B.Eng. Final Year Project

Literature Notes

Sam Moore, David Gow
Faculty of Engineering, Computing and Mathematics, University of Western Australia
March 2014

Contents

Chapter 1

Literature Summaries

1.1 Postscript Language Reference Manual [?]

Adobe's official reference manual for PostScript.

It is big.

- First version was published BEFORE the IEEE standard and used smaller floats than binary32
- Now uses binary32 floats.

1.2 Portable Document Format Reference Manual [?]

Adobe's official reference for PDF.

It is also big.

- Early versions did not use IEEE binary32 but 16-16 exponent/mantissa encodings (Why?)
- Current standard is restricted to binary32
- It specifically says PDF creators must use at most binary32 because higher precision is not supported by Adobe Reader.

1.3 IEEE Standard for Floating-Point Arithmetic [?]

The IEEE (revised) 754 standard.

It is also big.

Successes:

- Has been adopted by CPUs
- Standardised floats for programmers — accomplishes goal of allowing non-numerical experts to write reasonably sophisticated platform independent programs that may perform complex numerical operations

Failures:

- Adoption by GPUs slower[?]
- It specifies the maximum errors for operations using IEEE types but nothing about internal representations
- Many hardware devices (GPUs and CPUs) use non-IEEE representations internally and simply truncate/round the result

  - This isn't so much of a problem when the device uses additional bits but it is misleading when GPUs use less than binary32 and act as if they are using binary32 from the programmer's perspective.
  - Devices using **less** bits internally aren't IEEE compliant
- Thus the same program compiled and run on different architectures may give completely different results[**?**]
  - The ultimate goal of allowing people to write numerical programs in total ignorance of the hardware is not entirely realised
- This is the sort of thing that makes people want to use a virtual machine, and thus Java
  - Objectively I probably shouldn't say that using Java is in itself a failure
- Standards such as PostScript and PDF were slow to adopt IEEE representations
- The OpenVG standard accepts IEEE binary32 in the API but specifically states that hardware may use less than this[**?**]

1.4 Portable Document Format (PDF) — Finally... [?]

This is not spectacularly useful, is basically an advertisement for Adobe software.

Intro

- Visual communications has been revolutionised by computing
- BUT there have always been problems in exchanging formats
- Filetypes like text, rich text, IGES, DXF, TIFF, JPEG, GIFF solve problems for particular types of files only
- PDF solves everything for everyone; can include text, images, animation, sound, etc

PDF Features

- Raster Image Process (RIP) — For printing (presumably also displaying on screen)
- Originally needed to convert to PS then RIP, with PS 3 can now RIP directly.
- Reduced filesize due to compression
- Four major applications - Stoy 1999[**?**]
  1. Download files from internet
  2. Files on CDs
  3. Files for outputing to printers
  4. Conventional [commercial scale?] printing
- List of various (Adobe) PDF related software
  - Includes software for PS that converts to/from PDF
  - So PS was obviously pretty popular before PDF
- Can Optimize for screen/printer [not clear how]
- Can compress for size

1.5 Pixels or Perish [?]

"The art of scientific illustration will have to adapt to the new age of online publishing" And therefore, JavaScript libraries (D^3) are the future.

The point is that we need to change from thinking about documents as paper to thinking of them as pixels. This kind of makes it related to our paper, because it is the same way we are justifying our project. It does mention precision, but doesn't say we need to get more of it.

I get the feeling from this that Web based documents are a whole bunch of completely different design philosophies hacked together with JavaScript.

This paper uses Metaphors a lot. I never met a phor that didn't over extend itself.

Intro

- Drawings/Pictures are ornaments in science but they are not just ornamental
- Processes have changed a lot; eg: photographic plates → digital images
- "we are about to turn the page — if not close the book — on yet another chapter in publishing history." (HO HO HO)
- It would be cool to have animated figures in documents (eg: Population pyramid; changes with time); not just as "supplements"
- In the beginning, there was PostScript, 1970s and 1980s, John Warnock and Charles Geschke, Adobe Systems
- PS is a language for vector graphics; objects are constructed from geometric primitives rather than a discrete array of pixels
- PS is a complete programming language; an image is also a program; can exploit this to control how images are created based on data (eg: Faces)
- PDF is "flattened" PS. No longer programable. Aspires to be "virtual paper".
- But why are we using such powerful computing machines just to emulate sheets paper? (the author asks)

Web based Documents

- HTML, CSS, JavaScript - The Axis of Web Documents
  - HTML - Defines document structure
  - CSS - Defines presentation of elements in document
  - JavaScript - Encodes actions, allows dynamic content (change the HTML/CSS)
- `<canvas>` will let you draw anything (So in principle don't even need all of HTML/CSS)
  - Not device independent
  - "Coordinates can be specified with precision finer than pixel resolution" **(TODO: Investigate this?)**
  - JavaScript operators to draw things on canvas are very similar to the PostScript model
- SVG — Same structure (Document Object Model (DOM)) as HTML
  - "Noun language"
  - Nouns define lines/curves etc, rather than paragraphs/lists
  - Also borrows things from PostScript (eg: line caps and joints)
  - IS device independent, "very high precision" **(TODO: Investigate)**
  - JavaScript can be used to interact with SVG too
- D^3 (Data Driven Documents) - A JavaScript library
  - Idea is to create or modify elements of a DOM document using supplied data
  - ⟨https://github.com/mbostock/d3/wiki⟩

