Multiple Representation Document Development*

Pehong Chen†
Michael A. Harrison

Computer Science Division
University of California
Berkeley, CA 94720

July 30, 1987

Abstract

The world of electronic publishing software seems to divide into two camps: the conventional batch-oriented programming language approach versus the more elaborate direct manipulation paradigm. In this paper we indicate which aspects of document preparation are more conveniently handled under which model and point out several instances of a hybrid approach which takes advantage of multiple representations. We introduce a framework for analyzing the structure of multiple representation systems in general. Based upon this simple but robust framework, a top-down design methodology is derived. The design of a fairly sophisticated document development environment is discussed as a case study of the methodology.

1 Introduction

With advances in laser printer technology and the proliferation of high-performance workstations featuring high resolution displays, pointing devices in addition to the regular keyboard, and windowing environments, recent years have witnessed an explosive growth in the development of electronic publishing software. An important aspect of this development has been the exploitation of user interfaces commonly referred to as the WYSIWYG (*what-you-see-is-what-you-get*) type in document development, or as *direct manipulation* [37,58] in a more general context.

A key characteristic of WYSIWYG systems is the departure from a source language specification of document semantics (appearance). The system usually supports a diverse collection

*Sponsored in part by the National Science Foundation under Grant MCS-8311787, and by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871, monitored by Space Naval Warfare Systems Command, under Contract No. N00039-84-C-0089.

†Additional support has been provided by an IBM Graduate Fellowship.

of objects such as text, graphics, tables, formulas, etc. Editing, formatting, and related facilities for manipulating these objects are integrated in such a way that many events happen naturally and automatically. Operators are often encapsulated in a palette or menu-driven user interface so that a desired task can be accomplished by a few simple keystrokes or mouse clicks. These systems are highly interactive; the result of invoking an operation is instantaneously observed, thereby creating an illusion that the user is "directly" manipulating the underlying object.

This approach differs substantially from the traditional *source language model* in which documents are specified with interspersed textual commands much the same as ordinary high-level language programs. By and large, a document is first prepared using a text editor, the formatting and other related processors are then executed, usually in batch-mode, and the result is obtained. Superficially, the differences between the two models seem to stem from the degree of interactiveness and the level of integration. The former is the canonical interpreter versus compiler issue, and there is nothing prohibiting a document composition language from being incremental or interpreted to gain better interactive behavior. Integration is not the dominating issue either; with proper editor support, it has been shown that an effective integrated environment based on batch processors can be created [26].

A major distinguishing factor, therefore, is the explicit user manipulation of the document target appearance versus the manipulation of a programmable source representation. Advocates of the direct manipulation model claim that WYSIWYG systems bridge the gap between the user's perception and the actual task domain. Such systems relieve the user from any concerns of detail and are very easy to use. On the other hand, critics argue that the friendly user interface comes at the expense of generality and flexibility, or in another word, "power". A full-fledged symbolic language obviously offers more "expressiveness" than the finite set of operators likely to be incorporated in a direct manipulation user interface. Naturally, one should ask, what is the right approach?

As many have recognized (e.g. [18]), we believe the right approach is that of a hybrid model which employs multiple representations. There are certain aspects of document preparation that are best suited to a source language representation while others are easier to deal with using direct manipulation techniques. We like to view the direct manipulation user interface as simply another specification language. Thus, by blending a variety of languages including that of direct manipulation (encapsulated as palettes, menus, etc.) in an integrated environment, the user interface becomes a matter of choice, dictated by convenience or preference.

The primary purpose of this paper is to establish a framework for the analysis and design of multiple representation document development environments. We indicate which aspects of document preparation are more conveniently handled under which model and discuss several approaches to the hybrid paradigm which exploits multiple representations. First we identify the domain of basic tasks involved in the entire course of document development and review the pros and cons of the two individual models with respect to each identified task. We then examine a somewhat orthogonal issue generally referred to as the "degree of procedurality" (or equivalently the "degree of descriptiveness") which represents a sliding scale between "how" and "what" to do from a user's perspective. All these issues are summed up

subsequently by our design methodology based upon an abstract structure which captures the multiple views of representations, transformations, and user interfaces of document preparation. Finally we describe the design of VORTEX (for Visually-ORiented TEX) [25], a multiple representation document development environment being built at Berkeley, as a case study of this methodology.

