Number Representations and Precision in Vector Graphics

Author: Samuel Moore[1]
Partners: David Gow[2]
Supervisors: Prof Tim French, Dr Rowan Davies

THE UNIVERSITY OF WESTERN AUSTRALIA

Achieve International Excellence

October 10, 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⟩

Contents

List of Figures

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[3]. The ubiquitous Portable Document Format (PDF) standard provides many enhancements to PostScript taking into account desktop publishing requirements[4], but it is still fundamentally based on the same imaging model[5]. This idea of a document as a static "page" has lead to limitations on what could be achieved with a digital document viewers [6].

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[3, 5, 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.

We have implemented a proof of concept document viewer compatable with a subset of the SVG standard, which has allowed us to explore the limitations of floating point arithmetic and possible approaches to achieving arbitrary precision document formats. Using the Rational representation of the GNU Multiple Precision (GMP) library[**?**] we are able to implement correct rendering of SVG test images seperated by arbitrary distances. We demonstrate the trade off between performance cost and the accuracy of rendering

2. Literature Review

An overview will go here.

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 (see Section **??**).

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[10]. 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.

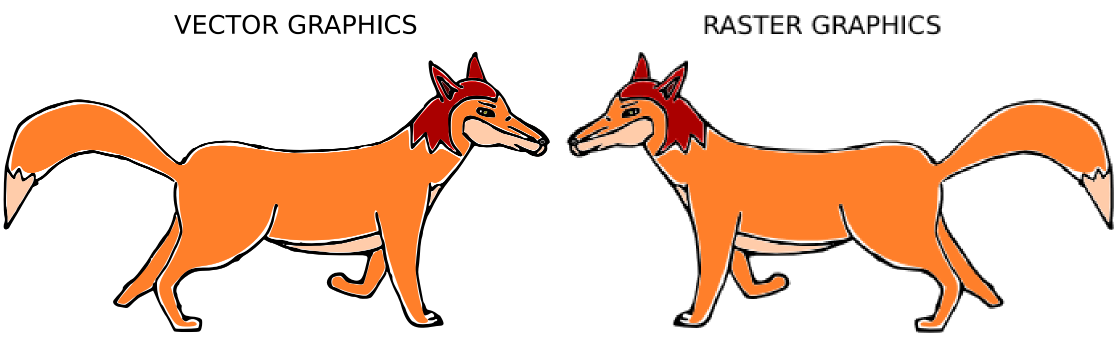

Figure 2.1: Original Vector and Raster Images

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

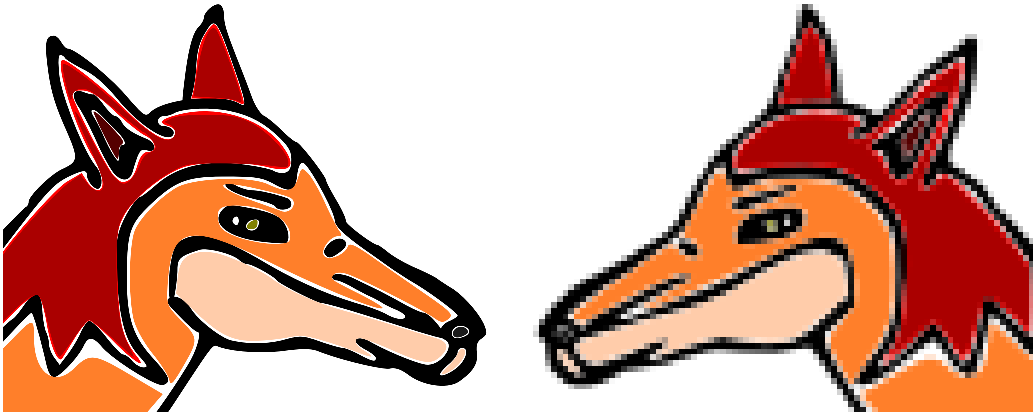

Figure 2.2: Scaled Vector and Raster Images

2.2 Rendering Vector Primitives

Hearn and Baker's textbook "Computer Graphics"[11] 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.

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[11].

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 costly floating point operations and rounding (See Section**??**). Modifications based on computing steps Δx and Δy eliminate the multiplication but are still less than ideal in terms of performance[11].

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[14]. 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[15].

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[16]. 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[15][2]. .

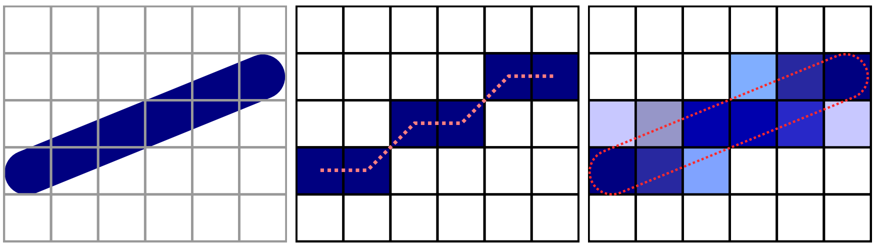

Figure 2.3: Rasterising a Straight Line

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[3], PDF[5] and SVG[7] standards which we will discuss in Section **??**. Cubic Béziers are also used to define vector fonts for rendering text in these standards and the TEX typesetting language [17, 18]. 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 computer aided design of automobile bodies[19].

