Adaptive Grid-Based Document Layout

Charles Jacobs[1] *Wilmot Li*[2] *Evan Schrier*[2] *David Bargeron*[1] *David Salesin*[1,2]

[1]Microsoft Research [2]University of Washington

Abstract

Grid-based page designs are ubiquitous in commercially printed publications, such as newspapers and magazines. Yet, to date, no one has invented a good way to easily and automatically adapt such designs to arbitrarily-sized electronic displays. The difficulty of generalizing grid-based designs explains the generally inferior nature of on-screen layouts when compared to their printed counterparts, and is arguably one of the greatest remaining impediments to creating on-line reading experiences that rival those of ink on paper. In this work, we present a new approach to adaptive grid-based document layout, which attempts to bridge this gap. In our approach, an adaptive layout style is encoded as a set of grid-based templates that know how to adapt to a range of page sizes and other viewing conditions. These templates include various types of layout elements (such as text, figures, etc.) and define, through constraint-based relationships, just how these elements are to be laid out together as a function of both the properties of the content itself, such as a figure's size and aspect ratio, and the properties of the viewing conditions under which the content is being displayed. We describe an XML-based representation for our templates and content, which maintains a clean separation between the two. We also describe the various parts of our research prototype system: a layout engine for formatting the page; a paginator for determining a globally optimal allocation of content amongst the pages, as well as an optimal pairing of templates with content; and a graphical user interface for interactively creating adaptive templates. We also provide numerous examples demonstrating the capabilities of this prototype, including this paper, itself, which has been laid out with our system.

CR Categories I.7.4 [Document and Text Processing]: Electronic Publishing; I.3.6 [Computer Graphics]: Methodology and Techniques; Interaction techniques.

Keywords Adaptive layout, templates, pagination, constraints, dynamic programming, HTML, PDF, XML, XSL, CSS.

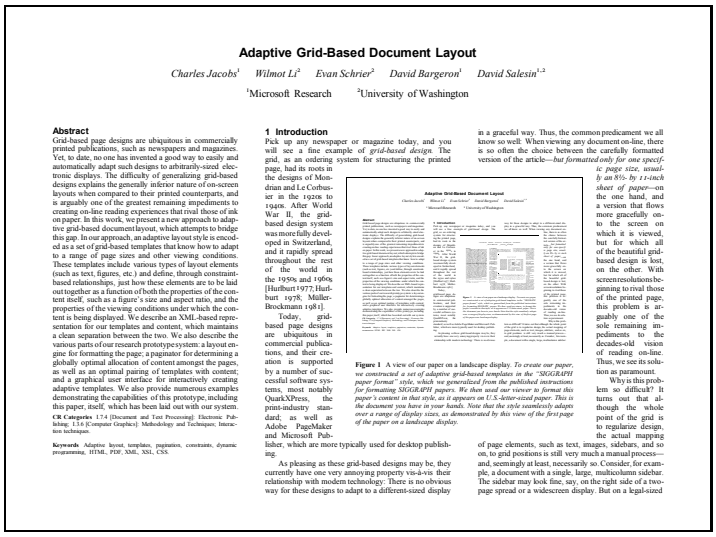

Figure 1 A view of our paper on a landscape display. *To create our paper, we constructed a set of adaptive grid-based templates in the "SIGGRAPH paper format" style, which we generalized from the published instructions for formatting SIGGRAPH papers. We then used our viewer to format this paper's content in that style, as it appears on U.S.-letter-sized paper. This is the document you have in your hands. Note that the style seamlessly adapts over a range of display sizes, as demonstrated by this view of the first page of the paper on a landscape display.*

1 Introduction

Pick up any newspaper or magazine today, and you will see a fine example of *grid-based design*. The grid, as an ordering system for structuring the printed page, had its roots in the designs of Mondrian and Le Corbusier in the 1920s to 1940s. After World War II, the grid-based design system was more fully developed in Switzerland, and it rapidly spread throughout the rest of the world in the 1950s and 1960s [Hurlburt 1977; Hurlburt 1978; Müller-Brockmann 1981].

Today, grid-based page designs are ubiquitous in commercial publications, and their creation is supported by a number of successful software systems, most notably QuarkXPress, the print-industry standard; as well as Adobe PageMaker and Microsoft Publisher, which are more typically used for desktop publishing.

As pleasing as these grid-based designs may be, they currently have one very annoying property vis-à-vis their relationship with modern technology: There is no obvious way for these designs to adapt to a different-sized display in a graceful way. Thus, the common predicament we all know so well: When viewing any document on-line, there is so often the choice between the carefully formatted version of the article—*but formatted only for one specific page size, usually an 8½- by 11-inch sheet of paper*—on the one hand, and a version that flows more gracefully onto the screen on which it is viewed, but for which all of the beautiful grid-based design is lost, on the other. With screen resolutions beginning to rival those of the printed page, this problem is arguably one of the sole remaining impediments to the decades-old vision of reading on-line. Thus, we see its solution as paramount.

Why is this problem so difficult? It turns out that although the whole point of the grid is to regularize design, the actual mapping of page elements, such as text, images, sidebars, and so on, to grid positions is still very much a manual process—and, seemingly at least, necessarily so. Consider, for example, a document with a single, large, multicolumn sidebar. The sidebar may look fine, say, on the right side of a two-page spread or a widescreen display. But on a legal-sized sheet of paper or on a portrait display, the "sidebar" may actually have to be placed at the bottom of the page so as not to squeeze out the main story. And on a Personal Digital Assistant (PDA), this same "sidebar" might have to be moved to a separate page entirely, perhaps made available through a hypertext

Figure 2 An article formatted in an adaptive "U.S. News & World Report" style, at various sizes. *The first row of the figure shows the first few pages of a document on U.S.-letter-sized paper. The second and third row show the same document formatted for a slightly smaller TabletPC screen, in two orientations: portrait and landscape. The fourth row shows the same document, once again, but formatted for a Personal Data Assistant (PDA) screen. The final row shows the same document, one last time, but formatted with larger fonts on a portrait TabletPC. The adaptive style actually accommodates a full continuous range of sizes and user preferences, of which these are a few examples. Notice how in addition to the obvious changes in the number and width of columns, and placement and scale of header text, images, and so on, the layout engine also makes more subtle adjustments, including selecting among a variety of different sizes and croppings of the images displayed on each page.*

link from the main page. The problem is of course compounded for more complex layouts, involving potentially multiple sidebars, figures, pull quotes, etc.—all being merged into a single page design.

