Adaptive Document Layout via Manifold Content

Charles Jacobs
Microsoft Research
cjacobs@microsoft.com

Wilmot Li
University of Washington

David H. Salesin
Microsoft Research
University of Washington

Abstract

We present and explore a simple idea for improving document layout on arbitrary devices of different resolutions and size. The key idea is to allow manifold representations of content: multiple versions of anything that might appear in a document, such as text, images, or even stylistic conventions. Content is then selected and formatted dynamically, on the fly, by a layout engine in order to best adapt to a given viewing situation. We propose a user interface for authoring and editing such manifold content, and sketch a few interesting new algorithms that make use of it.

1. Introduction

Paper documents are—by their very nature—static affairs: their physicality confines them to a single immutable layout on a single-size sheet of paper. Electronic documents, by contrast, can and should be much more dynamic. Most importantly, they should adapt seamlessly and attractively to the size and proportions of the display on which they appear—be it a standard monitor, a tiny PDA screen, or a certain format of paper.

Electronic documents today fall woefully short of this ideal. In general, they provide an impoverished layout in comparison with their traditional, physical counterparts. Moreover, they provide an exceedingly limited ability to adapt to different displays. Typically, either the width of the text is expanded to fill the available window, creating long, illegible lines of text, or the text area is kept fixed, which solves the first problem but requires inconvenient scrolling when the target display is too small. With the proliferation of new, differently sized display devices, this inability to adapt layouts to different display sizes is becoming an increasingly critical problem.

Why are we in this predicament? The answer is that adaptive layout is fundamentally a hard problem, and one of the key challenges—the one this paper addresses—involves the close coupling between content and layout. To format a document well for various page sizes, different content is often necessary. For example, a multicolumn sidebar may look fine on a widescreen display, but on a portrait display it may have to be placed at the bottom of the page, so as not to squeeze out the main story. On a PDA, the same "sidebar" might have to be moved to a separate page entirely. Even more tricky is the need for editorial changes to content to make a given layout work. As Knuth acknowledges [9], a computer should be able to achieve better results than a human typesetter—"unless we give this person the liberty to change the wording in order to obtain a better fit." Often one is forced to make last-minute changes to figures or text in order to, say, meet a page limit, produce better arrangements of figures and text, or eliminate annoying "widows" or "orphans."

In this paper, we suggest a simple, new approach to adaptive document layout. The key idea is to allow *manifold representations of content*—i.e. multiple versions of *anything* that might appear in a document, such as text, images, even stylistic conventions. This content is then selected and formatted dynamically to fit the target display device. To make this approach practical, we need a document *representation* that is flexible enough to represent manifold content (Section 2); a content *authoring system* (Section 3) that makes handling all of these multifarious versions natural; and finally, a *layout engine* that adapts and formats a document's manifold content automatically, on the fly (Section 4).

We are by no means the first to look at adaptive layout. Much of the earliest work in document layout focused on text formatting problems, such as how to arrange text into lines, paragraphs and higher-level semantic structures [6, 9, 12]. More recently, researchers have begun to focus on the *page layout* problem, whereby relational grammars [13], constraints [2, 3, 4, 5], or various forms of optimization [8] are used to arrange content and design elements onto a page based on some notion of "goodness." In addition, the World-Wide-Web Consortium (W3C) has recently adopted several standards (most notably, the Extensible Stylesheet Language (XSL) [1] and Cascading Style Sheets (CSS) [11]) that support the decoupling of a document's content from its formatting rules as a way to adapt how information is presented. Our idea of using manifold content is largely complementary to all of these earlier efforts, including our own recent work on producing adaptive page layouts that adhere to an underlying design grid [7].

2 Representation

To represent manifold content, we use a tree data structure, called the *document tree*, that contains two types of nodes: *content nodes*, which represent a continuous, flat piece of document content; and OR *nodes*, which group together alternate versions of content and are embedded within content nodes where the manifold representations can be inserted. Note that children of an OR node (which are, themselves, content nodes) can also contain alternate versions of content, resulting in a deeper document tree with nested OR nodes. In practice, we use an XML file format to specify this document structure. More syntactic details can be found in Li [10].

