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 18, 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 "infinite precision" document format, capable of including primitives created at an arbitrary level of zoom.

Keywords: document formats, precision, floating point, graphics, OpenGL, VHDL, PostScript, PDF, bootstraps

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 and PDF support rasterisation at different zoom levels[3, 5], 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].

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[]. 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 **??** gives the current state of our research and the progress towards the goals outlined in Chapter 1. In Chapter 5 we will conclude with a summary of our findings and goals.

2. Proposal

Most of this chapter is copy pasted from the project proposal ⟨http://szmoore.net/ipdf/documents/ProjectProposalSam.pdf⟩

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 in terms of 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.

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[10].

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. Both software and hardware techniques will be explored. 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"[11], and Goldberg's 1992 paper "The design of floating point data types"[9]. A more recent and comprehensive text book, "Handbook of Floating Point Arithmetic"[12], 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. Say more here?

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

Due to the relative immaturity and inconsistency of graphics drivers on mobile devices, our proof of concept will be developed for a conventional GNU/Linux desktop or laptop computer using OpenGL. However, the techniques explored could easily be extended to other platforms and libraries.

3. Literature Review

This chapter provides an overview of relevant literature. The areas of interest can be broadly grouped into two largely separate categories; Documents and Number Representations.

The first half of this chapter will be devoted to documents themselves, including: the representation and displaying of graphics primitives[10], and how collections of these primitives are represented in document formats, focusing on well known standards currently in use[3, 5, 13].

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

The research on arbitrary precision arithmetic applied to documents is very sparse; however arbitrary precision arithmetic itself is a very active field of research. Therefore, the second half of this chapter will be devoted to considering the IEEE-754 standard, its advantages and limitations, and possible alternative number representations to allow for arbitrary precision arithmetic.

In Chapter **??**, 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. Figures 3.1 and 3.2 attempt to illustrate this by comparing raster images to vector images in a similar way to Worth and Packard[14].

The right side of Figure 3.1 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".

In contrast, the left sides of Figures 3.1 and 3.2 are a vector image. 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 **??**). Vector images are well suited to high quality digital art[1] and text.