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

[2]Techniques for antialiasing primitives other than straight lines are discussed in some detail in Chapter 4 of "Computer Graphics" [11]

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 **??**.

De Casteljau's algorithm of 1959 is often used for decomposing Béziers into line segments[11, 17]. 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[20]:

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

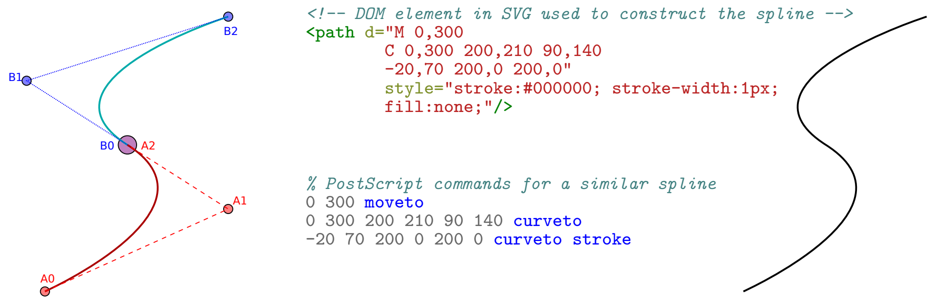

Figure 2.4: Constructing a Spline from two cubic Béziers
(a) Showing the Control Points (b) Representations in SVG and PostScript (c) Rendered Spline

2.2.3 Filled Paths

2.2.4 Compositing

FIXME Really won't have time to mention these? They are important, but we didn't end up implementing them anyway.

2.2.5 Fonts

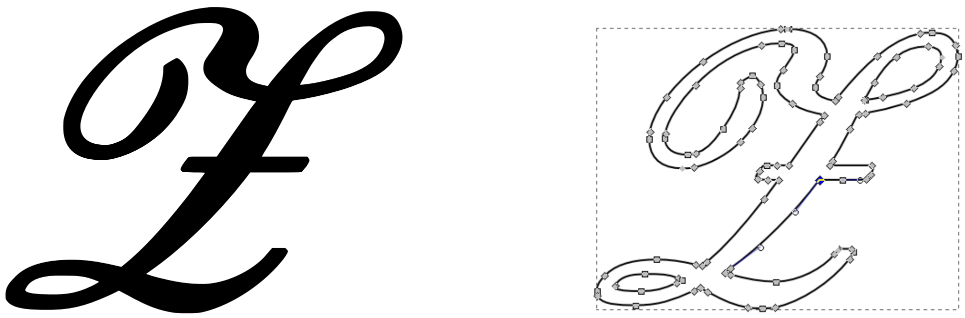

Figure 2.5: a) Vector glyph for the letter Z b) Screenshot showing Bézier control points in Inkscape

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[17]. 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[17]. Figure 2.5 shows a $\mathscr{Z}$ (script Z) produced by LATEX with Bézier cubics identified.

2.3 Coordinate Systems and Transformations

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 ??. A point (X, Y) in the document will transform to a point (x, y) in the view by:

$$X = \frac{x - v_x}{v_w} \qquad Y = \frac{y - v_y}{v_h} \tag{2.7}$$

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 $Z = 1$

$$\boldsymbol{X} = \boldsymbol{V}\boldsymbol{x} \tag{2.8}$$

$$\begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{v_w} & 0 & \frac{v_x}{v_w} \\ 0 & \frac{1}{v_h} & \frac{v_y}{v_h} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{2.9}$$

Moving the mouse[3] by a distance $(\Delta x, \Delta y)$ relative to the size of the view should translate it by the same amount[]:

$$v_x \rightarrow v_x + \Delta x \tag{2.10}$$

$$v_y \rightarrow v_y + \Delta y \tag{2.11}$$

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)[]:

$$v_x \rightarrow v_x + x_0 v_w (1 - s) \tag{2.12}$$

$$v_y \rightarrow v_y + y_0 v_h (1 - s) \tag{2.13}$$

$$v_w \rightarrow s v_w \tag{2.14}$$

$$v_h \rightarrow s v_h \tag{2.15}$$

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”.

TODO

- Intermediate coordinate systems...
- Write Matrix operations properly
- Link with the results where applying (2.7) directly leads to disaster
- This is because for $v_w << 1$, an error of $1ulp$ in $x - v_x$ is comparable with v_w, ie: Can increase to the order of the size of the display (or more)

2.4 Precision Specified by Document Standards

FIXME: Most of this stuff should probably be appendicised

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

[3] or on a touch screen, swiping the screen

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[21] 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[6]. 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.

2.4.1 Interpreted Models: PostScript and PDF

Adobe's PostScript Language Reference Manual defines a turing complete language for producing graphics output on an abstract "output device"[3]. 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 **??**), whilst text is treated as vector fonts described in Section **??**. 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[5, 4].

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[4].

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."[6].

2.4.2 The Document Object Model: SVG

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 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[22], HTML[23] 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)[24] for specifying the appearance of elements.

