Infinite Precision Document Formats

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

THE UNIVERSITY OF
WESTERN AUSTRALIA
Achieve International Excellence

May 8, 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

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Overview . . . 1

2 Proposal **2**

2.1 Aim . . . 2

2.1.1 Clarification of Terms . . . 2

2.2 Methods . . . 2

2.3 Software and Hardware Requirements . . . 4

2.4 Timeline . . . 5

3 Literature Review **6**

3.1 Raster and Vector Images . . . 7

3.2 Rasterising Vector Images . . . 8

3.2.1 Straight Lines . . . 9

3.2.2 Spline Curves . . . 10

3.2.3 Spline Curves . . . 11

3.2.4 Shading . . . 11

3.2.5 Rendering Vector Graphics on the GPU . . . 11

3.3 Document Representations . . . 11

3.3.1 Interpreted Model . . . 11

3.3.2 Crippled Interpreted Model . . . 12

3.3.3 Document Object Model . . . 12

3.3.4 Blurring the Line — Javascript . . . 13

3.3.5 Why do we still use static PDFs . . . 13

3.4 Precision in Modern Document Formats . . . 14

3.5 Representation of Numbers . . . 15

3.5.1 The IEEE Standard . . . 15

3.5.2 Floating Point Number Representations . . . 15

3.5.3 Limitations Imposed By CPU . . . 15

3.5.4 Limitations Imposed By Graphics APIs and/or GPUs . . . 16

3.5.5 Alternate Number Representations . . . 16

4 Progress Report **17**

4.1 Development of Testbed Software . . . 17

4.2 Design and Implementation of "Tests" . . . 17

4.3 Document Format . . . 17

4.4 Floating Point Number Representations 17
4.5 Virtual FPU 17
4.6 Version Control 17

5 Conclusion **18**
5.1 Acheived Milestones 18
5.2 Areas of further work 18
5.3 Witty Conclusion Goes Here 18

References **20**

List of Figures

3.1 Original Vector and Raster Images 8

3.2 Scaled Vector and Raster Images 8

3.3 Rasterising a Straight Line 10

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. We will refer to such a document format as "infinite precision". 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 "infinite 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 a document to be implemented using a single global coordinate system. However, 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.

2.4 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.
? 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.

[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

3. Literature Review

0. Here is a brilliant summary of the sections below

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].

Consider the right side of Figure 3.1. This is a raster image which should be recognisable as an animal defined by fairly sharp edges. Figure 3.2 shows that zooming on the animal's face causes these edges to appear jagged. 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" (See Section 3.2.1).[1]

In contrast, the left sides of Figures 3.1 and 3.2 are a vector image. A vector image contains information about a number of geometric shapes. To display this image on modern display hardware, the 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 ??). Thus, vector images are well suited to high quality digital art[2] and text.

[1] The exact appearance of the images at different zoom levels will depend greatly on the PDF viewer or printer used to display this report. On the author's display using the Atril (1.6.0) Document viewer, the top images appear to be pixel perfect mirror images at a 100% scale. In the bottom raster image, antialiasing is not applied at zoom levels above 125% and the effect of scaling is quite noticeable.

