Literature Review

David Gow

May 21, 2014

1 Introduction

Since mankind first climbed down from the trees, it is our ability to communicate that has made us unique. Once ideas could be passed from person to person, it made sense to have a permanent record of them; one which could be passed on from person to person without them ever meeting.

And thus the document was born.

Traditionally, documents have been static: just marks on paper, but with the advent of computers many more possibilities open up.

2 Document Formats

Most existing document formats — such as the venerable PostScript and PDF — are, however, designed to imitate existing paper documents, largely to allow for easy printing. In order to truly take advantage of the possibilities operating in the digital domain opens up to us, we must look to new formats.

Formats such as `HTML` allow for a greater scope of interactivity and for a more data-driven model, allowing the content of the document to be explored in ways that perhaps the author had not anticipated.[1] However, these data-driven formats typically do not support fixed layouts, and the display differs from renderer to renderer.

2.1 A Taxonomy of Document formats

The process of creating and displaying a document is a rather universal one (2.1), though different document formats approach it slightly differently. A document often begins as raw content: text and images (be they raster or vector) and it must end up as a set of photons flying towards the reader's eyes.

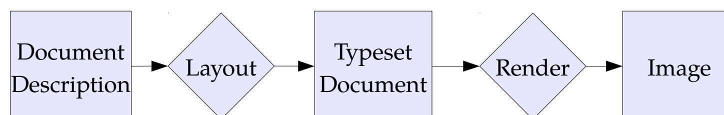

Figure 1: The lifecycle of a document

There are two fundamental stages by which all documents — digital or otherwise — are produced and displayed: *layout* and *rendering*. The *layout* stage is where the positions and sizes of text and other graphics are determined. The text will be *flowed* around graphics, the positions of individual glyphs will be placed, ensuring that there is no undesired overlap and that everything will fit on the page or screen.

The *display* stage actually produces the final output, whether as ink on paper or pixels on a computer monitor. Each graphical element is rasterized and composited into a single image of the target resolution.

Different document formats cover documents in different stages of this project. Bitmapped images, for example, would represent the output of the final stage of the process, whereas markup languages typically specify a document which has not yet been processed, ready for the layout stage.

Furthermore, some document formats treat the document as a program, written in a (usually turing complete) document language with instructions which emit shapes to be displayed. These shapes are either displayed immediately, as in PostScript, or stored in another file, such as with TEXor LATEX, which emit a `DVI` file. Most other forms of document use a *Document Object Model*, being a list or tree of objects to be rendered. `DVI`, `PDF`, `HTML`[1] and SVG[2]. Of these, only `HTML` and TEXtypically store documents in pre-layout stages, whereas even turing complete document formats such as PostScript typically encode documents which already have their elements placed.

TEX and LATEX Donald Knuth's typesetting language TEX is one of the older computer typesetting systems, originally conceived in 1977[3]. It implements a turing-complete language and is human-readable and writable, and is still popular due to its excellent support for typesetting mathematics. TEXonly implements the "layout" stage of document display, and produces a typeset file, traditionally in `DVI` format, though modern implementations will often target `PDF` instead.

This document was prepared in LATEX 2_ε.

DVI TEX traditionally outputs to the `DVI` ("DeVice Independent") format: a binary format which consists of a simple stack machine with instructions for drawing glyphs and curves[4].

A `DVI` file is a representation of a document which has been typeset, and `DVI` viewers will rasterize this for display or printing, or convert it to another similar format like PostScript to be rasterized.

HTML The Hypertext Markup Language (HTML)[5] is the widely used document format which underpins the world wide web. In order for web pages to adapt appropriately to different devices, the HTML format simply defined semantic parts of a document, such as headings, phrases requiring emphasis, references to images or links to other pages, leaving the *layout* up to the browser, which would also rasterize the final document.

The HTML format has changed significantly since its introduction, and most of the layout and styling is now controlled by a set of style sheets in the CSS[6] format.

[1] Some of these formats — most notably `HTML` — implement a scripting lanugage such as JavaScript, which permit the DOM to be modified while the document is being viewed.

PostScript Much like DVI, PostScript[7] is a stack-based format for drawing vector graphics, though unlike DVI (but like TEX), PostScript is text-based and turing complete. PostScript was traditionally run on a control board in laser printers, rasterizing pages at high resolution to be printed, though PostScript interpreters for desktop systems also exist, and are often used with printers which do not support PostScript natively.[8]

PostScript programs typically embody documents which have been type-set, though as a turing-complete language, some layout can be performed by the document.

PDF Adobe's Portable Document Format (PDF)[9] takes the PostScript rendering model, but does not implement a turing-complete language. Later versions of PDF also extend the PostScript rendering model to support translucent regions via Porter-Duff compositing[10].