Thus, the quandary we are in. Grid-based design systems, for good reason, do not support "document reflow." And, also for good reason, the systems that *do* support reflowing of document content—of which there are many well-known examples, most notably, Microsoft Word, the HyperText Markup Language (HTML), and Knuth's typesetting system, TEX [Knuth 1986]—treat the document's contents more or less as a single one-dimensional "flow" snaking from one page to the next, rather than supporting grid-based design.

In this paper, we propose a new approach to adaptive grid-based document layout that bridges this gap. The basic idea is to encode each adaptive layout "style" (such as the "New Yorker magazine" style) as a collection of *adaptive grid-based page templates* that include style attributes, layout elements (such as text, cartoons, advertisements, etc.), and constraint-based relationships among them. Each adaptive template is designed to adapt to a range of display dimensions, as well as to other types of viewing conditions, such as an increased font size. The document's content is then selected and formatted dynamically to fit the viewing situation at hand—i.e., the display device being used, as well as other user preferences. Figures 1 and 2 demonstrate the kind of results we have been able to achieve with this approach.

Making this approach practical has entailed a variety of research challenges. First, we need a *representation* for templates and content that is flexible enough to represent all the different style attributes, layout elements, and constraint-based relationships that may exist in a document. Second, we need a *layout engine* that makes good use of this representation to format a document's content automatically, on the fly. Third, we need a *paginator* that can determine the globally optimal pairing of content with templates. And finally, we need a *template authoring* tool for creating adaptive templates. We briefly review related work before we delve into the details of each of these areas.

2 Related work

Early work on software for document layout focused largely on text formatting—the arrangement of text into lines, paragraphs, and higher-level semantic structures [Furuta et al. 1982; Knuth and Plass 1981; Peels et al. 1985]. In contrast, our work on layout encompasses a richer and more complex document model that includes modern layout elements such as sidebars, overlapping figures and text, and floating figures.

At the heart of our system lies the decoupling of a document's content from its stylistic formatting rules. Several standards endorsed by the World-Wide-Web Consortium (W3C) have emerged to support this separation, most notably the Extensible Stylesheet Language (XSL) [Adler 2001] and Cascading Style Sheets (CSS) [Lie and Bos 1996]. A constraints-based extension to Cascading Style Sheets (CCSS) has also been proposed [Badros et al. 1999], and others have tried encapsulating stylistic rules in page templates [Purvis 2002]. Style is imposed top-down on document content in these systems, without consideration for the local characteristics of the content itself. In contrast, our templates support a protocol whereby alternate document content—for example, a wider version of an image or an optional drawing—can be chosen if it will improve the overall page layout.

Our templates encode two-dimensional relationships among layout elements as constraints that must be resolved to evaluate a particular layout. Other approaches have involved creating an underlying grid for page layout automatically by looking at the page's content [Feiner 1988] or through the use of relational grammars [Weitzman and Wittenburg 1993; Weitzman and Wittenburg 1996]. Other researchers have used constraints and/or rules along with some form of optimization to arrange different elements on a page while satisfying some notion of "goodness" [Graf et al. 1996; Kröner 1999; Kröner et al. 2002]. Our template representa-

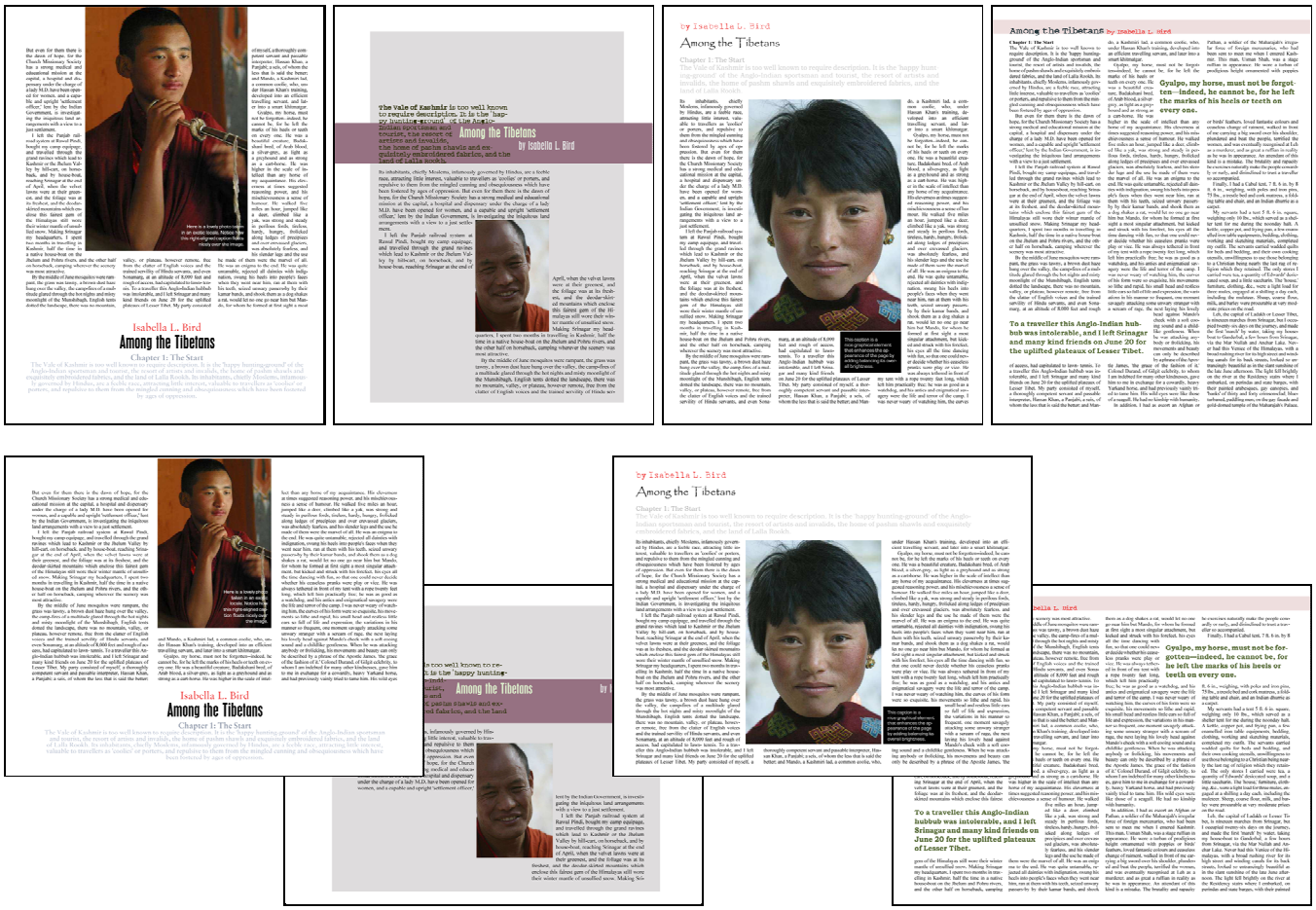

Figure 3 Examples of adaptive grid-based designs at two different page sizes. *Our system accommodates a wide range of modern page-design elements and ideas, including arrangements of text in unusual configurations, such as (from left to right) text that starts beneath its continuation and/or text that flows from regions with one style of formatting to another; text and captions that flow over colored backdrops or other images; design elements such as captions that carve out text regions; and pull quotes. Also, notice how each example can automatically adapt to portrait and landscape page orientations by changing the configuration of layout elements.*