- We are in a new Golden Age of data visualisation
- Why do we still use PDFs?
  - PDFs are "owned" by the author/reader; you download it, store it, you can print it, etc
  - HTML documents are normally on websites. They are not self contained. They often rely on remote content from other websites (annoying to download the whole document).
- **Conclusion** Someone should open up PDF to accept things like D^3 and other graphics formats (links nicely with [**?**])
- Also, Harry Potter reference

1.6 Embedding and Publishing Interactive, 3D Figures in PDF Files [?]

- Linkes well with [**?**]; I heard you liked figures so I put a figure in your PDF
- Title pretty much summarises it; similar to [**?**] except these guys actually did something practical

1.7 27 Bits are not enough for 8 digit accuracy [?]

Proves with maths, that rounding errors mean that you need at least q bits for p decimal digits. $10^p < 2^{q-1}$

- Eg: For 8 decimal digits, since $10^8 < 2^{27}$ would expect to be able to represent with 27 binary digits
- But: Integer part requires digits bits (regardless of fixed or floating point represenetation)
- Trade-off between precision and range
  - $9000000.0 \rightarrow 9999999.9$ needs 24 digits for the integer part $2^{23} = 83886008$
- Floating point zero = smallest possible machine exponent
- Floating point representation:

$$y = 0.y_1y_2...y_q \times 2^n$$

- Can eliminate a bit by considering whether $n = -e$ for $-e$ the smallest machine exponent (???)
  - Get very small numbers with the same precision
  - Get large numbers with the extra bit of precision

1.8 What every computer scientist should know about floating-point arithmetic [?]

- Book: *Floating Point Computation* by Pat Sterbenz (out of print... in 1991)
- IEEE floating point standard becoming popular (introduced in 1987, this is 1991)
  - As well as structure, defines the algorithms for addition, multiplication, division and square root
  - Makes things portable because results of operations are the same on all machines (following the standard)
  - Alternatives to floating point: Floating slasi and Signed Logarithm (TODO: Look at these, although they will probably not be useful)
- Base β and precision p (number of digits to represent with) - powers of the base can be represented exactly.
- Largest and smallest exponents e_{min} and e_{max}
- Need bits for exponent and fraction, plus one for sign
- "Floating point number" is one that can be represented exactly.
- Representations are not unique! $0.01 \times 10^1 = 1.00 \times 10^{-1}$ Leading digit of one $\implies$ "normalised"
- Requiring the representation to be normalised makes it unique, **but means it is impossible to represent zero**.

- Represent zero as $1 \times \beta^{e_{min}-1}$ - requires extra bit in the exponent

- **Rounding Error**
  - "Units in the last place" eg: 0.0314159 compared to 0.0314 has ulp error of 0.159
  - If calculation is the nearest floating point number to the result, it will still be as much as 1/2 ulp in error
  - Relative error corresponding to 1/2 ulp can vary by a factor of β "wobble". Written in terms of ϵ
  - Maths $\implies$ **Relative error is always bounded by** $\epsilon = (\beta/2)\beta^{-p}$
  - Fixed relative error $\implies$ ulp can vary by a factor of β . Vice versa
  - Larger $\beta \implies$ larger errors

- **Guard Digits**
  - In subtraction: Could compute exact difference and then round; this is expensive
  - Keep fixed number of digits but shift operand right; discard precision. Lead to relative error up to $\beta - 1$
  - Guard digit: Add extra digits before truncating. Leads to relative error of less than 2ϵ. This also applies to addition

- **Catastrophic Cancellation** - Operands are subject to rounding errors - multiplication
- **Benign Cancellation** - Subtractions. Error $< 2\epsilon$
- Rearrange formula to avoid catastrophic cancellation
- Historical interest only - speculation on why IBM used $\beta = 16$ for the system/370 - increased range? Avoids shifting
- Precision: IEEE defines extended precision (a lower bound only)
- Discussion of the IEEE standard for operations (TODO: Go over in more detail)
- NaN allow continuing with underflow and Infinity with overflow
- "Incidentally, some people think that the solution to such anomalies is never to compare floating-point numbers for equality but instead to consider them equal if they are within some error bound E. This is hardly a cure all, because it raises as many questions as it answers." - On equality of floating point numbers

1.9 Compositing Digital Images [?]

Perter and Duff's classic paper "Compositing Digital Images" lays the foundation for digital compositing today. By providing an "alpha channel," images of arbitrary shapes and images with soft edges or sub-pixel coverage information can be overlayed digitally, allowing separate objects to be rasterized separately without a loss in quality.

Pixels in digital images are usually represented as 3-tuples containing (red component, green component, blue component). Nominally these values are in the [0-1] range. In the Porter-Duff paper, pixels are stored as (R, G, B, α) 4-tuples, where alpha is the fractional coverage of each pixel. If the image only covers half of a given pixel, for example, its alpha value would be 0.5.

To improve compositing performance, albeit at a possible loss of precision in some implementations, the red, green and blue channels are premultiplied by the alpha channel. This also simplifies the resulting arithmetic by having the colour channels and alpha channels use the same compositing equations.

