# Precision In Document Formats

*Author:* Samuel Moore[1]
*Partners:* David Gow[2]
*Supervisor:* Prof Tim French


THE UNIVERSITY OF
WESTERN AUSTRALIA
*Achieve International Excellence*

May 23, 2014

# Abstract

At the fundamental level, a document is a means to convey information. The limitations on a digital document format therefore restrict the types and quality of information that can be communicated. Whilst modern document formats are now able to include increasingly complex dynamic content, they still suffer from early views of a document as a static page; to be viewed at a fixed scale and position. In this report, we focus on the limitations of modern document formats (including PDF, PostScript, SVG) with regards to the level of detail, or precision at which primatives can be drawn. We propose a research project to investigate whether it is possible to obtain an "arbitrary precision" document format, capable of including primitives created at an arbitrary level of zoom.



**Keywords:** *document formats, precision, floating point, vector images, graphics, OpenGL, VHDL, 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/documents/LitReviewSam.pdf⟩

# Contents

# List of Figures

# 1. Introduction

## 1.1 Motivation

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 limited precision in these and other traditional document formats.

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[6, 7]. 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[3, 5, 8], 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[9, 10].

We are now seeing a widespread use of mobile computing devices with touch screens, where the display size is typically much smaller than paper pages and traditional computer monitors; it seems that there is much to be gained by breaking free of the restricted precision of traditional document formats.

## 1.2 Overview

The remainder of this document will be organised as follows: In Chapter 2 we give an overview of the current state of the research in document formats, and the motivation for implementing "infinite precision" in a document format. We will outline our approach to research in collaboration with David Gow[2]. In Chapter 3 we provide more detailed background examining the literature related to rendering, interpreting, and creating document formats, as well as possible techniques for increased and possibly infinite precision. In Chapter 4 gives the current state of our research and the progress towards the goals outlined in Chapter 1.

# 2. Proposal

## 2.1 Aim

In this project, we will explore the state of the art of current document formats including PDF, PostScript, SVG, HTML, and the limitations of each with regards to precision.

We will consider designs for a document format allowing graphics primitives at an arbitrary level of zoom with no loss of detail. A viewer and editor will be implemented as a proof of concept; we adopt a low level, ground up approach to designing this viewer so as to not become restricted by any single existing document format. Although it is possible to produce three dimensional graphics using some of the technologies we will explore, we will focus on two dimensional graphics.

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.

### 2.1.1 Clarification of Terms

It may be necessary to clarify what we mean by the terms "arbitrary precision" and "document formats". Regarding the latter, we consider a document format to be any representation of visual information which is capable of being stored indefinitely. Regarding the former, we do not propose to be able to contain an infinite amount of information within such a document. The goal is to be able to render a primitive at the same level of detail it is specified by a document format, regardless of how precise this level is. For example, the precision of coordinates of primitives drawn in a graphical document editor will always be limited by the resolution of the display on which they are drawn, but not by the viewer.

## 2.2 Methods

Initial research and software development is being conducted in collaboration with David Gow[2]. Once a simple testbed application has been developed, we will individually explore approaches for introducing arbitrary levels of precision; these approaches will be implemented as alternate versions of the same software. The focus will be on drawing simple primitives (lines, polygons, circles). However, if time permits we will explore adding more complicated primitives (font glyphs, bezier curves, embedded bitmaps). Hearn and Baker's textbook "Computer Graphics" includes chapters providing a good overview of two dimensional graphics[11].

The process of rendering a document will be considered as a common area of research, whilst individual research will be conducted on means for allowing infinite precision. At this stage we have identified two possible areas for individual research:

1. **Arbitrary Precision real valued numbers** — Sam Moore

   We plan to investigate the representation of real values to a high or arbitary degree of precision. Such representations would allow for the coordinates of primitives to be relative to a single global coordinate system. We would expect a decrease in performance with increased

complexity of the data structure used to represent a real value. We will also consider the limitations imposed by performing calculations on the GPU or CPU.

Starting points for research in this area are Priest's 1991 paper, "Algorithms for Arbitrary Precision Floating Point Arithmetic"[12], and Goldberg's 1992 paper "The design of floating point data types"[10]. A more recent and comprehensive text book, "Handbook of Floating Point Arithmetic"[13], published in 2010, has also been identified as highly relevant.

2. **Local coordinate systems** — David Gow [2]

   An alternative approach involves segmenting the document into different regions using fixed precision floats to define primitives within each region. A quadtree or similar data structure could be employed to identify and render those regions currently visible in the document viewer.

We aim to compare these and any additional implementations considered using the following metrics:

1. **Performance vs Number of Primitives**

   As it is clearly desirable to include more objects in a document, this is a natural metric for the usefulness of an implementation. We will compare the performance of rendering different implementations, using several "standard" test documents.

