Precision in vector documents: a spatial approach

David Gow

This report is submitted as partial fulfilment of
the requirements for the Software Engineering Programmme of the
School of Computer Science and Software Engineering,
The University of Western Australia,
2014

Contents

CHAPTER 1

Introduction

Documents are an important part of day-to-day life: we use them for business and pleasure, to inform and entertain. We often use images or diagrams to illustrate our ideas, and to convey spatial or visual information.

On the printed page, there is a practical limit to the size and resolution of these images. In order to store, for example, maps of large areas with fine detail, we resort to an atlas: splitting the map up into several pages, each of which covers a different part of the map.

With the advent of computers, we can begin to alleviate these problems. The fixed size and resolution of the screen is worked around by having the screen be a *viewport* into a larger document, letting the document be *panned* (translated laterally) and *zoomed* (scaled to a particular magnification).

However, limits in the range and precision of the numbers used to store and manipulate documents artificially restrict the size and zoom-level of documents. While changing the numeric types used can solve these issues, here we investigate the use of a quadtree to take advantage of the spatial nature of the data. This is akin to maintaining the atlas of several different pages with different views of the map, but with the advantage of greater ease-of-use and continuity.

CHAPTER 2

Background

2.1 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 match how cameras, printers and monitors work.

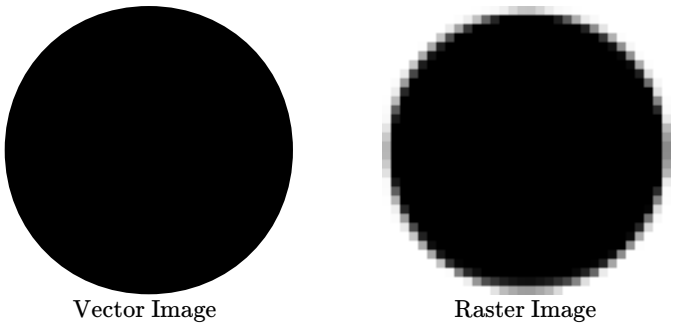

Figure 2.1: A circle as a vector image and a 32×32 pixel raster image

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 (see Figure 2.1) 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.

Vector graphics avoid many of these problems: the representation is independent of the output resolution, and rather an abstract description of what is being rendered, typically as a combination of simple geometric shapes like lines, arcs and glyphs.

As existing displays (and printers) are bit-mapped devices, vector documents must be *rasterized* into a bitmap at a given resolution. The resulting bitmap is then an approximation of the vector image at that resolution. This bitmap is then displayed or printed.

We will discuss the use of vector graphics, as they lend themselves more naturally to scaling, though many of the techniques will also apply to raster graphics.

Indeed, the use of quadtrees to compress bitmapped data is well established[44], and the view-reparenting techniques have been used to facilitate an infinite zoom into a (procedurally generated) document[38].

2.1.1 Rasterizing Vector Graphics

Before an vector document can be rasterized, the co-ordinates of any shapes must be transformed into *screen space* or *viewport space*[6]. On a typical display, many of these screen-space coordinates require very little precision or range. However, the co-ordinate transform must take care to ensure that precision is preserved during this transform.

After this transformation, the image is decomposed into its separate shapes, which are rasterized and then composited together. Most graphics formats support Porter-Duff compositing[41]. Porter-Duff compositing gives each element (typically a pixel) a "coverage" value, denoted α which represents the contribution of that element to the final scene. Completely transparent elements would have an α value of 0, and completely opaque elements have an α of 1. This permits arbitrary shapes to be layered over one another in the raster domain, while retaining soft-edges.

The rasterization process may then proceed on one object (or shape) at a time. There are special algorithms for rasterizing different shapes.

Line Segment Straight lines between two points are easily rasterized using Bresenham's algorithm[8]. Bresenham's algorithm draws a pixel for every point along the *long* axis of the line, moving along the short axis when the error exceeds $\frac{1}{2}$ a pixel.

Bresenham's algorithm only operates on lines whose endpoints lie on integer pixel coordinates. Due to this, line "clipping" may be performed to find endpoints of the line segment such that the entire line will be on-screen. However, if line clipping is performed naïvely without also setting the error accumulator correctly, the line's slope will be altered slightly, becoming dependent on the viewport.