Several binary compositing operations are defined:

- over
- in
- out
- atop
- xor
- plus

The paper further provides some additional operations for implementing fades and dissolves, as well as for changing the opacity of individual elements in a scene.

The method outlined in this paper is still the standard system for compositing and is implemented almost exactly by modern graphics APIs such as `OpenGL`. It is all but guaranteed that this is the method we will be using for compositing document elements in our project.

1.10 Bresenham's Algorithm: Algorithm for computer control of a digital plotter [?]

Bresenham's line drawing algorithm is a fast, high quality line rasterization algorithm which is still the basis for most (aliased) line drawing today. The paper, while originally written to describe how to control a particular plotter, is uniquely suited to rasterizing lines for display on a pixel grid.

Lines drawn with Bresenham's algorithm must begin and end at integer pixel coordinates, though one can round or truncate the fractional part. In order to avoid multiplication or division in the algorithm's inner loop,

The algorithm works by scanning along the long axis of the line, moving along the short axis when the error along that axis exceeds 0.5px. Because error accumulates linearly, this can be achieved by simply adding the per-pixel error (equal to (short axis/long axis)) until it exceeds 0.5, then incrementing the position along the short axis and subtracting 1 from the error accumulator.

As this requires nothing but addition, it is very fast, particularly on the older CPUs used in Bresenham's time. Modern graphics systems will often use Wu's line-drawing algorithm instead, as it produces antialiased lines, taking sub-pixel coverage into account. Bresenham himself extended this algorithm to produce Bresenham's circle algorithm. The principles behind the algorithm have also been used to rasterize other shapes, including Bézier curves.

1.11 Quad Trees: A Data Structure for Retrieval on Composite Keys [?]

This paper introduces the "quadtree" spatial data structure. The quadtree structure is a search tree in which every node has four children representing the north-east, north-west, south-east and south-west quadrants of its space.

1.12 Xr: Cross-device Rendering for Vector Graphics [?]

Xr (now known as Cairo) is an implementation of the PDF v1.4 rendering model, independent of the PDF or PostScript file formats, and is now widely used as a rendering API. In this paper, Worth and Packard describe the PDF v1.4 rendering model, and their PostScript-derived API for it.

The PDF v1.4 rendering model is based on the original PostScript model, based around a set of *paths* (and other objects, such as raster images) each made up of lines and Bézier curves, which are transformed by the "Current Transformation Matrix." Paths can be *filled* in a number of ways, allowing for different handling of self-intersecting paths, or can have their outlines *stroked*. Furthermore, paths can be painted with RGB colours and/or patterns derived from either previously rendered objects or external raster images. PDF v1.4 extends this to provide, amongst other features, support for layering paths and objects using Porter-Duff compositing[**?**], giving each painted path the option of having an α value and a choice of any of the Porter-Duff compositing methods.

The Cairo library approximates the rendering of some objects (particularly curved objects such as splines) with a set of polygons. An `XrSetTolerance` function allows the user of the library to set an upper bound on the approximation error in fractions of device pixels, providing a trade-off between rendering quality and performance. The library developers found that setting the tolerance to greater than 0.1 device pixels resulted in errors visible to the user.

1.13 Glitz: Hardware Accelerated Image Compositing using OpenGL [?]

This paper describes the implementation of an `OpenGL` based rendering backend for the `Cairo` library.

The paper describes how OpenGL's Porter-Duff compositing is easily suited to the Cairo/PDF v1.4 rendering model. Similarly, traditional OpenGL (pre-version 3.0 core) support a matrix stack of the same form as Cairo.

The "Glitz" backend will emulate support for tiled, non-power-of-two patterns/textures if the hardware does not support it.

Glitz can render both triangles and trapezoids (which are formed from pairs of triangles). However, it cannot guarantee that the rasterization is pixel-precise, as OpenGL does not proveide this consistently.

Glitz also supports multi-sample anti-aliasing, convolution filters for raster image reads (implemented with shaders).

Performance was much improved over the software rasterization and over XRender accellerated rendering on all except nVidia hardware. However, nVidia's XRender implementation did slow down significantly when some transformations were applied.

In [**?**], Kilgard mentions that Glitz has been abandoned. He describes it as "GPU assisted" rather than GPU accelerated, since it used the XRender (??) extension.

1.14 Boost Multiprecision Library

- "The Multiprecision Library provides integer, rational and floating-point types in C++ that have more range and precision than C++'s ordinary built-in types."
- Specify number of digits for precision as a template argument.
- Precision is fixed... **possible approach to project:** Use `boost::mpf_float<N>` and increase `N` as more precision is required?

1.15 A CMOS Floating Point Unit [?]

The paper describes the implentation of a FPU for PowerPC using a particular Hewlett Packard process (HP14B 0.5μm, 3M, 3.3V). It implements a "subset of the most commonly used double precision floating point instructions". The unimplemented operations are compiled for the CPU.

The paper gives a description of the architecture and design methods. This appears to be an entry to a student design competition.

Standard is IEEE 754, but the multiplier tree is a 64-bit tree instead of a 54 bit tree. " The primary reason for implementing a larger tree is for future additions of SIMD [Single Instruction Multiple Data (?)] instructions similar to Intel's MMX and Sun's VIS instructions".

HSPICE simulations used to determine transistor sizing.