2 Task Domain

There are quite a few tasks that an effective document development environment must be able to accomplish. Understanding them is important because evaluating the two individual models relies on the underlying task being clearly identified. These tasks can be divided into two major categories: *writing* and *reading*. The following is a list of what tasks we consider essential: items 1 through 6 belong to the writing category, item 7 belongs to the reading category, and item 8 covers both. It is by no means an exhaustive list, however.

1. *Editing*. This involves the editing of text and graphics in general and various classes of special objects like tables, mathematical or chemical formulas, data-driven statistical charts, fonts, bitmap images, musical scores, animation scenes, digitized audio signals, and so forth. Many of these objects are intermixed. For instance, tables and formulas are two special cases of text, graphics may appear in the middle of running text, text is likely to be included in graphical illustrations, and so on.

2. *Formatting*. The primary issue in formatting is document appearance, or equivalently, the layout of specific pages. At a global perspective, certain types of documents must obey certain styles. Consequently, some default styles must be provided to cover a wide range of commonly used documents. On the other hand, it is also desirable to support customization so that uncommon styles can be defined by the user. At a finer granularity, either the system or the user must be able to control the placement of objects within a page. This may be as trivial as setting a piece of text in a certain font, or as complicated as floating text around an arbitrarily shaped object according to a specified flow.

3. *Preprocessing*. This refers to operations which must be performed prior to formatting. Typical examples include spelling checking, writing style verification, bibliographical citations, etc. It may also include graphics, table, mathematics, or any other processing filters not integrated with their main formatting engine.

4. *Postprocessing*. These are tasks which cannot be carried out until the main document body has been formatted. Things like cross references and indexes depend on certain object permutations (e.g. page, section, or figure numberings); they cannot take place unless such numbering has been resolved by the formatting process.

5. *Imaging*. Another important task is imaging the formatted result onto either the display or the printer. This normally involves interpreting the document's certain intermediate

representation (its internal data structure or output file format) and rendering the bits onto the workstation display or translating it into a specific printer language.

6. *Filing.* This task concerns the filing of documents. There are two important issues. The first has to do with how to effectively save the internal state so that future invocations can be carried out incrementally. The second issue focuses on information interchange and system dependence, namely whether or not the filed document can be transmitted across and processed on different machine architectures.

7. *Dynamic Reading.* From the reader's point of view, a hardcopy document generated by the print medium is fixed and static. Electronic media such as a workstation-based environment provide an alternative which does not have to be reminiscent of its static print counterpart. A good deal of "dynamics" can be exploited in an integrated document development system. For example, instead of thumbing through the pages for a reference as one would do when reading a printed document, in an integrated environment it is possible to display and examine the target of a reference in a separate window when the source of the reference is being read. This kind of context-sensitive browsing, along with a number of other features not available in the print medium, require complex system support and deserve further investigation.

8. *Annotations and Narrations.* Annotations and narrations can be embedded in a document to convey more information than what is available in the main document body. This additional information can be represented in the form of text, graphics, voice, etc. in an electronic environment. More than one author can be involved in creating such information and the reader can be granted appropriate permission of access. For instance, in an instructional environment, a different set of narrations can be presented according to the level of a particular student. In a publication process, an author can be working with a paper annotated with comments from different referees while annotations intended for the editor are hidden from him or her. Providing these features will involve security and general distributed systems issues.

3 Pros and Cons — A Comparison

Most of the tasks listed in the last section can be carried out using direct manipulation techniques or by some programming language source code. In some cases, one approach may be more appropriate than the other, while in other cases a combined approach may make more sense. An analysis based on each task in the domain is given in the remainder of this section.