2. **Performance vs Visible Primitives**

   There will inevitably be an overhead to all primitives in the document, whether drawn or not. As the structure of the document format and rendering algorithms may be designed independently, we will repeat the above tests considering only the number of visible primitives.

3. **Performance vs Zoom Level**

   We will also consider the performance of rendering at zoom levels that include primitives on both small and large scales, since these are the cases under which floating point precision causes problems in the PostScript and PDF standards.

4. **Performance whilst translation and scaling**

   Whilst changing the view, it is ideal that the document be re-rendered as efficiently as possible, to avoid disorienting and confusing the user. We will therefore compare the speed of rendering as the standard documents are translated or scaled at a constant rate.

5. **Artifacts and Limitations on Precision**

   As we are unlikely to achieve truly "infinite" precision, qualitative comparisons of the accuracy of rendering under different implementations should be made.

## 2.3 Software and Hardware Requirements

Our proof of concept will be developed for a conventional GNU/Linux desktop or laptop computer using the OpenGL 3.1 API for rendering. However, the techniques explored could be extended to other platforms and libraries.

# 3. Literature Review

The first part of this chapter will be devoted to documents themselves, including: the representation and displaying of graphics primitives, and how collections of these primitives are represented in document formats, focusing on widely used standards.

We will find that although there has been a great deal of research into the rendering, storing, editing, manipulation, and extension of document formats, modern standards are content to specify at best single precision IEEE-754 floating point arithmetic.

The research on arbitrary precision arithmetic applied to documents is rather sparse; however arbitrary precision arithmetic itself is a very active field of research. Therefore, remainder of this chapter will be devoted to considering fixed precision floating point numbers as specified by the IEEE-754 standard, possible limitations in precision, and alternative number representations for increased or arbitrary precision arithmetic.

In Chapter 4, we will discuss our findings so far with regards to arbitrary precision arithmetic applied to document formats, and expand upon the goals outlined in Chapture 2.

## 3.1 Raster and Vector Images

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 3.1 and 3.2.

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 3.5.2).

Figures 3.1 and 3.2 illustrate the advantage of vector formats by comparing raster and vector images in a similar way to Worth and Packard[14]. On the right is a raster image which should be recognisable as an animal defined by fairly sharp edges. Figure 3.2 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".

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.

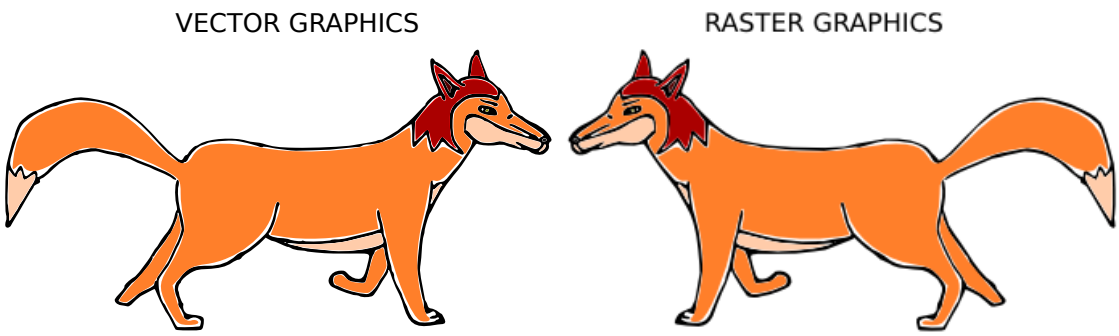


Figure 3.1: Original Vector and Raster Images

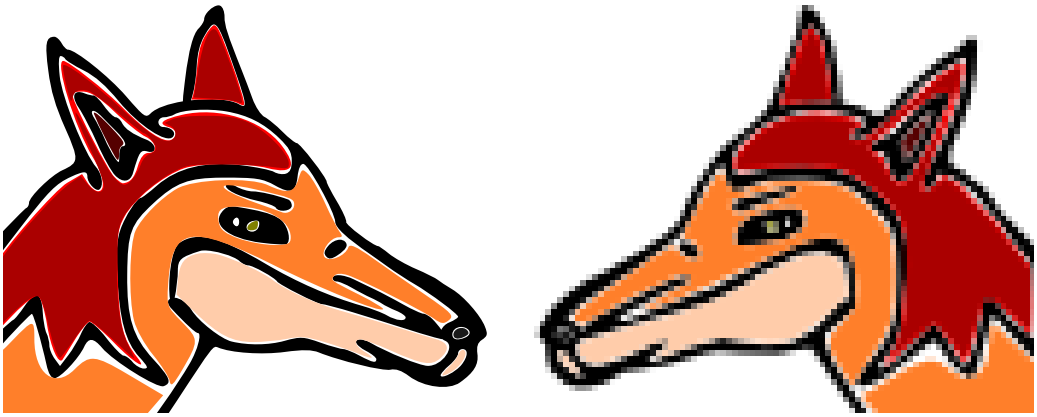

Figure 3.2: Scaled Vector and Raster Images

## 3.2 Rendering Vector Images

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[15]. Whilst curves can be drawn at high resolution on vector displays, a major disadvantage was shading[16]; by the early 90s the vast majority of computer displays were raster based[11].

