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 21, 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

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 widely used standards[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, 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 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 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 **??**, 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.

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

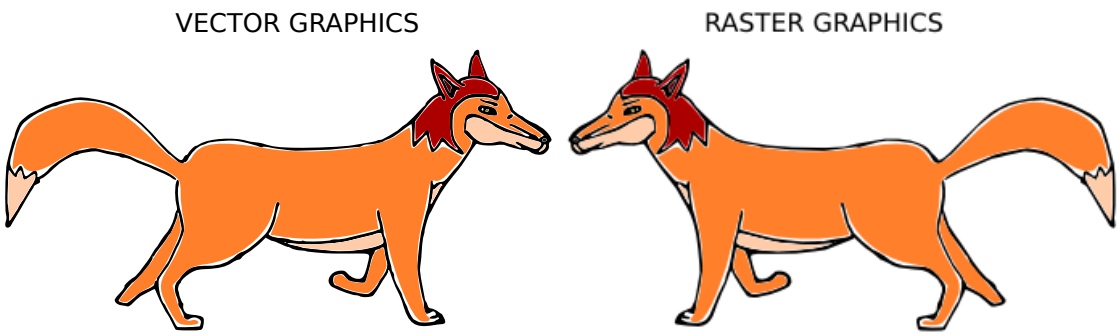

Figure 3.1: Original Vector and Raster Images

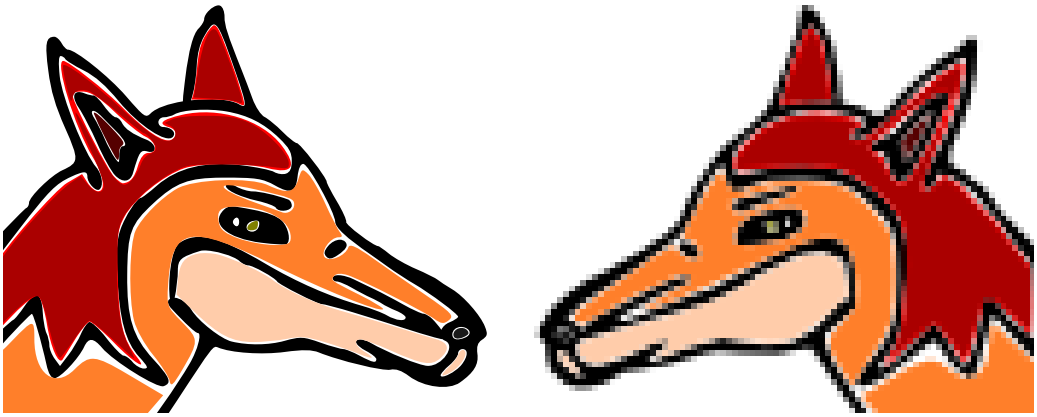

Figure 3.2: Scaled Vector and Raster Images

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]. 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.

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

It should be noted that algorithms for drawing lines can be based upon sampling $y(x)$ only if $|m| \leq 1$; if $|m| > 1$ then sampling at every integer for x would leave gaps in the line. However 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[16]. In 1991 Wu introduced an algorithm for drawing anti-aliased lines which, while equivelant in results to existing algorithms by Fujimoto and Iwata, set the state of the art in performance[19][2]. .

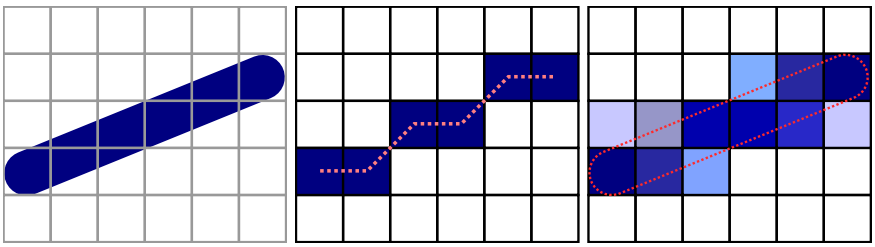

Figure 3.3: Rasterising a Straight Line

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

[2]Techniques for anti-aliasing primitives other than straight lines, including the anti-aliasing of images in a raster format shown at different scales (such as Figure **??**) are discussed in some detail in Chapter 4 of "Computer Graphics" [10]

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:

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

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, special polynomials may be defined using "control" points which themselves are not part of the curve; these are convenient for graphical based editors. Bezier splines are the most straight forward way to define a curve in the standards considered in Section 3.3

Bezier Curves

Cubic beziers form all curves in the PostScript[3], PDF[5] and SVG[13] standards which we will discuss in Section 3.3. One of the shapes in Figure 3.5 is a region defined by a cubic bezier spline. Beziers are also used to construct vector fonts for rendering text in these standards.