3 Authoring

Manifold content is useful only if it can be authored easily and effectively. Indeed, one reasonable objection to our whole approach might be that it sounds like too much work for authors to create not just one, but many possible versions of their document. However, we argue that, with the right authoring tools, this approach is not necessarily so onerous. In many cases, the author is already doing this kind of work, such as adapting a document for several different audiences (e.g., when preparing a journal article from an earlier conference paper). In this case, our approach could be used to keep the different versions in synch, rather than maintaining separate documents. In addition, as already mentioned, authors commonly make small changes to a document in order to fix up various formatting problems—in this case, our system helps keep around all of the possible versions. Finally, while manifold content may not always be worth the cost, one can easily imagine situations in which a document—e.g., some advertising copy—is written once for wide distribution over an electronic medium, and for which entering different versions of content is a minimal price to pay for the sake of a far greater visual impact.

3.1 The authoring user interface

We see two main challenges in designing an authoring tool. The first difficulty is in finding a way to display the full richness of the manifold content without overwhelming the user with its potential complexity. The second challenge is in coming up with a user interface that allows the author to edit content and specify alternate versions easily and naturally.

Our basic approach is to hide most of the document's structure at any given time and allow the user to interact with what appears to be, at first glance, just a single, linear view of the document. However, areas of manifold content within this *browser pane* are indicated by a faint, dotted line beneath that portion of the content; selecting an area of manifold content causes all other versions of the content to

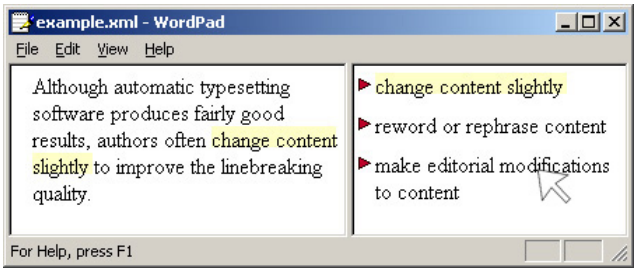

Figure 1. Screenshot of authoring user interface.

appear in a separate *alternate-version* pane, to the right of the browser pane. The currently selected version of content is indicated by yellow highlighting in both panes. To get a better sense of the interface, see Figure 1.

The author is always free to edit any text that appears in either pane. In addition, the author can select alternate versions of content by simply clicking on the selection in the alternate-version pane, in which case this version becomes highlighted and replaces the version currently appearing in the browser pane. In this way, by focusing on the browser pane, the author can get a good sense of how the selected version reads, or appears, alongside the other content in the document, without the distraction of any of its alternate versions. The author can also create new manifold content in the alternate-version pane (either by adding a new entry from scratch or by copying an existing version).

We have found these operations to feel quite natural in practice, once the user gains a little bit of familiarity with the system. Obviously, more formal user studies are required to evaluate and hone the interface more thoroughly.

3.2 Modifying the document tree

Most editing operations in the UI are supported by straightforward transformations of the document tree. For example, creating a new version for an existing region of manifold content is just a matter of adding a new child to the corresponding OR node in the tree. To create a new region of manifold content, a new OR node must be inserted into the appropriate content node, replacing the selected content. The selection is then moved beneath the OR node, becoming its only child. Deleting or editing versions of content are similarly straightforward.

The most complicated operation involves the selection of a new region of manifold content that spans several existing manifold content regions—and, in the worst case, intersects them in some arbitrary way. In our limited testing, the need for these worst-case non-hierarchical manifold content selections has not come up—although we do have an algorithm to support them should the need arise. A full description of this procedure can be found in Li [10].

4 Layout engine

With the benefit of manifold content, attractive text and page layout becomes a lot easier. Given a document tree and a rectangular display region as input, the layout algorithm starts by traversing the tree. As it encounteres content nodes of different specific types, it calls an appropriate *layout engine* to render the portion of the document represented by the content node, passing it the rectangular region in which to render the content.

Currently, we support two types of layout engines: a *composite-page layout engine*, and a *simple-page layout algorithm*. The composite-page layout engine determines the overall structure for the rectangular region it is passed. The content node that caused this engine to be called designates a set of alternative ways of dividing up the page into *panes* (specified via XSL templates). The content node also contains a collection of named content that is to be distributed among the different panes. The composite-page engine chooses among the different layout alternatives based on the dimensions of the rectangular region it is passed and the particular collection of content in the node. (For instance, if the rectangle is tall and skinny, the panes might be laid out vertically, whereas if it is short and wide, the engine might choose a horizontal layout instead.) The layout engines for the various pieces of named content are then called recursively in each pane. Within a pane, the simple-page layout engine is responsible for laying out the various primitive elements like figures and text. The rest of this section describes some of the more interesting aspects of the simple-page layout algorithm in more detail.