tion distinguishes our approach from most of this previous work, enabling us to perform optimal pagination efficiently, while still supporting a large class of grid-based designs. Furthermore, templates allow designers to author layout styles in an intuitive way via our graphical authoring tool; they give designers a type of direct and precise graphical control that is more difficult to attain using just rules and constraints. The problem of solving graphical constraints has been an active area of research for years, including investigation of iterative and direct numeric solvers [Sutherland 1963; Heydon and Nelson 1994, Borning et al. 2000; Badros et al. 2001], differential methods [Gleicher 1991], and discrete/continuous optimizations [Harada et al. 1995]. However, we have found that resolving constraints using simple local propagation (LP) [Sutherland 1963; Van Wyk 1981] has been sufficient for our purposes.

Before constraints can be resolved, however, they must be specified. While early systems allowed users to explicitly apply constraints in a graphical way [Sutherland 1963], more recent work has explored inferring constraints from layout snapshots [Kurlander and Feiner 1993] or user interaction [Karsenty et al. 1992]. Badros et al. [2000] allowed users to graphically apply and build up composite relationships in their constraint-based window manager. In our system, we focus on improving the user interaction model specifically for specifying the kinds of constraints needed for document layout.

Most document layout systems in use today rely either on user interaction to control pagination (e.g., Microsoft Word, Adobe PageMaker, and QuarkXPress) or some kind of "first fit" algorithm [Knuth 1986]. First-fit algorithms typically fill pages entirely, placing each figure in the document on the first page on which it fits following its first reference. Such paginations are sometimes adequate, but often they require substantial hand-tuning of the document content. Our algorithm begins with a first-fit solution, and then automatically tries to provide a globally optimal pagination if the first fit fails.

Much of the previous work on pagination has focused on specialized problem domains. In the mid-1990s, two groups independently proposed solutions for the "Yellow Pages" pagination problem, where block advertisements and textual phone listings are laid out on a sequence of pages subject to various constraints on their placement, using constraint satisfaction [Graf et al. 1996] and simulated annealing [Johari et al. 1997]. Though these systems worked well on that one specific problem, it is unclear how well they might work on the more general and sophisticated types of page layout that our system handles. More recently, Purvis tried using a genetic algorithm for custom document layout [2002], but her approach appears to use entire blocks of text as atomic units rather than considering individual lines, words, and hyphenation units, as our system does.

For more generalized pagination solutions, researchers have based their algorithms on dynamic programming. Plass [1981] outlined one such algorithm based upon his earlier work with Knuth on line-breaking algorithms [Knuth and Plass 1981]. Brüggeman-Klein, et al. [1998] based their approach on Plass's work; however, they assumed a simplified document model wherein all text lines and figures had the same, constant width. Our approach is also based on Plass, but we generalize this approach to allow a more expressive document model, one that gives the paginator flexibility to choose among various page templates, multiple versions of content (such as figure croppings), and optional content. We also describe several modifications and optimizations to the basic algorithm that turn out to accelerate the pagination by several orders of magnitude—optimizations that are essential in our more fine-grained version of the problem.

3 Representation

We maintain a clean separation between document content and presentation style in our system, and we further divide the specification of style into layout templates and style sheets. We describe our representation of each of these in turn.

3.1 Document content

Document content in our system is represented as a set of individual streams, each of which contains content that must be laid out sequentially. Streams are differentiated by media type and purpose—a magazine article might have five streams, for example, including "body text," "photos," "sidebar text," "pull quotes," and "photo credits"—and the content within each stream is decorated with special markup to indicate structure. Streams can also be nested hierarchically using the <atom> tag, which serves to group a collection of streams together as a *content atom* inside a parent stream. For example, an atom may group a title, a figure, a figure caption, some descriptive text, and a footer, all of which belong together in a sidebar, within a parent "sidebar" stream.

In addition to our standard markup, we also allow content elements to be annotated with custom attributes that alter the way an element is treated by the layout engine and templates. For example, an image could be marked with a custom "importance" attribute, and specially authored templates could check the value of this property to decide how large to make the image in the final layout.

Our document format also allows us to encode multiple versions of any piece of content. To accomplish this, each of the different versions of a piece of content are packaged inside a <multi> tag, and the layout system chooses one of these versions to use when formatting the page. Each of these different versions may be tagged with custom attributes, which allow the layout engine to choose the version of content most appropriate for a given template—for instance, the "summary" version of text for formatting a page on a small device. If there are no custom attributes, then the paginator and layout engine are free to choose the version that works best for formatting the document and/or individual page.

3.2 Templates

Document layout is described in our system using *page templates*. Each template is responsible for defining the layout for a single page containing a particular set of content (for instance, "a page with two figures, one sidebar, and some body text") across a range of page dimensions. Together, a collection of templates constitute a particular *layout style*. Our system supports a wide range of modern page-design ideas, as shown in Figure 3.

Each individual template is composed of *elements*, *constraints* defining relationships between elements, and *preconditions* that characterize the suitability of the template for the particular content of the page.

3.2.1 Elements

Elements represent rectangular areas of the page in which content can be placed. To specify which content to use, each element specifies a source stream from which its content is drawn. If multiple elements consume content from the same stream, then a *flow* is established, and the stream's content flows from one element to the next.

Grid-based page designs often have overlapping elements, or regions that appear to be cut out of other elements. For example, the text on this page flows around the figure in the center. We achieve this effect by allowing each element to specify its place in an element *z*-order. Elements that are higher in the *z*-order sit atop lower elements, and the area of the higher elements is subtracted from those of the elements underneath.

Elements themselves can also specify a layout template (or a collection of templates) that can be used to lay out content atoms. In this way, our layout infrastructure is fully recursive and can support everything from the very simple figure/caption combinations, as seen throughout the paper, to more complex recursive embeddings, as in Figure 1.

3.2.2 Constraints

The size and placement of each element in a template is determined by the evaluation of a set of interdependent constraints that form a directed acyclic graph. Consider, for example, a simple page template composed of a title element above a body text element. In the template, the title would be constrained to begin at the top of the page and to be as tall as required in order to fit the title text. The body element would be constrained to begin at the bottom of the title element, and to end at the bottom of the page.