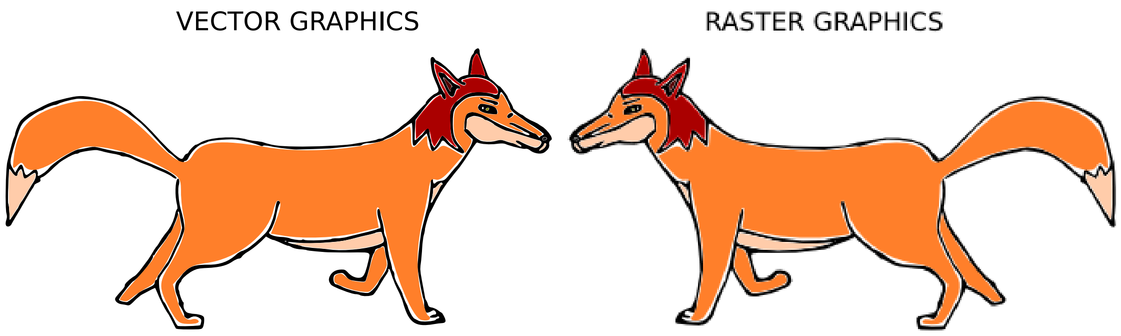

Figure 3.1: Original Vector and Raster Images

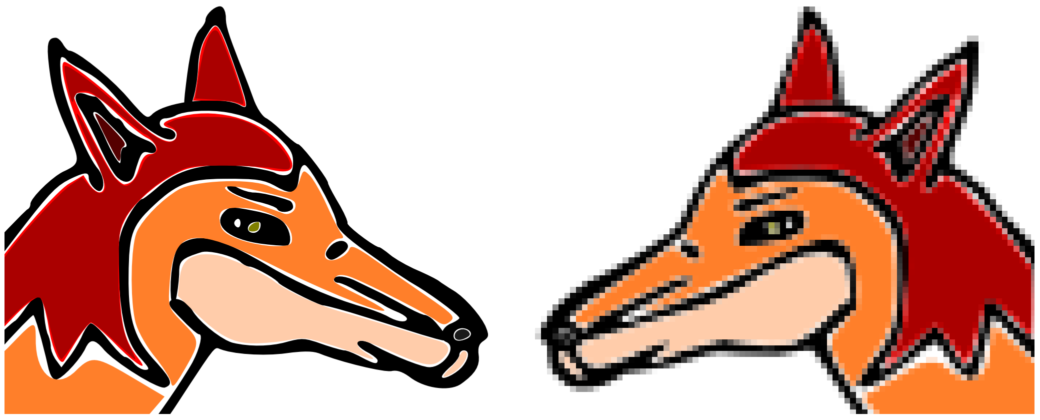

Figure 3.2: Scaled Vector and Raster Images

3.2 Rasterising Vector Images

Throughout Section **??** we were careful to refer to “modern” display devices, which are raster based. 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; by the early 90s the vast majority of computer displays were raster based[10].

Hearn and Baker’s textbook “Computer Graphics” [10] 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. Informal tutorials are abundant on the internet[16]. This section is by no means a comprehensive survey of the literature but intends to provide some idea computations which are required to render a document.

[1]Figure 3.1 is not to be taken as an example of this.

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:

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

$$\text{and } c = \tag{3.2}$$

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 Section**??**). Modifications based on computing steps δx and δy eliminate the multiplication but are still less than ideal in terms of performance[10].

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[**?**]. I studied this sort of thing in Physics, once upon a time... if you just say "Nyquist Sampling" and wave your hands people usually buy it.

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[16]. In 1991 Wu introduced an algorithm for drawing antialiased lines which, while equivelant in results to existing algorithms by Fujimoto and Iwata, set the state of the art in performance[19].

NOTE: Should I actually discuss how the algorithms work? Or just "review" the literature? Bearing in mind that how they actually work doesn't really relate to infinite precision document formats...

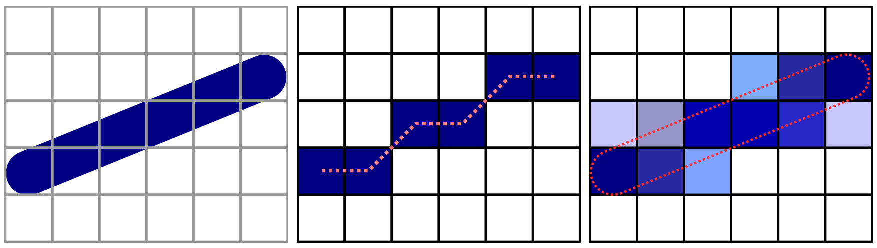

Figure 3.3: Rasterising a Straight Line

a) Before Rasterisation b) Bresenham's Algorithm c) Antialiased Line (Idealised)

3.2.2 Spline Curves

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:

$$y(x) = \sum_{k=0}^{n} a_k x^k \tag{3.3}$$

A straight line is simply a polynomial of 0th degree. Splines may be rasterised by sampling of $y(x)$ at a number of points x_i and rendering straight lines between (x_i, y_i) and (x_{i+1}, y_{i+1}) as discussed in Section 3.2.1. More direct algorithms for drawing splines based upon Brasenham and Wu's algorithms also exist[**?**].

There are many different ways to define a spline. One approach is to specify "knots" on the spline and solve for the cooefficients to generate a cubic spline ($n = 3$) passing through the points. Alternatively, there are many ways to specify a spline using "control" points which themselves are not part of the curve; these are convenient for graphical based editors.

Bezier Curves

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

$$P(t) = \sum_{j=0}^{n} B_j^n(t) P_j \tag{3.4}$$

From this definition it should be apparent $P(t)$ for a Bezier Curve of degree 0 maps to a single point, whilst $P(t)$ for a Bezier of degree 1 is a straight line between P_0 and P_1. $P(t)$ always begins at P_0 for $t = 0$ and ends at P_n when $t = 1$.

Figure **??** shows a Bezier Curve defined by the points $\{(0, 0), (1, 0), (1, 1)\}$.

A straightforward algorithm for rendering Bezier's is to simply sample $P(t)$ for some number of values of t and connect the resulting points with straight lines using Bresenham or Wu's algorithm (See Section 3.2.1). Whilst the performance of this algorithm is linear, in ???? De Casteljau derived a more efficient means of sub dividing beziers into line segments.