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 **??**) - the construction of these curves is very similar to PostScript (refer to Figure **??**). Again, text is created using vector fonts as described in Section **??**.

2.4.3 Precision Specified By Standards

TODO: Keep this subsection, appendicise rest of this section

2.4.4 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"[3]. There is no reference to the precision of mathematical operations, but the implementation limits *suggest* a range of $\pm10^{38}$ "approximate" and the smallest values not rounded to zero are $\pm10^{-38}$ "approximate".

2.4.5 PDF

PDF defines "Real" objects in a similar way to PostScript, but suggests a range of $\pm3.403 \times 10^{38}$ and smallest non-zero values of $\pm1.175 \times 10^{38}$[5]. 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".

2.4.6 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"[17]. [4] 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[25]. Beebe also suggested that TEX and METAFONT could now be modified to use IEEE-754 arithmetic.

2.4.7 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

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

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.

2.4.8 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, $2^64 -^5 3 + 3$) values, representing the double-precision 64-bit format IEEE 754 values as specified in the IEEE Standard for Binary Floating-Point Arithmetic"[26]. 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.

2.5 Fixed Point and Integer Number Representations

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

$$z = \sum_{i=-\infty}^{\infty} d_i \beta^i \tag{2.16}$$

Where each digit $d_i < \beta$ the base. 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 (2.16).

To express a real number using equation (2.16) 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 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):

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

FIXME Add Maths reference (Cantor's Diagonal argument) without going into all the Pure maths details

2.6 Floating Point Number Representations

Whilst a Fixed Point representation keeps the "point" (the location considered to be $i = 0$ in (2.16)) 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.

FIXME: Cite properly The use of floating point arithmetic in computer systems was pioneered by Knuth[], Goldberg[27], Dekker[], and others, but modern systems are largely compatable with the IEEE-754 standard pioneered by William Kahan in 1985 [28] and revised (also with contributions from Kahan) in 2008[29].

A floating point number x is commonly represented by a tuple of values (s, e, m) in base B as[30, 31]: $x = (-1)^s \times m \times B^e$

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[30]. Early computers had in fact used a variety of representations including $B = 3$ or even $B = 7$[**?**], and the revised IEEE-754 standard specifies a decimal representation $B = 10$ intended for use in financial applications[29][5]. From now on we will restrict ourselves to considering base 2 floats.

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[27].

2.6.1 Visualisation of Floating Point Representation

Figure **??** shows the positive real numbers which can be represented exactly by an 8 bit floating point number encoded in the IEEE-754 format. 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**??**. 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 as straight lines - the distance between points is always constant.

[5] Eg: The smallest valid unit of currency \$0.01 could not be represented exactly in base 2