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:

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

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 Font Rendering

Donald Knuth's 1986 textbook "Metafont" blargh

3.2.4 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 computationally expensive sorting of vertices[21].

On raster displays, shading is typically based upon Lane's algorithm of 1983[21]. Lane's algorithm relies on the ability to "subtract" fill from a region. This algorithm is now implemented in the GPU stencil buffer-y and... stuff [22]

3.2.5 Compositing and the Painter's Model

So far we have discussed techniques for rendering vector graphics primitives in isolation, with no regard to the overall structure of a document which may contain many thousands of primitives. A straight forward approach would be to render all elements sequentially to the display, with the most recently drawn pixels overwriting lower elements. Such an approach is particularly inconvenient for anti-aliased images where colours must appear to smoothly blur between the edge of a primitive and any drawn underneath it.

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[10]. In 1984, Porter and Duff introduced a fourth colour channel for rasterised images called the "alpha" channel, analogous to the transparency of a pixel[23]. 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, Porter and Duff's "over" operation is used when rendering one primitive over another[13]. 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:

$$a_T = 1 - (1 - a_1)(1 - a_2) \tag{3.4}$$

$$r_T = (1 - a_2)r_1 + r_2 \quad \text{(similar for } g_T \text{ and } b_T\text{)} \tag{3.5}$$

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 3.5 illustrates the painter's model for partially transparent shapes as they would appear in both the SVG and PDF models.

3.2.6 Rasterisation on the CPU and 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, 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.

In this section we will consider various approaches and motivations to specifying the structure and appearance of a document, including: early interpreted formats (PostScript, TEX, DVI), the Document Object Model popular in standards for web based documents (HTML, SVG), and Adobe's ubiquitous Portable Document Format (PDF). Some of these formats were discussed in a recent paper "Pixels Or Perish" by Hayes[**?**] who argues for greater interactivity in the PDF standard.

3.3.1 Interpreted Document Formats

TEXand Metafont

Donald Knuth's "The TEXBook" and "Metafont"

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. A special "graphics state" stack can be used to store primitive properties (such as colour, line thickness, the current cursor position). It is possible for the language to define new operators. Figure 3.4 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 recent 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 Section**??**, there is no concept of transparency. In fact, using tools to convert between the SVG image in Figure 3.5 and PostScript will simply rasterise the image and embed the rastered image in PostScript. 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"[?].

```
% 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.
 curveto 55.016 125.001 61.414 49.802 46.215 75.399 curveto 31.016
 56.613 126.598 curveto 56.613 126.598 88.613 166.598 56.613 137.8
109.001 -18.586 83.399 9.414 50.598 curveto 37.414 17.802 45.414 1
1.001 curveto closepath } bind def
% Set stroke properties
0.8 setlinewidth 0 setlinecap 0 setlinejoin [] 0.0 setdash 4 setmite
% 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
```

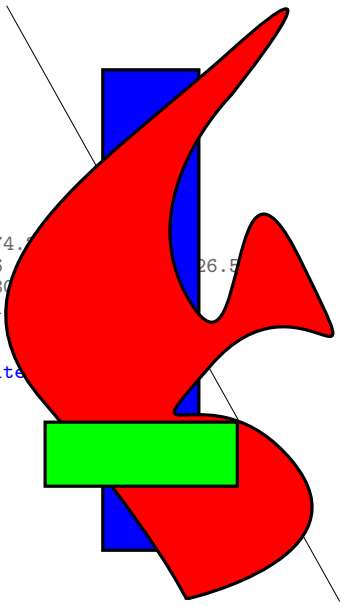

Figure 3.4: Vector image and a possible PostScript representation

- 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[3]
- 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]Desktop pdf viewers can still cope with PS, but I wonder if a smartphone pdf viewer would implement it?

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[27], HTML[28] and SVG[13]. 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)[29] 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 interpreted language Javascript included in the HTML5 standard can be used to dynamically alter the document as it is viewed in response to user input or other events such as communication with a remote server.

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 3.5 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.[4] 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

The W3C has produced a primer describing the use of HTML5 Javascript to produce interactive SVG's[30], and the HTML5 and SVG standards themselves include several examples discussing the use of Javascript to manipulate the DOM.

