bézier curves, a Geometry shader performs the subdivision on the GPU and the resultant points are drawn with lines.

The CPU renderer behaves similarly, with the exception that a custom "Renderer" class performs the function of all three shader programs. A bitmap is directly modified by the CPU and then uploaded to the GPU as a texture for displaying.

Figure 2.8 shows a comparison between the renderering of a circle performed by an x86-64 (CPU) and several GPUs in the IPDF software.

3.4 Coordinate Systems and Transformations

The literature discussed in Chapter 2 is primarily concerned with the rendering process for graphical primitives, namely outlines defined by Bézier curves. We have seen that basic vector primitives composed of Béziers may be rendered using only integer operations, once the starting and ending positions are rounded to the nearest pixel.

However, a complete document will contain many such primitives which in general cannot all be shown on a display at once. A "View" rectangle can be defined to represent the size of the display relative to the document. To interact with the document a user can change this view through scaling or translating with the mouse.

Primitives which are contained within the view rectangle will be visible on the display. This involves the transformation from coordinates within the document to relative coordinates within the view rectangle as illustrated in Figure 3.2. A point (X, Y) in the document will transform to a point (S_X, S_Y) in the display by:

$$S_X = \frac{X - V_x}{V_w} \qquad S_Y = \frac{Y - V_y}{V_h} \tag{3.1}$$

Where (V_x, V_y) are the coordinates of the top left corner and (V_w, V_h) are the dimensions of the view rectangle.

The transformation may also be written as a 3x3 matrix $\boldsymbol{V}$ if we introduce a third coordinate $z = 1$

$$\begin{pmatrix} S_X \\ S_Y \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{V_w} & 0 & \frac{V_x}{V_w} \\ 0 & \frac{1}{V_h} & \frac{V_y}{V_h} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \tag{3.2}$$

This transformation can be used not just for the view, but in any mapping of points from one coordinate system to another one which is defined by some bounds rectangle. In particular, our

implementations of Bézier rendering use this transformation to re-express control points relative to the bounding rectangle (whilst in the SVG standard, control points are specified relative to the document).

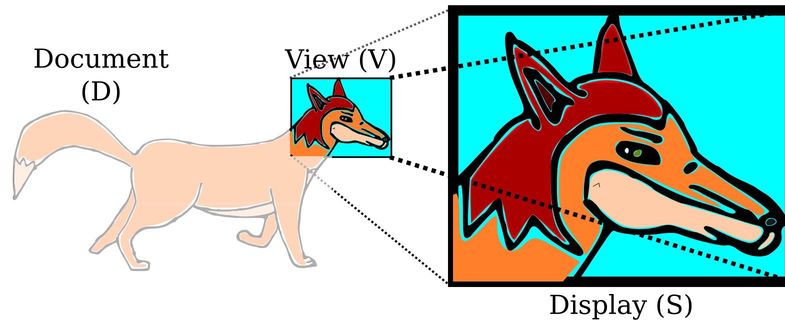


Figure 3.2: Illustration of view transformation (3.1)

3.4.1 View Transformations

Moving the mouse (or on a touch screen, swiping the screen) by a distance $(\Delta x, \Delta y)$ relative to the size of the view should translate it by the same amount:

$$V_x \to V_x + \Delta x \qquad V_y \to V_y + \Delta y \tag{3.3}$$

The document can be scaled by a factor of s about a point (x_0, y_0) specified relative to the view (such as the position of the mouse cursor):

$$V_x \to V_x + x_0 V_w (1 - s) \qquad V_y \to V_y + y_0 V_h (1 - s) \tag{3.4}$$

$$V_w \to s V_w \qquad V_h \to s V_h \tag{3.5}$$

The effect of this transformation is that, measured relative to the view rectangle, the distance of primitives with coordinates (x, y) to the point (x_0, y_0) will decrease by a factor of s. For $s < 1$ the operation is “zooming out” and for $s > 1$, “zooming in”.

As we will see in Chapter 4, the application of the transformations discussed in this section can cause issues with the rendering of vector graphics even when the primitives are specified with coordinates compliant with the SVG standard.

3.5 Interactivity and Obtaining Results

There are two basic ways to control the IPDF software; manually through use of keyboard and mouse and a Qt4 [44] based control panel, or automatically by reading a script containing a sequence of commands to transform the view or insert test SVGs. More complex control can be obtained by using the Python `subprocess` module to produce the commands and analyse performance results.

All results presented in Chapter 4 were obtained on a conventional Debian GNU/Linux laptop with an AMD/ATI Radeon series GPU. An attempt was made to cross compile the software for the

Windows operating system, but at the time of publication there were difficulties with the Windows 7 OpenGL drivers on the author's system.

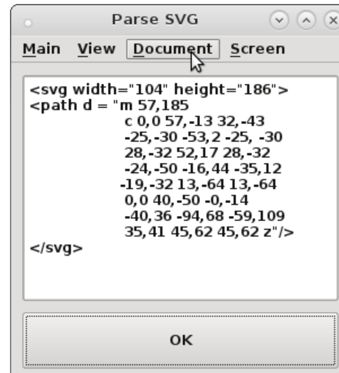


Figure 3.3: The Qt4 Control Panel provides basic interactivity

3.6 Version Control

The Git version control system was used to collaborate and back up work on this project; the main repository may be viewed at ⟨http://git.ucc.asn.au/?p=ipdf/code.git⟩ or on Github at ⟨http://github.com/szmoore/ipdf-code⟩.

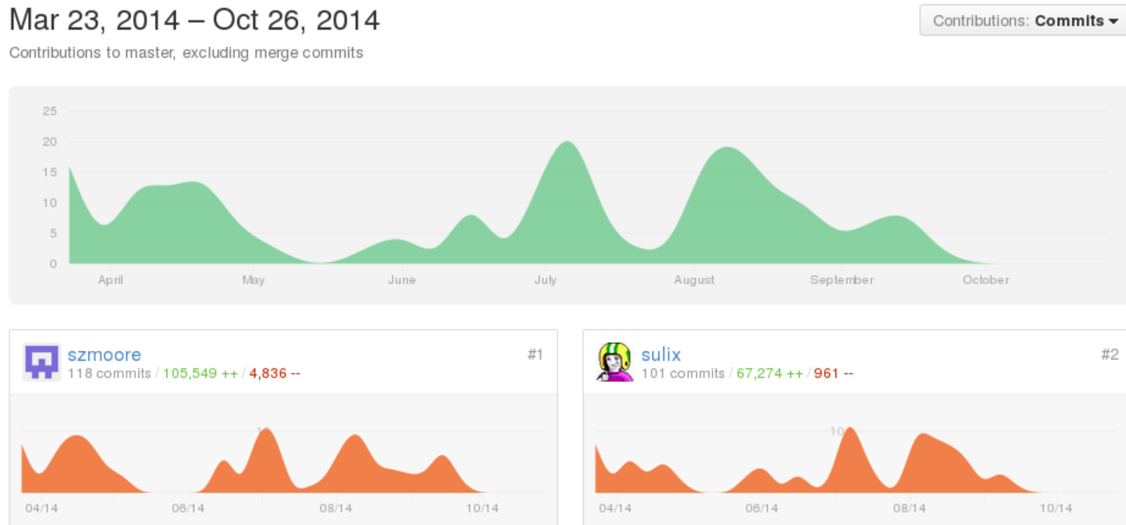


Figure 3.4: Commit statistics from the repository at Github (this author is "szmoore")

3.7 Approaches to Arbitrary Precision

3.7.1 Naïve Approach

A naïve approach would be to replace all floating point operations with arbitrary precision operations, and this was in fact tried in early experiments. This approach requires use of the CPU renderer, as GLSL is restricted to floating point representations. A type definition `Real` on the CPU can be selected at compile time.