In our implementation, the constraint system comprises a pool of *constraint*

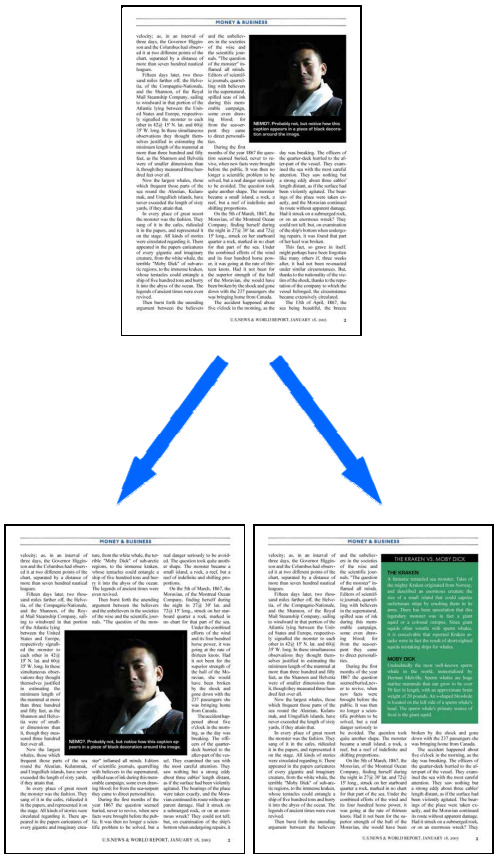

Figure 4 Automatic selection of templates. *The layout engine can adapt the layout automatically according to the content of the document. The topmost frame shows the second page of the layout on a TabletPC in portrait mode, from Figure 2. If the image on that page is tagged as "important," the layout engine uses the template for "important" figures, which enlarges the figure and centers it on the page. If the content formatted with the page includes a sidebar instead of a figure, then a different subtemplate is chosen, one that contains the stylistic information required for displaying a sidebar instead.*

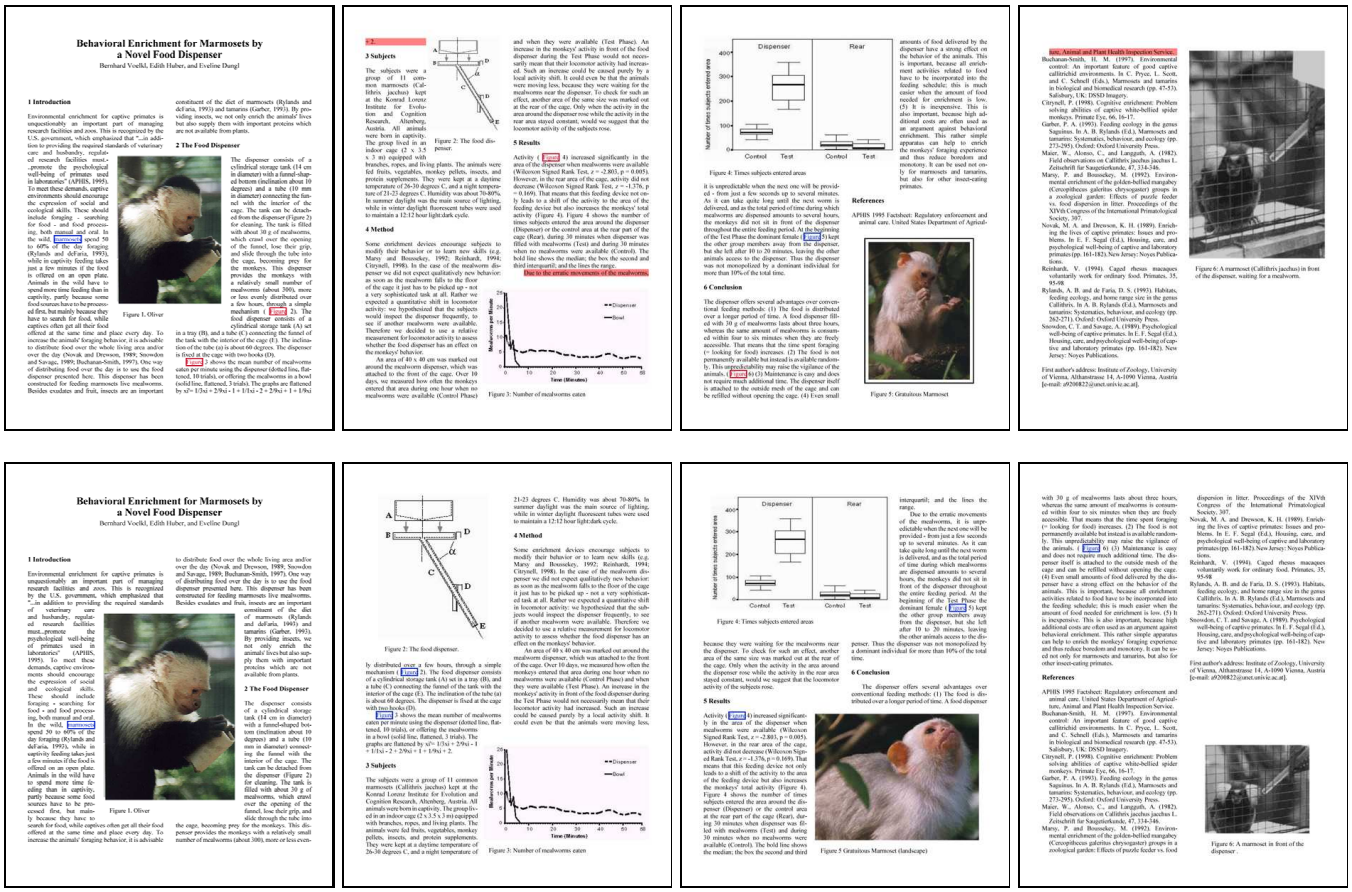

Figure 5 Optimal pagination. *The first row shows a "first fit" pagination of a four-page scientific journal article (using a "SIGGRAPH" formatting style for a TabletPC-size display in portrait mode). Figure references whose referent is laid out on the same page are rendered in blue, while those whose referents were placed on some other page are rendered in red. Similarly, widowed and orphaned text lines are highlighted in red. Note how, by selecting from among a small variety of templates and alternate versions of images, the paginator was able to achieve an "optimal" layout, with every figure on the same page as its reference and no widows or orphans. This optimal pagination, shown in the second row, was computed in 5 seconds on a Pentium IV processor.*

variables, whose values are determined by a mathematical expression in terms of the other constraint variables. This configuration is known as a one-way constraint system.