Recently, Goldman presented an argument that Bezier's could be considered as fractal in nature, a fractal being the fixed point of an iterated function system[20]. Goldman's proof depends upon a modification to the De Casteljau Subdivision algorithm which expresses the subdivisions as an iterated function system.

3.2.3 Shading

Algorithms for shading on vector displays involved drawing equally spaced lines in the region with endpoints defined by the boundaries of the region[15]. Apart from being unrealistic, these techniques required a the sorting of vertices and a

On raster displays, shading is typically based upon Lane's algorithm of 1983[21] which is implemented in the GPU [22]

6. Sort of starts here... or at least background does

3.2.4 Compositing

In 1984, Porter and Duff introduced Digital Compositing for rastered images[23].

Traditionally, vector graphics have been rasterized by the CPU before being sent to the GPU for drawing[22]. Lots of people would like to change this [14, 24, 25, 22, 26].

2. Here are the ways documents are structured ... we got here eventually

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 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[15]. In contrast, modern papers such as Barnes et. al's recent paper on embedding 3d images in PDF documents[**?**] can themselves be an interactive proof of concept.

Hayes' 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 documents although the line between these two philosophies is being blurred. We shall restrict ourselves to considering the standards discussed by Hayes.

3.3.1 Interpreted Model

- This model treats a document as the source code program which produces graphics
- Arose from the desire to produce printed documents using computers (which were still limited to text only displays).
- Typed by hand or (later) generated by a GUI program

- PostScript — largely supersceded by PDF on the desktop but still used by printers[2]
- TEX— Predates PostScript, similar idea
  - Maybe if LATEXwere more popular there would be desktop viewers that converted LATEXdirectly into graphics
- Potential for dynamic content, interactivity; dynamic PostScript, enhanced Postscript
- Problems with security — Turing complete, can be exploited easily

3.3.2 Crippled Interpreted Model

I'm pretty sure I made that one up

- PDF is PostScript but without the Turing Completeness
- Solves security issues, more efficient

3.3.3 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 DOM is described by the W3C XML (extensible markup language) standard[?]. XML itself is a general language which is intended for representing any tree-like structure using the DOM, whilst languages such as HTML[?] and SVG[?] are specific XML based languages for representing visual information.

The HyperText Markup Language (HTML) was, as its name implies, originally intended mostly for text. When combined with Cascading Style Sheets (CSS) control over the positioning and style of the text can be acheived. Images stored in some other format can be rendered within a HTML document, but HTML does not include ways to specify graphics primitives or their coordinates.

The Scalable Vector Graphics (SVG) standard was designed to represent a vector image. 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.

Modifying the DOM — Javascript

Javascript is now ubiquitous in web based documents, and is essentially used to make the DOM interactive. This can be done by altering the attributes of elements, or adding and removing elements from the DOM, in response to some event such as user input or communication with

[2]Desktop pdf viewers can still cope with PS, but I wonder if a smartphone pdf viewer would implement it?

a HTTP server. In the HTML5 standard, it is also possible to draw directly to a region of the document defined by the `<canvas>` tag; as Hayes points out, this is similar to the use of PostScript to specify the appearance of a document using low level drawing operators.

3.3.4 Scientific Computation Packages

The document and the code that produces it are one and the same.

- Numerical computation packages such as Mathematica and Maple use arbitrary precision floats
  - Mathematica is not open source which is an issue when publishing scientific research (because people who do not fork out money for Mathematica cannot verify results)
  - What about Maple? [12] and [27] both mention it being buggy.
  - Octave and Matlab use fixed precision doubles
- IPython is pretty cool guys

3.4 Precision in Modern Document Formats

We briefly summarise the requirements of standard document formats in regards to the precision of number representations:

- **PostScript** predates the IEEE-754 standard and originally specified a floating point representation with ? bits of exponent and ? bits of mantissa. Version ? of the PostScript standard changed to specify IEEE-754 binary32 "single precision" floats.
- **PDF** has also specified IEEE-754 binary32 since version ?. Importantly, the standard states that this is a maximum precision; documents created with higher precision would not be viewable in Adobe Reader.
- **SVG** specifies a minimum of IEEE-754 binary32 but recommends more bits be used internally
- **Javascript** uses binary32 floats for all operations, and does not distinguish between integers and floats.