Paper has a block diagram that sort of vaguely makes sense to me. The rest requires more background knowledge.

1.16 Simply FPU[?]

This is a webpage at one degree of seperation from wikipedia.

It talks about FPU internals, but mostly focuses on the instruction sets. It includes FPU assembly code examples (!)

It is probably not that useful, I don't think we'll end up writing FPU assembly?

FPU's typically have 80 bit registers so they can support REAL4, REAL8 and REAL10 (single, double, extended precision).

Note: Presumably this is referring to the x86 80 bit floats that David was talking about?

1.17 Floating Point Package User's Guide [?]

This is a technical report describing floating point VHDL packages ⟨http://www.vhdl.org/fphdl/vhdl.html⟩

In theory I know VHDL (cough) so I am interested in looking at this further to see how FPU hardware works. It might be getting a bit sidetracked from the "document formats" scope though.

The report does talk briefly about the IEEE standard and normalised / denormalised numbers as well.

See also: Java Optimized Processor[**?**] (it has a VHDL implementation of a FPU).

1.18 Low-Cost Microarchitectural Support for Improved Floating-Point Accuracy[?]

Mentions how GPUs offer very good floating point performance but only for single precision floats. (NOTE: This statement seems to contradict [**?**].

Has a diagram of a Floating Point adder.

Talks about some magical technique called "Native-pair Arithmetic" that somehow makes 32-bit floating point accuracy "competitive" with 64-bit floating point numbers.

1.19 Accurate Floating Point Arithmetic through Hardware Error-Free Transformations [?]

From the abstract: "This paper presents a hardware approach to performing ac- curate floating point addition and multiplication using the idea of error- free transformations. Specialized iterative algorithms are implemented for computing arbitrarily accurate sums and dot products."

The references for this look useful.

It also mentions VHDL.

So whenever hardware papers come up, VHDL gets involved... I guess it's time to try and work out how to use the Opensource VHDL implementations.

This is about reduction of error in hardware operations rather than the precision or range of floats. But it is probably still relevant.

This has the Fast2Sum algorithm but for the love of god I cannot see how you can compute anything other than $a + b = 0 \forall a, b$ using the algorithm as written in their paper. It references Dekker[?] and Kahn; will look at them instead.

1.20 Floating Point Unit from JOP [?]

This is a 32 bit floating point unit developed for JOP in VHDL. I have been able to successfully compile it and the test program using GHDL[?].

Whilst there are constants (eg: `FP_WIDTH = 32, EXP_WIDTH = 8, FRAC_WIDTH = 23`) defined, the actual implementation mostly uses magic numbers, so some investigation is needed into what, for example, the "52" bits used in the sqrt units are for.

1.21 GHDL [?]

GHDL is an open source GPL licensed VHDL compiler written in Ada. It had packages in debian up until wheezy when it was removed. However the sourceforge site still provides a `deb` file for wheezy.

This reference explains how to use the `ghdl` compiler, but not the VHDL language itself.

GHDL is capable of compiling a "testbench" - essentially an executable which simulates the design and ensures it meets test conditions. A common technique is using a text file to provide the inputs/outputs of the test. The testbench executable can be supplied an argument to save a `vcd` file which can be viewed in `gtkwave` to see timing diagrams.

Sam has successfully compiled the VHDL design for an FPU in JOP[?] into a "testbench" executable which uses standard i/o instead of a regular file. Using unix domain sockets we can execute the FPU as a child process and communicate with it from our document viewing test software. This means we can potentially simulate alternate hardware designs for FPUs and witness the effect they will have on precision in the document viewer.

Using `ghdl` the testbench can also be linked as part a C/C++ program and run using a function; however there is still no way to communicate with it other than forking a child process and using a unix domain socket anyway. Also, compiling the VHDL FPU as part of our document viewer would clutter the code repository and probably be highly unportable. The VHDL FPU has been given a seperate repository.

1.22 On the design of fast IEEE floating-point adders [?]

This paper gives an overview of the "Naive" floating point addition/subtraction algorithm and gives several optimisation techniques:

TODO: Actually understand these...

- Use parallel paths (based on exponent)
- Unification of significand result ranges
- Reduction of IEEE rounding modes
- Sign-magnitude computation of a difference
- Compound Addition

- Approximate counting of leading zeroes
- Pre-computation of post-normalization shift

They then give an implementation that uses these optimisation techniques including very scary looking block diagrams.

They simulated the FPU. Does not mention what simulation method was used directly, but cites another paper (TODO: Look at this. I bet it was VHDL).

The paper concludes by summarising the optimisation techniques used by various adders in production (in 2001).

This paper does not specifically mention the precision of the operations, but may be useful because a faster adder design might mean you can increase the precision.

1.23 Re: round32 (round64 (X)) ?= round32 (X) [?]

I included this just for the quote by Nelson H. F. Beebe:

"It is too late now to repair the mistakes of the past that are present in millions of installed systems, but it is good to know that careful research before designing hardware can be helpful."

This is in regards to the problem of double rounding. It provides a reference for a paper that discusses a rounding mode that eliminates the problem, and a software implementation.

It shows that the IEEE standard can be fallible!

Not sure how to work this into our literature review though.

1.24 Basic Issues in Floating Point Arithmetic and Error Analysis [?]

These are lecture notes from U.C Berkelye CS267 in 1996.