Unfortunately truly arbitrary precision number representations (including custom implementations of Rationals, and the GMP library's rationals) were found to be far too inefficient for practical purposes, and indeed unnecessary. The results shown in Chapter 4 were produced using the GPU renderer, since this naïve approach was discarded.

3.7.2 Intermediate Coordinate Systems

When an object is visible on the screen it is only necessary to render it accurately to within the nearest pixel. As shown in Chapter 4, introducing an intermediate coordinate system for a large number of objects and applying transformations to this coordinate system instead of individual objects produces the best results both in terms of reduced rounding errors using floating point arithmetic, and reduced number of required arbitrary precision operations.

3.7.3 Quadtree Document Division

An approach identified by Gow[1] is to construct intermediate coordinate systems as the user manipulates the view in a spatial structure called a "Quadtree". This involves dividing the initial view into four quadrants when the document is scaled by a required amount, and only rendering those quadrant(s) that are visible. The process repeats with additional scaling. With each division objects must be added to the appropriate quadrant, or in the case of objects which span a boundary, clipped. The advantages and disadvantages of this implementation will be explored by Gow[1].

3.8 Libraries Used

- SDL2 - Simple Direct media Library

  SDL2 is a cross-platform library commonly used in games for window management and to obtain an OpenGL context. We have also made some use of the SDL2 bitmap handling functions to save screenshots.

- Qt4 (optional) — Open source toolkit for Dialog based applications

  The control panel shown in Figure 3.3 was created using Qt4. Use of Qt4 can cause difficulties in compiling the software, so it can be disabled at compile time.

- OpenGL (4.4) — The standard API for controlling GPUs

- PugiXML — Open source XML parsing library used to implement parsing of SVGs

- GNU Multiple Precision (GMP)

  As discussed in Sections 2.4.1, 2.6, 2.7 GMP implements arbitrary precision integers, floats, and rationals. Although we did explore these representations by producing custom implementations, examining the GMP source code reveals that it is highly optimised using CPU specific assembly instructions, and vastly outperformed straight forward C++ implementations of Big Integers and Rationals.

- MPFR — Arbitrary precision floats built on GMP but ensures IEEE-754 consistent rounding behaviour. The IPDF software may be compiled with MPFR floats in place of IEEE-754 floats. The precision (size of mantissa) must be set to an arbitrary large but fixed size at compile time.

4. Results and Discussion

4.1 Qualitative Rendering Accuracy

Our ultimate goal is to be able to insert detail at an arbitrary point in the document. Therefore, we are interested in how the same test SVG would appear when scaled to the view coordinates, as the view coordinates are varied.

Throughout this section we will use IEEE-754 single precision (binary32) floats unless otherwise stated. Although double precision (binary64) would allow for greater precision, one could still choose coordinates for which similar results can be obtained.

4.1.1 Applying the view transformation directly

Figure 4.1 shows the rendering of a vector image[1]. Transformation (3.1) is applied to the coordinates of Bézier bounds, with default IEEE-754 rounding behaviour (to nearest). The loss of precision in the second figure is obvious.

In this case, the precision loss occurs when the test SVG is added to the document; the inverse of (3.1) must be applied. That is (for the x coordinate, with the same equations applying for the y coordinate):

$$X = V_w \times \mathrm{SVG}_x + V_x$$

Where V represents the view, X is the coordinate in the document, and SVG_x is the coordinate in the test SVG at original scale. In Figure 4.1, the multiplication $V_w \times \mathrm{SVG}_x$ has a smaller exponent than V_x. The error of the addition operation is comparable to one ulp, ie: $\frac{V_x}{2}$. In this case, the rounding error is dominating the calculation. The division by $V_w = 10^6$ in (3.1) is merely increasing this rounding error.

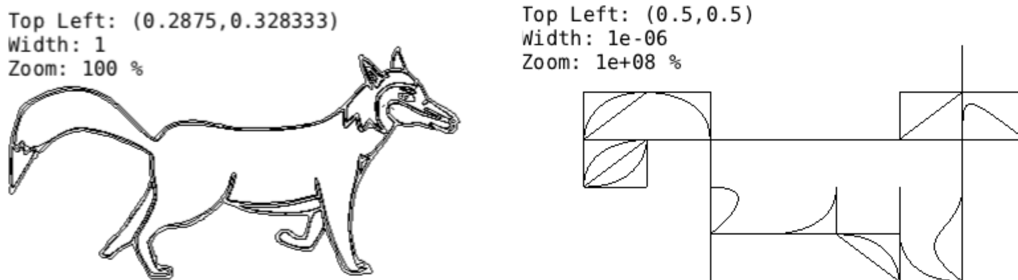


Figure 4.1: The vector image from Figure 2.1 under two different scales

4.1.2 Applying cumulative transformations to all Béziers

Rather than applying (3.1) to object coordinates specified relative to the document, we can store the bounds of objects relative to the view and modify these bounds according to the transformations

[1]Unfortunately, since a rendered vector image is a raster image and this figure must be scaled to fit the PDF, the figure as seen here is not a pixel perfect representation of the actual rendering. Most notably, antialiasing effects will be apparent

discussed in Section 3.4 as the view is changed. This is convenient for an interactive document, as detail is typically added by inserting objects into the document within the view rectangle. As a result this approach makes the rendering of detail added to the document independent of the view coordinates — until the view is moved.