4. Here is IEEE-754 which is what these standards use

3.5 Real Number Representations

We have found that PostScript, PDF, and SVG document standards all restrict themselves to IEEE floating point number representations of coordinates. This is unsurprising as the IEEE standard has been successfully adopted almost universally by hardware manufactures and programming language standards since the early 1990s. In the traditional view of a document as a static, finite sheet of paper, there is little motivation for enhanced precision.

In this section we will begin by investigating floating point numbers as defined in the IEEE standard and their limitations. We will then consider alternative number representations including fixed point numbers, arbitrary precision floats, rational numbers, p-adic numbers and symbolic representations. Oh god I am still writing about IEEE floats let alone all those other things

3.5.1 Floating Point

A floating point number x is commonly represented by a tuple of integers (s, e, m) in base B as[12, 28]:

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

Where s is the sign and may be zero or one, m is commonly called the "mantissa" and e is the exponent. The name "floating point" refers to the equivelance of the $\times B^e$ operation to a shifting of a decimal point along the mantissa. This contrasts with a "fixed point" representation where x is the sum of two fixed size numbers representing the integer and fractional part.

3.5.2 The IEEE-754 Standard

Although the concept of a floating point representation has been attributed to various early computer scientists including Charles Babbage[**?**], it is widely accepted that William Kahan and his colleagues working on the IEEE-754 standard in the 1980s are the "fathers of modern floating point computation"[**?**]. The IEEE standard specifies the encoding, number of bits, rounding methods, and maximum acceptable errors for the basic floating point operations. It also specifies "exceptions" — mechanisms by which a program can detect an error such as division by zero.

In the IEEE-754 standard, for a base of $B = 2$, numbers are encoded in continuous memory by a fixed number of bits, with s occupying 1 bit, followed by e and m occupying a number of bits specified by the precision; 5 and 10 for a binary16 or "half precision" float, 8 and 23 for a binary32 or "single precision" and 15 and 52 for a binary64 or "double precision" float[12, 28]. The IEEE-754 standard also specifies a base 10 encoding (useful in financial software[**?**]), but since this is subject to similar limitations, we will restrict ourselves to the simpler base 2 encodings.

3.5.3 Precision and Rounding

Real values which cannot be represented exactly in a floating point representation must be rounded. The results of a floating point operation may be such values and thus there is a rounding error possible in any floating point operation. Goldberg's assertively titled 1991 paper "What Every Computer Scientist Needs to Know about Floating Point Arithmetic" provides a comprehensive overview of issues in floating point arithmetic and relates these to the 1984 version of the IEEE-754 standard[8]. More recently, after the release of the revised IEEE-754 standard,

Figure **??** shows the real numbers which can be represented exactly by an 8 bit base $B = 2$ floating point number; and illustrates that a set of fixed precision floating point numbers forms a discrete approximation of the reals. There are only $2^8 = 256$ numbers in this set, which means it is easier to see some of the properties of floats that would be unclear using one of the IEEE-754 encodings. The first set of points corresponds to using $(1, 2, 5)$ to encode (s, e, m) whilst the second set of points corresponds to a $(1, 3, 4)$ encoding. This allows us to see the trade off between the precision and range of real values represented.

3.5.4 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[?]. The improvement of FPUs remains highly active in several areas including: efficiency[29]; accuracy of operations[30]; and even the adaptation of algorithms originally used in software for reducing the overal error of a sequence of operations[31]. In this section we will consider the algorithms for floating point operations without focusing on the hardware implementation of these algorithms.

3.5.5 Limitations Imposed By CPU

CPU's are restricted in their representation of floating point numbers by the IEEE standard.