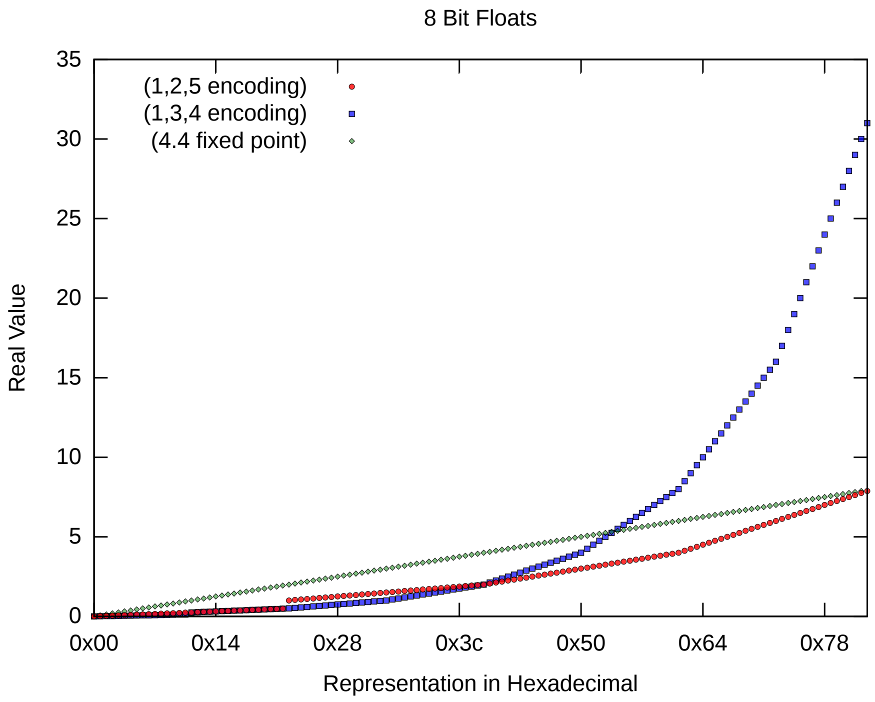

Figure 2.6: Positive 8-Bit Number Representations

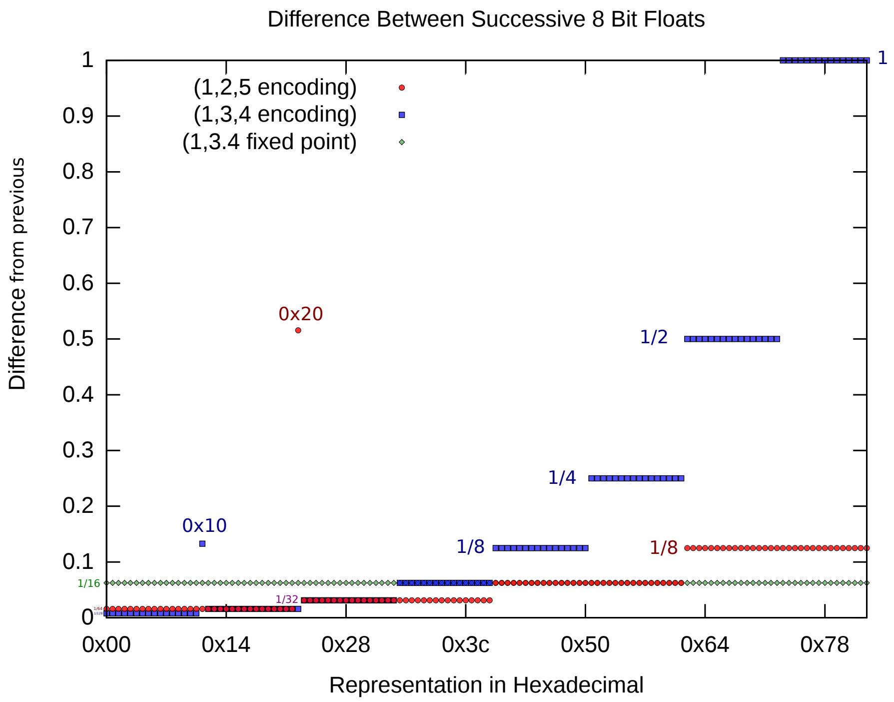

Figure 2.7: Difference between successive numbers

2.6.2 Floating Point Operations

FIXME: Appendix?

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. Referring to Figure **??** 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].

Put this stuff in an Appendix?

2.6.3 Addition and Subtraction

According to the IEEE-754 standard, if $e_1 < e_2$, then the preferred form of $f_1 + f_2$ is:

$$m_1\beta^{e_1} \pm m_2\beta^{e_2} = (m_1 \pm \beta^{e_2-e_1}m_2)\beta^{e_1} \tag{2.17}$$

This is equivelant to shifting the fixed point in m_2 by $e_2 - e_1$ to the left, and then performing fixed point addition or subtraction. If the result of the addition/subtraction requires a carry/borrow, divide result by β (ie: shift digits by 1 the right) and increment/decrement exponent. Then normalise the result (subtract leading zeros in mantissa from the exponent). Lastly perform the rounding operation; if this would generate a carry/borrow, shift right and increment/decrement exponent again, repeat.

2.6.4 Multiplication and Division

$$m_1\beta^{e_1} \times m_2\beta^{e_2} = (m_1 \times m_2)\beta^{e_1+e_2} \tag{2.18}$$

$$m_1\beta^{e_1} \div m_2\beta^{e_2} = (m_1 \div m_2)\beta^{e_1-e_2} \tag{2.19}$$

Multiplication and Division are not inverses.

Floating point operations can in principle be performed using integer operations, but specialised Floating Point Units (FPUs) are an almost universal component of modern processors[32]. The improvement of FPUs remains highly active in several areas including: efficiency[33]; accuracy of operations[34]; and even the adaptation of algorithms originally used in software, such as Kahan's Fast2Sum algorithm[35].

2.6.5 Arbitrary Precision Floating Point Numbers

Arbitrary precision floating point numbers are implemented in a variety of software libraries which will dynamically allocate extra bits for the exponent or mantissa as required. An example is the GNU MPFR library discussed by Fousse in 2007[36]. 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.

As we have seen, it is trivial to find real numbers that would require an infinite number of bits to represent exactly. Implementations of "arbitrary" precision must carefully determine at what point rounding should occur so as to balance performance with memory usage.

2.7 Rational Number Representations

$$Q = \frac{N}{D} \tag{2.20}$$

- N and D are arbitrary precision integers

$$N = \sum_{i=0}^{S} d_i \beta^i \tag{2.21}$$

- d_i are fixed size integers, $\beta = 2^{64}$
- Size S grows as needed
- Operations are always exact
- Implemented by GNU Multiple Precision Library

2.8 Floating Point Operations on the CPU and GPU

FIXME: I feel this section is important but I'm not quite sure where to place it; it could almost work as a paper by itself (in fact I sort of wrote one for it already...)

Traditionally, vector images have been rasterized by the CPU before being sent to a specialised Graphics Processing Unit (GPU) for drawing[11]. Rasterisation of simple primitives such as lines and triangles have been supported directly by GPUs for some time through the OpenGL standard[37]. 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[37] API[38, 39, 40, 41, 42, 43]. Recently Mark Kilgard of the NVIDIA Corporation described an extension to OpenGL for NVIDIA GPUs capable of drawing and shading vector paths[44, 45]. 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[6]. Although the OpenGL API does use IEEE-754 number representations, research by Hillesland and Lastra in 2004 suggested that many GPUs were not internally compliant with the standard[46].

