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 8, 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.

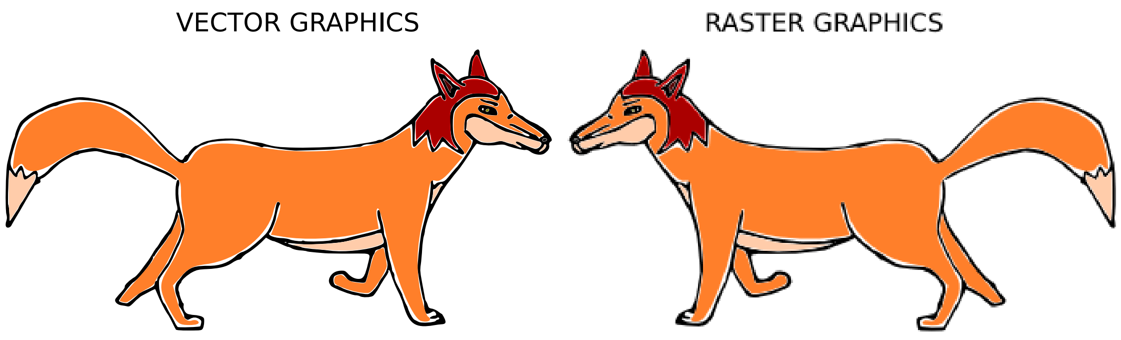


Figure 2.1: Original Vector and Raster Images

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

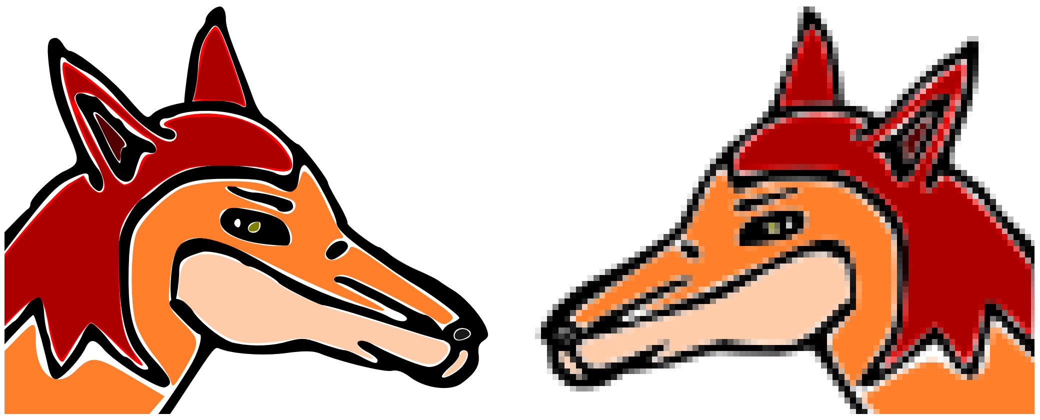


Figure 2.2: Scaled Vector and Raster Images

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]. .

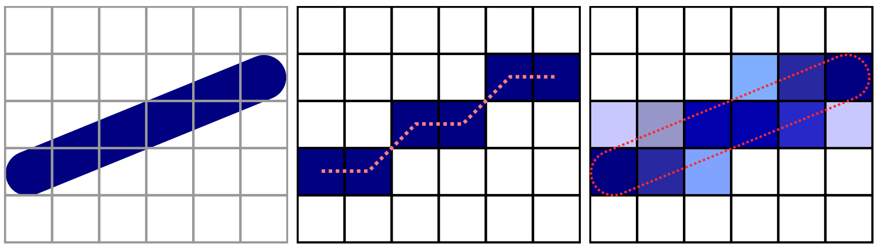


Figure 2.3: Rasterising a Straight Line

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:

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

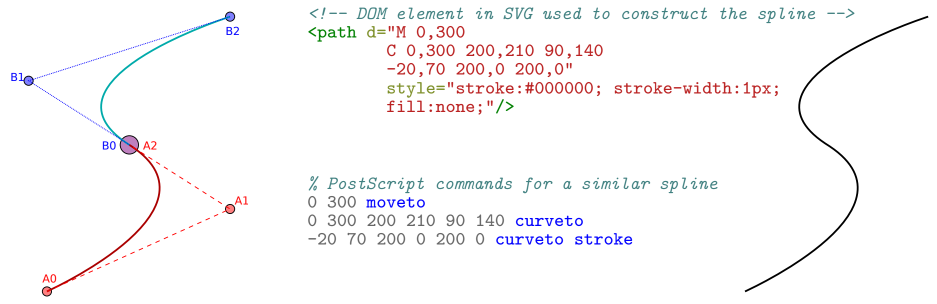
$$\text{Where } \binom{n}{j} = \frac{n!}{n!(n-j)!} \quad \text{(The Binomial Coefficients)} \tag{2.5}$$