A template's constraint variables can be divided into two types: *input* and *output*. The input variables are used to tell the template about the context in which it is being used. For example, the page dimensions are related to the template by the use of input variables, *page.width* and *page.height*. Also, if content is tagged with custom attributes, these values are reflected as additional variables in the constraint system. The most important output variables in our system are the variables that indicate the bounding rectangle for each element. An element's size is set by the variables, *element.width* and *element.height*. Another important output variable is called *template.score*, which allows a template to express its fitness in terms of the inputs.

3.2.3 Preconditions

In order for the system to be able to choose from among the multitude of templates constituting a given layout style, each template uses *preconditions* to express when it is valid. *Content preconditions* indicate the amount of content from a given stream that must be present, and *value preconditions* indicate the range of values within which a given constraint variable must lie. With this mechanism we can specify, for example, that a particular template is valid only if there are exactly two figures available to put on the page, and only if the page dimensions fall somewhere between an A4 and U.S.-letter size page. In fact, we rely quite heavily on value preconditions to determine the range of page sizes for which each template is valid, and to help determine which set of templates is used for laying out a document's first page, which often requires special treatment.

3.3 Style sheets

In order to allow document authors to apply specific styling to text, such as **bold** or SMALL CAPS, we provide a simple stylesheet language that is similar in spirit to CSS [Lie and Bos 1996]. (We may eventually adopt CSS or XSL-FO [Adler 2001] as the stylesheet language, but our simple language has been sufficient for our purposes.) Runs of text in the document can be tagged with a style identifier, and these identifiers can in turn be associated with formatting rules in a separate stylesheet file. The section headings, figure references, italics, citations, and other parts of this paper were styled in this way.

3.4 Bringing it all together

Together, document content, templates, and stylesheets comprise the raw data on which our system operates. These data are fed to the paginator, which in turn relies on the layout engine to render the actual page layouts. In the next two sections we take a more in-depth look at how the layout engine and paginator do their jobs.

4 Layout

The layout engine is responsible for combining a page-full of content, which it receives from the paginator, together with the templates and stylesheets that define the document's layout style, to produce a collection of potential page layouts.

The first step in this process is to determine the appropriate set of candidate templates to try, from among all the templates in the collection. The layout engine does this by evaluating all of the templates' preconditions against the content at hand (see Figure 4). Often, more than one template is valid for the given content, in which case the content is laid out according to each of these candidate templates.

For each valid template, the layout engine determines the size and position of each element by setting the template's input variables and then propagating these values forward through the template's constraint graph using simple greedy local propagation. Once this is done, the layout engine computes the 2-dimensional regions of the page into which content is to be flowed, shaving the regions down according to the various elements' overlap and *z*-ordering. Finally, content is flowed into each of the determined regions.

4.1 Flowing into elements

Our prototype layout engine uses simple rules for flowing content into regions. For images, it simply scales the image to fit the appropriate element's bounding rectangle, and then displays the image, cropped by the element's content region. For text elements, it flows the text into the element's region using Knuth and Plass's optimal line-breaking algorithm [Knuth and Plass 1981]. For inline figures—figures that occur within a text flow—the layout engine places each figure at the position at which it is referenced in the text, and stretches the figure to fill the whole column. If there is no room left in the current element, the figure is displayed in the next element in the flow.

4.2 Self-sizing elements

The layout engine also supports elements that automatically adjust their height to fit their content. For an element that can contain a single image, the layout engine sets constraint variables on the element to inform it of the pixel dimensions of the image being placed inside. The pixel dimensions are then used to compute the image's aspect ratio, and this is multiplied by the element's width in the template to come up with the element's height.

For a text element, if it is marked with a special "resize-to-content" tag, the layout engine first sets the element's height variable to the maximum allowable value. Then, if there is not enough text to fill the element entirely, the element's height is reset to be the actual height of the text.

For elements that contain compound content and require templates for layout, the situation is only slightly more complex. In this case, the template employed to lay out the element's content exposes a special output variable called *template.outheight*, which, after the element has been laid-out, can be used to set the final height of the element within its template.

4.3 Template scoring

For each template used to lay out a given page of content, the layout engine calculates a score based on how well the content fits the template (by evaluating the *template.score* variable) and how many widows and orphans there are in the page layout. Once the layout engine has calculated scores for all of the candidate templates, it reports these back to the paginator, which uses them—along with template scores for previous and subsequent pages—to calculate an optimal sequence of templates to use for paginating a whole document.

5 Pagination

The pagination task is the mapping of content from the document streams to a set of discrete pages, subject to various constraints such as the sequential ordering of words in the text stream, the finite capacity of the pages, and dependencies between content streams (e.g., textual references to figures and tables). When only text is present in a document, a superior pagination can avoid widowed and orphaned lines by slightly under- or over-filling pages to

Figure 6 Automatic page filling. *This figure, along with Figure 7, demonstrates how multiple template choices and optional content can be used by the paginator to completely fill pages in multiple formats. The first row shows the content laid out in an adaptive "New Yorker" style using first fit. The blank space at the end of the last page is something that would never appear in the printed magazine. In the second row, the layout has been optimized to fill the last page, through a combination of "underfilling" the text (i.e., increasing the bottom margin of the page) and the inclusion of "bubbles" (optional image content) to inflate the text area.*

Figure 7 Automatic page filling, continued. *The same content and style as in Figure 6, adapted to a different-sized display. Once again, the top row shows the unoptimized "first fit" layout, whereas the bottom row shows the result that our paginator can achieve using multiple template choices and optional content.*

move the page breaks to more desirable places. When one or more additional types of content, such as figures or tables, are present there are many more choices that can be made to affect the pagination, and finding a desirable pagination is non-trivial.

In order to find an optimal pagination, a measure of "goodness" must be defined and then maximized by systematic or heuristic search, or by constraint optimization. One simple measure of "goodness" is the "total page turns" metric, which counts the total number of page turns that would be required to both read through the text, and also turn to any additional content that is referenced by the text from the place where it is referenced.

We typically use a metric that combines total number of page turns with other measures that reflect the quality of the appearance of the page. In general, the more flexibility available to an algorithm—i.e., the more choices that can be made in laying out the content—the higher the "goodness" of the optimal solution is likely to be.

