Pinkney, Alexander J. and Bagley, Steven R. and Brailsford, David F. (2013) No need to justify your choice: pre-compiling line breaks to improve eBook readability. In: ACM Symposium on Document Engineering (DocEng '13), 10-13 Sept 2013, Florence, Italy.

No Need to Justify Your Choice: Pre-compiling Line Breaks to Improve eBook Readability

Alexander J. Pinkney
Document Engineering Lab.
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, UK
azp@cs.nott.ac.uk

Steven R. Bagley
Document Engineering Lab.
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, UK
srb@cs.nott.ac.uk

David F. Brailsford
Document Engineering Lab.
School of Computer Science
University of Nottingham
Nottingham, NG8 1BB, UK
dfb@cs.nott.ac.uk

ABSTRACT

Implementations of eBooks have existed in one form or another for at least the past 20 years, but it is only in the past 5 years that dedicated eBook hardware has become a mass-market item.

New screen technologies, such as e-paper, provide a reading experience similar to those of physical books, and even backlit LCD and OLED displays are beginning to have high enough pixel densities to render text crisply at small point sizes. Despite this, the major element of the physical book that has not yet made the transition to the eBook is high-quality typesetting.

The great advantage of eBooks is that the presentation of the page can adapt, at rendering time, to the physical screen size and to the reading preferences of the user. Until now, simple first-fit line-breaking algorithms have had to be used in order to give acceptable rendering speed whilst conserving battery life.

This paper describes a system for producing well-typeset, scalable document layouts for eBook readers, without the computational overhead normally associated with better-quality typesetting. We precompute many of the complex parts of the typesetting process, and perform the majority of the 'heavy lifting' at document compile-time, rather than at rendering time. Support is provided for floats (such as figures in an academic paper, or illustrations in a novel), for arbitrary screen sizes, and also for arbitrary point-size changes within the text.

Categories and Subject Descriptors

I.7.2 [**Document and Text Processing**]: Document Preparation—*format and notation, markup languages*; I.7.4 [**Document and Text Processing**]: Electronic Publishing

General Terms

Algorithms, Documentation, Experimentation

Keywords

eBooks, Document layout, Typesetting

FINAL DRAFT of Short Paper accepted for
DocEng'13, September 10–13, 2013, Florence, Italy.
 Published version is at:
http://dx.doi.org/10.1145/2494266.2494310.

1. INTRODUCTION

Many studies [9, 5, 17, 13] have shown that good typography is the key to readability. In particular, Bill Hill in *The Magic of Reading* [9] emphasises that both the regularity of whitespace between words and the evenness of line lengths are of particular importance. Traditionally, good typography can be ensured as the document is being created — as is the case with documents stored as PDF — since the size and shape of the output medium is fixed. However, non-PDF eBook formats, such as EPUB and Amazon's Kindle format, do not have a fixed presentation associated with their textual content and therefore they rely on the eBook readers themselves to perform the typesetting and layout.

It would be hoped that eBook readers, being dedicated reading devices, would strive to typeset a book's content in the most readable form possible. Unfortunately, most eBook readers seem to rely on a simple first-fit line-breaking algorithm, which means that the whitespace between words can vary wildly between consecutive lines of text: extremely wide spacing onsome lines,andextremelytightspacingonothers.Thiscanbeavoided by using more complex line-breaking algorithms that can identify optimal breakpoints in the text, such as that detailed by Knuth and Plass in [12]. These algorithms are not currently used in eBook readers because they are computationally complex: the Knuth-Plass algorithm, for example, runs in O(n^2) time, whereas first-fit runs in O(n). (Both of these are per paragraph, where n is the number of possible breakpoints in the paragraph.) With some pruning, the effective complexity of the Knuth-Plass algorithm can be reduced to linear time [10, 7, 11], but large constant factors still make the algorithm slow in practice. In any case, the Knuth-Plass algorithm is certainly not the last word in line breaking algorithms. Mittelbach and Rowley [14] mention some limitations of the Knuth-Plass algorithm: as an example, it has no mechanism to avoid (nor indeed any knowledge of) vertical rivers of whitespace. Inevitably, adding support to avoid rivers, and for any of the other nuances used by hand compositors, would add further complexity. Portable eBook readers have both limited processing power and are powered by batteries; implementing a more computationally complex algorithm would not only make page turns and point-size changes noticeably slow, but it would also noticeably increase the drain on the battery.