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]:

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



```
<!-- DOM element in SVG used to construct the spline -->
<path d="M 0,300
        C 0,300 200,210 90,140
        -20,70 200,0 200,0"
        style="stroke:#000000; stroke-width:1px;
        fill:none;"/>
```

```
% PostScript commands for a similar spline
0 300 moveto
0 300 200 210 90 140 curveto
-20 70 200 0 200 0 curveto stroke
```

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

2.2.5 Fonts

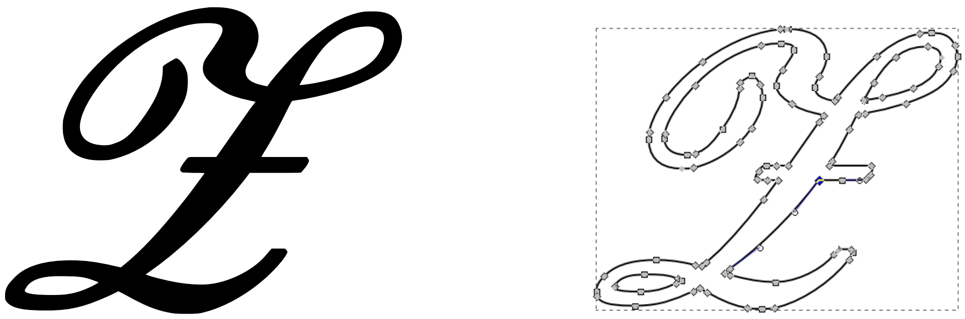


Figure 2.5: a) Vector glyph for the letter Z b) Screenshot showing Bézier control points in Inkscape

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".

2.4 Precision Specified by Document Standards

We briefly summarise the requirements of the standards discussed so far in regards to the precision of mathematical operations.

2.4.1 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"[3]. 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".

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

2.4.2 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 $\pm 1.175 \times 10^{38}$[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”.

2.4.3 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.

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 integers 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):

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

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.

A floating point number x is commonly represented by a tuple of values (s, e, m) in base B as[21, 22]: $x = (-1)^s \times m \times B^e$

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[21].

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[23].

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.