### 3.2.1 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:

$$y(x) = mx + c \quad \text{on } x \in [x_1, x_2] \text{ for } m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \text{ and } c = y_1 - mx_1 \tag{3.1}$$

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 3.1 will require costly floating point operations and rounding (See Section3.5.2). Modifications based on computing steps $\Delta x$ and $\Delta y$ 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[17]. 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. Bresenham himself points out that rasterisation processes have existed since long before the first computer displays[18].

In Figure 3.3 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[19].

Figure 3.3 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[20]. 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[19][1]. .

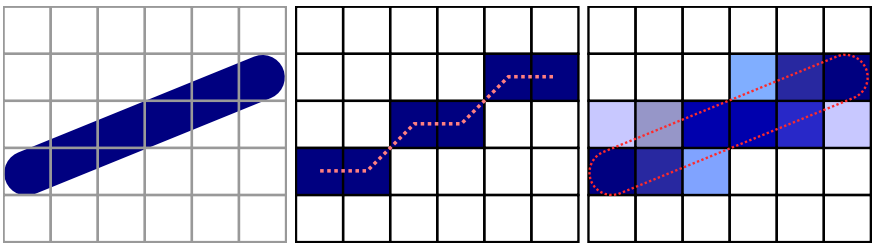

Figure 3.3: Rasterising a Straight Line

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

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

### 3.2.2 Spline Curves and Béziers

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

$$y(x) = \sum_{k=0}^{n} a_k x^k \tag{3.2}$$

Cubic and Quadratic Bézier Splines are used to define curved paths in the PostScript[3], PDF[5] and SVG[8] standards which we will discuss in Section 3.3. Cubic Béziers are also used to define vector fonts for rendering text in these standards and the TEX typesetting language [21, 22]. 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[23].

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:

$$P(t) = \sum_{j=0}^{n} B_j^n(t) P_j \tag{3.3}$$

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

$$B_j^n(t) = \binom{n}{j} t^j (1-t)^{n-j} \qquad j = 0, 1, ..., n \tag{3.4}$$

$$\text{Where } \binom{n}{j} = \frac{n!}{n!(n-j)!} \quad \text{(The Binomial Coefficients)} \tag{3.5}$$

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 3.2.1.

De Casteljau's algorithm of 1959 is often used for approximating Béziers[11, 21]. This algorithm subdivides the original $n$ control points $\{P_0, ...P_n\}$ into $2n$ 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[24]:

$$Q_i = \left( \frac{\binom{n}{j}}{2^j} \right) P_i \text{ and } R_i = \left( \frac{\binom{n-j}{n-k}}{2^{n-j}} \right) P_i \tag{3.6}$$

In much of the literature it is taken as trivial that it is only necessary to specify the control points of a Bézier in order to be able to render it at any level of detail[21, 11]. Recently, Goldman

presented an argument that Bézier's could be considered as fractal in nature, because the De Casteljau algorithm may be modified to be expressed the polynomial $P(t)$ as the result of iterated function system[24]. If this argument is correct, any primitive that can be described soley in terms of Bézier Curves may also be considered as fractal in nature. Ideally all these primitives may be rendered at any level of detail or "zoom" desired; however, computation of the pixel locations of the curve will be subject to the precision limits of the numerical representation which is used; we discuss these issues in Section **??**.

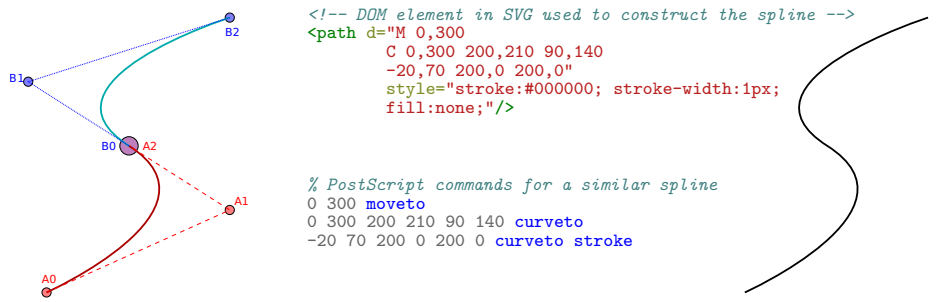


Figure 3.4: Constructing a Spline from two cubic Béziers
(a) Showing the Control Points (b) Representations in SVG and PostScript (c) Rendered Spline