Repeated transformations on the view will cause an accumulated error on the coordinates of object bounds. This is most noticable when zooming *out* and then back into the document; the object coordinates will gradually underflow and eventually round to zero. An example of this effect is shown in Figure 4.2 b)

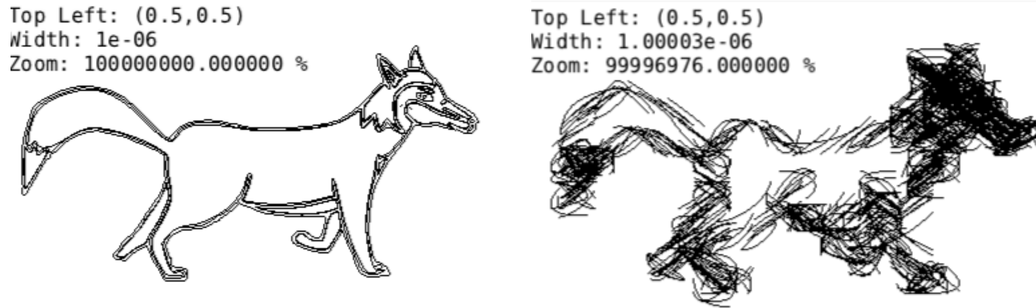


Figure 4.2: The effect of applying cumulative transformations to all Béziers

4.1.3 Applying cumulative transformations to Paths

In Figure 4.1, transformations are applied to the bounds of each Bézier. Figure 4.3 a) shows the effect of introducing an intermediate coordinate system expressing Bézier coordinates relative to the path which contains them. In this case, the rendering of a single path is accurate, but the overall positions of the paths drift as the view is moved.

We can correct this drift whilst maintaining performance by using an arbitrary or high precision number representation to express the coordinates of the paths - but maintaining the floating point coordinates for Bézier curves relative to their path. This is shown in Figure 4.3 b).

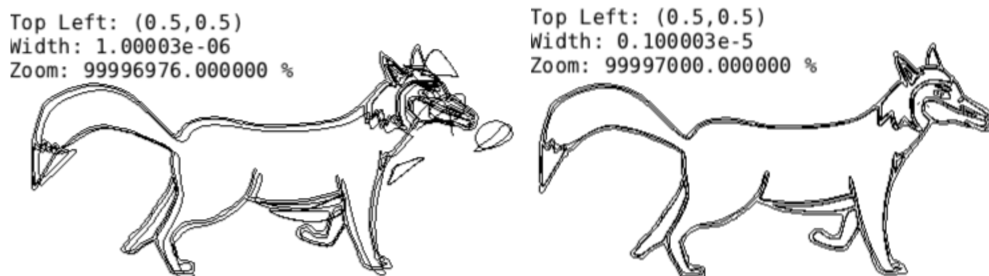


Figure 4.3: Effect of cumulative transformations applied to Paths
a) Path bounds represented using floats b) Path bounds represented using GMP Rationals

4.2 Quantitative Measurements of Rendering Accuracy

To quantitatively measure rendering accuracy, we can record the coordinates of objects in *display space* and measure how these drift as the same collection of objects is added to the document at different view locations. Alternately, since rounding errors causes different coordinates to round to the same value in display space, we may count the number of distinct object bounds in display space.

A useful test SVG is a simple grid of horizontal and vertical lines seperated by 1 pixel. When this SVG is correctly scaled to a view, all that should be visible is a coloured rectangle filling the screen. Increasing the magnification will reveal the grid of lines indicating how the original size of a pixel is scaled.

Figure 4.4 illustrates the effect of applying the view transformation (3.1) directly to the grid, as discussed above in Section 4.1.1. When the grid is correctly rendered, as in Figure 4.4 a) it appears as a black rectangle. Further from the origin , not all pixels in the grid can be represented and individual lines become visible. As the distance from the origin increases, fewer pixel locations can be represented exactly after performing the view transformation.

We should note that with the view top left corner close to $(0, 0)$ as in Figure 4.4 a), detail can be represented more precisely due to the use of IEEE-754 denormals near the origin (see Section 2.5).

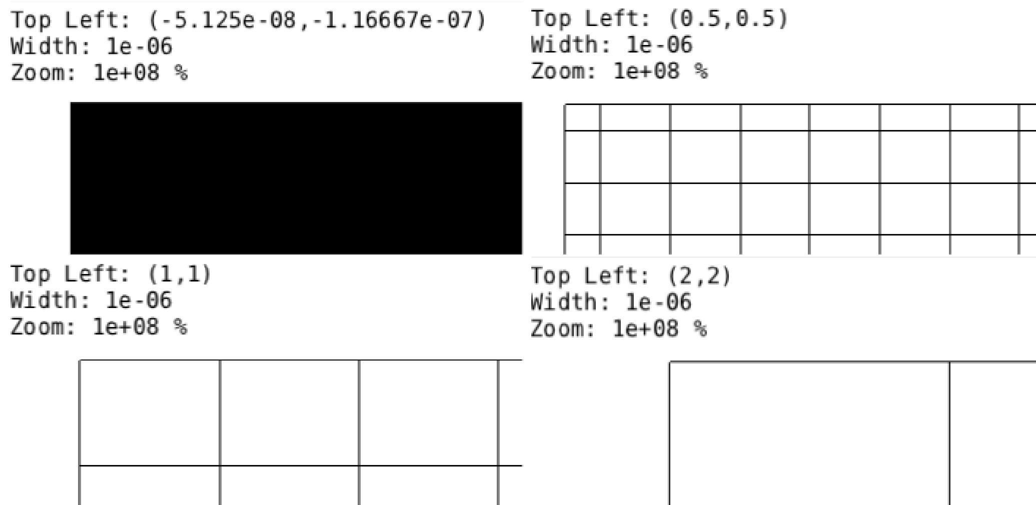


Figure 4.4: Effect of applying (3.1) to a grid of lines seperated by 1 pixel
a) Near origin (denormals) b), c), d) Increasing the exponent of (v_x, v_y) by 1

4.2.1 Precision for Fixed View

By counting the number of distinctly representable lines within a particular view, we can show the degradation of precision quantitatively. The test grid is added to each view rectangle with increasingly smaller width and height.

Figure 4.5 shows how precision degrades with $(V_x, V_y) = (0.5, 0.5)$ for different precision settings using MPFR floating point values to represent the view coordinates. A constant line at 1401 grid

locations indicates no loss of precision. From this figure it should be clear how merely setting the precision of the floating point representation to a higher (but fixed) value will not allow insertion of detail at an arbitrary point; using 1024 bits of precision will still leave no lines representable above magnifications of 10^{300}.