1.25 Charles Babbage [?, ?]

Tributes to Charles Babbage. Might be interesting for historical background. Don't mention anything about floating point numbers though.

1.26 GPU Floating-Point Paranoia [?]

This paper discusses floating point representations on GPUs. They have reproduced the program *Paranoia* by William Kahan for characterising floating point behaviour of computers (pre IEEE) for GPUs.

There are a few remarks about GPU vendors not being very open about what they do or do not do with

Unfortunately we only have the extended abstract, but a pretty good summary of the paper (written by the authors) is at: ⟨www.cs.unc.edu/ ibr/projects/paranoia/⟩

From the abstract:

"... [GPUs are often similar to IEEE] However, we have found that GPUs do not adhere to IEEE standards for floating-point op- erations, nor do they give the information necessary to establish bounds on error for these operations ... "

and "...Our goal is to determine the error bounds on floating-point op- eration results for quickly evolving graphics systems. We have cre- ated a tool to measure the error for four basic floating-point opera- tions: addition, subtraction, multiplication and division."

The implementation is only for windows and uses glut and glew and things. Implement our own version?

1.27 A floating-point technique for extending the available precision [?]

This is Dekker's formalisation of the Fast2Sum algorithm originally implemented by Kahn.

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

There is a version for multiplication.

I'm still not quite sure when this is useful. I haven't been able to find an example for x and y where $x + y \neq \mathrm{Fast2Sum}(x, y)$.

1.28 Handbook of Floating-Point Arithmetic [?]

This book is amazingly useful and pretty much says everything there is to know about Floating Point numbers. It is much easier to read than Goldberg or Priest's papers.

I'm going to start working through it and compile their test programs.

1.28.1 A sequence that seems to converge to a wrong limit - pgs 9-10, [?]

$$
u_n = \begin{cases} u_0 = 2 \\ u_1 = -4 \\ u_n = 111 - \frac{1130}{u_{n-1}} + \frac{3000}{u_{n-1}u_{n-2}} \end{cases}
$$

The limit of the series should be 6 but when calculated with IEEE floats it is actually 100 The authors show that the limit is actually 100 for different starting values, and the error in floating point arithmetic causes the series to go to that limit instead.

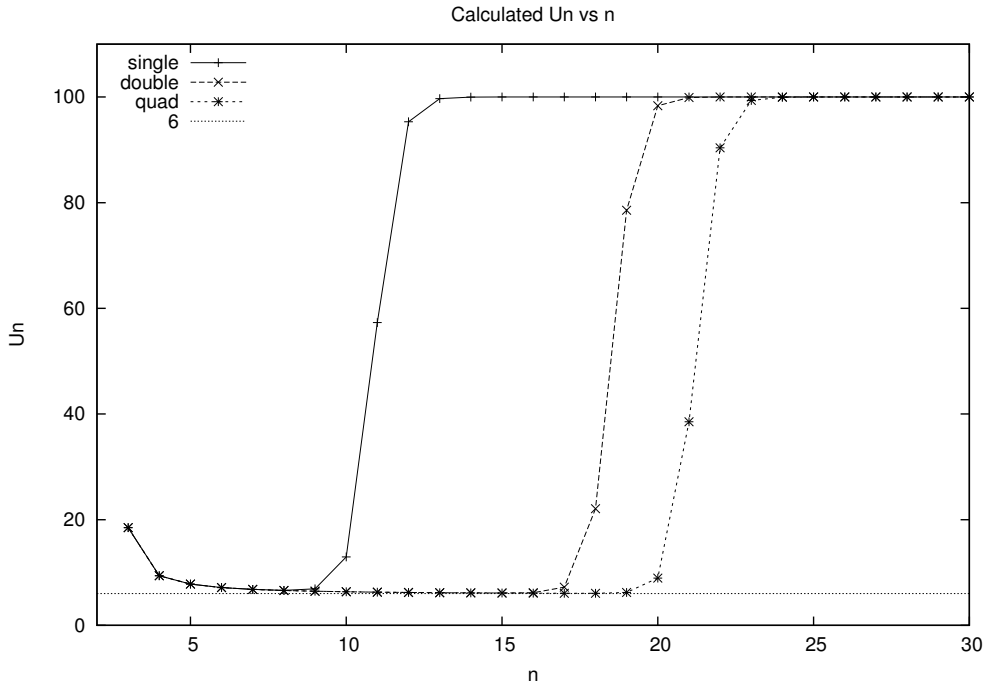

Figure 1.1: Output of Program 1.1 from *Handbook of Floating-Point Arithmetic*[**?**] for various IEEE types

1.28.2 Mr Gullible and the Chaotic Bank Society pgs 10-11 [?]

This is an example of a sequence involving e. Since e cannot be represented exactly with FP, even though the sequence should go to 0 for $a_0 = e - 1$, the representation of $a_0 \neq e - 1$ so the sequence goes to $\pm\infty$.

To eliminate these types of problems we'd need an *exact* representation of all real numbers. For *any* FP representation, regardless of precision (a finite number of digits) there will be numbers that can't be represented exactly hence you could find a similar sequence that would explode.

IE: The more precise the representation, the slower things go wrong, but they still go wrong, **even with errorless operations**.