### 3.2.3 Font Glyphs

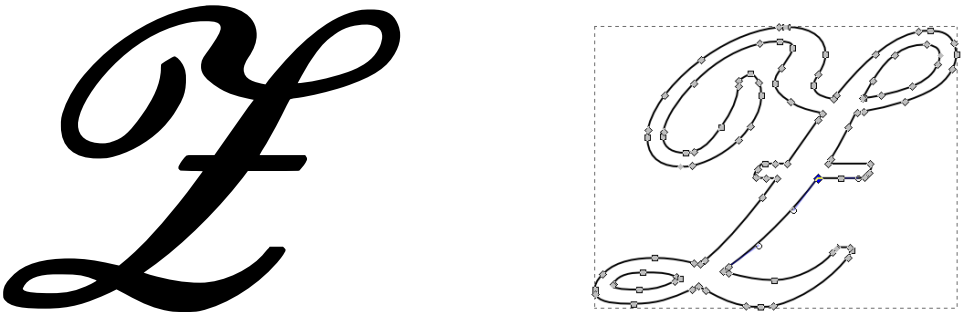

Figure 3.5: a) Vector glyph for the letter Z b) Screenshot showing Bézier control points in Inkscape

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[21]. Previously, only rasterised font images (glyphs) were popular; as can be seen from the zooming in Figure 3.2 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[21]. Figure 3.5 shows a $\mathscr{Z}$ (script Z) produced by LATEXwith Bézier cubics identified.

### 3.2.4 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[11]. In 1984, Porter and Duff introduced a fourth colour channel for rasterised images called the "alpha" channel, analogous to the transparency of a pixel[25]. 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[8]. 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:

$$a_T = 1 - (1 - a_1)(1 - a_2) \tag{3.7}$$

$$r_T = (1 - a_2)r_1 + r_2 \quad \text{(similar for } g_T \text{ and } b_T\text{)} \tag{3.8}$$

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[3]. Figure 3.7 illustrates the painter's model for partially transparent shapes as they would appear in both the SVG and PDF models.

### 3.2.5 Rasterisation 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[11]. Rasterisation of simple primitives such as lines and triangles have been supported directly by GPUs for some time through the OpenGL standard[26]. 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[26] API[27, 28, 29, 30, 31, 32]. Recently Mark Kilgard of the NVIDIA Corporation described an extension to OpenGL for NVIDIA GPUs capable of drawing and shading vector paths[33, 34]. 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[2]. 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[35].

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

## 3.3 Document Representations

The representation of information, particularly for scientific purposes, has changed dramatically over the last few decades. For example, Brassel's 1979 paper referenced earlier[15] 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[7] 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 static sheets of paper (Programmed Documents) or as a dynamic and interactive way to convey information, using the Document Object Model. We will now explore these two approaches and the extent to which they overlap.

### 3.3.1 Programmed Documents

#### PostScript

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. Each symbol is pushed onto a stack as it is read. Special symbols called "operators" can act upon this stack and/or the output device. An internal "graphics state" stack can be constructed to store styling information (such as colour, line thickness, the current cursor position). It is possible for the language to define new operators. Figure 3.6 shows a vector image and one possible way to express this image in PostScript. PostScript was and is still widely used in printing of documents onto paper; many printers execute postscript directly, and newer formats including PDFs must still be converted into PostScript by printer drivers[5, 4].

There are some limitations in PostScript's model. As mentioned in Section3.2.4, 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 3.7 and PostScript will simply rasterise the image and embed the rastered image in PostScript[3]

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"[6].

[3]For Figure 3.7 converted using the Inkscape SVG editor: ⟨http://szmoore.net/ipdf/figures/shape-svg-converted-to.ps⟩

```
%!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
```

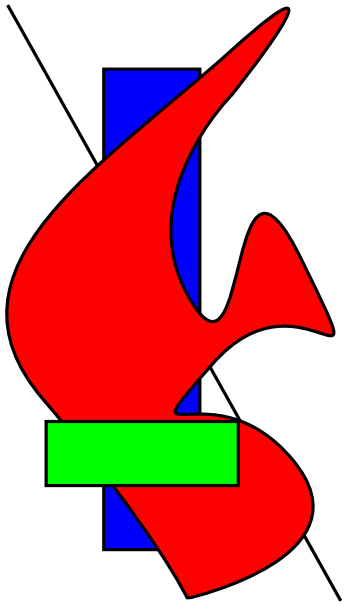

Figure 3.6: Vector image and a possible PostScript representation

### TEX, METAFONT and LATEX

Knuth's "The TEXbook"[22] and "The METAFONT book"[21] 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"[11]).