5.1 Original algorithm

Pagination is solvable by dynamic programming because it has the "optimal subproblem" property, where any optimal solution of n pages contains an optimal solution of the first $n-1$ pages. A dynamic programming paginator will start with the empty solution and incrementally solve larger and larger subproblems, while keeping a table of each subproblem's score, and a pointer back to the preceding subproblem in the optimal solution. The basic algorithm evaluates a new subproblem S by scanning the table for the preceding subproblem S' with the best score that can legally precede the last page in S, and stores the score for the new last page in the table (if there is a valid predecessor). Since the number of subproblems that can precede the last page is bounded by the page size, there is just a constant number of predecessors that must be examined for each subproblem. When the table is completed, the entry in the last cell contains an optimal pagination (if one exists), and the pointers can be followed back to recover the optimal solution.

Plass [1981] was not concerned with the specifics of validating a page, but commented that the operation had to be very low-cost. Brüggeman-Klein et al. [1998] kept this cost low by restricting the document model. Our model, on the other hand, requires an incredibly complex inner loop in which we must both solve a constraint system to lay out each template and also line-break the text to measure the template's capacity. Therefore, we had to rethink the basic dynamic programming algorithm to evaluate far fewer of these very expensive operations.

5.2 Our algorithm

The basic algorithm just described calls an "evaluate page" function on each of the possible predecessors of each subproblem—of which there can be thousands, the vast majority of which are not even valid predecessors. Our approach was to restructure the algorithm so that the evaluation is performed only for valid pages. Since there is no way to know which pages are valid looking backwards from an endpoint, we start instead with each completed subproblem (again, beginning with the empty problem) and calculate all possible ending points for the following page. A reference back to the originating subproblem is placed in the table for each ending point, along with the score for that partial solution. If an entry for the ending point already exists in the table, it is replaced if the new entry has a better score. The revised algorithm ensures that when a new subproblem is considered, all subproblems that could possibly precede it in a solution will have already been solved, and the entry pointing back to the optimal predecessor will be in the table. If no entry exists for a subproblem when it is reached, then the subproblem can be passed over with no computation. The new algorithm is still linear in the product of the sizes of the content streams, but many of these operations now require no substantial work.

Performance is further improved by pruning partial solutions from the table whose score is worse than some threshold. This pruning is helpful because there are relatively few "good" solutions in the valid solution space. A *conservative* pruning strategy uses the score from a cheap "first fit" solution as a pruning threshold, which typically provides a large speedup and guarantees that a solution will always be found. Another strategy is the *optimistic* pruning

strategy, which chooses an approximate, near-perfect threshold of "goodness" and iteratively reduces this threshold if no solution is found. When lots of flexibility is available to the paginator (typically because a rich choice of templates or content options is present), the likelihood of a near-perfect solution existing becomes high, and this "optimistic" approach is preferred.

To implement this algorithm the layout engine must provide a *FindValidEndingPoints*() call that enumerates the places the next page can end, given the starting point and the number of figures to be placed on the page. The *FindValidEndingPoints*() call is responsible for enumerating the valid template choices and returning a vector of valid ending points together with the template used to arrive at each ending point and a quality score for the resulting layout. The quality can reflect the template's output score, the presence of widows and orphans, and any other measures of layout quality, such as penalties for under-filled pages.

Like the original algorithm, our improved algorithm can easily be expanded for additional content streams by adding extra dimensions to the table and additional nested loops to the algorithm. Optional content streams can be handled with no additional programming by having templates available that display content from optional streams. If these templates are included in the "style" and optional content is available in the document, the paginator will include them in the solution whenever they improve the optimal pagination. We have found that having a small number of templates with optional content can vastly improve pagination quality (see Figures 5–7).

5.3 Analysis

To test our pagination algorithm, we created a set of synthetic test documents using a statistical distribution of words. Figures were chosen for each document from a pool of samples with varying aspect ratios and at a variety of densities, and references to these figures were randomly distributed over the document. These documents were then paginated by a greedy first-fit algorithm to establish a base pagination quality, then optimal solutions were generated using our paginator. As expected, overall, our algorithm produced consistently superior paginations using the "total page turns" metric.

We first tested our algorithm using a simplified layout engine that employed greedy line-breaking and a small collection of templates. A set of 50 test documents containing between 5,000 and 20,000 words and between 10 and 45 figures were tested. These documents were found to have an average greedy pagination quality of 3.2 "penalty page turns," that is, page turns beyond the ones required to simply page through the text. This count was reduced to an average of 1.4 page turns by the optimal algorithm when constrained to nearly full pages. With more flexibility available (equivalent to a 90% minimum fill) this count was further reduced to an average below 0.1 page turns, although occasionally adding an extra page to the document. A set of large documents with unusually high figure density had an average penalty of 6.0 reduced to 1.7 with 100% fill. Using the new algorithm described here yielded running times of under three seconds on a Pentium IV. These timings represent a speed-up of over 5,000 from the running time of the standard dynamic programming algorithm.

Paginations generated with the full system are highly dependent upon the selection of templates associated with the documents. We tested the fully-featured system with two sets of templates: our standard "SIGGRAPH" style used in Figure 5 and a "rich SIGGRAPH" style augmented with ten additional templates offering more layout options. We tested documents of two sizes, 5,000 and 10,000 words, and at two densities, 1.2 and 2.5 figures per thousand words. Overall, the documents with the basic template set averaged 4.4 penalty turns per document with greedy pagination, which improved to 1.6 page turns with our algorithm. With the richer template set, the documents averaged 2.7 penalty turns with the greedy solution and 0.45 page turns with the optimal. We found that widowed and orphaned lines occured about 1.2 times per thousand words in the greedy paginations and were reduced to about 0.4 times per thousand words in optimal solutions. For timing results, we used the optimistic pruning strategy, since it dramatically outperformed the conservative strategy most of the time. (However, this approach slows down considerably when no good optimal solution exists.) Using the optimistic pruning strategy, the full layout system solved the small (5,000 word) test files in an average of 21 seconds with 6 figures and 43 seconds with 12 figures. The large files (10,000 words) averaged 94 seconds with 12 figures and 472 seconds with 25 figures. Our experimental prototype has not been optimized for speed, and we expect a tuned system to perform much faster.

6 Authoring templates

Although constraints represent a powerful means for specifying flexible layouts, this increased expressiveness does place more responsibility on the template designer, who must now consider how a page should look over a range of different aspect ratios and sizes, rather than at just a single static size. To make this potentially difficult task more manageable, we have developed a constraint-based graphical authoring tool that allows the user to draw and arrange layout elements, specify how they adapt to different page sizes, and preview this adaptation interactively. In addition, our system also offers some support for authoring template preconditions and adding style attributes to elements.