4.1 Manifold text formatting

To format paragraphs of text, we use a modified form of Knuth's dynamic-programming-based line-breaking algorithm [9]. The original algorithm takes as input a paragraph of text and first determines a set of potential break points $B = \{b_1, b_2, \ldots, b_m\}$. This set includes all inter-word spaces as well as legal hyphenation positions within words. Knuth's algorithm uses dynamic programming to find, in $O(m)$ time, the set of breaks $B_O \subseteq B$ that result in the best paragraph, as determined by some measure of goodness. (For justified text, this measure simply considers how well each line of text fits into the available space.)

To incorporate alternate versions, we include additional break points in B that correspond to the various wordings specified in each OR node of the document tree. For each b in B, we determine all preceding break points that should not appear together with b in a valid solution (i.e., all breaks in different child subtrees of a common OR ancestor), and record these in the *conflict list* for b. During the dynamic programming loop, we use the conflict lists to quickly discard invalid lines. With these modifications, the algorithm will find the best line-breaking solution over all choices of content in linear time (assuming a uniform distribution of OR nodes over the paragraph). Thus, depending on the size and shape of the region into which the text must flow, the system may choose different versions of content to optimize the line-breaking quality. Note that a naive algorithm that tries all combinations of alternate content is exponential in the number of OR nodes within the paragraph.

4.2 Placing manifold figures

By contrast, we use a simple, brute-force approach for selecting among manifold (floating) figures and placing them onto the page. Even though such an algorithm is exponential in the number of figures on the page, we have so far not encountered documents with a huge number of these (although we acknowledge that such documents are certainly possible). For each combination of figures, we compute a score for the "goodness" of the page layout incorporating those figures. Once all combinations have been tried, we simply pick the best one.

Determining the "goodness" of a page involves measuring distances between figures and their textual references, and penalizing figures that do not fall on the same page as their reference. In addition, a score is computed for the formatting of each line of text. Extra penalties are added for any widows or orphans. There are also parts of the metric that the page designer can tune according to the document's style—e.g. metrics that prefer small or large figures, or figures that together consume a certain proportion of the page.

5 Results

To evaluate our system, we created a simple manifold-content version of a real magazine article and formatted it at different display sizes using our layout engine (Figure 2). Note how the layout engine chooses different content to optimize the layout for the different displays. In particular, differently cropped versions of the main image are automatically selected to help achieve the specified style for both portrait and landscape orientations. Similarly, the layout engine chooses a smaller title with no teaser paragraph for the PDA-sized display.

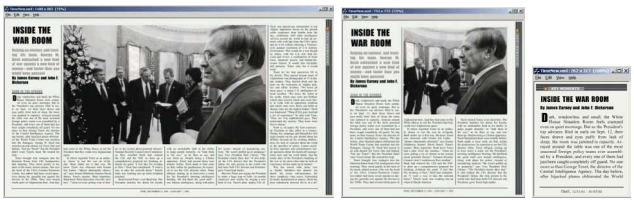

Figure 2. Article from TIME magazine rendered using our layout engine at different sizes. Note how different content has been selected to optimize page quality for the various displays.