1.28.3 Rump's example pg 12 [?]

This is an example where the calculation of a function $f(a, b)$ is not only totally wrong, it gives completely different results depending on the CPU. Despite the CPU conforming to IEEE.

1.29 Scalable Vector Graphics (SVG) 1.1 (Second Edition) [?]

Scalable Vector Graphics (SVG) is a XML language for describing two dimensional graphics. This document is ⟨http://www.w3.org/TR/2011/REC-S
the latest version of the standard at the time of writing.

Three types of object

1. Vector graphics shapes (paths)
2. Images (embedded bitmaps)
3. Text

Rendering Model and Precision

SVG uses the "painter's model". Paint is applied to regions of the page in an order, covering the paint below it according to rules for alpha blending.

"Implementations of SVG are expected to behave as though they implement a rendering (or imaging) model cor- responding to the one described in this chapter. A real implementation is not required to implement the model in this way, but the result on any device supported by the implementation shall match that described by this model."

SVG uses **single precision floats**. Unlike PDF and PostScript, the standard specifies this as a **minimum** range from `-3.4e+38F` to `+3.4e+38F`

"It is recommended that higher precision floating point storage and computation be performed on operations such as coordinate system transformations to provide the best possible precision and to prevent round-off errors."

There is also a "High Quality SVG Viewers" requirement to use at least **double** precision floats.

1.30 OpenVG Specification 1.1 [?]

"OpenVG is an application programming interface (API) for hardware-accelerated two- dimensional vector and raster graphics developed under the auspices of the Khronos Group ⟨www.khronos.org⟩."

Specifically, provides the same drawing functionality required by a high-performance SVG document viewer (SVG Tiny 1.2) TODO: Look at that $\neq$ SVG 1.1

It is designed to be similar to OpenGL.

Precision

"All floating-point values are specified in standard IEEE 754 format. However, implementations may clamp extremely large or small values to a restricted range, and internal processing may be performed with lesser precision. At least 16 bits of mantissa, 6 bits of exponent, and a sign bit must be present, allowing values from $\pm 2^{-30}$ to $\pm 2^{31}$ to be represented with a fractional precision of at least 1 in 2^{16}."

IEEE but with a non standard number of bits.

Presumably the decreased precision is for increased efficiency the standard states that example applications are for ebooks.

1.31 Document Object Model — pugixml 1.4 manual [?]

Pugixml is a C++ library for parsing XML documents[?]. XML is based on the Document Object Model (DOM); this is explained pretty well by the pugixml manual.

The document is the root node of the tree. Each child node has a type. These may

- Have attributes
- Have child nodes of their own
- Contain data

In the case of HTML/SVG an XML parser within the browser/viewer creates the DOM tree from the XML (plain text) and then interprets that tree to produce the objects that will be rendered.

Example of XML → DOM tree given at[?]. Example of XML parsing using pugixml is in `code/src/tests/xml.cpp`

```
<?xml version="1.0"?>
<mesh name="mesh_root">
    <!-- here is a mesh node -->
    some text
    <![CDATA[someothertext]]>
    some more text
    <node attr1="value1" attr2="value2" />
    <node attr1="value2">
        <innernode/>
    </node>
</mesh>
<?include somedata?>
```

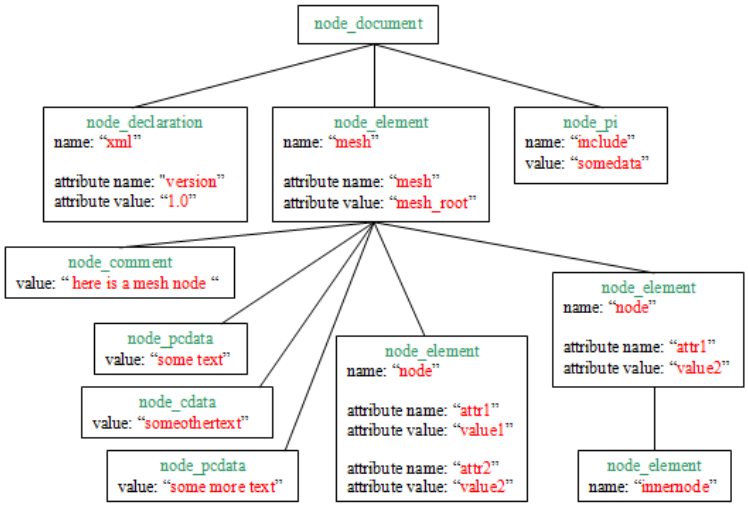

Figure 1.2: Tree representation of the above listing [?]

1.32 An Algorithm For Shading of Regions on Vector Display Devices [?]

All modern display devices are raster based and therefore this paper is mainly of historical interest. It provides some references for shading on a raster display.

The algorithm described will shade an arbitrary simply-connected polygon using one or two sets of parallel lines.

The "traditional" method is:

1. Start with a N vertex polygon, rotate coords by the shading angle
2. Determine a bounding rectangle
3. For M equally spaced parallel lines, compute the intersections with the boundaries of the polygon
4. Rotate coordinates back
5. Render the M lines

This is pretty much exactly how an artist would shade a pencil drawing. It is $O(M \times N)$.

The algorithm in this paper does:

1. Rotate polygon coords by shading angle
2. Subdivide the polygon into trapezoids (special case triangle)
3. Shade the trapezoids independently
4. Rotate it all back

It is more complicated than it seems. The subdivision requires a sort to be performed on the vertices of the polygon based on their rotated x and y coordinates.