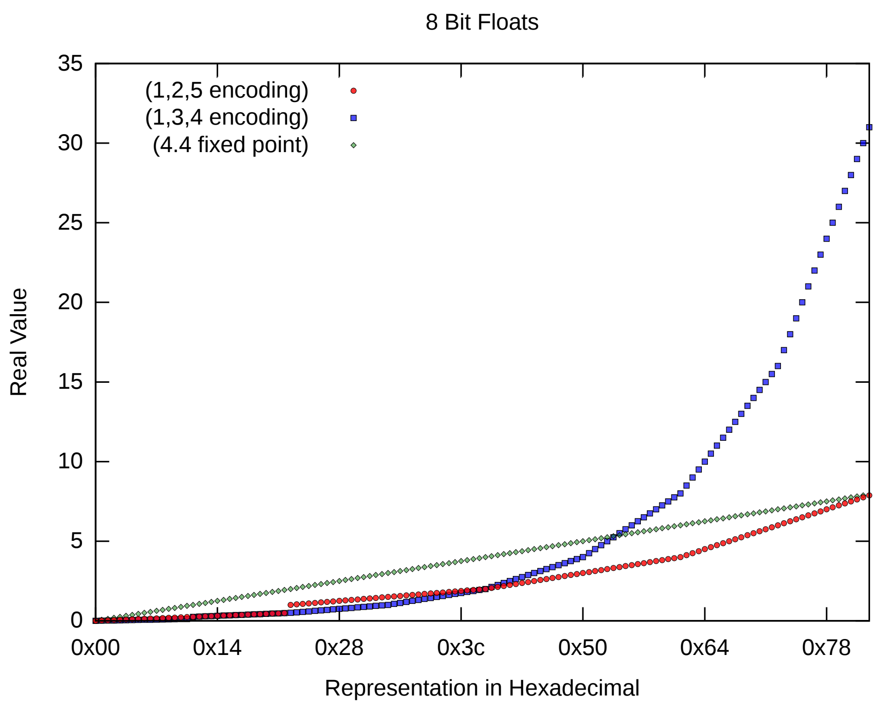


Figure 2.6: Positive 8-Bit Number Representations

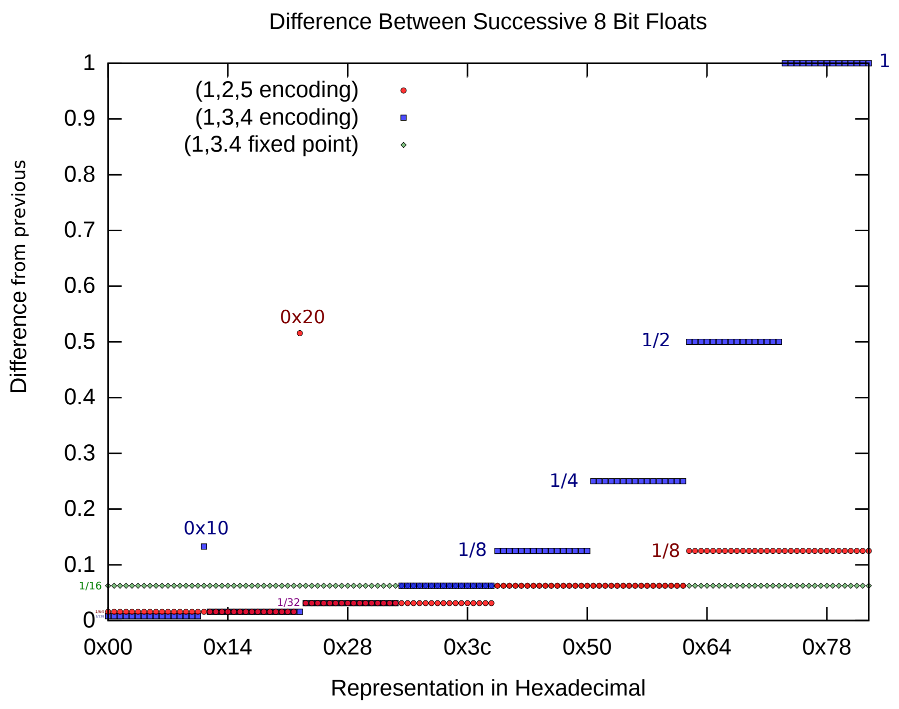


Figure 2.7: Difference between successive numbers

2.6.2 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[24]. 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

3. Methods and Design

3.1 Softare Overview

UML diagram.

3.2 Design Process

3.2.1 Timeline

A timeline will go here.

3.2.2 Version Control

We used Git.

3.2.3 Debugging Methods

We used GDB and Valgrind and copious amounts of `Debug` (ie: `printf`) calls.

3.3 Coordinate Systems and Bounds Transformations

Equation (2.7) involves a division operation; in general, the result cannot be represented exactly in either a fixed or floating point format. However, if the coordinates involved are expressed as rational numbers, the result will always be exact.

1. Store static object bounds, transform to view before rendering
    - Straight foward approach
    - Causes the most obvious rounding errors; impossible to render objects accurately below epsilon
    - The transformation can be performed by the GPU or CPU; the GPU is restricted to floats
    - CPU can perform transformation to arbitrary precision, but this requires all object bounds to be expressed with arbitrary precision
2. Modify all object bounds, no transformation required before rendering
    - The rounding error on floats does not accumulate as quickly
    - Instead of one division by a very small number there are repeated divisions by larger numbers
    - However, if the document is scaled so object bounds $\to 0$ then the object will not be recoverable
    - For arbitrary precision we still need to apply all bounds transformations on the GPU

3. Keep object bounds static to some intermediate object and transform the bounds of that object
    - We have used SVG `<path>` to construct the intermediate objects; all beziers are relative to their parent path.
    - One could instead use the entire SVG bounds; however for an SVG with a high level of detail this would have problems elaborate.
    - We only need to apply some bounds transformations on the CPU
    - However as the document grows we still need to apply transformations to all paths, even those that are not visible
4. Quad tree
    - Divide space up into a quad tree
    - Refer to David's stuff
    - The advantage over Path based transformations is that the bounds of objects which are not visible do not need to be recalculated

3.4 Measurements

3.4.1 Measurement Techniques

- Control panel allows inserting test images, screenshots, saving bound coordinates, changing program behaviour
- For more automated tests, a basic scripting language is implemented
- It has goto

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.

applying Transformation (2.7). We will use single precision floats for coordinates unless otherwise stated.