In order to explore this, we implemented a simple fragment shader to render a circle. Points $x^2+y^2 < 1$ should be black. When scaled to bounds of width $\approx 10^{-6}$ the edges of the circle become jagged due to imprecision. However, the behaviour is quite different depending on GPU model. A CPU renderer was also implemented to evaluate the same function using IEEE-754 singles.

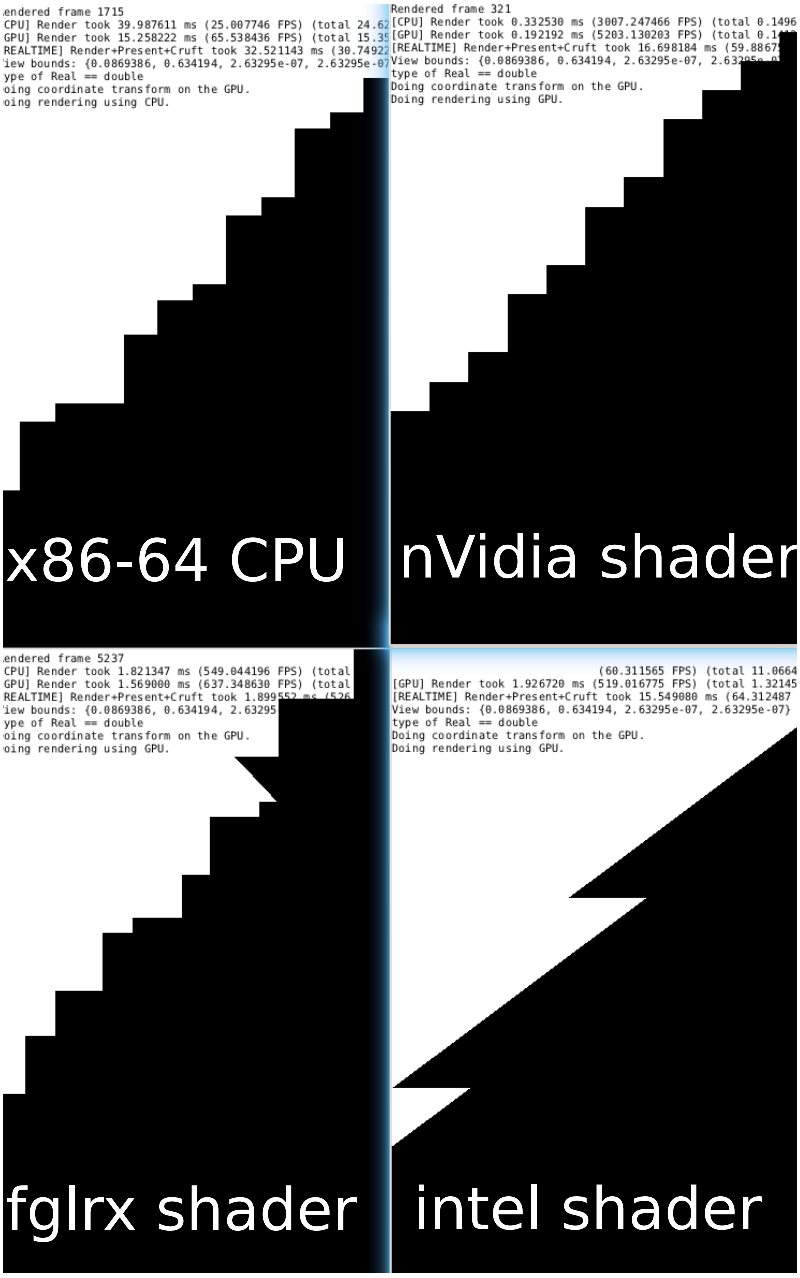

Figure 2.8: Difference in evaluating $x^2 + y^2 < 1$ for the x86_64 and various GPUs
The view bounds are identical

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

3. Methods and Design

TODO Write most of this section. I suspect I will have to be very selective about what to fit in considering the word limit.

3.1 Collaborative Process

- Collaborated with David Gow on the design and implementation of the SVG viewer
- Individual work: Applying GMP Rationals (Sam), Quadtree (David)
  - CPU renderer, SVG parsing, Control Panel, Python Scripts - Sam
  - Most of the OpenGL stuff, Scaling/Translating controls - David
  - Other parts were worked on by everyone
- Used git to collaborate ⟨https://git.ucc.asn.au⟩
- Used preprocessor defines to not interfere with each other's code too much
- David used a `goto` letting the team down

3.2 Structure of Software

- CPU and GPU renderer supported
  - See figure in "Floating Point Operations on the CPU and GPU"
- Rendering of Cubic Béziers (no antialiasing)
- Partial implementation of shading Paths on CPU (abandoned)
- Ability to move the view around the document with the mouse
- Ability to insert an SVG into the view location
- `typedef` for number representations
- Ability to control program through scripts or stdio
- Hacky python scripts to produce plots by abusing this

3.3 Approaches to Arbitrary Precision

- Replace *all* operations with arbitrary precision (ie: Rationals) - Horrendously slow
- Change approach to applying coordinate transform (2.7)
- Apply view transformations directly to objects as the view is transformed, rather than just before rendering
  - Allows much better precision and range with just regular IEEE-754 floats