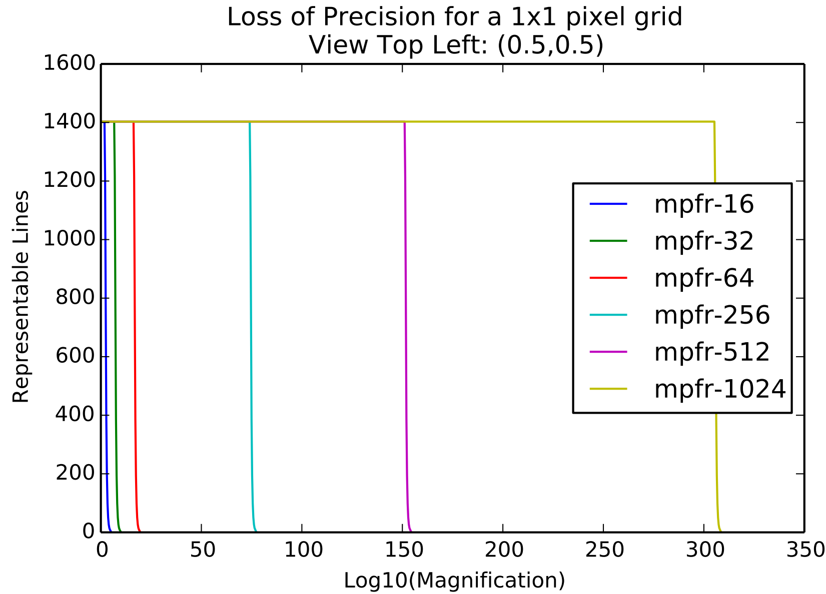


Figure 4.5: Loss of precision of the grid

4.2.2 Accumulated error after changing the View

Using the cumulative transformation approach discussed in Section 4.1.2 means that detail inserted into a fixed view will always render correctly. A fairer test of this approach is to test the rendering accuracy after applying repeated scaling to the document.

Figure 4.6 shows the total error in the coordinates of each line in the grid after the view is scaled (zooming *out*) by repeated transformations. A constant line at 0 would indicate no accumulated error.

In this case, using an arbitrary precision representation such as GMP Rationals (`path-rat`) does not totally eliminate error. This is simply because the final coordinate transformation requires the conversion of rationals to IEEE-754 floats before rendering. Since the total final error for 1042 lines is less than 10^{-2}, and the width of the display is 1, this would represent a negligable difference in the rendering of the grid.

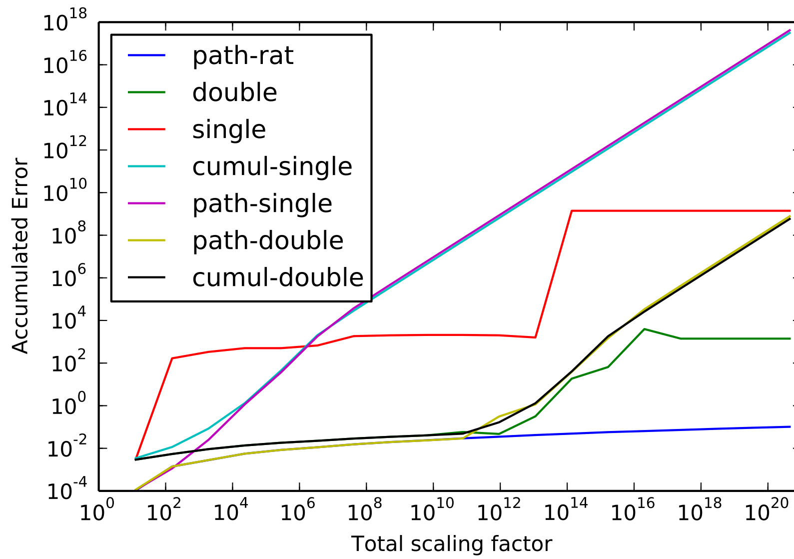


Figure 4.6: Error in the coordinates of the grid **Note:** Logarithmic Axes

4.3 Performance Measurements

4.3.1 Performance of Static Detail at Different View Locations

As discussed above, we succeeded in preserving rendering accuracy as defined above for extremely large ranges of coordinates in the document.

However this comes at a performance cost, as the size of the Rational number representation must grow accordingly. Figures 4.7 a) and b) were obtained by repeatedly resetting the document, scaling, and adding a fixed number of Bézier curves. It appears that the GMP representation increases memory usage linearly, with the speed decreasing faster than linear. The `mpfr-1024` number representation performs much better in terms of a static memory usage and speed; however as discussed in Section 4.2.1, due to the fixed precision it cannot represent detail seperated by a truly arbitrary distance.

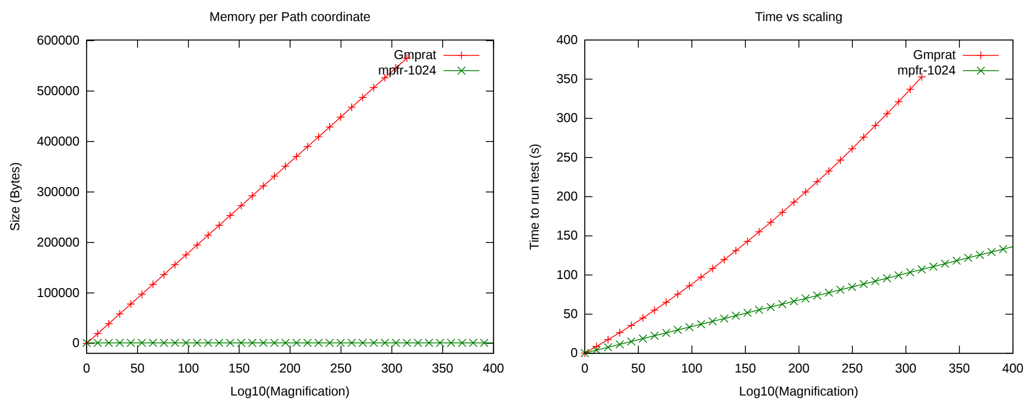


Figure 4.7: a) Memory used per Path coordinate and b) Time taken to scale

4.3.2 Performance whilst adding Detail

For a static document containing only a few imported test SVGs, the use of GMP rationals for path coordinates was not a noticable performance detriment compared to the implementations using floating point coordinates. Figure 4.8 measures the time taken for a script to scale the document to a point at which it will insert an additional copy of a test SVG (Figure 4.9).

We have included the Naïve approach discussed in Section 3.7.1 with GMP rationals (`Gmprat`) and MPFR using 1024 bits of precision (`mpfr-1024`) to illustrate its impracticality. The `Gmprat` is removed from Figure 4.8 b).

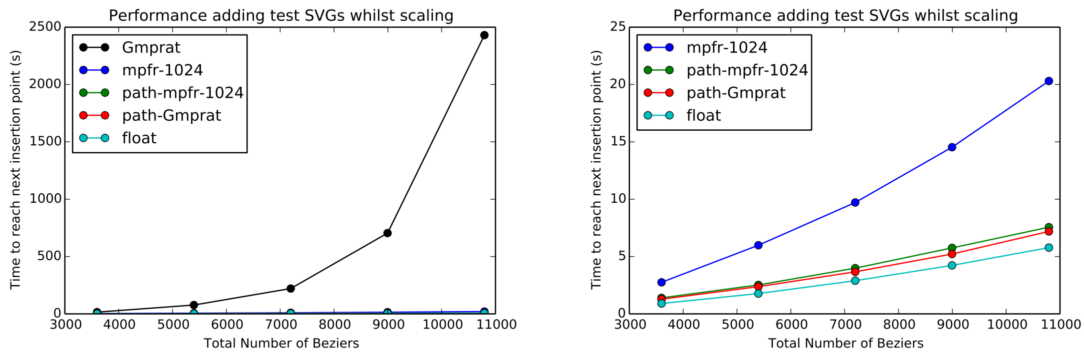


Figure 4.8: a) Performance including Naïve Implementations b) Excluding `Gmprat` data
Legend is in descending order to correspond with the height of the curves

From these results it is clear that our implementation using arbitrary precision arithmetic only for path coordinates is comparable to the straight forward floating point implementation. It is interesting to note that despite Figure 4.7, GMP rationals are slightly faster than MPFR with 1024 bits for this test. This is possibly because the GMP rationals only grow in size as needed, whilst MPFR operations always use the full 1024 bits per number.