6.1 Creating and arranging layout elements

The user interface for our system presents a schematic representation of a page template that can be interactively resized. (See Figure 8 and the accompanying video or CD-ROM to get a sense of the interface and style of interaction.) To create a new layout element, the designer simply draws a rectangular region on the screen and then (using standard "drawing tool" operations) manipulates it to its desired size and position. Since templates are adaptive, most elements will need to be constrained relative to the page and/or other elements in order to maintain the integrity of the layout at different form factors. For example, a designer might want to specify that columns of text remain evenly distributed across the horizontal extent of the page, or that a caption remain below a figure and aligned with its right edge.

To specify page-level constraints, the user defines a page grid by drawing horizontal or vertical *guides* and then, using a snap-dragging interface [Bier et al. 1986], constrains elements relative to this grid. The user can create guides that either scale relative to the page (e.g., a horizontal "hang line" for figures that always remains one-third of the way down the page), or maintain a constant offset (e.g., a vertical guide that defines a constant margin on the left side of the page). Guides can also be specified relative to other guides, allowing the user to define a hierarchical page grid. To constrain one element relative to another, the user again uses a snap-dragging interaction to align the elements as desired.

Not all elements can be fully constrained using the page grid and other elements alone. As mentioned previously, templates often include layout elements that size themselves based on content rather than on the dimensions of the page. For example, figure elements typically must respect the native aspect ratio of an image, and caption elements usually expand to fit the available text. To handle these situations, the user can constrain one of an element's dimensions (usually the width) as described above, and then speci-

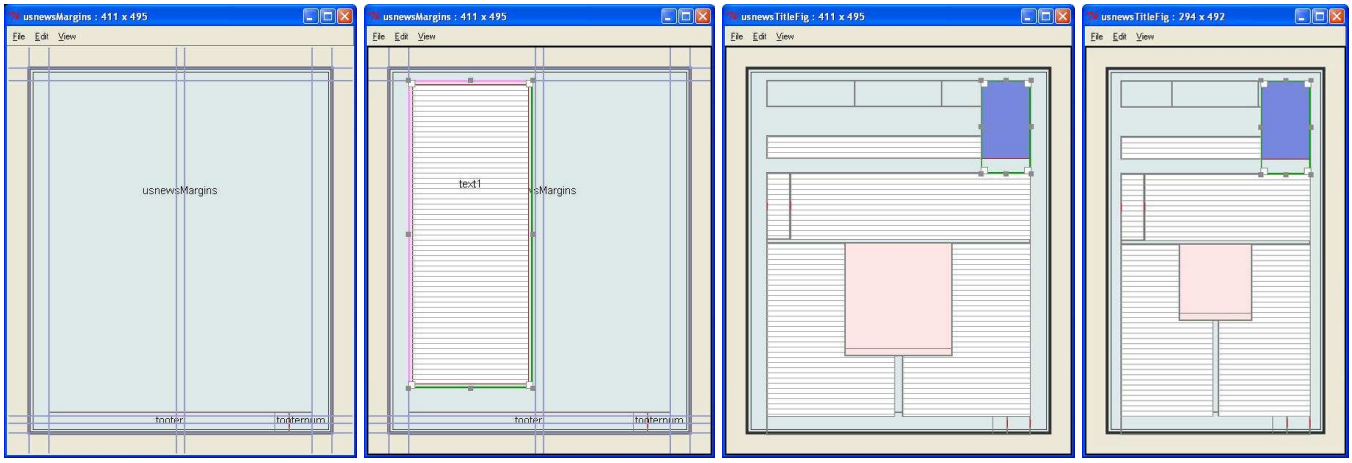

Figure 8 Template authoring. *The authoring interface allows the user to construct a grid over the page using a set of guides. Here are some snapshots from the process of creating a template for the "U.S. News & World Report" style. From left to right: a simple two-column grid is loaded from an existing template; a text element is snapped to the grid, the pink lines indicating aligned features that will be constrained to abut if the mouse button is released with the element in this position; the completed template with guide lines turned off; and the resulting constraint-based template adapting to different display sizes as the window is resized. (Please see the video or CD-ROM for a real-time depiction of this figure.)*

fy that the other dimension be computed based on content. Another common type of content-specific constraint involves alignment based on the implicit grid defined by text. Our system allows the user to align the top or bottom of one element to the precise position of the top, bottom, or center of a set of lines of text within another.

The system keeps track of the specified relationships by maintaining a multi-way constraint graph. When elements are manipulated or the page is resized, we use local propagation (LP) to update the constraint system, and then we re-render the page to reflect the changes. We chose LP as our constraint resolution technique because it is simple and supports arbitrary constraint functions. The primary drawback of LP, however, is its inability to correctly resolve simultaneous constraints. A more sophisticated constraint solving engine could also be incorporated into our tool should the need arise.

6.2 Template selection

As mentioned in Section 3, how suitable a template is for a particular page size and selection of content is defined using preconditions and a scoring function. Based on the content sources associated with each element, the content preconditions for a template can be automatically computed. Using a simple dialog box, the user can specify additional preconditions based on the value of any variable in the constraint system. For instance, there is typically a range of page dimensions for which a given template is suitable; to specify this common precondition, the tool allows the user to take snapshots of the template at its smallest and largest acceptable sizes.

Often, certain templates are optimized for content with specific attributes. For example, magazines usually have special layouts designed for particularly important figures that are different than the ones used for regular figures. Using our tool, a designer can add attribute preferences to elements that influence the score that the page template receives for a given selection of content. Specifically, these preferences are incorporated into the expression for computing *template.score*. When more than one attribute preference is specified for an element, the user is asked to rank them in order of importance. Although we could easily allow the user to specify actual numeric weights for each of the attributes, we have not found this to be necessary, and we have been wary of exposing too many controls unnecessarily in a tool of this kind.

7 Results

All of the figures in this paper—as well as the paper, itself—were generated using our adaptive grid-based document layout system. Other examples are shown on the accompanying video or CD-ROM.

Currently, we have no special authoring system for creating content in our markup format. Instead, we have used Microsoft Word to author the documents and converted them to our markup using a macro.

8 Conclusion and future work

Computer display hardware is rapidly approaching the point at which—at least from a technical standpoint—the on-screen reading experience will rival the printed page. Moreover, there is every reason to believe that, once documents can look as good on the screen as they do in print, the on-screen reading experience will even surpass the experience of reading on paper, as computers provide all sorts of opportunities for customization of style and content—not to mention additional capabilities, like animation and interactivity—that should make the on-screen experience superior in many ways.

Yet, oddly, one of the biggest impediments to achieving this decades-old vision of a paperless world may turn out to be a deceptively simple two-dimensional computer graphics and user interface problem: How can you define a type of grid-based design that adapts elegantly and seamlessly to any viewing conditions?