Bézier Curve A Bézier curve is a smooth (i.e. infinitely differentiable) curve between two points, represented by a Bernstein polynomial. The coefficients of this Bernstein polynomial are known as the "control points."

Bézier curves are typically rasterized using De Casteljau's algorithm[19] Line Segments are a first-order Bézier curve.

2.1.2 GPU Rendering

While traditionally, rasterization was done entirely in software, modern computers and mobile devices have hardware support for rasterizing lines and triangles designed for use rendering 3D scenes. This hardware is usually programmed with an API like `OpenGL`[46].

More complex shapes like Bézier curves can be rendered by combining the use of bitmapped textures (possibly using signed-distance fields[34][20][25]) strtched over a triangle mesh approximating the curve's shape[35][36].

Indeed, there are several implementations of entire vector graphics systems using OpenGL:

- The OpenVG standard[43] has been implemented on top of OpenGL ES[40];
- the Cairo[48] library, based around the PostScript/PDF rendering model, has the "Glitz" OpenGL backend[39]
- and the SVG/PostScript GPU renderer by nVidia[31] as an OpenGL extension[30].

2.2 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[1].

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:

$$\underbrace{0101}_{\text{integer component}} . \underbrace{1100}_{\text{fractional component}} = 5.75 \tag{2.1}$$

Floating-point numbers[24, 33] are a superset of scientific notation, originally used by the Babylonians (in base 60) perhaps as early as 1750 BC. Each number n (in a base β) consists of integers e (the *exponent*) and m (the *mantissa*) such that

$$n = \beta^e \times m_f \tag{2.2}$$

Both e and m typically have a fixed width q and p respectively in base β. For notational convenience, we also represent m as a fraction $m_f = \beta^{-p} m$ so $|m_f| < 1$. We further call a floating point number *normalised* if the first digit of m is nonzero. The exponent e is usually allowed to be either positive or negative (either by having a sign bit, or being offset a fixed amount). The value of a floating point number n must therefore be $-\beta^{e_max} < n < \beta^{e_max}$. The smallest possible magnitude of a normalised floating point number is β^{e_min-1}, as m_f must be at least 0.1. Non-normalised numbers may represent 0, and many normalised floating-point include zero in some way, too.

The IEEE 754 standard[1] defines several floating-point data types which are used[2] 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)

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

[2] Many systems implement the IEEE 754 standard's storage formats, but do not implement arithmetic operations in accordance with this standard[46, 29].

formats[3], which can store approximately 7 and 15 decimal digits of precision respectively. The IEEE 754 standard also introduces an implicit 1 bit in the most significant place of the mantissa when the exponent is not e_min.

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. Furthermore, due to the limited precision, and the different "alignment" of operands in arithmetic, several algebraic properties of rings we take for granted in the integers do not exist, including the property of assosciativity[33].

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 *subnormal* values, which trade precision for range when dealing with very small numbers by not normalising numbers when the exponent is e_min. Indeed, with these values, IEEE 754 floating-point equality does not form an equivalence relation, which can cause issues when not considered carefully[23].

There also exist formats for storing numbers with arbitrary precising and/or range. Some programming languages support "big integer"[14] types which can represent any integer that can fit in the system's memory. Similarly, there are arbitrary-precision floating-point data types[13][37] which can represent any number of the form

$$\frac{n}{2^d} \quad n, d \in \mathbb{Z} \tag{2.3}$$

These types are typically built from several native data types such as integers and floats, paired with custom routines implementing arithmetic primitives.[42] Operations on these types, therefore, are usually slower than those performed on native types.

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:

$$\underbrace{0}_{\text{integer part}} . \underbrace{01}_{\text{recurring part}} \quad 01\ 01\ 01 \ldots \tag{2.4}$$

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.

While traditionally, GPUs have supported some approximation of IEEE 754's 32-bit floats, modern graphics processors also support 16-bit[9] and 64-bit[10] IEEE floats, though some features of IEEE floats, like denormals and NaNs are not always supported. Note, however, that some parts of the GPU are not able to use all 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.[17]

[3]The 2008 revision to this standard[2] adds some additional formats, but is less widely supported in hardware.

Figure 2.2: The lifecycle of a document

2.3 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.[26] However, these data-driven formats typically do not support fixed layouts, and the display differs from renderer to renderer.