In the absence of new battery technologies, one method to maximise battery life, and hence reading time, is to minimise the complexity of any algorithms that run on portable devices, and so letting their CPUs idle as much as possible. Given these constraints and desires (to maximise battery life and increase readability), we suggest a system whereby certain parts of the typesetting of the document are performed at "compile time" of the document, so that less processing power and battery power needs to be devoted at "view time" on the device.

In a previous paper [15], we described a simple system that enables documents to be partially pre-rendered, whilst retaining a reasonable degree of flowability. The paper outlined the concept of producing multiple partial renderings of a document, which effectively causes the system to 'compile out' the line-breaking algorithm. In this paper, we have reimplemented and extended this system, so that it can be tested on a wider range of devices, and can produce document layouts on the fly that previously would not have been possible. For convenience, in the remainder of this paper, we shall refer to the partially pre-compiled documents as *malleable documents*.

The rest of this paper details the implementation of our new system. Section 2 recaps the implementation of our previous system and also discusses related work in this area. Sections 3 and 4 describe how our malleable documents are generated and viewed respectively. Section 5 evaluates the performance of our system, while section 6 concludes by discussing future work.

2. RELATED WORK

Much work has been done in the field of automated document layout, though most systems are geared towards producing documents with one fixed presentation. Although these documents may have wonderful layout, they cannot easily be scaled to fit every one of the plethora of devices upon which the documents may be viewed.

A fairly comprehensive review of the literature around automatic document formatting was published in [11]. Since most (if not all) literature in this field is geared towards producing documents with static layouts, the computational complexity of the algorithms used has never been of huge concern. It seems that until now, not much thought has been given to the development of document formatting systems that run in real time, or on battery powered devices.

2.1 Our Previous Work

In our 2011 paper [15], we outlined the concept of pre-rendering a document's text multiple times, into multiple *galleys* of text. Each galley is effectively one long column, and each galley is rendered at a different width.

Once the text had been typeset into a galley, the resultant lines of text could be treated as atomic units. These units could then be placed onto the page individually; in particular, if the height of the page was altered, lines could be added to or removed from the bottom of the page, allowing the text to "flow" in a vertical direction.

The resultant document is composed of multiple galley renderings of the source document, each with a different width. At runtime, the renderer chooses the most appropriate width of galley to display, based upon the screen size of the device. If the screen size permits, the rendering algorithm may choose to display multiple columns, in order to best fill the available space. This system, very much a proof-of-concept prototype, provided no support for floats, and no support for any items (for example headers or footers) to span multiple columns.

The system was initially implemented within Component-Object Graphic (COG) PDF, a system developed at Nottingham that provides encapsulation for objects within PDF documents, and allows for their dynamic modification [16, 4, 3, 1]. This medium was chosen simply because the tools for creating these documents were readily available to us, and because PDF provides a reasonable guarantee that its contents will be rendered identically in any PDF viewer.

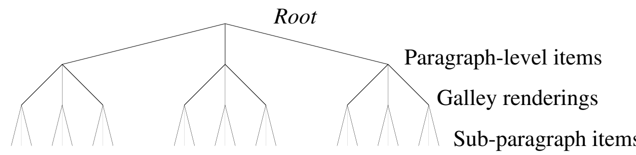

Figure 1: A simple document structure tree. The first level below the root represents all paragraph-level items: headings, paragraphs, figures etc. These items have one child for each galley rendering of the document. These in turn have one child for each sub-component of their content, for example the lines of a paragraph.

3. DOCUMENT GENERATION

Since the previous paper, the system has been reimplemented in HTML, JavaScript and CSS, as well as being extended to add support for floating items, and for items to span across multiple columns.

Previously, the generation of malleable documents involved a manual process built around a pre-existing solution developed at Nottingham. However, in order for the system to be usable, an automated process is required, through which a malleable document can be generated. The underlying principles of the system remain the same, for example the paragraph-tree structure described in [15], as shown in figure 1, is retained.

In our implementation, the source document is described in terms of separate logical blocks; a block is either designated as a 'float', or as a 'paragraph'. Floats are currently limited to referencing images only (with an optional size parameter). Paragraphs, on the other hand, are described by their desired textual content. This is deliberately simplistic, as an aid to testing. (We envisage that in a real system, the source document would have a richer language, perhaps marked up in a form similar to LaTeX source, or in XML.)