4.4 Video Demonstrations

Realtime videos of the IPDF software showing the results presented in this chapter can be found at ⟨http://szmoore.net/ipdf/sam/videos⟩. The performance tests in Section 4.3.2 were taken using the same script running in these videos.

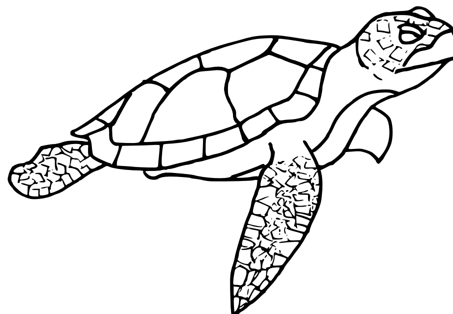


Figure 4.9: The test SVG used to produce the videos

5. Conclusion

5.1 Work Achieved

In this project we have explored and identified the issues limiting precision in vector graphics formats. A primitive such as a Bézier curve can be decomposed into straight lines and rendered on a display to sub-pixel accuracy using IEEE-754 floating point representations, but creating a document using coordinates specified with a floating point representation limits the locations at which detail can be included.

By implementing a proof of concept SVG viewer, we have shown how the approach to applying coordinate transformations affects the accuracy of rendering. Using arbitrary precision GMP rationals to form the coordinates of SVG path elements, expressing Bézier bounds relative to the paths, we have demonstrated the ability to reduce rendering error and include detail accross an extremely large scale. Performance tests show that this implementation is comparable to a straight forward implementation for a very large number of Bézier's in the document.

5.2 Limitations and Future Work

As seen in Figure 4.6 the error for an implementation based on GMP rationals and path coordinates still increases very slowly with scaling, as the final transformation requires conversion to IEEE-754 floats. Our tests have typically been limited to ranges of values represented by IEEE-754 double precision floats, due to time constraints. Future work should address this gradual accumulation of error, particularly outside the range of IEEE-754 values.

Another limitation of our implementation is that only straight lines with starting and ending coordinates in the display will be rendered accurately using Bresenham's algorithm. When "zooming in" to a point within a path, the size of lines increases beyond that of the display, and a rounding error is present. This may be a minor problem unless the view is scaled to an intersection point of lines, in which case the intersection may move. Future work should explore this issue.

The MPFR arbitrary precision floating point implementation allows for individually altering the precision of a number. As an alternative to GMP rationals, which automatically increase in size as needed, algorithms for dynamically increasing the precision of MPFR floats could be explored.

Gow has been exploring a spatial approach to constructing a document, which allows detail to be scaled indefinitely whilst using only IEEE-754 single precision floating point representations [1]. This implementation could be compared to ours in more detail. It may be possible to apply the Quad tree approach to perform sub division of the path based coordinate systems to overcome some of the limitations of both approaches.

Arbitrary precision arithmetic is well understood, but from our review of the literature we have found little evidence of it's application to vector graphics. The well known graphics standards mention precision in passing, if at all. Ideally this work would motivate future document standards which can include detail at potentially arbitrary precision.

References

[1] David Gow. Precision in vector documents: a spatial approach. ⟨http://davidgow.net/stuff/DavidFYPThesis.pdf⟩, 2014.

[2] Adobe Systems Incorporated. *PostScript Language Reference*. Addison-Wesley Publishing Company, 3rd edition, 1985 - 1999.

[3] Michael A. Wan-Lee Cheng. Portable document format (PDF) – finally, a universal document exchange technology. *Journal of Technology Studies*, 28(1):59 – 63, 2002.

[4] Adobe Systems Incorporated. *PDF Reference*. Adobe Systems Incorporated, 6th edition, 2006.

[5] Brian Hayes. Pixels or perish. *American Scientist*, 100(2):106 – 111, 2012.

[6] David G. Barnes, Michail Vidiassov, Bernhard Ruthensteiner, Christopher J. Fluke, Michelle R. Quayle, and Colin R. McHenry. Embedding and publishing interactive, 3-dimensional, scientific figures in portable document format (pdf) files. *PLoS ONE*, 8(9):1 – 15, 2013.

[7] Erik Dahlstóm, Patric Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, Jonathon Watt, Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. Scalable vector graphics (svg) 1.1 (second edition). *W3C Recommendation*, August 2011. Retrieved 2014-05-23.

[8] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[9] David Goldberg. The design of floating-point data types. *ACM Lett. Program. Lang. Syst.*, 1(2):138–151, June 1992.

[10] T Granlund. Gnu mp: The gnu multiple precision arithmetic library. Mar 2014. ⟨http://gmplib.org/gmp-man-6.0.0a.pdf⟩.

[11] Carl Worth and Keith Packard. Xr: Cross-device rendering for vector graphics. In *Linux Symposium*, page 480, 2003.

[12] Kurt E. Brassel and Robin Fegeas. An algorithm for shading of regions on vector display devices. *SIGGRAPH Comput. Graph.*, 13(2):126–133, August 1979.

[13] J. M. Lane and R. and M. Rarick. An algorithm for filling regions on graphics display devices. *ACM Trans. Graph.*, 2(3):192–196, July 1983.

[14] Donald Hearn and M Pauline Baker. *Computer Graphics*. Prentice Hall, Inc, Upper Saddle River, New Jersey 07458, USA, 2 edition, 1997.

[15] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[16] Xiaolin Wu. An efficient antialiasing technique. *SIGGRAPH Comput. Graph.*, 25(4):143–152, July 1991.

[17] Hugo Elias. Graphics. ⟨http://freespace.virgin.net/hugo.elias/graphics/x_main.htm⟩ accessed May 2014.

[18] Donald Knuth. *The METAFONT Book*. Addison-Wesley, 2 edition, 1983.

[19] Donald Knuth. *The TEX Book*. Addison-Wesley, 2 edition, 1983.

[20] Pierre E. Bézier. A personal view of progress in computer aided design. *SIGGRAPH Comput. Graph.*, 20(3):154–159, July 1986.