2.3.1 A Taxonomy of Document formats

The process of creating and displaying a document is a rather universal one (**??**), 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 stream of photons flying towards the reader's eyes.

There are two fundamental stages (as shown in Figure 2.2) 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`[4] and SVG[15]. 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[32]. 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[21].

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[7] format.

PostScript Much like DVI, PostScript[27] 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.[47]

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

PDF Adobe's Portable Document Format (PDF)[28] 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[41].

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

[4]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.

SVG Scalable Vector Graphics (SVG) is a vector graphics document format[15] which uses the Document Object Model. It consists of a tree of matrix transforms, with objects such as vector paths (made up of Bézier curves) and text at the leaves.

2.3.2 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)[3]. 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[21] 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[27] 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[28] 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 $\pm 1.175 \times 10^{-38}$ with approximately 5 decimal digits of precision *in the fractional part*. [5] 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.

The SVG specification[15] specifies numbers as strings with a decimal representation of the number. It is stated that a "Conforming SVG Viewer" must have "all visual rendering accurate to within one device pixel to the mathematically correct result at the initial 1:1 zoom ratio" and that "it is suggested that viewers attempt to keep a high degree of accuracy when zooming." A "Conforming High-Quality SVG Viewer" must use "double-precision floating point[6]" for computations involving co-

[5] The PDF specification mistakenly leaves out the negative in the exponent here.
[6] Presumably the 64-bit IEEE 754 "double" type.

ordinate system transformations.

2.4 Quadtrees

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

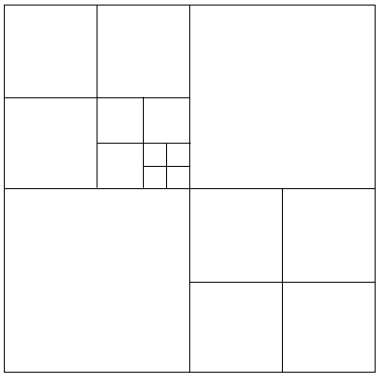

Figure 2.3: A simple quadtree.

The quadtree[18]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) is split into four equal-sized subregions, and points attached to the region which contains them.

Quadtrees have been used in computer graphics for both culling — excluding objects in nodes which are not visible — and "level of detail", where different levels of the quadtree store different quality versions of objects or data[49]. Typically the number of points in a node exceeding a maximum triggers this split, though in our case likely the level of precision required exceeding that supported by the data type in use.

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[7] within the quadtree structure, eliminating redundancy in the coordinates of nearby objects.

Other spatial data structures exist, such as the KD-tree[4], which partitions the space on any axis-aligned line; or the BSP tree[22], 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.

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

CHAPTER 3

IPDF: A Document Precision Playground

In order to investigate the precision issues present in document formats and viewers, we developed a document viewer IPDF. IPDF is a C++ program which supports rendering TrueType font outlines and a subset of the SVG format.

At its core, IPDF breaks documents down into a set of *objects*, each with rectangular bounds (x, y, w, h) and with several *types*:

1. **Rectangle:** A simple, axis-aligned rectangle (either filled with a solid colour or left as an outline) covering the object bounds.
2. **Ellipse:** A filled, axis-aligned ellipse, rendered parametrically.
3. **Bézier Curve:** A cubic Bézier curve. Bézier curve control points are stored relative to the coordinate system provided by the document bounds, and several objects can share these coefficients.

IPDF can be compiled using different number representations, to allow for comparison between not only different-precision floating point numbers, but also arbitrary-precision types such as rationals.

Documents in IPDF may be rendered either on the CPU, using a custom software rasterizer built on the numeric types supported, or on the GPU with OpenGL 3.2 shaders, using the GPU's default numeric representation.

Furthermore, IPDF allows SVG files (and text rendered with TrueType fonts) to be inserted at any point and scale in the document, allowing for the same images to be compared at different scales.

3.1 GPU Floating Point Rounding Behaviour

While the IEEE-754 standard specifies both the format of floating-point numbers and the operations they perform. However, the OpenGL specification[46], while requiring the same basic format for single-precision floats, does not specify the behaviour of denormals, nor requires any support for NaN or infinite values. Similarly, no support for floating point exceptions is required, with the note that no operation should ever halt the GPU.