Next, the source document is passed through a program to produce the output that becomes the malleable document itself. This program passes the text of each paragraph through an implementation of a line-breaking algorithm (we use Knuth-Plass, but this could be replaced by any other algorithm that performs line breaking and justification). Each paragraph is rendered multiple times, once for each galley width, in order to produce the document's multiple galley renderings. Each line of each rendering of every paragraph is converted into a list of its composite words. All of these words have an associated offset value, which is later used when drawing the text to ensure that each word is positioned on the line with the correct spacing.

The content of the floats is largely left unchanged. A reference to the image, along with its required dimensions, is simply passed through to the output. If dimensions were not explicitly specified in the source document, the pixel size of the image itself is used.

Finally, once the whole of the source document has been processed, the rendered content is output — in the form of the document structure tree shown in figure 1 — encoded in JSON. This becomes the data representing the source document, which, in conjunction with the viewer defined in the next section, becomes a *malleable document*.

4. THE VIEWER

In order to circumvent the browser's default text layout algorithm, and to ensure that our "high quality" pre-computed text layout is used, the viewer must be able to specify the absolute position of every word on each line, in a manner not dissimilar to the internals of a PDF file. The document generator described in the previous section ensures that all the information needed to lay out the text is contained within the generated JSON object representing the document structure tree.

When the viewer is launched, it decides which is the most appropriate galley rendering to display, based on some metric of which rendering will be most aesthetically pleasing. Since we feel it works well, we have not altered the metric defined in [15], which attempts to balance a penalty for excessive inter-column whitespace against a penalty for too many columns.

Although every galley is rendered in the same point size this can be scaled up or down, at view time, based on the preference of the user, to simulate point-size changes. The gaps between words are scaled proportionally, to allow the text to remain correctly justified.

4.1 Floats with a Queue

Our initial attempt at supporting floats took inspiration from TEX, which places floats into a queue until it finds somewhere it deems appropriate to place the first float. In order to emulate this, we define two queues: the *float queue*, and the *line queue*. ('Line queue' is perhaps a slight misnomer, but it is somewhat snappier than 'non-floating items queue'.)

If both queues are empty, as they will be at the start of the layout process, the document structure tree is traversed, and when the first paragraph-level item (see figure 1) is encountered, its subcomponents (of the chosen galley rendering) are added to the requisite queue: lines to the line queue, and floats to the float queue.

When at least one of the queues is not empty, document layout begins. If the float queue is nonempty, and the first float in the queue will fit below the last typeset item, it is placed on the page. If not, items from the line queue are placed one by one, until no more will fit in the current column. When this happens, a new column is started, and the first float in the float queue is output. Whenever the line queue is depleted, and no floats in the float queue will fit at the current point on the page, all subcomponents of the next paragraph-level item from the document structure tree are queued.

Pagination is reasonably simple with this queueing system: as soon as a page is full, the layout can be restarted at the origin of the page using the current status of both queues and the document structure tree. It is entirely possible that floats may appear on pages subsequent to their callout point in the text, but this effect should be no worse than in many current typesetting systems.

While this approach does produce reasonable layouts, and handles floats well without the need for backtracking, it is not particularly conducive to producing layouts with floats that span multiple columns. The queue-based layout described above is rather simplistic: it knows about the size of each component that it lays out but it does not remember the history of the positions of any of the components that are already laid out. This makes it difficult to have items that span more than one column, because there is no mechanism to mark space on the page as being reserved. In order to do this, we must take another approach.

4.2 A Grid-Based Layout

A simple method for allowing parts of a page to be reserved is to break it up into a grid. Grid-based layouts are useful in many situations [6]; one place of particular note is that of modern-day newspapers. We follow the example set by these newspapers, and define our grid's row height to be the same as the leading of the document's body text, and the grid's column width to be the measure of one text column plus the required gutter space.

2. PROBLEMS WITH CURRENT EBOOK READERS

Three formats currently dominate the eBook market: EPUB and Mobipocket, which allow the document to be formatted to fit the device, and PDF, which does not. (Amazon's proprietary Kindle format is derived from Mobipocket; PDF and EPUB are open standards.) Both the EPUB and Mobipocket formats are largely based on XHTML. Whilst the use of XML-derived formats allow the semantic structure of documents to be very well defined, in general their presentation can only be specified in a very loose manner. The user is often presented with a choice of typefaces and point sizes, allowing the reader software to render the document in essentially any arbitrary way it chooses.