3.1 Text Editing

Display-oriented editors can be regarded as direct manipulation systems when the underlying task is restricted to text editing. The popular screen editors *vi* [38] and GNU *Emacs* [60]

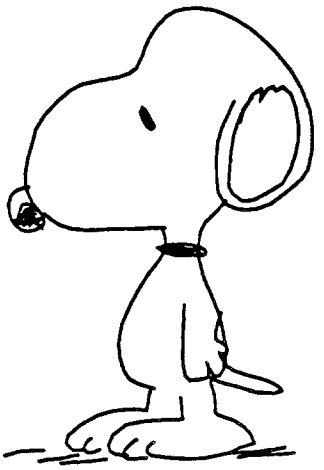


Figure 1: *Snoopy*. The picture of Snoopy created using a direct manipulation graphics editor.

are typical examples of this kind. They are superior to the old-fashioned line-oriented text editors because a full screenful of text can be directly manipulated.

Emacs also supports a source representation; a Lisp programming subsystem is embedded underneath direct manipulation. Each simple operation corresponds to a Lisp primitive upon which more complex operations can be coded, which can then be bound to user level commands in terms of a few keystrokes or mouse clicks. This makes *Emacs* customizable and extensible and thus a very powerful text editing tool [59].

3.2 Graphics Specification

The repertoire of techniques for specifying and generating graphics is very rich. Some of the techniques are language-based, others exploit direct manipulation user interfaces, while a few have employed a hybrid model. There are relative strengths and weaknesses for each approach, as described below.

3.2.1 WYSIWYG Graphics Editing

In general, it is easier to specify freehand drawings such as the Snoopy shown in Figure 1 using a WYSIWYG graphics editor like *MacPaint* [8] than directly programming it with a graphics language like *pic* [39] or *ideal* [64]. Also, it is more convenient to create technical drawings such as the one shown in Figure 2 with a direct manipulation editor like *MacDraw* [7] than with a noninteractive graphics programming language when visual feedback concerning operations such as object placement and orientation is essential.