However, an extension to the specification, `GL_ARB_shader_precision`[29] in 2010 allows programs to require stricter precision requirements. Notably, support for infinite values is required and maximum relative error in ULPs.

Different GPU vendors and drivers have different rounding behaviour, and most hardware (both CPU and GPU) provide ways of disabling support for some IEEE features to improve performance[12, 16].

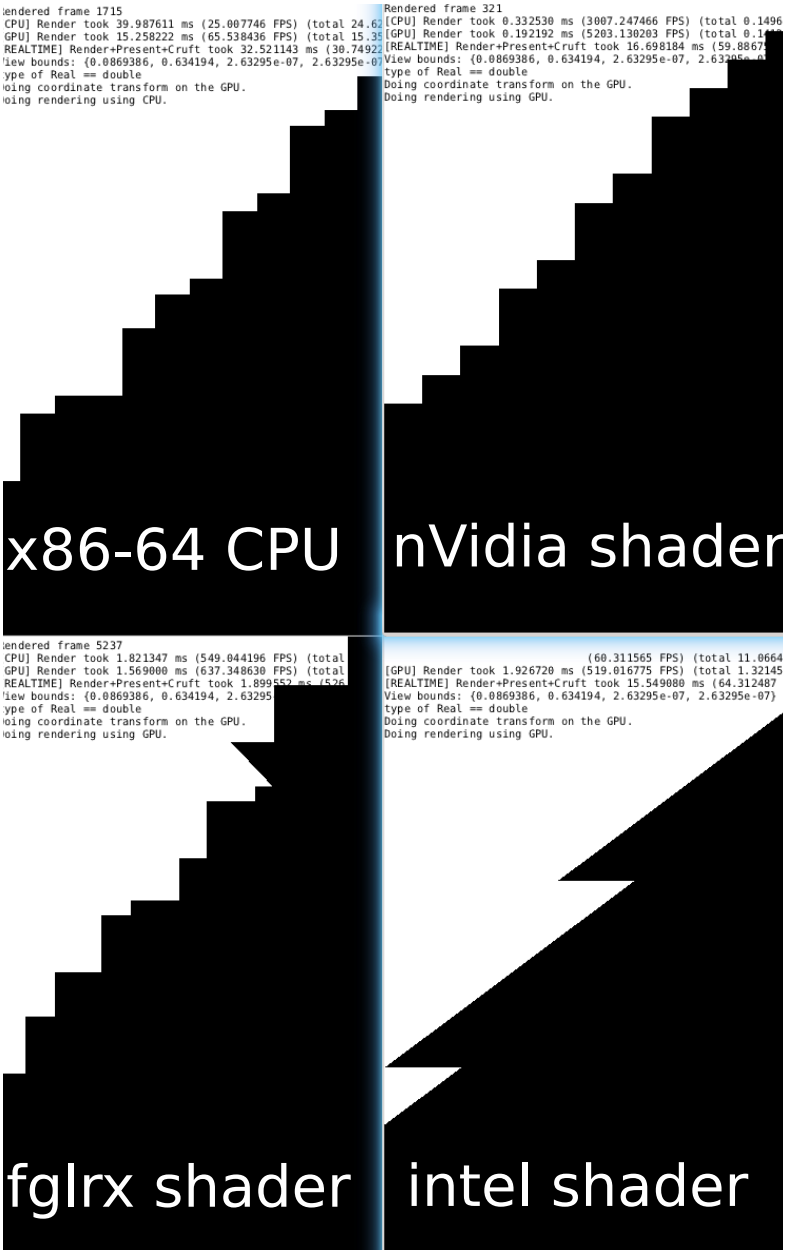

Figure 3.1: The edges of a unit circle viewed through bounds (x,y,w,h) = (0.0869386,0.634194,2.63295e-07,2.63295e-07)

Many GPUs now support double-precision floating-point numbers[1] as specified in the `GL_ARB_gpu_shader_fp64`[10] OpenGL extension. However, many of the fixed-function parts of the GPU do not support double-precision floats, making it impractical to use them.