CHAPTER 1

Loomings

Call me Ishmael. Some years ago—never mind how long precisely— having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself

1%

Conversely, PDF is entirely presentation-oriented, stemming from its origin of being compiled PostScript. PDF, therefore, will often include no informationon the semantic structure of the document, and will consist simply of drawing operators which describe the document pages. There is no compulsion for these drawing operators to render the page in an order thatmight be considered sensible: for example, if a PDF generator program decided to render every character on a page in alphabetical order, or radially outwards from the centre, the resulting file would still be semantically valid, and the result might well be unnoticeable to the end user. This lack of imposed semantic structure can make it difficult to infer the best way to 'unpick' PDF files to allow their content to be reflowedinto a new layout.

Since an XHTML-derived format has no fixed presentation associated with it, this must be calculated each time the document is displayed, in a similar manner to the way an interpreted programminglanguage needs to be interpreted each time it runs. For an eBook reader to maximise its batterylife (the human reader will be annoyed if the device dies just before the climax of a novel!), the 'interpretation' needs to be as simple as possible - i.e. the algorithm used must not be too complex, since the more CPU cycles spent executing it, the less time the CPU can spend idle, and hence the greater the drain on the battery. Furthermore, the longer that is spent formattingthe output, the longer the delay between page turns on the device, and with the speed of CPUs used in these devices (< 500 MHz) it does not take too large an increase in computation for the page turn to become noticeable.

2.1 Hyphenationand Line-Breaking

eBook readers typically use a 'greedy' algorithm to lay out their text - that is, they place as many words as will fit onto the current line without exceeding it, then start a new line and continue. Although this algorithm is optimal in that it will always fit text onto the fewest possible lines, it often causes consecutive lines to have wildly varying lengths, accentuating either the 'ragged-right' effect of the text, or, in the case of justified text, the inter-word spacing. In general, eBook readers will only hyphenate in extreme cases - indeed the Kindle 3 seems not to do so at all. Knuth and Plass[7] developed a more advanced line-breaking algorithm(now used by TEX) which attempts to minimise large discrepancies between consecutive lines by considering each paragraph as a whole. TEX also uses the hyphenationalgorithm designed by Liang[8], which has been ported to many other applications.

To AV V. Wa fi fl
To AV V. Wa fi fl

2.2 Other Typographical Techniques

Other techniques employed during hand-typesetting and highquality electronic typesetting include the use of kerning and of ligatures. Kerning involves altering the spacing between certain glyph pairs in order to produce more consistent letter spacing, whilst ligatures are single-glyph replacements for two or more single glyphs which may otherwise have clashing components. Some examples of these are shown in figure2. Kerning requiresa table of kernpairs, specific to each font; values from this table must then be looked up for every pair of adjacent glyphs in the document. Ligatures may or may not need to be inserted: if the componentcharacters of the ligaturelie over a potential hyphenation point, it cannot be decided whether to replace them with the ligature until it is known whether the hyphenation point needs to be used.

3. A GALLEY-BASED APPROACH

Our proposed solution, of precomputing several text variants, revisits an approach to typesetting from before the advent of desktop publishing. In the days before DTP, newspaper articles were typeset into long columns known as galleys. Since all columns in the newspaper would be of uniformwidth, all articles couldbe typeset into galleys of the same measure, and then broken as necessary between lines, in order to slot into the final layout of the newspaper. Once the text has been set in this manner, with appropriate hyphenation and justification, the individual lines can be treated as atomic units that will never have to be re-typeset. In essence, each article is 'compiled' only once, but can be used anywhere in the final layout without penalty.

It is this behaviour we wish to emulate. So long as the atomic components of the document are tightly specified, and the reader software can obey the associated drawing instructions (essentially treating them as pre-typeset blocks), the resulting display of the document will be of as high typographic quality as that of the original galley, and the requirement for further computation will be vastly reduced. In order to permit aesthetic layout for a wider range of screen sizes, it seems sensible to create a document containing multiple renderings of the same content, and simply choosing the 'best fit' rendering when the document is displayed.

3.1 A Sample Implementation

Our sample implementation is built around our existing work in PDF and Component-Object Graphics (COGs)[1], but there is no reason why it could not be implemented in any other format capable of tightly specifying page imaging operations. It builds on existing software, principally pdfdit, in conjunction with COGManipulator, as these tools are already capable of producing modular documents with tightly specified rendering.