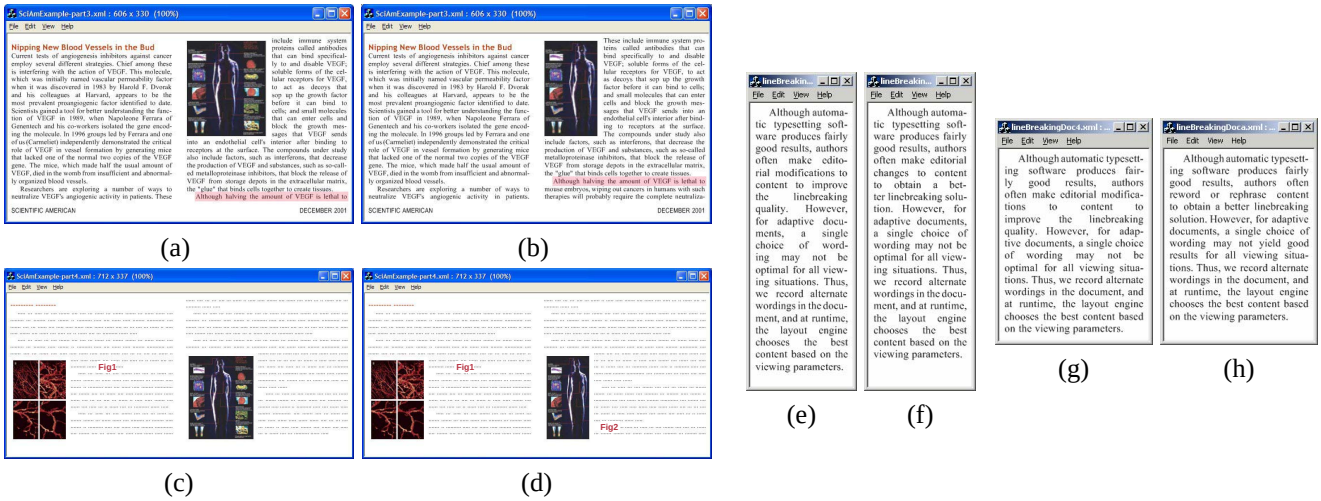

Figure 3. Fixing layout problems using manifold content. By choosing a different figure in (b), the layout engine can eliminate the end-of-page widow in (a). Similarly, choosing a narrower figure in (d) allows "Fig 2" to appear on the same page as its textual reference. Finally, our linebreaking algorithm uses alternate wordings in (f) and (h) to fix the poor inter-word spacing in (e) and (g), which do not have manifold content.

Figure 3 shows some other ways in which the layout engine's optimization procedure can use alternate content to prevent typical layout problems. Figure 3(a) shows a layout that contains a widow at the bottom of the second column. Since widows/orphans result in poor layout scores the optimizer is able to eliminate this artifact by choosing an alternate version of the second image (Figure 3(b)).

Another common layout problem occurs when a figure gets pushed off the page on which a reference to it appears, or vice versa. Figure 3(c) shows a layout in which the reference to "Fig 2" has been pushed off the page by the very figure it references. With an alternate version of the second image specified in the document, the layout engine succeeds in reuniting the errant reference (Figure 3(d)).

Figures 3(e)–(h) show how our linebreaking algorithm takes advantage of alternate content to produce the best possible results for justified text at different display sizes. For comparison, we show the results obtained using a version of the paragraph that does not contain alternate content.

6 Conclusions

While this paper contains at least one or two algorithmic innovations, as well as several new user interface ideas, the primary value of this paper is in its new, different and fundamentally very simple approach towards adaptive layout. Although we have not, by any means, fully explored all of the possible document layout scenarios available via this approach, our early experimental results, presented here, provide some promising indications of the approach's potential as a viable, versatile tool.

References

[1] S. Adler. Extensible stylesheet language, 2001.

[2] G. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint cascading style sheets for the web. In *Proceedings of UIST 99*, 1999.

[3] G. Badros, J. Nichols, and A. Borning. Scwm - an intelligent constraint-enabled window manager. In *Smart Graphics 00*, 2000.

[4] A. Borning, R. Lin, and K. Marriott. Constraint-based document layout for the web. *Multimedia Systems*, 8:177–189, 2000.

[5] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of UIST 97*, pages 87–96, 1997.

[6] R. Furuta, J. Schofield, and A. Shaw. Document formatting systems: Survey, concepts and issues. *ACM Computing Surveys*, pages 417–472, 1982.

[7] C. Jacobs, W. Li, , E. Schrier, D. H. Salesin, and D. M. Bargeron. Adaptive grid-based document layout. In *Proceedings of SIGGRAPH 03*, 2003. (To appear).

[8] R. Johari, J. Marks, A. Partovi, and S. Shieber. Automatic yellow-pages pagination and layout. Technical report, MERL, 1996.

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

[10] W. Li. Adaptive multi-representation documents. Master's thesis, University of Washington. 2002.

[11] H. W. Lie and B. Bos. Cascading Style Sheets, Level 1, 1996.

[12] A. J. H. Peels, N. T. M. Janssen, and W. Nawijn. Document architecture and text formatting. *ACM Transactions on Information Systems*, 1985.

[13] L. Weitzman and K. Wittenburg. Automatic presentation of multimedia documents using relational grammars. In *Proceedings of ACM Multimedia Conference*, 1994.