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[4] — this report itself has been produced using the CTAN LATEX packages[5].

[4]The site ⟨http://tex.stackexchange.com⟩ (accessed 2014-05-22) is devoted to TEX and LATEX

[5]The complete TEX source code to produce this document can be found at ⟨http://szmoore.net/ipdf/sam/⟩

### 3.3.2 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 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[36], HTML[37] and SVG[8]. 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 visual information to humans. These languages make use of Cascading Style Sheets (CSS)[38] 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 [39]) 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. Figure 3.7 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.[6] 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.

#### 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 3.7 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 interactively using HTML. The example generates successive iterations of a particular type of

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

fractal curve first described by Koch[40] in 1904 and a popular example in modern literature [24]. Unfortunately as it is currently possible to directly include W3C HTML in a PDF, we are only able to provide some examples of the output as static images in Figure 3.8. The W3C has produced a primer describing the use of HTML5 and Javascript to produce interactive SVG's[41], 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[7]. As Hayes points out, there is some similarity between the `<canvas>` API and the PostScript interpreted approach to drawing[**?**].

```
<?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>
```

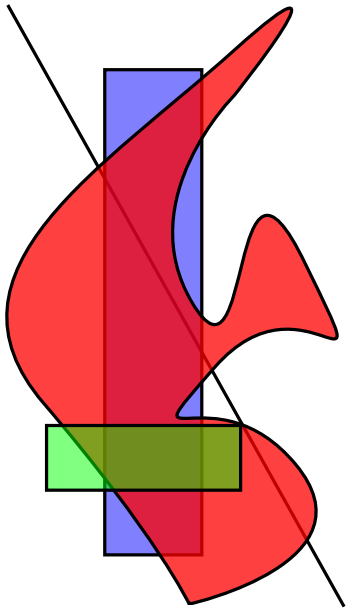

Figure 3.7: Vector image and a possible SVG representation

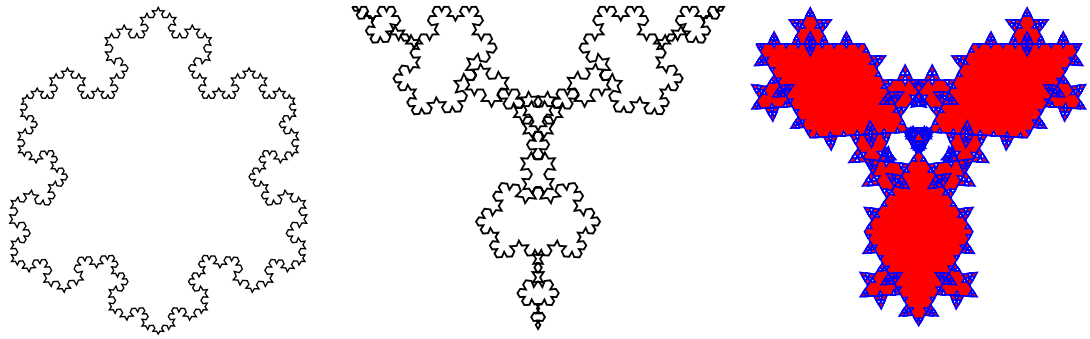

Figure 3.8: Koch "snowflakes" generated using Javascript to modify an SVG DOM. The interactive HTML5 document can be found at ⟨http://szmoore.net/ipdf/sam/figures/koch.html⟩

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

### 3.3.3 The Portable Document Format

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[**?**].

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]. 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[42]. There is some evidence in the literature of attempts to exploit these features, with mixed success[7, 6].

## 3.4 Precision required by Document Formats

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

### 3.4.1 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".

### 3.4.2 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".

### 3.4.3 TEXand 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"[21]. There is no mention of precision in "The TEXbook". 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[43]. Beebe also suggests that TEXand METAFONT could now be modified to use IEEE-754 arithmetic.

### 3.4.4 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."[8] An SVG Viewer may refer to itself as "High Quality" if it uses a minimum of "double precision" floats.

### 3.4.5 Javascript

According to the EMCA-262 standard, "The Number type has exactly 18437736874454810627 (that is, $2^64 - 5 + 3$) values, representing the double-precision 64-bit format IEEE 754 values as specified in the IEEE Standard for Binary Floating-Point Arithmetic"[39]. 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.