  - But there is an accumulated rounding error, particularly when zooming out and back in, which is bad
- As above, but introduce intermediate coordinate system; use the Path elements
  - Rendering of individual paths is consistent but overall they drift apart
- As above, but specify Path coordinates with arbitrary precision rationals
  - Works well, rationals slow down though

3.4 Number Representations Trialed

- IEEE-754 single, double, extended
- Custom implementation of Rationals with `int64_t`
  - Very limited since the integers grow exponentially and overflow
- Custom implementation of Rationals with custom Arbitrary precision integers
  - Actually works
  - Implementation of division is too slow to be feasible
- Custom rationals but with GMP arbitrary precision integers
  - Our implementation of GCD is not feasible
- Paranoid Numbers; store a operation tree of IEEE-754 floats and simplify the tree wherever `FE_INEXACT` is *not* raised
  - This was a really, really, really, bad idea
- Just use GMP rationals already
  - Works
- MPFR floats
  - They work, but they don't truly give arbitrary precision
  - Because you have to specify the maximum precision
  - However, this can be changed at runtime
  - Future work: Trial MPFR floats changing the precision as needed

3.5 Libraries Used

- SDL2 - Simple Direct media Library
  - Used for window management and to obtain an OpenGL context
  - Also provides BMP handling which is useful
- Qt4 (optional)

  - Open source toolkit for Dialog based applications
  - We can optionally compile with a Qt4 based control panel
  - This is useful for interacting with the document
  - Has way more features than we actually use it for
- OpenGL - Standard API for rendering on GPUs
  - Using GLSL shaders
  - Béziers are rendered using a Geometry shader which produces line segments
- PugiXML - Open source XML parsing library
  - Used to parse SVGs
- GNU Multiple Precision (GMP)
  - Implements arbitrary precision integers, floats, and rationals
  - We can use the arbitrary precision integers with a custom rational type
  - Or just use the GMP rational type (much better)
  - We don't use the floats, because they are hardware dependent
- MPFR
  - MPFR is built on GMP but ensures IEEE-754 consistent rounding behaviour
  - (Not hardware dependent)
  - We can compile with MPFR floats but the precision is currently fixed at compile time

3.6 Design of Performance Tests

- This is mostly covered in the Results chapter
- Control the program through stdin using a python script
- Results plotted with matplotlib

4. Results and Discussion

Note: Need to be more consistent, I often refer to Béziers and Objects interchangably (since the original design was based around an Object and Bézier was just one possible Object, but we have moved on to pretty much only caring about Béziers now)

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.

4.1.1 Applying the view transformation directly

Figure 4.1 shows the rendering of a vector image[1]. Transformation (2.7) is applied to object coordinates with default IEEE-754 rounding behaviour (to nearest). The loss of precision in the second figure is obvious. This is because division by 10^{-6} increases the rounding error in $x - v_x$, by 10^6, so the total error is of the order 10^6 ulp which is of the order 0.25

TODO: Calculate that properly, shouldn't be hard

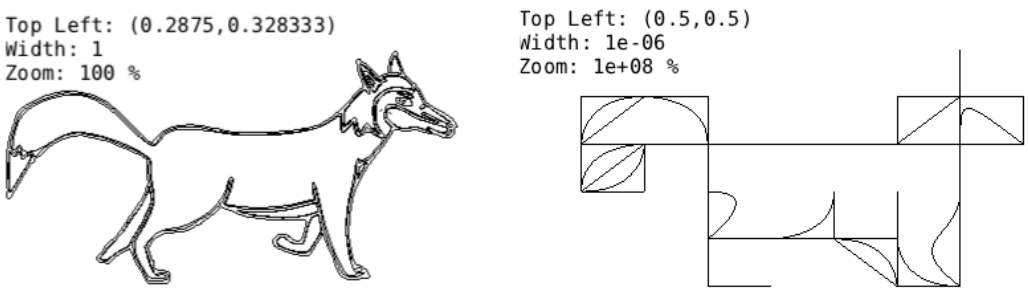

Figure 4.1: The vector image from Figure 2.1 under two different scales

4.1.2 Applying cumulative transformations to all Béziers

Rather than applying (2.7) 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 transformations (**??**) and (**??**) 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)

[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

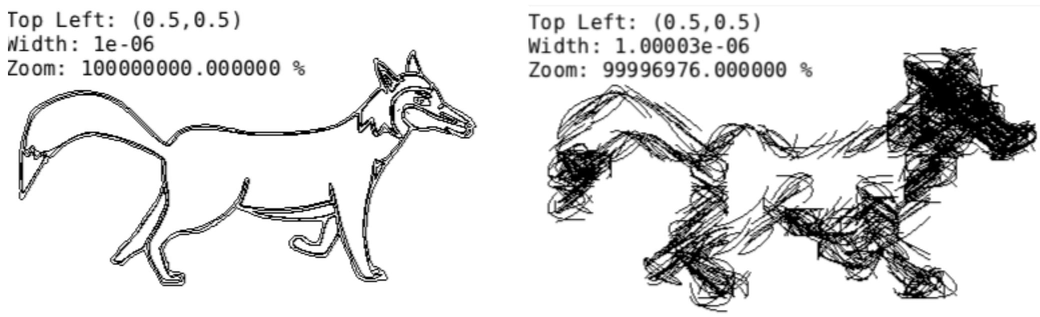

Figure 4.2: The effect of applying cumulative transformations to all Béziers

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. As we will discuss in Section **??**, this offers an acceptable trade off between rendering accuracy and performance.