[21] Ron Goldman. The fractal nature of bezier curves. The de Casteljau subdivision algorithm is used to show that Bezier curves are also attractors (ie: fractals). A new rendering algorithm is derived for Bezier curves.

[22] Nelson Beebe. Extending TEX and METAFONT with floating-point arithmetic. *TUGboat*, 28(3), 2007.

[23] ECMA International. *ECMAScript Language Specification*. ⟨http://www.ecma-international.org⟩ accessed 2014-05-22, 5.1 edition, June 2011.

[24] Moshe Zadka and Guido van Rossum. Unifying long integers and integers. ⟨http://legacy.python.org/dev/peps/pep-0237/⟩, 2007.

[25] Oracle Corporation. java.math.BigInteger. ⟨http://docs.oracle.com/javase/6/docs/api/java/math/BigInteger.htr
Retrieved 2014-05-19.

[26] Kaiyong Zhao and Xiaowen Chu. Gpump: A multiple-precision integer library for gpus. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 1164–1168, June 2010.

[27] Sam Moore and David Gow. Arbitrary sized integers. ⟨http://szmoore.net/ipdf/documents/ArbitraryIntegers.pdf⟩. Incomplete work.

[28] I. Bennett Goldberg. 27 bits are not enough for 8-digit accuracy. *Commun. ACM*, 10(2):105–106, February 1967.

[29] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston Inc., Cambridge, MA, USA, 2010.

[30] IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985*, 1985.

[31] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[32] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[33] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[34] Mark Segal, Kurt Akely, and Jon Leech. *The OpenGL® Graphics System: A Specification*. The Kronos Group, Inc, 2014.

[35] Mathieu Robart. OpenVG paint subsystem over OpenGL ES shaders. In *Consumer Electronics, 2009. ICCE'09. Digest of Technical Papers International Conference on*, pages 1–2. IEEE, 2009.

[36] F Leymarie and Martin D Levine. Fast raster scan distance propagation on the discrete rectangular lattice. *CVGIP: Image Understanding*, 55(1):84–94, 1992.

[37] Sarah F Frisken, Ronald N Perry, Alyn P Rockwood, and Thouis R Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254. ACM Press/Addison-Wesley Publishing Co., 2000.

[38] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses*, pages 9–18. ACM, 2007.

[39] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Transactions on Graphics (TOG)*, 24(3):1000–1009, 2005.

[40] Charles Loop and Jim Blinn. Rendering vector art on the gpu. *GPU gems*, 3:543–562, 2007.

[41] Mark J Kilgard and Jeff Bolz. GPU-accelerated path rendering. *ACM Transactions on Graphics (TOG)*, 31(6):172, 2012.

[42] Mark J Kilgard. Programming with NV path rendering: An annex to the SIGGRAPH paper GPU-accelerated path rendering. *heart*, 300:300.

[43] Karl E Hillesland and Anselmo Lastra. Gpu floating-point paranoia. *Proceedings of GP 2004*, 2004.

[44] Qt Project. Qt project website. ⟨https://qt-project.org/⟩.

[45] Adobe Systems Incorporated. *Adobe Acrobat Reader SDK*, April 2007.

[46] W3C. Extensible markup language (xml) 1.0 (fifth edition). *W3C Recommendation*, November 2008.

[47] W3C. Html5 - developer view - a vocabulary and associated apis for html and xhtml. *W3C Candidate Recommendation*, April 2014.

[48] W3C. Cascading style sheets level 2 revision 1 (css 2.1) specification. *W3CRecommendation*, June 2011.

[49] H Von Koch. Sur une courbe continue sans tangente, obtenue par une construction gomtrique lmentaire. *Archiv fr Matemat., Astron. och Fys.*, pages 681–702, 1904.

[50] W3C. An svg primer for today's browsers. *WC3 Primer (Editor's Draft)*, September 2010.

[51] Thomas Porter and Tom Duff. Compositing digital images. In *ACM SIGGRAPH Computer Graphics*, volume 18, pages 253–259. ACM, 1984.

Appendices

A. An Overview of Document Standards

Together with Section 2.3 this Appendix forms an overview of the well known standard document formats.

The representation of information, particularly for scientific purposes, has changed dramatically over the last few decades. For example, Brassel's 1979 paper on shading polygons[12] has been produced on a mechanical type writer. Although the paper discusses an algorithm for shading on computer displays, the figures illustrating this algorithm have not been generated by a computer, but drawn by Brassel's assistant. In contrast, modern papers such as Barnes et. al's 2013 paper on embedding 3d images in PDF documents[6] can themselves be an interactive proof of concept.

Haye's 2012 article "Pixels or Perish" discusses the recent history and current state of the art in documents for scientific publications[5]. Hayes argued that there are currently two different approaches to representing a document: As a sequence of commands for producing an image on a static sheets of paper (Interpreted Model) or as a dynamic and interactive way to convey information, using the Document Object Model.

A.1 Interpreted Models

Adobe's PostScript Language Reference Manual defines a turing complete language for producing graphics output on an abstract "output device"[2]. A PostScript document is treated as a procedural program; an interpreter executes instructions in the order they are written by the programmer. In particular, the document specifies the locations of enclosed curves using Bézier splines (Section 2.2.2), whilst text is treated as vector fonts described in Section 2.2.3. PostScript was and is still widely used in printing of documents onto paper; many printers execute postscript directly, and newer formats including PDF must still be converted into PostScript by printer drivers[4, 3].

Adobe's Portable Document Format (PDF) is currently used almost universally for sharing documents; the ability to export or print to PDF can be found in most graphical document editors and even some plain text editors[3].

Hayes describes PDF as "... essentially 'flattened' PostScript; its whats left when you remove all the procedures and loops in a program, replacing them with sequences of simple drawing commands."[5]. Consultation of the PDF 1.7 standard shows that this statement does not a give a complete picture — despite being based on the Adobe PostScript model of a document as a series of "pages" to be printed by executing sequential instructions, from version 1.5 the PDF standard began to borrow some ideas from the Document Object Model.

For example, interactive elements such as forms may be included as XHTML objects and styled using CSS. "Actions" are objects used to modify the data structure dynamically. In particular, it is possible to include Javascript Actions. Adobe defines the API for Javascript actions seperately to the PDF standard[45]. There is some evidence in the literature of attempts to exploit these features, with mixed success[6, 5].

Figure A.1 shows a vector image and one possible way to express this image in PostScript.

There are some limitations in PostScript's model. As mentioned in Section A.3, since PostScript predates Porter and Duff Compositing, there is no concept of transparency. In fact, using tools to

convert between the SVG image in Figure A.2 and PostScript will simply rasterise the image and embed the rastered image in PostScript[1]

Another limitation of PostScript is that the model of a document as a static page, convenient for printers which literally produce static pages, is unable to include interactive or dynamic elements. Dynamic PostScript attempted to fix this problem, but "never caught on"[5].