PDF documents represent a particular layout, and must be rasterized before display.

2.2 Precision in Document Formats

Existing document formats — typically due to having been designed for documents printed on paper, which of course has limited size and resolution — use numeric types which can only represent a fixed range and precision. While this works fine with printed pages, users reading documents on computer screens using programs with "zoom" functionality are prevented from working beyond a limited scale factor, lest artefacts appear due to issues with numeric precision.

TEXuses a 14.16 bit fixed point type (implemented as a 32-bit integer type, with one sign bit and one bit used to detect overflow)[11]. This can represent values in the range $[-(2^14), 2^14-1]$ with 16 binary digits of fractional precision.

The DVI files TEX produces may use "up to" 32-bit signed integers[4] to specify the document, but there is no requirement that implementations support the full 32-bit type. It would be permissible, for example, to have a DVI viewer support only 24-bit signed integers, though many files which require greater range may fail to render correctly.

PostScript[7] supports two different numeric types: *integers* and *reals*, both of which are specified as strings. The interpreter's representation of numbers is not exposed, though the representation of integers can be divined by a program by the use of bitwise operations. The PostScript specification lists some "typical limits" of numeric types, though the exact limits may differ from implementation to implementation. Integers typically must fall in the range $[-2^{31}, 2^{31}-1]$, and reals are listed to have largest and smallest values of $\pm 10^{38}$, values closest to 0 of $\pm 10^{-38}$ and approximately 8 decimal digits of precision, derived from the IEEE 754 single-precision floating-point specification.

Similarly, the PDF specification[9] stores *integers* and *reals* as strings, though in a more restricted format than PostScript. The PDF specification gives limits for the internal representation of values. Integer limits have not changed from the PostScript specification, but numbers representable with the *real* type have been specified differently: the largest representable values are $\pm 3.403 \times 10^{38}$, the smallest non-zero representable values are [2] $\pm 1.175 \times 10^{-38}$ with approximately

[2] The PDF specification mistakenly leaves out the negative in the exponent here.

5 decimal digits of precision *in the fractional part.* Adobe's implementation of PDF uses both IEEE 754 single precision floating-point numbers and (for some calculations, and in previous versions) 16.16 bit fixed-point values.

3 Rendering

Computer graphics comes in two forms: bit-mapped (or raster) graphics, which is defined by an array of pixel colours; and *vector* graphics, defined by mathematical descriptions of objects. Bit-mapped graphics are well suited to photographs and are match how cameras, printers and monitors work. However, bitmap devices do not handle zooming beyond their "native" resolution — the resolution where one document pixel maps to one display pixel —, exhibiting an artefact called pixelation where the pixel structure becomes evident. Attempts to use interpolation to hide this effect are never entirely successful, and sharp edges, such as those found in text and diagrams, are particularly affected.

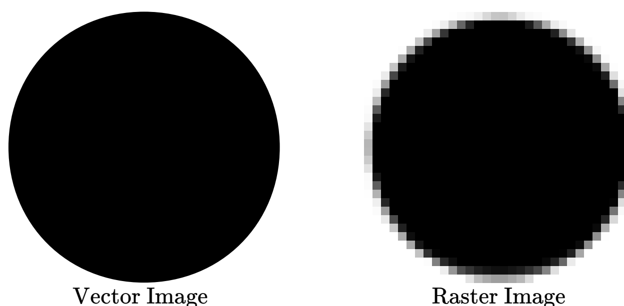

Figure 2: A circle as a vector image and a 32×32 pixel raster image

Vector graphics lack many of these problems: the representation is independent of the output resolution, and rather an abstract description of what it is being rendered, typically as a combination of simple geometric shapes like lines, arcs and "Bézier curves"[12]. As existing displays (and printers) are bit-mapped devices, vector documents must be *rasterized* into a bitmap at a given resolution. This bitmap is then displayed or printed. The resulting bitmap is then an approximation of the vector image at that resolution.

This project will be based around vector graphics, as these properties make it more suited to experimenting with zoom quality.

The rasterization process typically operates on an individual "object" or "shape" at a time: there are special algorithms for rendering lines[13], triangles[14], polygons[15] and Bézier Curves[16]. Typically, these are rasterized independently and composited in the bitmap domain using Porter-Duff compositing[10] into a single image. This allows complex images to be formed from many simple pieces, as well as allowing for layered translucent objects, which would otherwise require the solution of some very complex constructive geometry problems.

While traditionally, rasterization was done entirely in software, modern computers and mobile devices have hardware support for rasterizing some basic

primitives — typically lines and triangles —, designed for use rendering 3D scenes. This hardware is usually programmed with an API like `OpenGL`[17].

More complex shapes like Bézier curves can be rendered by combining the use of bitmapped textures (possibly using signed-distance fields[18][19][20]) with polygons approximating the curve's shape[21][22].