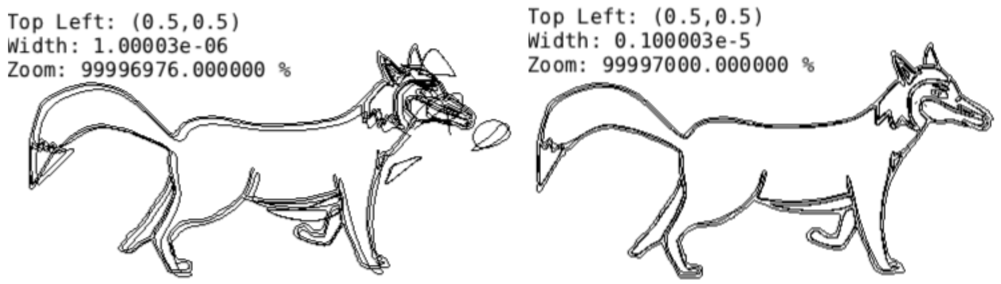

Figure 4.3: Effect of cumulative transformations applied to Paths
a) Path bounds represented using floats b) Path bounds represented using Rationals

4.2 Quantitative Measurements of Rendering Accuracy

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 (2.7) directly to the grid. 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.

An error of 1 ulp is increased by a factor of 10^6 to end up comparable to the size of the display $(0 \to 1)$.

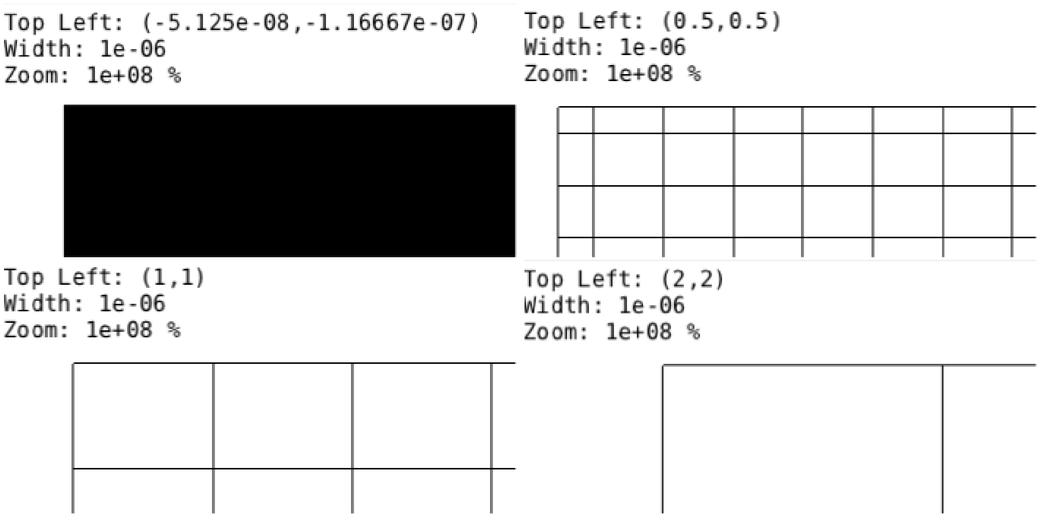

Figure 4.4: Effect of applying (2.7) to a grid of lines seperated by 1 pixel
a) Near origin (denormals) b), c), d) Increasing the exponent of (v_x, v_y) by 1

4.2.1 Names of programs in figures

- single - Single precision IEEE-754 with (2.7) applied directly
- double - Double precision IEEE-754 with (2.7) applied directly
- cumul-single - Single precision IEEE-754 with cumulative transforms to Béziers
- cumul-double - Double precision IEEE-754 with cumulative transforms to Béziers
- path-single - Single precision IEEE-754 with cumulative transforms to Paths
- path-double - Single precision IEEE-754 with cumulative transforms to Paths
- path-rat - GNU MP Rationals with cumulative transforms to Paths

4.2.2 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.

Figure 4.5 shows how precision degrades with $(v_x, v_y) = (0.5, 0.5)$. A constant line at 1401 grid locations indicates no loss of precision.