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).

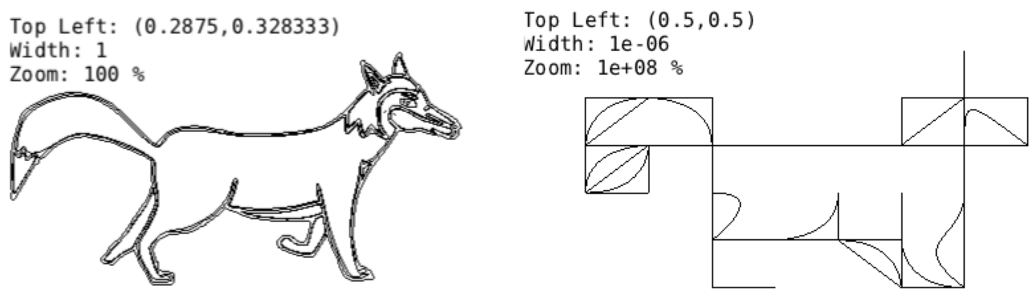


Figure 4.1: The vector image from Figure 2.1 under two different scales

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 approach significantly improves the range over which an image can be rendered correctly, and it is convenient for interactively created documents as no transformation must be applied to add new detail.

However, 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 bounds coordinates will gradually underflow and eventually round to zero. An example of this effect is shown in Figure 4.2 b)

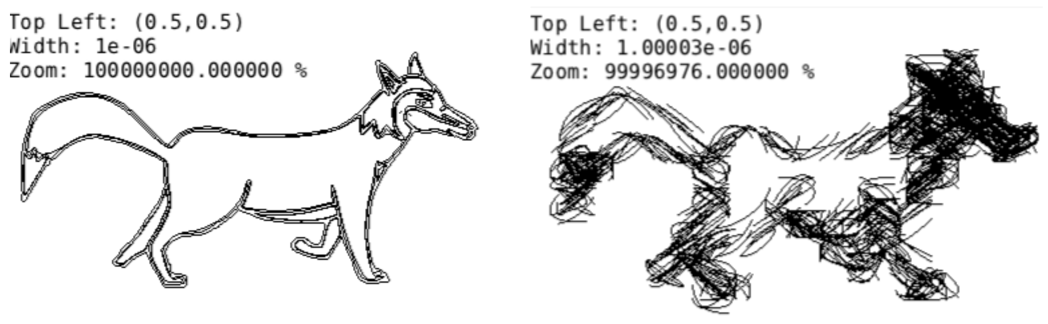


Figure 4.2: The effect of applying cumulative transformations to all Béziers

[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

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.

We can correct this drift whilst maintaining performance by using an arbitrary 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.

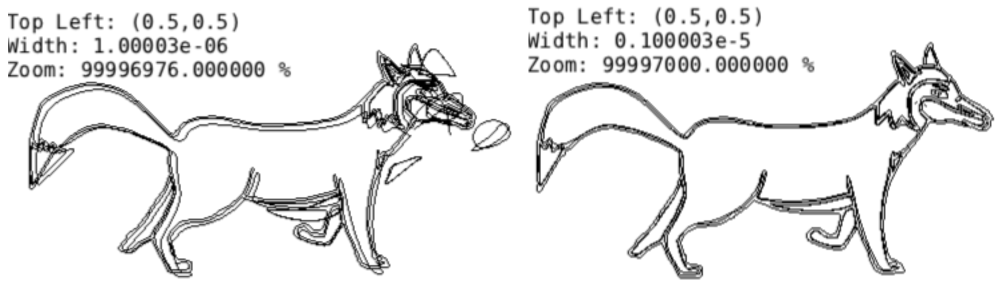


Figure 4.3: Effect of cumulative transformations applied to Paths
a) Path bounds represented using floats b) Path bounds represented using Rationals

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.

Figures **??** show the effect of scaling the grid to different view coordinates using single precision. This illustrates the trade off between precision and range; as the top left corner of the view moves further away from the origin, the width for which the grid appears unaltered decreases, or if the width is kept fixed, then there are fewer locations on the grid that can be correctly transformed from document to view space.

Figure **??** shows a plot of the number of lines visible in the grid as a function of distance from the origin for IEEE-754 floats and several different precision settings for the MPFR library.

Figure **??** shows the obvious

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.

5. Conclusion

5.1 Summary of Work and Results

5.2 Considerations for Future Work

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] 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.

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

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

[24] 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.

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