3.5.6 Limitations Imposed By Graphics APIs and/or GPUs

Traditionally algorithms for drawing vector graphics are performed on the CPU; the image is rasterised and then sent to the GPU for rendering[]. Recently there has been a great deal of literature relating to implementation of algorithms such as bezier curve rendering[] or shading[] on the GPU. As it seems the trend is to move towards GPU

6. Here are ways GPU might not be IEEE-754 — This goes *somewhere* in here but not sure yet

- Internal representations are GPU dependent and may not match IEEE[32]
- OpenGL standards specify: binary16, binary32, binary64
- OpenVG aims to become a standard API for SVG viewers but the API only uses binary32 and hardware implementations may use less than this internally[25]
- It seems that IEEE has not been entirely successful; although all modern CPUs and GPUs are able to read and write IEEE floating point types, many do not conform to the IEEE standard in how they represent floating point numbers internally.

7. Sod all that, let's just use an arbitrary precision library (AND THUS WE FINALLY GET TO THE POINT)

3.5.7 Arbitrary Precision Floating Point Numbers

An arbitrary precision floating point number simply uses extra bits to store extra precision. Do it all using MFPR[27], she'll be right.

8. Here is a brilliant summary of sections 7- above

Dear reader, thankyou for your persistance in reading this mangled excuse for a Literature Review. Hopefully we have brought together the radically different areas of interest together in some sort of coherant fashion. In the next chapter we will talk about how we have succeeded in rendering a rectangle. It will be fun. I am looking forward to it.

Oh dear this is not going well

4. Progress Report

This chapter outlines the current state of our research in relation to the aims outlined in Chapter 1.

4.1 Literature Review

We have examined a range of literature that can be broadly classed into three different areas:

1. Rendering Vector Graphics
2. Representations of Vector Documents
3. Floating Point number representations

In summary, we have found:

- Rasterisation of Vector Graphics is non-trivial but well understood
- Traditionally rasterisation has been performed on the CPU and rendering on a dedicated GPU; current interest is in techniques for utilising the GPU directly to rasterise vector graphics.
- The popular standards for document formats including PostScript, PDF, HTML, SVG require IEEE-754 binary32 precision
- Fixed precision floating point numbers make a trade off between precision and range
- IEEE-754 is widely used although there are instances of languages or processors which do not conform exactly to the standard
- GPUs in particular may not conform to IEEE-754, trading some accuracy of operations for performance

4.2 Development of Testbed Software

We have produced a basic Document Viewer capable of rendering simple primitives under translation and scaling. OpenGL 3.1 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. Transformations can be performed on either the GPU or CPU

5. Performance of rendering can be measured

We have found the performance of coordinate transforms on the GPU to be far superior to the CPU. However, at large enough scales it becomes apparent that the GPU is performing operations at a lower precision than the CPU. See Figure **??**.

4.3 Floating Point Precision

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 arbitrary real numbers (which may themselves 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.

4.3.1 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 closest 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.

4.4 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⟩ and ⟨david's website probably I guess⟩[1]

4.5 Timeline

Deadlines enforced by the faculty of Engineering Computing and Mathematics are italicised. Tasks completed as of the submission of this report are struck through. [2].

[1]These are all actually on the same filesystem but it sounds impressive anyway

[2]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

Date	**Milestone**
1st May	Testbed Software (basic document format and viewer) completed and approaches for extending to allow infinite precision identified.
? 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 infinite 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 infinite 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.

5. Conclusion

This report has provided motivation for considering approaches to achieving an infinite level of zoom in a document.

5.1 Acheived Milestones

5.2 Areas of further work

- Continue looking for relevant literature
- Implement all those tests mentioned in Chapter 1
- Actually identify the techniques I will use **THIS ONE SHOULD BE DONE BEFORE I HAND IN THE LITERATURE REVIEW!**
- Possible Ultimate Goal: Implement (a subset) of SVG and then show an SVG document that we can render but a browser can't
  - This means extending our viewer to be able to read (a subset) SVG
  - Can already read XML, so this shouldn't actually be too bad
    - Emphasis on **subset**
    - (I've seen the SVG standard; I'm talking about implementing the 18 pages under "Basic Shapes". The other 818 pages can complain to someone who cares.)
  - Suggestion to David that he probably won't like (or read): Make his octree structure specifiable as an SVG extension

5.3 Witty Conclusion Goes Here

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

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

[12] 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.

[13] 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). WC3 Recommendation, August 2011.

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

[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] 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] 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.

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

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

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

[25] Daniel Rice and RJ Simpson. Openvg specification, version 1.1. Khronos Group, 2008.

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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