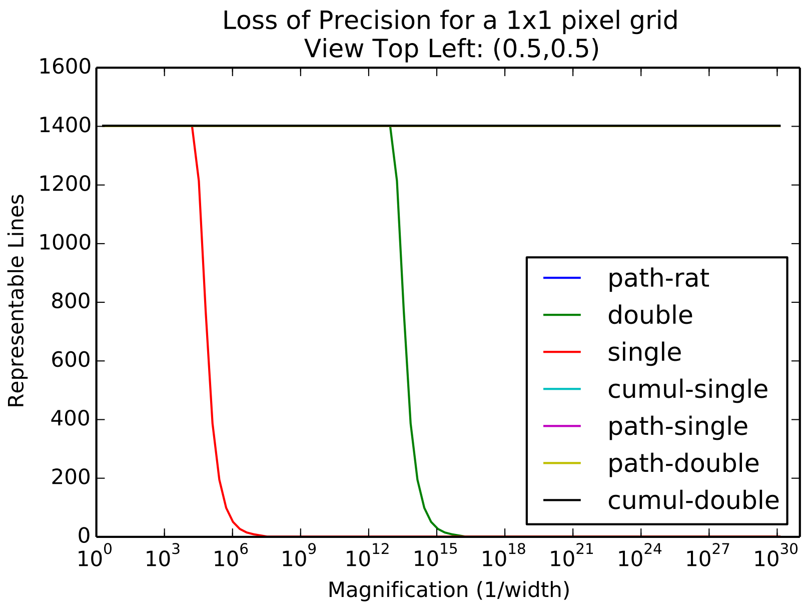

Figure 4.5: Loss of precision of the grid

4.2.3 Accumulated error after changing the View

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 indicates no accumulated error.

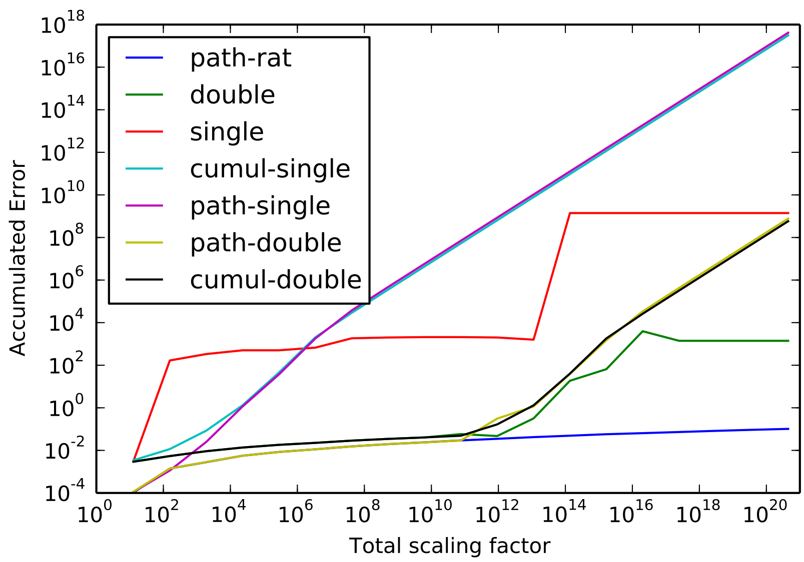

Figure 4.6: Error in the coordinates of the grid

By considering Figure 4.5 and 4.6, `path-rat` is the winner.

4.3 Performance Measurements whilst Rendering

As discussed above, we succeeded in preserving rendering accuracy as defined above for an arbitrary view. However this comes at a performance cost, as the size of the number representation must grow accordingly.

TODO: Insert performance measurements here

TODO: Also, would be nice to show a graph (log scale) where something goes past $10^{\pm 320}$ **(absolute limit for doubles, previous figures are all within range of representable floats**

5. Conclusion

- What we have done?
  - Implemented a basic SVG viewer
  - Demonstrated how precision affects rendering vector graphics
  - Showed how the choice of transformations to apply affects rendering
  - Using GMP rationals, demonstrated the ability to render SVGs scaled to an arbitrary position in a document
- Possible future work
  - Implement more of the SVG standard (eg: Shading)
  - Trial alternative number representations, eg: MPFR with algorithm to set precision
  - Allow for saving and loading SVGs with arbitrary precision
  - Deal with zooming very far in to intersection of lines (requires subdividing paths)
  - Compare with David's Quadtree

References

[1] Sam Moore. Infinite precision document formats (project proposal). ⟨http://szmoore.net/ipdf/documents/ProjectProposalSam.pdf⟩, 2014.

[2] David Gow. Infinite-precision document formats (project proposal). ⟨http://davidgow.net/stuff/ProjectProposal.pdf⟩, 2014.

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

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

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

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

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

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

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

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

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

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

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

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

[20] 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.

[21] 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.

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

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

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

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

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

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

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

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

[30] 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.

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

[32] Michael J. Kelley, Matthew A. Postiff, Advisor Richard, and B. Brown. A cmos floating point unit, 1997.

[33] P.-M. Seidel and G. Even. On the design of fast ieee floating-point adders. In *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pages 184–194, 2001.

[34] William R. Dieter, Akil Kaveti, and Henry G. Dietz. Low-cost microarchitectural support for improved floating-point accuracy. *IEEE Comput. Archit. Lett.*, 6(1):13–16, January 2007.

[35] Edin Kadric, Paul Gurniak, and André DeHon. Accurate parallel floating-point accumulation. In *Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on*, pages 153–162. IEEE, 2013.

[36] 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.

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

[38] 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.

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

[40] 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.

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

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

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

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

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

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

Note: We have collated most of these references at ⟨http://szmoore.net/ipdf/documents/references/⟩