In 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 3.5 would change the fill colour of the curved shape from red `#ff0000` to black `#000000`:

[4] The exact details of classification of these languages (such as why HTML cannot be defined as a subset of XML) are beyond the scope of this report.

```
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 our own example to generate an SVG interactively using HTML. The example generates successive iterations of a particular type of fractal curve first described by Koch[31] in 1904 and a popular example in modern literature [**?**]. 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 3.6.

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[5]. 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">

<svg id="svg_example" xmlns="http://www.w3.org/2000/svg"
        version="1.1" width="104" height="186">
<path id="straightline" d = "m 0, 0 104, 186"
        style="stroke:#000000;"/>
<rect id="rect1"
        x = "30" y = "20" width = "30" height = "150"
        style = "fill:#0000ff; fill-opacity:0.5;
                stroke:#000000;"/>
<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;"/>
<rect id="rect2"
        x = "12" y = "130" width = "60" height = "20"
        style = "fill:#00ff00; fill-opacity:0.5;
                stroke:#000000;"/>
</svg>
```

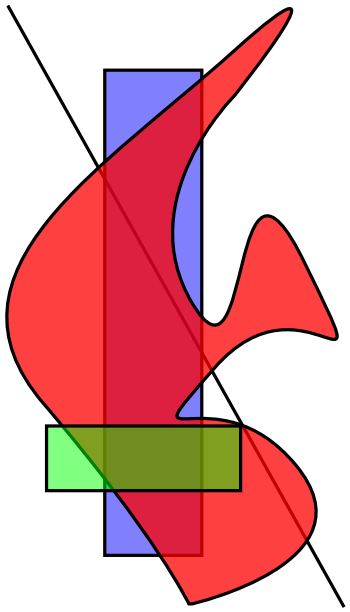

Figure 3.5: Vector image and a possible SVG representation

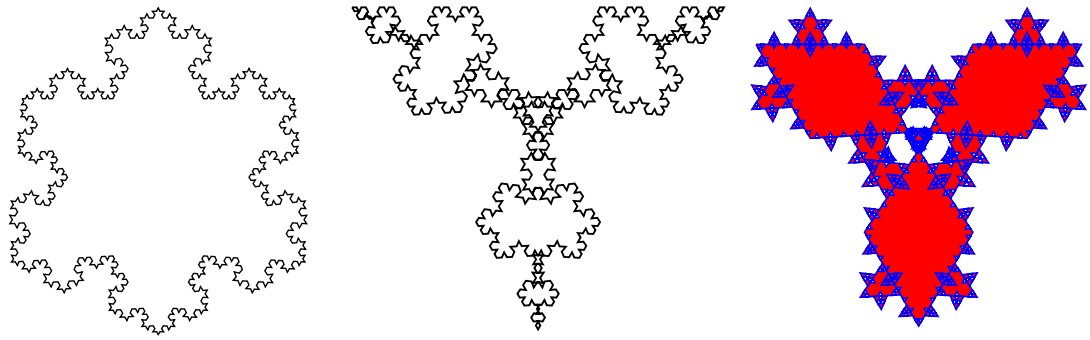

Figure 3.6: Koch “snowflakes” generated using Javascript to modify an SVG DOM. The interactive HTML5 document can be found at ⟨http://szmoore.net/ipdf/sam/figures/koch.html⟩

[5]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 used almost universally for sharing documents; the ability to export or print to PDF can be found in most graphical document editors, even 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."[**?**]. 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 discussed in Section 3.3.2. 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[32]. There is evidence in the literature of attempts to exploit these features, with mixed success[7, **?**].

To quote Adobe's PDF 1.7 reference manual, "A PDF file should be thought of as a flattened representation of a data structure consisting of a collection of objects that can refer to each other in any arbitrary way"[5].

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 [33] 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 the standards discussed so far in regards to the precision of mathematical operations:

- **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.
- **Python** uses binary64 floats
- **Matlab** uses binary64 floats
- **Mathematica** uses some kind of terrifying symbolic / arbitrary float combination
- **Maple** is similar but by many accounts horribly broken

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

Reorder to start with Integers, General Floats, then go to IEEE, then other things

3.5.1 IEEE Floating Points

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 original IEEE-754 standard specified the encoding, number of bits, rounding methods, and maximum acceptable errors for the basic floating point operations for base $B = 2$ floats. It also specifies "exceptions" — mechanisms by which a program can detect an error such as division by zero[6]. We will restrict ourselves to considering $B = 2$, since it was found that this base in general gives the smallest rounding errors[12], although it is worth noting that different choices of base had been used historically[**?**], and the IEEE-854 and later the revised IEEE-754 standard specify a decimal representation $B = 10$ intended for use in financial applications.