1.33 An Algorithm For Filling Regions on Graphics Display Devices [?]

This gives an algorithm for for polygons (which may have "holes"). It requires the ability to "subtract" fill from a region; this is (as far as I can tell) difficult for vector graphics devices but simple on raster graphics devices, so the paper claims it is oriented to the raster graphics devices.

If the polygon is defined by (x_i, y_i) then this algorithm iterates from $i = 2$ and alternates between filling and erasing the triangles $[(x_i, y_i), (x_{i+1}, y_{i+1}), (x_1, y_1)]$. It requires no sorting of the points.

The paper provides a proof that the algorithm is correct and is "optimal in the number of pixel updates required for convex polygons". In the conclusion it is noted that trapezoids could be used from a fixed line and edge of the polygon, but this is not pixel optimal.

This paper doesn't have a very high citation count but it is cited by the NVIDIA article [**?**]. Apparently someone else adapted this algorithm for use with the stencil buffer.

1.34 GPU-accelerated path rendering [?, ?]

Vector graphics on the GPU; an NVIDIA extension. [**?**] is the API.

Motivations:

- The focus has been on 3D acceleration in GPUs; most path rendering is done by the CPU.
- Touch devices allow the screen to be transformed rapidly; CPU rastering of the path becomes inefficient
  - The source of the ugly pixelated effects on a smartphone when scaling?
- Especially when combined with increased resolution of these devices
- Standards such as HTML5, SVG, etc, expose path rendering
- Javascript is getting fast enough that we can't blame it anymore (the path rendering is the bottleneck not the JS)
- GPU is more power efficient than the CPU

Results show the extension is faster than almost every renderer it was compared with for almost every test image.

Comparisons to other attempts:

- Cairo and Glitz [**?**] (abandoned)
- Direct2D from Microsoft uses CPU to tesselate trapezoids and then renders these on the GPU
- Skia in Android/Chrome uses CPU but now has Ganesh which is also hybrid CPU/GPU
- Khronos Group created OpenVG[**?**] with several companies creating hardware units to implement the standard. Performance is not as good as "what we report"

1.35 A Multiple Precision Binary Floating Point Library With Correct Rounding [?]

This is what is now the GNU MPFR C library; it has packages in debian wheezy.

The library was motivated by the lack of existing arbitrary precision libraries which conformed to IEEE rounding standards. Examples include Mathematica, GNU MP (which this library is actually built upon), Maple (which is an exception but buggy).

TODO: Read up on IEEE rounding to better understand the first section

Data:

- (s, m, e) where e is signed
- Precision p is number of bits in m
- $\frac{1}{2} \leq m < 1$
- The leading bit of the mantissa is always 1 but it is not implied
- There are no denormalised numbers
- Mantissa is stored as an array of "limbs" (unsigned integers) as in GMP

The paper includes performance comparisons with several other libraries, and a list of literature using the MPFR (the dates indicating that the library has been used reasonably widely before this paper was published).

1.36 Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic [?]

11 years after the IEEE 754 standard becomes official, Kahan discusses various misunderstood features of the standard and laments at the failure of compilers and microprocessors to conform to some of these.

I am not sure how relevant these complaints are today, but it makes for interesting reading as Kahan is clearly very passionate about the need to conform *exactly* to IEEE 754.

Issues considered are: Rounding rules, Exception handling and NaNs (eg: The payload of NaNs is not used in any software Kahan is aware of), Bugs in compilers (mostly Fortran) where an expression contains floats of different precisions (the compiler may attempt to optimise an expression resulting in a failure to round a double to a single), Infinity (which is unsupported by many compilers though it is supported in hardware)...

An example is this Fortran compiler "nasty bug" where the compiler replaces s with x in line 4 and thus a rounding operation is lost.

1.36.1 The Baleful Influence of Benchmarks [?] pg 20

This section discusses the tendency to rate hardware (or software) on their speed performance and neglect accuracy.

Is this complaint still relevant now when we consider the eagerness to perform calculations on the GPU? ie: Doing the coordinate transforms on the GPU is faster, but less accurate (see our program).

Benchmarks need to be well designed when considering accuracy; how do we know an inaccurate computer hasn't gotten the right answer for a benchmark by accident?

A proposed benchmark for determining the worst case rounding error is given. This is based around computing the roots to a quadratic equation. A quick scan of the source code for paranoia does not reveal such a test.

As we needed benchmarks for CPUs perhaps we need benchmarks for GPUs. The GPU Paranoia paper[**?**] is the only one I have found so far.

1.37 Prof W. Kahan's Web Pages [?]

William Kahan, architect of the IEEE 754-1985 standard, creator of the program "paranoia", the "Kahan Summation Algorithm" and contributor to the revised standard.

Kahan's web page has more examples of errors in floating point arithmetic (and places where things have not conformed to the IEEE 754 standard) than you can poke a stick at.

Wikipedia's description is pretty accurate: "He is an outspoken advocate of better education of the general computing population about floating-point issues, and regularly denounces decisions in the design of computers and programming languages that may impair good floating-point computations."