Indeed, there are several implementations of entire vector graphics systems using OpenGL: OpenVG[23] on top of OpenGL ES[24]; the Cairo[25] library, based around the PostScript/PDF rendering model, has the "Glitz" OpenGL backend[26] and the SVG/PostScript GPU renderer by nVidia[27] as an OpenGL extension[28].

4 Numeric formats

On modern computer architectures, there are two basic number formats supported: fixed-width integers and *floating-point* numbers. Typically, computers natively support integers of up to 64 bits, capable of representing all integers between 0 and $2^{64} - 1$, inclusive[3].

By introducing a fractional component (analogous to a decimal point), we can convert integers to *fixed-point* numbers, which have a more limited range, but a fixed, greater precision. For example, a number in 4.4 fixed-point format would have four bits representing the integer component, and four bits representing the fractional component:

$$\underbrace{0101}_{\text{integer component}} \quad . \quad \underbrace{1100}_{\text{fractional component}} = 5.75 \tag{1}$$

Floating-point numbers[29] are the binary equivalent of scientific notation: each number consisting of an exponent (e) and a mantissa (m) such that a number is given by

$$n = 2^e \times m \tag{2}$$

The IEEE 754 standard[30] defines several floating-point data types which are used[4] by most computer systems. The standard defines 32-bit (8-bit exponent, 23-bit mantissa, 1 sign bit) and 64-bit (11-bit exponent, 53-bit mantissa, 1 sign bit) formats[5], which can store approximately 7 and 15 decimal digits of precision respectively.

Floating-point numbers behave quite differently to integers or fixed-point numbers, as the representable numbers are not evenly distributed. Large numbers are stored to a lesser precision than numbers close to zero. This can present problems in documents when zooming in on objects far from the origin.

IEEE floating-point has some interesting features as well, including values for negative zero, positive and negative infinity, the "Not a Number" (NaN) value and *denormal* values, which trade precision for range when dealing with very small numbers. Indeed, with these values, IEEE 754 floating-point equality does

[3] Most machines also support *signed* integers, which have the same cardinality as their *unsigned* counterparts, but which represent integers in the range $[-(2^{63}), 2^{63} - 1]$

[4] Many systems' implement the IEEE 754 standard's storage formats, but do not implement arithmetic operations in accordance with this standard.

[5] The 2008 revision to this standard[31] adds some additional formats, but is less widely supported in hardware.

not form an equivalence relation, which can cause issues when not considered carefully.[32]

There also exist formats for storing numbers with arbitrary precising and/or range. Some programming languages support "big integer"[33] types which can represent any integer that can fit in the system's memory. Similarly, there are arbitrary-precision floating-point data types[34][35] which can represent any number of the form

$$\frac{n}{2^d} \quad n, d \in \mathbb{Z} \tag{3}$$

These types are typically built from several native data types such as integers and floats, paired with custom routines implementing arithmetic primitives.[36] These, therefore, are likely slower than the native types they are built on.

While traditionally, GPUs have supported some approximation of IEEE 754's 32-bit floats, modern graphics processors also support 16-bit[37] and 64-bit[38] IEEE floats. Note, however, that some parts of the GPU are only able to use some formats, so precision will likely be truncated at some point before display. Higher precision numeric types can be implemented or used on the GPU, but are slow.[39]

Pairs of integers $(a \in \mathbb{Z}, b \in \mathbb{Z} \setminus 0)$ can be used to represent rationals. This allows values such as $\frac{1}{3}$ to be represented exactly, whereas in fixed or floating-point formats, this would have a recurring representation:

$$\underbrace{0}_{\text{integer part}} \quad . \quad \underbrace{01}_{\text{recurring part}} \quad 01 \;\; 01 \;\; 01 \ldots \tag{4}$$

Whereas with a rational type, this is simply $\frac{1}{3}$. Rationals do not have a unique representation for each value, typically the reduced fraction is used as a characteristic element.

5 Quadtrees

When viewing or processing a small part of a large document, it may be helpful to only processs — or *cull* — parts of the document which are not on-screen.

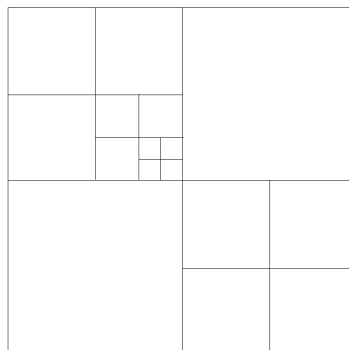

Figure 3: A simple quadtree.