[6]Kahan has argued that exceptions in IEEE-754 are conceptually different to Exceptions as defined in several programming languages including C++ and Java. An IEEE exception is intended to prevent an error by its detection, whilst an exception in those languages is used to indicate an error has already occurred[]

3.5.2 Floating Point Definition

A floating point number x is commonly represented by a tuple of integers (s, e, m) in base B as[12, 34]:

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

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.

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, 34].

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 will in general 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 in 2008, 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[12].

Figure **??** shows the positive 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 $2^7 = 256$ 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 2 and 5 bits to encode e and m whilst the second set of points corresponds to a 3 and 4 bit encoding. This allows us to see the trade off between the precision and range of real values represented.

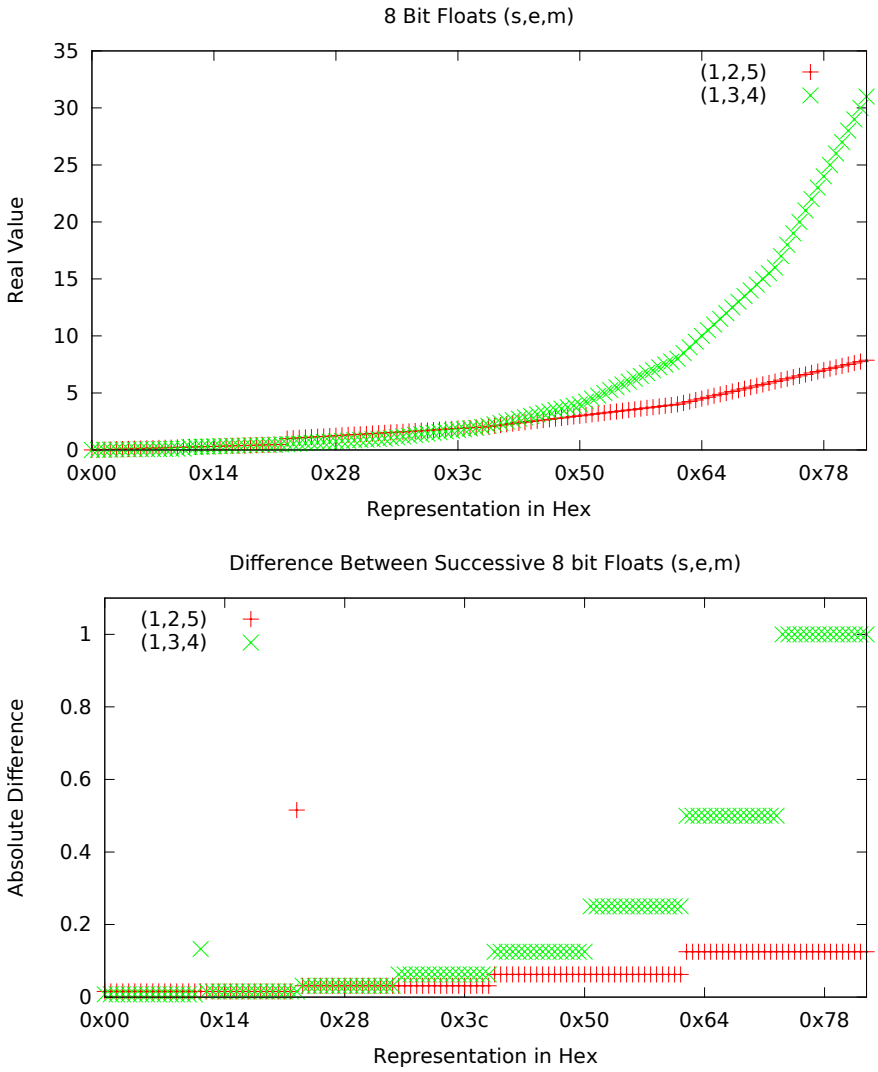

Figure 3.7: The mapping of 8 bit floats to reals

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[35]; accuracy of operations[36]; and even the adaptation of algorithms originally used in software for reducing the overal error of a sequence of operations[37]. In this section we will consider the algorithms for floating point operations without focusing on the hardware implementation of these algorithms.

3.5.5 Some sort of Example(s) or Floating Point Mayhem

Eg: $f(x) = |x|$ calculated from sqrt and squaring

Eg: Massive rounding errors from calculatepi

Eg: Actual graphics things :S

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[38]
- 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.
- Blog post alert ⟨https://dolphin-emu.org/blog/2014/03/15/pixel-processing-problems/⟩

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

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

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

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

[31] 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.

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

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

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