Kahan's articles are written with almost religious zeal but they are backed up by examples and results, a couple of which I have confirmed.[1] This is the unpublished work. I haven't read any of the published papers yet.

The articles are filled sporadically with lamentation for the decline of experts in numerical analysis (and error) which is somewhat ironic; if there were no IEEE 754 standard meaning any man/woman and his/her dog could write floating point arithmetic and expect it to produce platform independent results[2] this decline would probably not have happened.

These examples would be of real value if the topic of the project were on accuracy of numerical operations. They also explain some of the more bizarre features of IEEE 754 (in a manner attacking those who dismiss these features as being too bizarre to care about of course).

It's somewhat surprising he hasn't written anything (that I could see from scanning the list) about the lack of IEEE in PDF/PostScript (until Adobe Reader 6) and further that the precision is only binary32.

I kind of feel really guilty saying this, but since our aim is to obtain arbitrary scaling, it will be inevitable that we break from the IEEE 754 standard. Or at least, I (Sam) will. Probably. Also there is no way I will have time to read and understand the thousands of pages that Kahan has written.

Therefore we might end up doing some things Kahan would not approve of.

Still this is a very valuable reference to go in the overview of floating point arithmetic section.

[1] One that doesn't work is an example of wierd arithmetic in Excel 2000 when repeated in LibreOffice Calc 4.1.5.3

[2] Even if, as Kahan's articles often point out, platforms aren't actually following IEEE 754

Chapter 2

General Notes

2.1 The DOM Model

A document is modelled as a tree. The root of the tree is the document. This has nodes of varying types. Some nodes may have children, attributes, and data.

2.2 Floating-Point Numbers[?, ?, ?, ?]

A set of FP numbers is characterised by:

1. Radix (base) $\beta \geq 2$
2. Precision
3. Two "extremal" exponents $e_min < 0 < e_max$ (generally, don't have to have the 0 in there)

Numbers are represented by **integers**: (M, e) such that $x = M \times \beta^{e-p+1}$

Require: $|M| \leq \beta^p - 1$ and $e_min \leq e \leq e_max$.

Representations are not unique; set of equivelant representations is a cohort.

β^{e-p+1} is the quantum, $e - p + 1$ is the quantum exponent.

Alternate represetnation: (s, m, e) such that $x = (-1)^s \times m \times \beta^e$ m is the significand, mantissa, or fractional part. Depending on what you read.

2.3 Rounding Errors

They happen. There is ULP and I don't mean a political party.

TODO: Probably say something more insightful. Other than "here is a graph that shows errors and we blame rounding".

2.3.1 Results of calculatepi

We can calculate pi by numerically solving the integral:

$$\int_0^1 \left(\frac{4}{1+x^2}\right) dx = \pi$$

Results with Simpson Method:

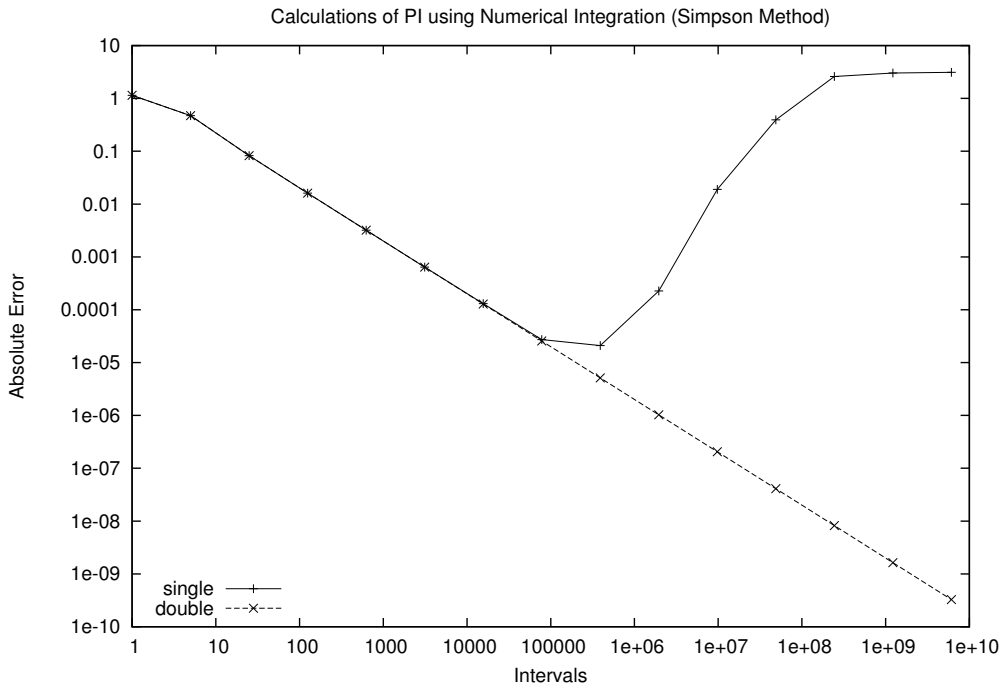

Figure 2.1: Example of accumulated rounding errors in a numerical calculation

Tests with `calculatepi` show it's not quite as simple as just blindly replacing all your additions with Fast2Sum from Dekker[**?**]. ie: The graph looks exactly the same for single precision. `calculatepi` obviously also has multiplication ops in it which I didn't change. Will look at after sleep maybe.