To see the issues, we rendered the edge of a circle (calculated by discarding pixels with $x^2 + y^2 > 1$) on several GPUs, as well as an `x86-64` CPU, as seen in Figure 3.1. Of these, the nVidia GPU came closest to the CPU rendering, whereas Intel's hardware clearly performs some optimisation which produces quite different artefacts. The diagonal distortion in the AMD rendering may be a result of different rounding across the two triangles across which the circle was rendered.

3.2 Distortion and Quantisation at the limits of precision

TODO: Fix these figures, explain properly.

(a) Intended rendering (b) With artefacts

Figure 3.2: Floating point errors affecting the rendering of an image

When trying to insert fine detail into a document using fixed-width floats, some precision is lost. In particular, the control points of the curves making up the image get rounded to the nearest[2] representable point. Figure 3.2 shows what happens when an image is small enough to be affected by this quantisation. The grid structure becomes very apparent.

These artefacts also become prevalent when an object is far from the origin, as more bits would be required to store the position, so the lower order bits of the position must be discarded. As the position of the viewport and the object will share many of the same initial digits, catastrophic cancellation[23] can occur.

[1] Some drivers, however, do not yet support this feature, including one of our test machines.
[2] Other rounding modes are available, but they all suffer from similar artefacts.

CHAPTER 4

View Reparenting and the Quadtree

One of the important parts of document rendering is that of coordinate transforms.

Traditionally, the document exists in its own coordinate system, **b**. We then define a coordinate system **v** to represent the view.

To eliminate visible error due to floating point precision, we need to ensure that any point (including control points) must be representable as a float both in the document and in view coordinates, i.e:

1. have a limited magnitude,
2. be precise enough to uniquely identify a pixel on the display.

Despite a point not being representable with floats in one coordinate system, it may be representable in another. We can therefore split a document up into several coordinate systems, such that each point is completely representable.

Similarly, the points making up the visible document need to all be representable (to at least one pixel's worth of precision). To achieve this, we use a quadtree where each node stores points within its bounds in its own coordinate system. Objects which span multiple nodes are clipped, such that no points[1] lie outside the quadtree node.

Setting aside the possibility that an object might span multiple nodes for the time being, let's investigate how the coordinates of a point are affected by placing it in the quadtree.

Suppose $p = (p_x, p_y)$ is a point in a global document coordinate system, which we'll consider the root node of our quadtree. p_x and p_y are represented by a finite list of binary digits $x_0 \dots x_n$, $y_0 \dots y_n$. Consider now the pair (x_0, y_0):

$$x_0 = \begin{cases} 0 & \text{if } p_x \text{ is in the left half of } d \\ 1 & \text{if } p_x \text{ is in the right half of } d \end{cases}$$

$$y_0 = \begin{cases} 0 & \text{if } p_y \text{ is in the bottom half of } d \\ 1 & \text{if } p_y \text{ is in the top half of } d \end{cases}$$

We have therefore found which child node of our quadtree contains p. The coordinates of p within that node are $(x_1 \dots x_n, y_1 \dots y_n)$. By the principle of mathematical

[1] except some control points

induction, we can repeat the process to move more bits from the coordinates of p into the structure of the quadtree, until the remaining coordinates may be precisely represented.

This implies that the quadtree is equivalent to an arbitrary precision integer datatype. By reversing the process, any point representable in the quadtree can be stored as a fixed-length bit string.

Furthermore, we can get approximations of points by inserting them into higher levels of the quadtree, with their coordinates rounded. By then viewing the document at a certain level of the quadtree, we then not only do not need to consider bits of precision which are not required to represent the point to pixel resolution, we can also cull objects outside the visible nodes at that level.

Indeed, we can guarantee that, by selecting the level of the quadtree we view such that the view width and height lie within the range $(0.5, 1]$, we can ensure that at most four nodes need to be rendered.

4.1 Clipping cubic Béziers

In order to ensure that the quadtree maintains precision in this fashion, we need to ensure that all objects are entirely contained within a quadtree node. This is not always the case and, indeed, when zooming in on any object, eventually the object will span multiple quadtree nodes.

TODO: We can show this by showing that, given $(a, b) \in \mathbb{R}, \exists$ binary fraction c such that $a < c < b$.

We therefore need a way to subdivide objects into several objects, each of which is contained within one node. To do this, we need to *clip* the cubic Bézier curves from which our document is formed to a rectangle.