The quadtree[40]is a data structure — one of a family of *spatial* data structures — which recursively breaks down space into smaller subregions which can

be processed independently. Points (or other objects) are added to a single node, which if certain criteria are met — typically the number of points in a node exceeding a maximum, though in our case likely the level of precision required exceeding that supported by the data type in use — is split into four equal-sized subregions, and points attached to the region which contains them.

In this project, we will be experimenting with a form of quadtree in which each node has its own independent coordinate system, allowing us to store some spatial information[6] within the quadtree structure, eliminating redundancy in the coordinates of nearby objects.

Other spatial data structures exist, such as the KD-tree[41], which partitions the space on any axis-aligned line; or the BSP tree[42], which splits along an arbitrary line which need not be axis aligned. We believe, however, that the simpler conversion from binary coordinates to the quadtree's binary split make it a better avenue for initial research to explore.

References

[1] Brian Hayes. Pixels or perish. *American Scientist*, 100(2):106 – 111, 2012.

[2] Erik Dahlstóm, Patric Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, Jonathon Watt, Jon Ferraiolo, Fujisawa Jun, and Dean Jackson. Scalable vector graphics (svg) 1.1 (second edition). *WC3 Recommendation*, August 2011.

[3] Donald Knuth. Preliminary preliminary description of TEX. `http://www.saildart.org/TEXDR.AFT[1,DEK]1`, 1977.

[4] David Fuchs. The format of TEX's DVI files. *TUGBoat*, 3(2), 1982.

[5] Tim Berners-Lee and Daniel Connolly. Hypertext markup language – 2.0. *Internet RFC 1866*, 1995.

[6] Bert Bos, Håkon Wium Lie, Chris Lilley, and Jacobs Ian. Cascading style sheets, level 2, CSS2 specification. `http://www.w3.org/TR/1998/REC-CSS2-19980512/`.

[7] Adobe Systems Incorporated. *PostScript Language Reference*. Addison-Wesley Publishing Company, 3rd edition, 1985 - 1999.

[8] Artifex Software. Ghostscript, an interpreter for the postscript language and pdf. `http://www.ghostscript.com/`, 1988.

[9] Adobe Systems Incorporated. *PDF Reference*. Adobe Systems Incorporated, 6th edition, 2006.

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

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

[6] One bit per-coordinate, per-level of the quadtree

[12] Edwin Earl Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, 1974. AAI7504786.

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

[14] Fabien Giesen. Triangle rasterization in practice. `http://fgiesen.wordpress.com/2013/02/08/triangle-rasterization-in-practice/`, 2013.

[15] Juan Pineda. A parallel algorthim for polygon rasterization. *ACM Computer Graphics*, 22(4):17–20, 1988.

[16] 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.

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

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

[19] Sarah F Frisken, Ronald N Perry, Alyn P Rockwood, and Thouis R Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 249–254. ACM Press/Addison-Wesley Publishing Co., 2000.

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

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

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

[23] Mathieu Robart. OpenVG paint subsystem over OpenGL ES shaders. In *Consumer Electronics, 2009. ICCE'09. Digest of Technical Papers International Conference on*, pages 1–2. IEEE, 2009.

[24] Aekyung Oh, Hyunchan Sung, Hwanyong Lee, Kujin Kim, and Nakhoon Baek. Implementation of OpenVG 1.0 using OpenGL ES. In *Proceedings of the 9th international conference on Human computer interaction with mobile devices and services*, pages 326–328. ACM, 2007.

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

[26] Peter Nilsson and David Reveman. Glitz: Hardware accelerated image compositing using OpenGL. In *USENIX Annual Technical Conference, FREENIX Track*, pages 29–40, 2004.

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

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

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

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

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

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

[33] Oracle Corporation. java.math.BigInteger. `http://docs.oracle.com/javase/6/docs/api/java/math/BigInteger.html`.

[34] Oracle Corporation. java.math.BigDecimal. `http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`.

[35] John Maddock and Christopher Kormanyos. Boost multiprecision library. `http://www.boost.org/doc/libs/1_53_0/libs/multiprecision/doc/html/boost_multiprecision/`.

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

[37] Pat Brown. NV_half_float. `http://www.opengl.org/registry/specs/NV/half_float.txt`, 2002.

[38] Pat Brown, Barthold Lichtenbelt, Bill Licea-Kane, Bruce Merry, Chris Dodd, Eric Werness, Graham Sellers, Greg Roth, Jeff Bolz, Nick Haemel, Pierre Boudier, and Piers Daniell. ARB_gpu_shader_fp64. `http://www.opengl.org/registry/specs/ARB/gpu_shader_fp64.txt`, 2010.

[39] Niall Emmart and Charles Weems. High precision integer multiplication with a graphics processing unit. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–6. IEEE, 2010.

[40] Raphael A. Finkel and Jon Louis Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[41] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[42] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '80, pages 124–133, New York, NY, USA, 1980. ACM.