3.1.1 The COG Model

The Component Object Graphic (COG) model was developed to enable the reuse of semantic components within PDF documents, by breaking the traditional graphically-monolithic PDF page into a series of distinct, encapsulated graphical blocks, termed COGs. In its original incarnation, the COG model did not account for any relationship between individual COGs - it was simply designed as a method by which document components could be easily reused or reordered. The COGs it generates are largely at the granularity of a paragraph, and can still be imaged onto the page in any arbitrary order, independent of reading order.

In order to implement our galley-based design, it is necessary to decrease this granularity, such that each line of text is represented by a separate COG. However, it is also important that the semantic structure of the document is explicitly stored. This is principally so that the reading order of the COGs is maintained, and also so that the reader software can identify paragraphs, headings etc. to enable them to be laid out

Figure 2: An excerpt from our 2011 paper, typeset and rendered by our new system.

The viewer uses the dimensions of the float, as specified in the document structure tree (see section 3), to determine how many columns it should span. The float is scaled to span the integer multiple of column widths that most closely matches its 'natural' size, though for reasons that should hopefully be obvious, this number is limited to a minimum of 1, and a maximum of the number of columns on the page. Additionally, checks are made to ensure that the scaling will not cause the height of the figure to exceed that of the page.

An advantage of this grid-based approach is that it no longer requires the use of queues, either for lines, or for floats. The viewer simply traverses the document structure tree, placing each item in the first available place in the grid. In the case of floats, or other items larger than multiples of the main leading, spaces in the grid can be marked as reserved, to prevent other items from trampling over their reserved space. If a float will not fit directly below the previous item to be placed, the grid is walked over until a gap of sufficient size can be found. Figure 2 shows an example of a document laid out with this system.

Pagination becomes a little trickier when floats are allowed to span multiple columns. For example, if a float, whose natural size would lead it to span n columns, is encountered in the document structure tree when there are $(n - 1)$ or fewer columns remaining to be typeset on the page, it must be decided how best to handle the situation. Three obvious options present themselves: alter the float to span fewer columns; delay the placement of the float until the start of the next page; or backtrack and check whether there is room to move the float back one or more columns, by shunting non-floatable text lines forwards.

The first option is clearly not desirable behaviour, given that shrinking a float may well reduce its legibility. Additionally, if this becomes a common problem, it it likely to be noticeable that floats spanning into the rightmost column of the page appear shrunken. The second option (delaying placement until the following page) is a reasonable compromise, though it will increase float-drift (whereby floats become separated from their callout points in the text), which is not ideal. The third option (backtracking and shunting) is likely to produce the most desirable output, although some computational overhead will be added. One approach is simply to check whether there is enough space immediately to the left (specifically a gap between other, already placed, floats) into which the current float can be placed, with the displaced lines being shunted forwards. This method will not produce layouts as optimal as methods that use full backtracking and check all possibilities, but it will run in much quicker time. A combination of all three of the above options is likely to work best in practice.

5. EVALUATION

Currently, our solution suffers from significantly bloated filesizes: each rendering requires one full copy of the document's source text, in addition to positioning data for each word. Even after compression, this leads to filesizes that are an order of magnitude larger than the source text. We have not yet given great consideration to space efficiency: it is possible that by separating the text from the positioning data with the use of pointers (similar to the approach described in [2]) a more space-efficient encoding could be devised.

Another drawback of our system is that the choice of typeface must be set at document compile-time. There is no reason, in principle, why renderings in multiple typefaces could not be included within the file, though clearly it would be impractical to render the document in every typeface known to mankind. A carefully chosen selection of serif and sans-serif typefaces should hopefully cover most eventualities.

5.1 Computational Performance

The layout system described herein works in a similar manner to a first-fit line-breaking algorithm, in that it places elements on the page in order, in the first place they will fit. Items that are the same size as a single grid cell, such as lines of text set in the main point size, can simply be placed in the first empty slot in the current column, or the first empty slot in the next column, should there be no empty spaces. For the placement of items that are larger than a single grid cell, there is some overhead required to step through the grid until a suitable position can be found. Once a position has been found, each grid cell that it overlaps must be marked as being reserved.

Whilst this algorithm does have a greater-than-linear time complexity, the problem size is actually reduced in comparison to a first-fit text layout algorithm, since our system uses lines of text as its atomic units, rather than individual words. For this reason, we feel that our algorithm should still be efficient enough to merit use on portable eBook readers.