Cubic Bézier curves are defined parametrically as two cubics: $x(t)$ and $y(t)$, with $0 \leq t \leq 1$. We clip these to a rectangular bounding box in stages. We first find the intersections of the curve with the clipping rectangle by finding the roots[2] of the cubic shifted to match the corners of the rectangle. This produces some spurious points, as it assumes the edges of the clipping rectangle are lines with infinite extent, but this at worst introduces some minor inefficiency in the process and does not affect the result.

Once the values of the parameter t which intersect the clipping rectangle have been determined, the curve is split into segments. To do this, the values 0 and 1 (representing the endpoints) are added to the list of intersecting t values, which is then sorted. Adjacent values t_0 and t_1 form a segment. The midpoint of that segment (with the value $\frac{t_0+t_1}{2}$) is evaluate and if it falls outside the clipping rectangle, the segment is discarded.

Finally, the segments are re-parametrised by subdividing the curve using De Casteljau's algorithm[45]. By re-parameterising the curves such that $0 \leq t \leq 1$,

[2]While there is a method for solving cubics exactly[11], we instead use numeric methods to avoid the need for square root operations, which cannot be done exactly on some of the numeric types we used.

we ensure that the first and last coefficients have the endpoints' coodinates, and therefore lie in the quadtree node.

TODO: Prove that the other control points' magnitude is reduced, and try to quantify it, prove that it will never overflow.

4.2 Implementation Details

- Store object ID ranges.
- Pointers to children and parent.
- Linked-list of "overlay" nodes for mutation.
- Have billions of bugs.

CHAPTER 5

Experimental Results

These are all iPython-y at the moment.

Roughly 3s/frame for GMP rationals, 16ms for Quadtree which is still slightly broken.

5.1 Performance per object

5.2 Performance per onscreen object

5.3 Performance per zoom-level

5.4 Stability of performance

CHAPTER 6

Further Work and Conclusion

The use of spatial data structures to specify documents with arbitrary precision has been validated as a viable alternative to the use of arbitrary precision numeric types where an arbitrary (rather than infinite) amount of precision is needed. Once constructed, they are faster in many circumstanced, and the structure can also be used for culling. When the viewport moves and scales smoothly, the cost of constructing new nodes is amortised over the movement. Unfortunately, the mutation of the quadtree is difficult and slow, and discontinuous movement can result in a large number of nodes needing to be created.

Quadtree seems to be viable and is really performant.

Loop-blinn shading.

Bibliography

[1] IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Std 754-1985* (1985).

[2] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70.

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

[4] Bentley, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept. 1975), 509–517.

[5] Berners-Lee, T., and Connolly, D. Hypertext markup language – 2.0. *Internet RFC 1866* (1995).

[6] Blinn, J. A trip down the graphics pipeline: Grandpa, what does "viewport" mean? *Computer Graphics and Applications, IEEE 12*, 1 (Jan 1992), 83–87.

[7] Bos, B., Wium Lie, H., Lilley, C., and Ian, J. Cascading style sheets, level 2, CSS2 specification. `http://www.w3.org/TR/1998/REC-CSS2-19980512/`. Retrieved 2014-05-22.

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

[9] Brown, P. NV_half_float. `http://www.opengl.org/registry/specs/NV/half_float.txt`, 2002. Retrieved 2014-05-20.

[10] Brown, P., Lichtenbelt, B., Licea-Kane, B., Merry, B., Dodd, C., Werness, E., Sellers, G., Roth, G., Bolz, J., Haemel, N., Boudier, P., and Daniell, P. ARB_gpu_shader_fp64. `http://www.opengl.org/registry/specs/ARB/gpu_shader_fp64.txt`, 2010. Retrieved 2014-05-20.

[11] Cardano, G. Artis magnae sive de regulis agebraicis: liber unus, 1545.

[12] Corporation, I. 3D/Media — 3D Pipeline (Ivy Bridge). *Intel OpenSource HD Graphics Programmer's Reference Manual (PRM) 2*, 1 (2012).

[13] Corporation, O. java.math.BigDecimal. `http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`. Retrieved 2014-05-19.

[14] Corporation, O. java.math.BigInteger. `http://docs.oracle.com/javase/6/docs/api/java/math/BigInteger.html`. Retrieved 2014-05-19.