## 3.5 Number Representations

Consider a value of $7.25 = 2^2 + 2^1 + 2^0 + 2^{-2}$. In binary (base 2), this could be written as $111.01_2$ Such a value would require 5 binary digits (bits) of memory to represent exactly in computer hardware. Some values, for example 7.3 can not be represented exactly in one base (decimal) but not another; in binary the sequence $111.010..._2$ will never terminate. A rational value such as $\frac{7}{3}$ could not be represented exactly in any base, but could be represented by the combination of a numerator $7 = 111_2$ and denominator $3 = 11_2$. Lastly, some values such as $e \approx 2.81...$ can only be expressed exactly using a symbolical system — in this case as the result of an infinite summation — $e = \sum_n = 0^\infty \frac{1}{n!}$

Modern computer hardware typically supports integer and floating-point number representations and operations. Due to physical limitations, the size of these representations is limited; this is the fundamental source of both limits on range and precision in computer based calculations.

### 3.5.1 Floating Point Definitions

Whilst a Fixed Point representation keeps the "point" at the same position in a string of bits, Floating point representations can be thought of as analogous to 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[13, 44]:

$$x = (-1)^s \times m \times B^e$$

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 $\pm e_max$, 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[13]. 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[45][8]. 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[46].

Figure **??**[9] shows the positive real numbers which can be represented exactly by an 8 bit floating point number encoded in the IEEE-754 format[10], and the distance between successive floating point numbers. 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 on the right. 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.

The earlier example 7.25 would be converted to a (1,3,4) floating point representation as follows:

1. Determine the fixed point representation $7.25 = 111.01_2$
2. Determine the sign bit; in this case $s = 0$
3. Calculate the exponent by shifting the point $111.01_2 = 1.1101_2 \times 2^2 \implies e = 2 = 10_2$

[8] Eg: The smallest valid unit of currency $0.01 could not be represented exactly in base 2
[9] In a digital PDF viewer we suggest increasing the zoom level — the graphs were created from SVG images
[10] Not quite; we are ignoring the IEEE-754 definitions of NaN and Infinity for simplicity

4. Determine the exponent encoding; in IEEE-754 equal to the number of exponent bits is added so $e_{enc} = e + 3 = 5 = 101_2$
5. Remove the implicit bit if the encoded exponent $\neq 0$; $1.1101_2 \rightarrow .1101_2$
6. Combine the three bit strings$0, 101, 1101$
7. The final encoding is $01011101 \equiv$ 0x5D

This particular example can be encoded exactly; however as there are an infinite number of real values and only a finite number of floats, in general a value must be 7.26 must be rounded or truncated at Step 3.

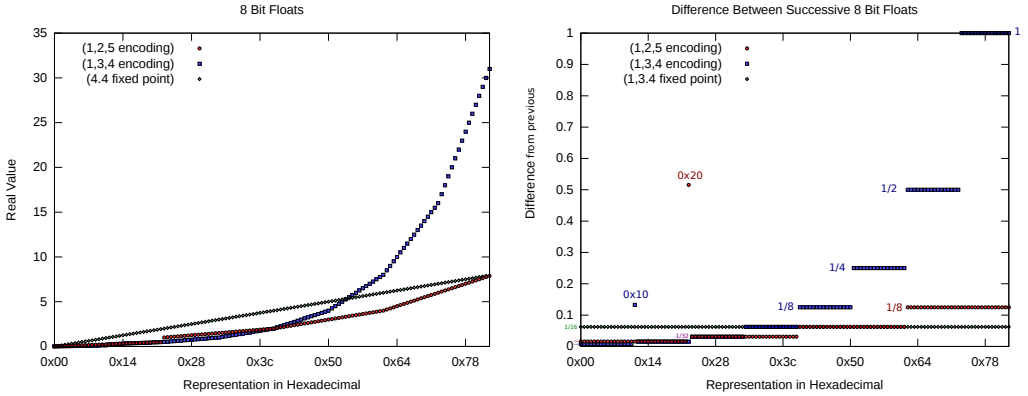


Figure 3.9: 8 bit float and fixed point representations a) As mapped to real values b) The distance between each representation

### 3.5.2 Precision and Rounding

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 3.9 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.

Goldberg's assertively titled 1991 paper "What Every Computer Scientist Needs to Know about Floating Point Arithmetic"[9] provides a comprehensive overview of issues in floating point arithmetic and relates these to requirements of the IEEE-754 1985 standard[47]. More recently, after the release of the revised IEEE-754 standard in 2008[45], a textbook "Handbook Of Floating Point Arithmetic" has been published which provides a thourough review of literature relating to floating point arithmetic in both software and hardware[13].

William Kahan, one of the architects of the IEEE-754 standard in 1984 and a contributor to its revision in 2010, has also published many articles on his website explaining the more obscure features of the IEEE-754 standard and calling out software which fails to conform to the

standard[11][48, 49], as well as examples of the limitations of floating point computations[50].