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

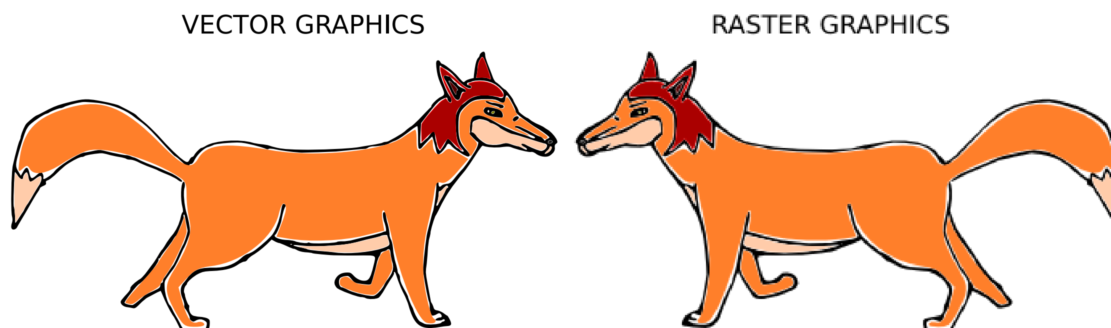

Figure 3.1: Original Vector and Raster Images

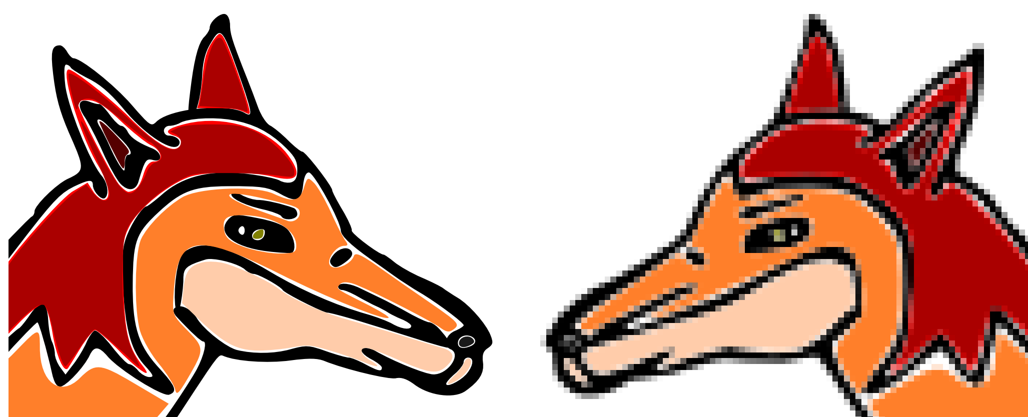

Figure 3.2: Scaled Vector and Raster Images

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].

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:

$$y(x) = mx + b \quad \text{on } x \in [x_1, x_2] \tag{3.1}$$

$$\text{for } m = (y_2 - y_1)/(x_2 - x_1) \tag{3.2}$$

$$\text{and } b \tag{3.3}$$

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...

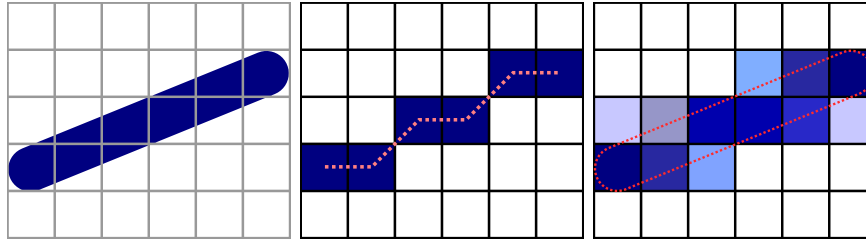

Figure 3.3: Rasterising a Straight Line

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:

$$y(x) = \sum_{k=0}^{n} a_k x^k \tag{3.4}$$

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[**?**].

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:

$$P(t) = \sum_{j=0}^{n} B_j^n(t) P_j \tag{3.5}$$

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. The cost of this modification is that the algorithm is no longer $O(n)$ but $O(n^2)$; although it is not explicitly stated by Goldman it seems clear that the modified algorithm is mainly of theoretical interest.

3.2.3 Spline Curves

A spline is a series of piecewise continuous bezier curves passing through "knot" points.

3.2.4 Shading

Algorithms for shading on vector displays involve drawing equally spaced lines within a region; this is limited both in the complexity of shading and the performance required to compute the lines[15].

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

3.2.5 Rendering Vector Graphics on the GPU

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, 23, 24, 22, 25] ... All of these are things David found except kilgard which I thought I found and then realised David already had it :S

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

3.3 Document Representations

The file format can be either human readable[3] or binary[4]. Can also be compressed or not. Here we are interested in how the document is interpreted or traversed in order to produce graphics output.

3.3.1 Interpreted Model

Did I just invent that terminology or did I read it in a paper? Is there actually existing terminology for this that sounds similar enough to "Document Object Model" for me to compare them side by side?

- This model treats a document as the source code program which produces graphics

[3]For some definition of human and some definition of readable
[4]So, our viewer is basically a DOM style but stored in a binary format

- 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[5]
- TeX— Predates PostScript! LaTeX is being used to create this very document and until now I didn't even have it here!
  - I don't really want to go down the path of investigating the billion steps involved in getting LaTeXinto an actually viewable format
  - There are interpreters (usually WYSIWYG editors) for LaTeXthough
  - Maybe if LaTeXwere more popular there would be desktop viewers that converted LaTeXdirectly into graphics
- Potential for dynamic content, interactivity; dynamic PostScript, enhanced Postscript
- Scientific Computing — Mathematica, Matlab, IPython Notebook — The document and the code that produces it are stored together
- 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

- DOM = Tree of nodes; node may have attributes, children, data
- XML (SGML) is the standard language used to represent documents in the DOM
- XML is plain text
- SVG is a standard for a vector graphics language conforming to XML (ie: a DOM format)
- CSS style sheets represent more complicated styling on objects in the DOM

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

3.3.4 Blurring the Line — Javascript

- The document is expressed in DOM format using XML/HTML/SVG
- A Javascript program is run which can modify the DOM
- At a high level this may be simply changing attributes of elements dynamically
- For low level control there is canvas2D and even WebGL which gives direct access to OpenGL functions
- Javascript can be used to make a HTML/SVG interactive
  - Overlooking the fact that the SVG standard already allows for interactive elements...
- Javascript is now becoming used even in desktop environments and programs (Windows 8, GNOME 3, Cinnamon, Game Maker Studio) (**shudder**)
- There are also a range of papers about including Javascript in PDF "Pixels or Perish" being the only one we have actually read[6]
  - I have no idea how this works; PDF is based on PostScript... it seems very circular to be using a programming language to modify a document that is modelled on being a (non turing complete) program
  - This is yet more proof that people will converge towards solutions that "work" rather than those that are optimal or elegant
  - I guess it's too much effort to make HTML look like PDF (or vice versa) so we could phase one out

3.3.5 Why do we still use static PDFs

Despite their limitations, we still use static, boring old PDFs. Particularly in scientific communication.