5.2 Aesthetic Performance

Aesthetically speaking, our system produces layouts that we feel most people would consider to be 'good'. The system can guarantee use of a high-quality line breaking algorithm, since it has effectively been compiled in, and so the only remaining concern is that the columns of text and floats are laid out in a pleasing manner.

Harrington et al. [8] identified nine aesthetic measures for automated document layout. A number of these measures (alignment, regularity, uniform separation, white-space free-flow, uniformity) are particularly well satisfied by our system, due to its use of a grid to provide regular layout.

We intend to run a user study to assess the more qualitative aspects of our system.

6. FUTURE WORK

The system as described in this paper has only very basic support for floats. A particular limitation is that unlike paragraphs, each float has only one rendering, which must be scaled up or down as required, to fit across multiples of columns. Whilst for image-based figures or illustrations, this is probably already the desired behaviour, other types of floats, such as tables or code listings, would almost certainly benefit from the inclusion of multiple width renderings, with the choice of which rendering to display to be made at view-time.

Since the malleable document and viewer are composed entirely from HTML, CSS, and JavaScript — the core technologies behind EPUB — modifying our system to produce self-contained EPUB files seems an obvious next step.

7. REFERENCES

[1] S. R. Bagley. COG extractor. In *Proceedings of the 2006 ACM Symposium on Document Engineering*, page 31. ACM Press, 2006.

[2] S. R. Bagley. Lessons from the dragon: Compiling PDF to machine code. In *Proceedings of the 2010 ACM Symposium on Document Engineering*, 2010.

[3] S. R. Bagley and D. F. Brailsford. Demo abstract: The COG scrapbook. In *Proceedings of the 2005 ACM Symposium on Document Engineering*, pages 233–234. ACM Press, 2005.

[4] S. R. Bagley, D. F. Brailsford, and M. R. B. Hardy. Creating reusable well-structured PDF as a sequence of component object graphic (COG) elements. In *Proceedings of the 2003 ACM Symposium on Document Engineering*, pages 58–67. ACM Press, 2003.

[5] R. Bringhurst. *The Elements of Typographic Style (v 3.2)*. Hartley & Marks, 2008.

[6] D. Collier. *Collier's Rules for Desktop Design and Typography*. Addison-Wesley, 1991.

[7] D. Eppstein and Z. Galil. Sparse dynamic programming II: Convex and concave cost functions. *J. ACM*, 39(3):546–567, 1992.

[8] S. J. Harrington, J. F. Naveda, R. P. Jones, P. Roetling, and N. Thakkar. Aesthetic measures for automated document layout. In *Proceedings of the 2004 ACM Symposium on Document Engineering*, pages 109–111. ACM Press, 2004.

[9] B. Hill. The magic of reading. Technical report, Microsoft, 1999.

[10] D. S. Hirschberg and L. L. Larmore. The least weight subsequence problem. *SIAM J. Comput.*, 16(4):628–638, 1987.

[11] N. Hurst, W. Li, and K. Marriott. Review of automatic document formatting. In *Proceedings of the 2009 ACM Symposium on Document Engineering*, 2009.

[12] D. E. Knuth and M. F. Plass. Breaking paragraphs into lines. *Software — Practice and Experience*, 11:1119–1184, 1981.

[13] G. E. Legge and C. A. Bigelow. Does print size matter for reading? A review of findings from vision science and typography. *Journal of Vision*, 11(5):8:1–22, 2011.

[14] F. Mittelbach and C. Rowley. The pursuit of quality — how can automated typesetting achieve the highest standards of craft typography? In *EP92 (Proceedings of Electronic Publishing)*, pages 261–273. Cambridge University Press, 1991.

[15] A. J. Pinkney, S. R. Bagley, and D. F. Brailsford. Reflowable documents composed from pre-rendered atomic components. In *Proceedings of the 11th ACM Symposium on Document Engineering*, DocEng '11, pages 163–166, New York, NY, USA, 2011. ACM.

[16] P. N. Smith and D. F. Brailsford. Towards structured, block-based PDF. *Electronic Publishing — Origination, Dissemination and Design*, 8(2 and 3):153–165, June/September 1995.

[17] G. Voorhees. Congeniality of reading on digital devices. Master's thesis, Rochester Institute of Technology, 2011.