In Figure 3.10 we show the effect of accumulated rounding errors on the computation of $\pi$ through a numerical integration[12] using 32 bit "single precision" floats and 64 bit "double precision" floats.

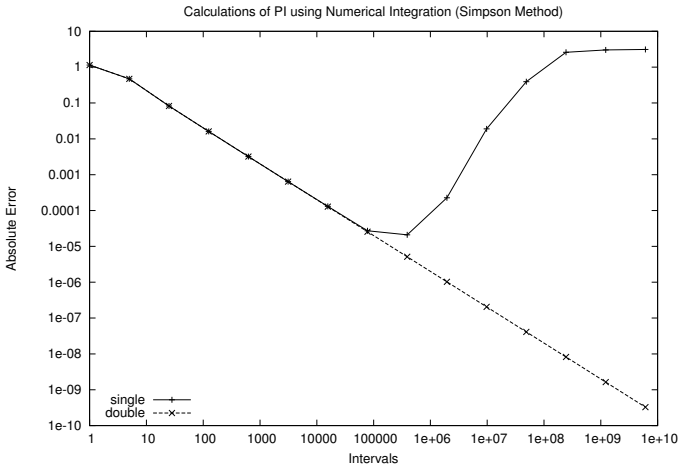


Figure 3.10: Numerical calculation of $\pi$

### 3.5.3 Floating Point Operations

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[51]. The improvement of FPUs remains highly active in several areas including: efficiency[52]; accuracy of operations[53]; and even the adaptation of algorithms originally used in software, such as Kahan's Fast2Sum algorithm[54].

In 1971 Dekker formalised a series of algorithms including the Fast2Sum method for calculating the correction term due to accumulated rounding errors[55]. The exact result of $x + y$ may be expressed in terms of floating point operations with rounding as follows:

$$\begin{aligned} z &= \mathrm{RN}(x+y) \quad w = \mathrm{RN}(z-x) \\ zz &= \mathrm{RN}(y-w) \quad \implies x+y = zz \end{aligned}$$

### 3.5.4 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[**?**]. 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.

[11] In addition to encodings and acceptable rounding behaviour, the standard also specifies "exceptions" — mechanisms by which a program can detect and report an error such as division by zero

[12] This is not intended to be an example of a good way to calculate $\pi$

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.

# 4. Progress Report

We describe the current state of our project in relation to the aims outlined in Chapter 1. At this stage work on the project has been done in collaboration with David Gow; however the Project Proposals and Literature Reviews were produced individually.

## 4.1 Literature Review

The literature examined in Chapter3 can broadly classed into three different areas (with major references indicated):

1. Rendering Vector Graphics [11, 21, 33]
    - Rasterisation of Vector Graphics is non-trivial but well understood
    - Traditionally most rasterisation has been performed on the CPU and drawing on a dedicated GPU; current interest is in techniques for utilising the GPU directly to rasterise vector graphics
2. Representations of Vector Documents [6, 3, 22, 8, 5]
    - Traditional approaches are be based on a programmatic model (PostScript, TEX, DVI)
    - The Document Object Model (DOM) used by web technologies is a powerful way to produce dynamic documents (HTML5, SVG, Javascript)
    - These approaches can overlap (PDF)
3. Number Representations [45, 13, 9, 56]
    - Most document standards either specify, suggest, or imply a IEEE-754 floating point representation (TEX is an exception)
    - IEEE-754 is widely used, although there are instances of languages or processors which do not conform exactly to the standard
    - Some GPUs in particular may not conform to IEEE-754, possibly trading some accuracy for performance
    - Arbitrary precision floating point arithmetic is possible through several libraries

To improve the Literature Review we could consider the following topics in more detail:

1. Additional approaches to arbitrary precision possibly including symbolic computation
    - The Mathematica computational package claims to use symbolic computation, but we have yet to explore this field
2. Floating point errors in the context of computing Bézier Curves or similar
3. Algorithms for reducing overall error other than Fast2Sum
4. Alternative number representations such as rationals (eg: $\frac{1}{3}$)
5. How well GPUs conform or do not conform to IEEE-754 in more detail
6. Additional aspects of rendering vector documents including shading

## 4.2 Development of Testbed Software

We have produced a basic Document Viewer capable of rendering simple primitives under translation and scaling. The OpenGL 3.1 API is used to interface with graphics hardware. This software has the following features:

1. A type name `Real` is used in place of the standard floating point types `float`, `double` or `long double`. This type name can be redefined to refer to one of the standard types or a custom real number representation, allowing us to easily recompile and test our software for different representations.
2. Screenshots can be overlaid on top of each other to get a pixel comparison of the graphical output of different versions of the program
3. Test documents can be loaded and saved so that we can compare different versions of the program on identical inputs
4. The time for rendering can be measured
5. Coordinate transformations may be performed on either the GPU or CPU

We have noticed the CPU produces more precise coordinate transformations at large "zoom" levels, but is significantly slower than the GPU. We have yet to quantitatively measure this difference.