[15] Dahlström, E., Dengler, P., Grasso, A., Lilley, C., McCormack, C., Schepers, D., Watt, J., Ferraiolo, J., Jun, F., and Jackson, D. Scalable vector graphics (svg) 1.1 (second edition). *W3C Recommendation* (August 2011). Retrieved 2014-05-23.

[16] Dawson, B. That's Not Normal — the Performance of Odd Floats. https://randomascii.wordpress.com/2012/05/20/thats-not-normalthe-performance-of-odd-floats/ accessed 2014-10-18, 2012.

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

[18] Finkel, R. A., and Bentley, J. L. Quad trees a data structure for retrieval on composite keys. *Acta informatica 4*, 1 (1974), 1–9.

[19] Foley, J. *Computer Graphics: Principles and Practice*. Addison-Wesley systems programming series. Addison-Wesley, 1996.

[20] Frisken, S. F., Perry, R. N., Rockwood, A. P., and Jones, T. R. 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* (2000), ACM Press/Addison-Wesley Publishing Co., pp. 249–254.

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

[22] Fuchs, H., Kedem, Z. M., and Naylor, B. F. On visible surface generation by a priori tree structures. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1980), SIGGRAPH '80, ACM, pp. 124–133.

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

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

[25] Green, C. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 courses* (2007), ACM, pp. 9–18.

[26] Hayes, B. Pixels or perish. *American Scientist 100*, 2 (2012), 106 – 111.

[27] Incorporated, A. S. *PostScript Language Reference*, 3rd ed. Addison-Wesley Publishing Company, 1985 - 1999.

[28] Incorporated, A. S. *PDF Reference*, 6th ed. Adobe Systems Incorporated, 2006.

[29] Kessenich, J. GL_ARB_shader_precision. https://www.opengl.org/registry/specs/ARB/shader_precision.txt accessed 2014-10-17, 2010.

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

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

[32] Knuth, D. Preliminary preliminary description of TEX. http://www.saildart.org/TEXDR.AFT[1,DEK]1, 1977. Retrieved 2014-05-20.

[33] Knuth, D. *Seminumerical Algorithms*, 3rd ed., vol. 2 of *The Art of Computer Programming*. Addison–Wesley, 1998.

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

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

[36] Loop, C., and Blinn, J. Rendering vector art on the gpu. *GPU gems 3* (2007), 543–562.

[37] Maddock, J., and Kormanyos, C. Boost multiprecision library. http://www.boost.org/doc/libs/1_53_0/libs/multiprecision/doc/html/boost_multiprecision/.

[38] Munroe, R. Pixels. http://xkcd.com/1416/ accessed 2014-09-03.

[39] Nilsson, P., and Reveman, D. Glitz: Hardware accelerated image compositing using OpenGL. In *USENIX Annual Technical Conference, FREENIX Track* (2004), pp. 29–40.

[40] Oh, A., Sung, H., Lee, H., Kim, K., and Baek, N. Implementation of OpenVG 1.0 using OpenGL ES. In *Proceedings of the 9th international conference on Human computer interaction with mobile devices and services* (2007), ACM, pp. 326–328.

[41] Porter, T., and Duff, T. Compositing digital images. In *ACM SIGGRAPH Computer Graphics* (1984), vol. 18, ACM, pp. 253–259.

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

[43] Robart, M. OpenVG paint subsystem over OpenGL ES shaders. In *Consumer Electronics, 2009. ICCE'09. Digest of Technical Papers International Conference on* (2009), IEEE, pp. 1–2.

[44] Salomon, D., Motta, G., and Bryant, D. *Data Compression: The Complete Reference*. Molecular biology intelligence unit. Springer, 2007.

[45] Sederberg, T. W. Computer aided geometric design course notes, 2007.

[46] Segal, M., Akely, K., and Leech, J. *The OpenGL® Graphics System: A Specification*. The Kronos Group, Inc, 2014.

[47] Software, A. Ghostscript, an interpreter for the postscript language and pdf. http://www.ghostscript.com/, 1988. Retrieved 2014-05-21.

[48] Worth, C., and Packard, K. Xr: Cross-device rendering for vector graphics. In *Linux Symposium* (2003), p. 480.

[49] Zerbst, S., and Düvel, O. *3D Game Engine Programming.* Premier Press, 2004.