In this paper, we take some significant first steps toward addressing this timely question. However, our prototype system is not, by any means, the final word on adaptive grid-based page layout. In fact, we feel this work opens up a lot of new and interesting directions, which we hope others will join us in exploring. These include:

- What kind of editor could be built for creating and editing content, as well as adaptive styles? How do you allow the user to move seamlessly between the two modes?
- How do you allow the interactive editing of multiple versions of content? And how do you keep track of all the different versions? And what about interactions among the various choices? (See also [Anderson 2002].)
- How do you create a higher-level editor for entire layout "styles," defined as collections of templates?

- How do you design styles for tables, defined as arbitrary two-dimensional arrays of elements? (See also [Anderson and Sobti 1999; Wang 1996].)
- How do you design a style that can accommodate hand-drawn annotations that the user might add?
- How do you add more continuous forms of optimization to the paginator and layout engine in order to achieve more nuanced effects, such as, for example, scaling an image by a tiny fraction in order to achieve a better page break later on?
- How do you incorporate additional metrics into the layout engine, such as achieving some kind of "compositional balance"?
- Is it possible to infer a set of templates, or an entire style, automatically from a scanned set of documents?

9 Acknowledgements

The authors wish to thank the following people for helpful discussions at various points in this project: Maneesh Agrawala, P. Anandan, Richard Anderson, Alan Borning, Michael Cooper, Michael Duggan, Sam Epstein, Bill Hill, Nathan Hurst, Joe King, Eliyezer Kohen, Kim Marriott, Marc McDonald, Tomer Moscovich, Raman Narayanan, Radoslav Nickolov, Patrice Simard, and Geraldine Wade. We would also like to thank the SIGGRAPH anonymous reviewers for their comments and suggestions.

References

Adler, S. 2001. Extensible stylesheet language (XSL) version 1.0. W3C recommendation. http://www.w3.org/TR/xsl/.

Anderson, R.J. 2002. The power of choice: Content selection in page layout. Technical report, University of Washington.

Anderson, R.J., and Sobti, S. 1999. The table layout problem. In *Proceedings of the 15th ACM Symposium on Computational Geometry*, 115–123.

Badros, G.J., Borning, A., Marriott, K., and Stuckey, P. 1999. Constraint cascading style sheets for the web. *Proceedings of UIST '99*, 73–82.

Badros, G.J., Borning, A., and Stuckey, P.J. 2001. The Cassowary linear arithmetic constraint solving algorithm. In *Computer-Human Interaction 8 (4)*, 267–306.

Badros, G.J., Nichols, J., and Borning, A. 2000. Scwm—An intelligent constraint-enabled window manager. In *Proceedings of SmartGraphics '00*.

Bier, E.A., Stone, M.C. 1986. Snap-dragging. In *Proceedings of SIGGRAPH '86*, 233–240.

Borning, A., Lin, R., and Marriott, K. 2000. Constraint-based document layout for the web. In *Multimedia Systems 8.3*, 177–189.

Brüggeman-Klein, A., Klein, R., and Wohlfeil, S. 1998. On the pagination of complex documents. Technical report, Fernuniversität Hagen [University of Hagen].

Feiner, S. 1988. A grid-based approach to automating display layout. In *Proceedings of Graphics Interface '88*, 192–197.

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

Gleicher, M. and Witkin, A. 1991. Differential manipulation. In *Proceedings of Graphics Interface '91*, 61–67.

Graf, W. H. 1992. Constraint-based graphical layout of multimodal presentations. In *Proceedings of AVI '92*, 356–387.

Graf, W.H., Neurohr, S., Goebel, R. 1996. YPPS—A constraint-based tool for the pagination of yellow-page directories. In *Proceedings of the KI-96 Workshop on Declarative Constraint Programming,* 87–97.

Harada, M., Witkin, A., and Baraff, D. 1995. Interactive physically-based manipulation of discrete/continuous models. In *Proceedings of SIGGRAPH '95*, 199–208.

Heydon, A., and Nelson, G. 1994. The Juno-2 constraint-based drawing editor. DEC SRC technical report 131a, Digital Systems Research Center.

Hurlburt, A. 1977. *Layout: The Design of the Printed Page.* Watson-Guptill Publications. New York.

Hurlburt, A. 1978. *The Grid.* Van Nostrand Reinhold Company. New York.

Johari, R., Marks, J., Partovi, A., and Shieber, S. 1997. Automatic yellow-pages pagination and layout. In *Journal of Heuristics 2 (4)*, 321–342.

Karsenty, S., Landay, J.A., and Weikart, C. 1992. Inferring graphical constraints with Rockit. In *Proceedings of HCI '92*, 137–153.

Knuth, D.E. 1986. *TEX: The Program*, Volume B of *Computing and Typesetting.* Addison Wesley. New York.

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

Kröner, A. 1999. The DesignComposer: Context-based automated layout for the internet. In *AAAI 1999 Fall Symposium Series: Using Layout for the Generation, Understanding or Retrieval of Documents.*

Kröner, A., Brandmeier, P., and Rist, T. 2002. Managing layout constraints in a platform for customized multimedia content packaging. In *Proceedings of AVI '02*, 89–93.

Kurlander, D., and Feiner, S. 1993. Inferring constraints from multiple snapshots. In *ACM Transactions on Graphics*, October, 227–304.

Lie, H.W., and Bos, B. 1996. Cascading style sheets, level 1. W3C recommendation. http://www.w3.org/Style/CSS/.

Lok, S., and Feiner, S. 2001. A Survey of automated layout techniques for information presentations. In *SmartGraphics '01*, 61–68.

Müller-Brockmann, J. 1981. *Grid Systems in Graphic Design.* Hastings House Publishers. New York.

Peels, A.J.H., Janssen, N.T.M., and Nawijn, W. 1985. Document architecture and text formatting. In *ACM Transactions on Information Systems*, 347–369.

Plass, M.F. 1981. Optimal pagination techniques for automatic typesetting systems, technical report STAN-CS-81-870, Department of Computer Science, Stanford University.

Purvis, L. 2002. A genetic approach to automated custom document assembly, In *Proceedings of ISDA '02*.

Sutherland, I.E. 1963. SketchPad: A man-machine graphical communication system. In *Proceedings of AFIPS 23*, 323–328.

Van Wyk, C.J. 1981. IDEAL user's manual. Bell Laboratories.

Wang, X. 1996. *Tabular Abstraction, Editing and Formatting.* PhD thesis, University of Waterloo.

Weitzman, L., and Wittenburg, K. 1993. Relational grammars for interactive design. In *Proceedings of the IEEE Workshop on Visual Languages*, 4–11.

Weitzman, L., and Wittenburg, K. 1996. Grammar-based articulation for multimedia document design. In *Multimedia Systems 4*, 99–111.