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 0 -1 85 150
% These lines are comments to aid in human understanding
% Define an operator to produce a rectangular path
/re { exch dup neg 3 1 roll 5 3 roll moveto 0 rlineto
      0 exch rlineto 0 rlineto closepath } bind def
% Operator to produce the path for the first rectangle
/re1 { 24.613 133.001 24 -120 re } bind def
% Operator to produce the path for the second rectangle
/re2 { 10.215 45.001 48 -16 re } bind def
% Operator which will produce the curved path
/curve { 46.215 1.001 moveto
        46.215 1.001 91.812 11.399 71.812 35.399 curveto
        51.812 59.399  29.414 33.802 51.812 59.399 curveto
        74.215 85.001 93.414 45.802 74.215 85.001 curveto
        55.016 125.001 61.414 49.802 46.215 75.399 curveto
        31.016 101.001 56.613 126.598 56.613 126.598 curveto
        56.613 126.598 88.613 166.598 56.613 137.802 curveto
        24.613 109.001 -18.586 83.399 9.414 50.598 curveto
        37.414 17.802 45.414 1.001 45.414 1.001 curveto
closepath } bind def
% Set stroke properties
0.8 setlinewidth 0 setlinecap 0 setlinejoin []
        0.0 setdash 4 setmiterlimit
% Draw the straight line
0 setgray 0.613 149.001 moveto 83.812 0.2 lineto fill
% Fill and outline the first rectangular path
0 0 1 setrgbcolor re1 fill 0 setgray re1 stroke
% Fill and outline the curved shape
1 0 0 setrgbcolor curve fill 0 setgray curve stroke
% Fill and outline the second rectangle
0 1 0 setrgbcolor re2 fill 0 setgray re2 stroke
showpage
```

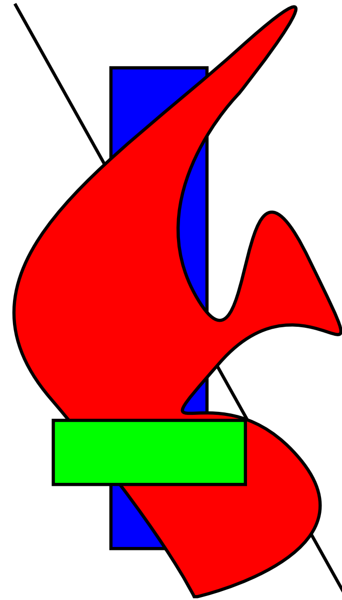


Figure A.1: Vector image and a possible PostScript representation

TeX, METAFONT and LaTeX

Knuth's "The TeXbook"[19] and "The METAFONT book"[18] define two complementary programming languages for typesetting documents. Wheras PostScript may be considered an interpreted language, in that it can be produced in a human readable form which is also readable by an interpreter, TeX is a compiled language; a program parses human readable TeX to produce a machine readable format DVI ("DeVice Independent"). A DVI interpreter might be thought of as a virtual "Display Processor" for drawing vector graphics directly (as defined in the earlier editions of "Computer Graphics"[14]).

DVI itself is not a widely used format for sharing documents. However, an system based upon TeX called LaTeX which includes libraries for advanced typesetting and programs that ultimately produce PDF output is particularly popular for producing technical reports and papers[2] — this report itself has been produced using the CTAN LaTeX packages[3].

A.2 The Document Object Model

The Document Object Model (DOM) represents a document as a tree like data structure with the document as a root node. The elements of the document are represented as children of either

[1]For Figure A.2 converted using the Inkscape SVG editor: ⟨http://szmoore.net/ipdf/figures/shape-svg-converted-to.ps⟩
[2]The site ⟨http://tex.stackexchange.com⟩ (accessed 2014-05-22) is devoted to TeX and LaTeX
[3]The complete TeX source code to produce this document can be found at ⟨http://szmoore.net/ipdf/sam/⟩

this root node or of a parent element. In addition, elements may have attributes which contain information about that particular element.

The World Wide Web Consortium (W3C) is an organisation devoted to the development of standards for structuring and rendering web pages based on industry needs. The DOM is used in and described by several W3C recommendations including XML[46], HTML[47] and SVG[7]. XML is a general language which is intended for representing any tree-like structure using the DOM, whilst HTML and SVG are specifically intended for representing text documents and more general graphics respectively. These languages make use of Cascading Style Sheets (CSS)[48] for specifying the appearance of elements.

Version 5 of the Hypertext Markup Language (HTML5) is currently a candidate recommendation which aims to standardise the state of the art in technologies relating to web based documents. In HTML5 it is possible to achieve almost any level of control over both the structure and rendering of a document desirable. In particular, the language Javascript (based upon ECMAScript [23]) can be used to dynamically alter a HTML5 document in response to user input or other events, including communication with HTTP servers.

The Scalable Vector Graphics (SVG) recommendation defines a language for representing vector images using the DOM. This is intended not only for stand alone images, but also for inclusion within HTML documents. In the SVG standard, each graphics primitive is an element in the DOM, whilst attributes of the element give information about how the primitive is to be drawn, such as path coordinates, line thickness, mitre styles and fill colours.

In the SVG representation, general shapes can be specified by locations of enclosed curves using Bézier splines (Section 2.2.2) - the construction of these curves is very similar to PostScript (refer to Figure 2.4). Again, text is created using vector fonts as described in Section 2.2.3.

Figure A.2 shows an example of an SVG image as rendered (left) and represented as text. The textual representation is syntactically a subset of XML and is similar to HTML.[4] Here we have used `<rect>` elements to position rectangles and `<path>` elements to define a straight line and a filled region bounded by a cubic bezier spline; note that the points and type of curves are defined as a data attribute.

A.2.1 Javascript and the DOM

Using Javascript, an element in the DOM can be selected by its type, class, name, or unique identifier, each of which may be specified as an attribute in the original DOM. Once an element is selected Javascript can be used to modify its attributes, add children below it in the DOM, or remove it from the DOM entirely.

For example, the following Javascript acting on the DOM described in Figure A.2 will change the fill colour of the curved region.

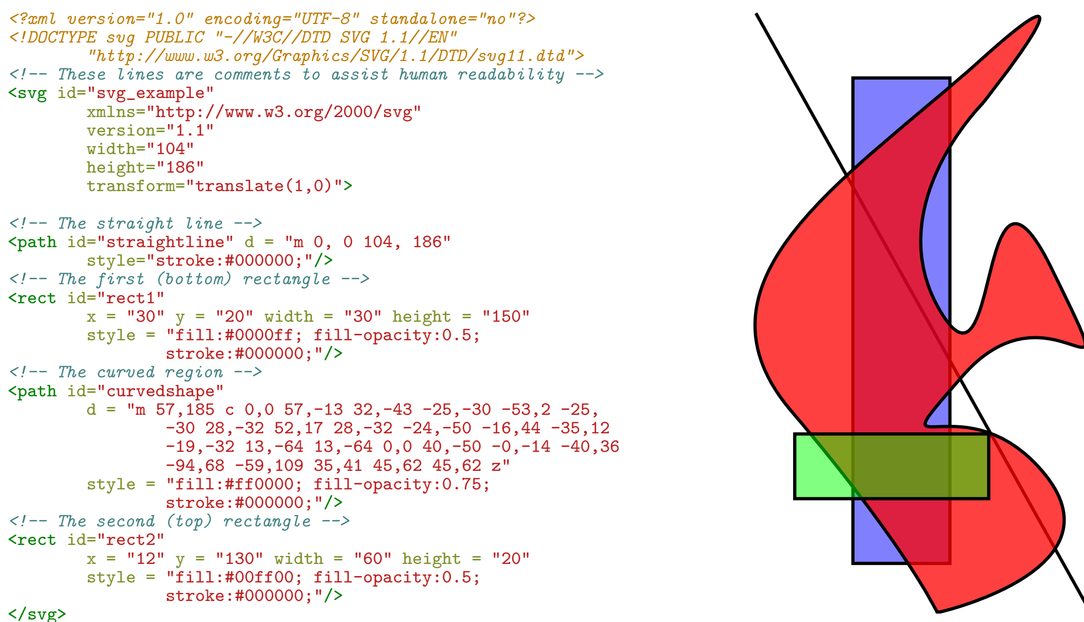
```
var node = document.getElementById("curvedshape"); // Find the node by its unique id
node.style.fill = "#000000"; // Change the ``style'' attribute and set the CSS fill colour
```

To illustrate the power of this technique we have produced an example to generate an SVG

[4]The details of distinctions between these languages are beyond the scope of this report.

interactively using HTML. The example generates successive iterations of a particular type of fractal curve first described by Koch[49] in 1904 and a popular example in modern literature [21]. Unfortunately as including W3C HTML directly in a standard PDF is not possible, we are only able to provide some examples of the output as static images in Figure A.3. The W3C has produced a primer describing the use of HTML5 and Javascript to produce interactive SVG's[50], and the HTML5 and SVG standards themselves include several examples.

In HTML5, Javascript is not restricted to merely manipulating the DOM to alter the appearance of a document. The `<canvas>` tag and associated API provide a means to directly set the values of pixels on a display. This sort of low level API is inteded for performance intensive graphical applications such as web based games[5]. As Hayes points out, there is some similarity between the `<canvas>` API, the SVG path descriptions and the PostScript interpreted approach to drawing[5].



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
        "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<!-- These lines are comments to assist human readability -->
<svg id="svg_example"
        xmlns="http://www.w3.org/2000/svg"
        version="1.1"
        width="104"
        height="186"
        transform="translate(1,0)">

<!-- The straight line -->
<path id="straightline" d = "m 0, 0 104, 186"
        style="stroke:#000000;"/>
<!-- The first (bottom) rectangle -->
<rect id="rect1"
        x = "30" y = "20" width = "30" height = "150"
        style = "fill:#0000ff; fill-opacity:0.5;
                stroke:#000000;"/>
<!-- The curved region -->
<path id="curvedshape"
        d = "m 57,185 c 0,0 57,-13 32,-43 -25,-30 -53,2 -25,
                -30 28,-32 52,17 28,-32 -24,-50 -16,44 -35,12
                -19,-32 13,-64 13,-64 0,0 40,-50 -0,-14 -40,36
                -94,68 -59,109 35,41 45,62 45,62 z"
        style = "fill:#ff0000; fill-opacity:0.75;
                stroke:#000000;"/>
<!-- The second (top) rectangle -->
<rect id="rect2"
        x = "12" y = "130" width = "60" height = "20"
        style = "fill:#00ff00; fill-opacity:0.5;
                stroke:#000000;"/>
</svg>
```