- They are portable; you can write an amazing document in Mathematica/Matlab but it
- Scientific journals would need to adapt to other formats and this is not worth the effort
- No network connection is required to view a PDF (although DRM might change this?)
- All rescources are stored in a single file; a website is stored accross many separate files (call this a "distributed" document format?)

- You can create PDFs easily using desktop processing WYSIWYG editors; WYSIWYG editors for web based documents are worthless due to the more complex content
- Until Javascript becomes part of the PDF standard, including Javascript in PDF documents will not become widespread
- Once you complicate a PDF by adding Javascript, it becomes more complicated to create; it is simply easier to use a series of static figures than to embed a shader in your document. Even for people that know WebGL.

3. Here are the ways document standards specify precision (or don't)

3.4 Precision in Modern Document Formats

All the above is very interesting and provides important context, but it is not actually directly related to the problem of infinite precision which we are going to try and solve. Sorry to make you read it all.

- Implementations of PostScript and PDF must by definition restrict themselves to IEEE binary32 "single precision"[6] floating point number representations in order to conform to the standards[3, 5].
- Implementations of SVG are by definition required to use IEEE binary32 as a **minimum**. "High Quality" SVG viewers are required to use at least IEEE binary64.[13]
- 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 [26] both mention it being buggy.
  - Octave and Matlab use fixed precision doubles

The use of IEEE binary32 floats in the PostScript and PDF standards is not surprising if we consider that these documents are oriented towards representing static pages. They don't actually need higher precision to do this; 32 bits is more than sufficient for A4 paper.

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

[6] The original IEEE-754 defined single, double and extended precisions; in the revision these were renamed to binary32, binary64 and binary128 to explicitly state the base and number of bits

3.5 Representation of Numbers

Although this project has been motivated by a desire for more flexible document formats, the fundamental source of limited precision in vector document formats is the restriction to IEEE floating point numbers for representation of coordinates.

Whilst David Gow will be focusing on structures and the use of multiple coordinate systems to represent a document so as to avoid or reduce these limitations[2], the focus of our own research will be increased precision in the representation of real numbers so as to get away with using a single global coordinate system.

3.5.1 The IEEE Standard

3.5.2 Floating Point Number Representations

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

$B = 2$, although IEEE also defines decimal representations for $B = 10$ — these are useful in financial software[27].

Aside: Are decimal representations for a document format eg: CAD also useful because you can then use metric coordinate systems?

Precision

The floats map an infinite set of real numbers onto a discrete set of representations.

Figure: 8 bit “minifloats” (all 255 of them) clearly showing the “precision vs range” issue

The most a result can be rounded in conversion to a floating point number is the units in last place; $m_N \times B^e$.

Even though that paper that claims double is the best you will ever need because the error can be as much as the size of a bacterium relative to the distance to the moon[] there are many cases where increased number of bits will not save you.[12]

5. Here are limitations of IEEE-754 floating point numbers on compatible hardware

3.5.3 Limitations Imposed By CPU

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

3.5.4 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[28]
- 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[24]
- 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)

3.5.5 Alternate Number Representations

They exist[12].

Do it all using MFPR[], 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.

4. Progress Report

This chapter outlines the current state of our research in relation to the aims outlined in Chapter 1. It will serve as an explanation for where the Figures in Chapter 3 came from. It will just be a short summary of the implementation details.

4.1 Development of Testbed Software

We wrote a very simple OpenGL 1.1 program to experiment with, and then David Gow converted it to OpenGL 3.1 and I have no idea how it works anymore.

4.2 Design and Implementation of "Tests"

- Compile by swapping out `main()` for a tester
- There are tests for doing some of the things in Chapter 1 but most still aren't written yet.

4.3 Document Format

Currently we effectively have a DOM format but with the following non-features:

- Binary file format (non standard; not XML)
- Only rectangles.

4.4 Floating Point Number Representations

I have[1] some figures that I would prefer to include in Chapter 3 when I am talking about the papers that inspired them. This section will probably briefly talk about how they were created and just refer back to them.

- `calculatepi.test`
- `typedef` of `Real`

4.5 Virtual FPU

Techniques for dealing with FP numbers can be implemented in software (CPU) or on dedicated hardware (FPU). We are able to run FP arithmetic on arbitrary simulations of FPUs created using VHDL. Hopefully explore this a bit in Chapter 3.

4.6 Version Control

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⟩

[1] Ok... "will have"

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

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

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

[26] 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.

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

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