## 4.3 Floating Point Arithmetic

Algorithms for floating point arithmetic may be implemented in software (CPU) or on dedicated hardware (FPU). We have made progress towards both approaches.

An open source Virtual FPU implemented in the VHDL language has been successfully compiled and can be substituted into our testbed software in place of native arithmetic running on the CPU. The timing diagram for this FPU throughout the execution of test programs can be extracted. Currently the virtual FPU is restricted to 32 bit floats and the square root operation is unimplemented.

Mainly motivated by producing Figure **??** we have also implemented functions to convert an arbitrary `Real` type (which may be IEEE-754 floats) to and from a fixed size floating point representation of our choosing. We have not implemented any operations for floating point arithmetic using these representations.

By using the functions to convert real numbers to variable precision floats as an interface for the virtual FPU, we hope to illustrate the limitations of floating point arithmetic more clearly than would be possible using IEEE-754 binary32 as is native to the C and C++ languages. Using the virtual FPU instead of a CPU based software library will prove useful for determining the exact performance of floating point operations.

## 4.4 Prototype Document Formats

Our testbed software is capable of reading primitive attributes from either a binary file or XML plain text file. Our format is conceptually similar to the Document Object Model, although there is currently only one generation in the tree as no primitives can contain other elements as of yet.

If time permits, we plan to extend our XML format to cover a subset of the SVG standard. This may allow us to compare the rasterisation of an SVG using our own software and traditional software relying on IEEE-754 floats.

Some of the figures produced for Chapter 3 may prove useful as standard test images for comparing the qualitative performance of versions of our software.

## 4.5 Version Control and Backup of Work

Git is a distributed version control system widely used in the development of open source software. All rescources created for or used by this project have been placed in git repositories on several servers. The repositories are publically accessable at ⟨http://git.ucc.asn.au⟩, ⟨http://szmoore.net/ipdf⟩.

## 4.6 Timeline

Deadlines enforced by the faculty of Engineering Computing and Mathematics are *italicised*.[1].

| **Date** | **Milestone** |
|---|---|
| 1st May | Testbed Software (basic document format and viewer) completed and approaches for extending to allow infinite precision identified. |
| 17th May | Draft Progress Report and Literature Review |
| 26th May | *Progress Report and Literature Review due.* |
| 9th June | Demonstrations of limitations of floating point precision in the Testbed software. |
| 1st July | At least one implementation of arbitrary precision for basic primitives (lines, polygons, curves) completed. Other implementations, advanced features, and areas for more detailed research identified. |
| 1st August | Experiments and comparison of various arbitrary precision implementations completed. |
| 1st September | Advanced features implemented and tested, work underway on Final Report. |
| TBA | *Conference Abstract and Presentation due.* |
| 10th October | *Draft of Final Report due.* |
| 27th October | *Final Report due.* |

[1]David Gow is being assessed under the 2014 rules for a BEng (Software) Final Year Project, whilst the author is being assessed under the 2014 rules for a BEng (Mechatronics) Final Year Project; deadlines and requirements as shown in Gow's proposal[2] may differ

# 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] 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.

[8] 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.

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

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

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

[12] D.M. Priest. Algorithms for arbitrary precision floating point arithmetic. In *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*, pages 132–143, Jun 1991.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] J. Bresenham. Pixel-processing fundamentals. *Computer Graphics and Applications, IEEE*, 16(1):74–82, Jan 1996.

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

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

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

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

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

[24] 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.

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

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

[27] 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.

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

[29] 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.

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

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

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

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

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

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

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

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

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

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

[40] 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.

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

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

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

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

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

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

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

[48] W Kahan. Prof w kahan's webpages. ⟨http://www.cs.berkeley.edu/ wkahan/⟩ accessed April 2014.

[49] W Kahan. Lecture notes on the status of ieee standard 754 for binary floating-point arithmetic. ⟨http://http.cs.berkeley.edu/ wkahan/ieee754status/ieee754.ps⟩ accessed April 2014, May 1996.

[50] W Kahan. Why is floating-point computation so hard to debug when it goes wrong? ⟨http://www.cs.berkeley.edu/ wkahan/WrongR.pdf⟩ accessed April 2014, 2007.

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

[52] 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.

[53] 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.

[54] 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.

[55] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[56] 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.