Figure A.2: Vector image and a possible SVG representation

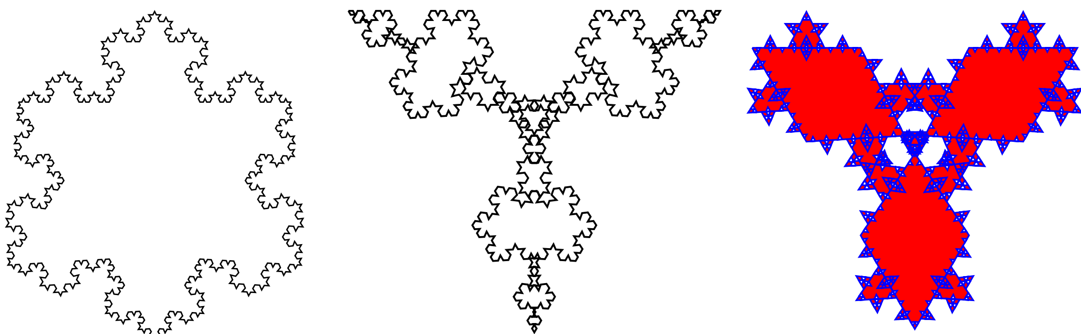


Figure A.3: Koch "snowflakes" generated using Javascript to modify an SVG DOM. The interactive HTML5 document can be found at ⟨http://szmoore.net/ipdf/sam/figures/koch.html⟩

[5]For an example by the author including both the canvas2d and experimental WebGL APIs see ⟨http://rabbitgame.net⟩

A.3 Compositing

Colour raster displays are based on an additive red-green-blue (r, g, b) colour representation which matches the human eye's response to light[14]. In 1984, Porter and Duff introduced a fourth colour channel for rasterised images called the "alpha" channel, analogous to the transparency of a pixel[51]. In compositing models, elements can be rendered seperately, with the four colour channels of successively drawn elements being combined according to one of several possible operations.

In the "painter's model" as described by the SVG standard the "over" operation is used when rendering one primitive over another[7]. Given an existing pixel P_1 with colour values (r_1, g_1, b_1, a_1) and a pixel P_2 with colours (r_2, g_2, b_2, a_2) to be painted over P_1, the resultant pixel P_T has colours given by:

$$a_T = 1 - (1 - a_1)(1 - a_2) \tag{A.1}$$

$$r_T = (1 - a_2)r_1 + r_2 \quad \text{(similar for } g_T \text{ and } b_T\text{)} \tag{A.2}$$

It should be apparent that alpha values of 1 correspond to an opaque pixel; that is, when $a_2 = 1$ the resultant pixel P_T is the same as P_2. When the final pixel is actually drawn on an rgb display, the (r, g, b) components are $(r_T/a_T, g_T/a_T, b_T/a_T)$.

The PostScript and PDF standards, as well as the OpenGL API also use a painter's model for compositing. However, PostScript does not include an alpha channel, so $P_T = P_2$ always[2]. Figure A.2 illustrates the painter's model for partially transparent shapes as they would appear in both the SVG and PDF models.