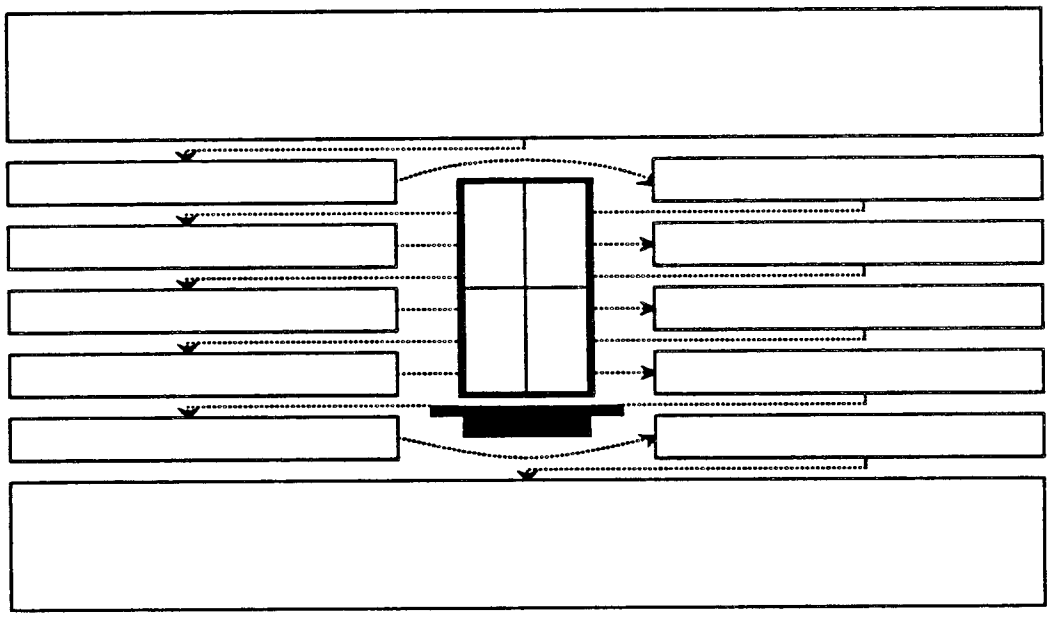


Figure 2: *Layout of Windowed Text.* A technical diagram created by a direct manipulation graphics editor. This diagram is intended for explaining an exotic page layout, or windowed text, in document formatting. The window in the center is a piece of graphics. The two large boxes on top and bottom contain a paragraph's *lintel* (leading text) and its *sill* (trailing text) which may each contain multiple lines. Each narrow box in the two sides holds a single line of text. Finally, the dotted arrows represent the flow of text.

MacPaint is a typical example of graphics editors specifically designed for creating artistic drawings. Once specified, the notion of objects disappears in these editors; all that's left is a raster image. Drawings of this kind are obviously confined by the device resolution with which they are created. By contrast, *MacDraw* represents another class of editors more suitable for creating technical diagrams. In *MacDraw*, objects and their structure are maintained throughout the editing session. When the drawing is done, it is the description of the objects and their structure, rather than the image, that gets saved. The advantage is that the image can be reproduced on a variety of devices with different resolutions.

A significant intermediate system is the Adobe *Illustrator* [2,27] which accepts a raster image, but allows the user to recover the underlying mathematical description by tracing the image. From that point on, the drawing can be manipulated in an object-oriented fashion, thereby making it possible for the user to fine tune images of any kind.

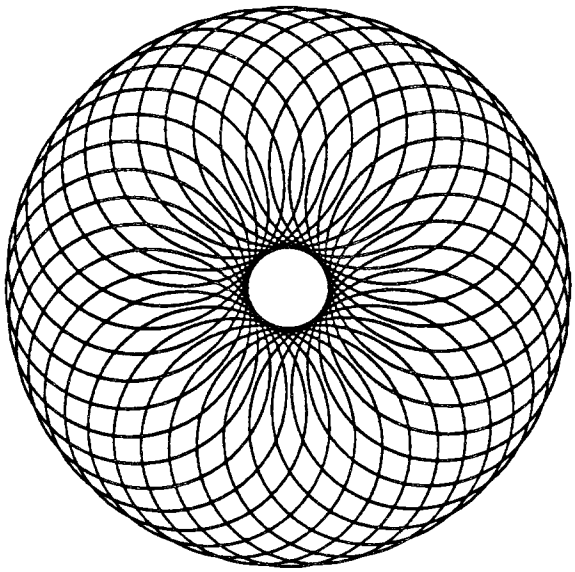
There is no question, however, that drawings created by any type of direct manipulation editors can also be described by graphics programming languages or some meaningful textual representations. In fact, drawings created with most of these editors will eventually be translated into a source language representation or some textual format for filing and document interchange purposes. The issue here is that in order to create such pictures efficiently, direct responses from the drawing apparatus in terms of its underlying objects' *placement* and *orientation* are crucial. Direct manipulation interfaces, in this respect, act as an interactive agent between the user and the task domain and are more effective than attempting to do the programming at the source level.

3.2.2 Graphics Programming

Direct manipulation editing breaks down when a great deal of *regularity*, a finer degree of *control*, any sort of *naming*, or other basic building blocks of programming languages are required. Figure 3 demonstrates the superiority of a graphics programming language in expressing something of high regularity. This is a circle repeated many times in rotation. The PostScript source code needed to describe it is a "one liner" shown at the bottom of the figure. One can imagine how cumbersome it would be to create this picture by direct manipulation.

Another good example which shows the advantage of working with programs is the *Pico* picture editor [36]. It is an interactive editor for digitized graphic images. Instead of directly manipulating the pixels involved as many bitmap editors do, *Pico* treats a raster image as a two-dimensional array of pixels. Typical operations such as changing contrast, masking or enhancing pixels, and merging, fading, or transforming the image are defined in terms of a C-style expression language. The user edits images by entering programs expressed in this language with references to the pixel array. This programming approach is able to produce startling effects on images very difficult to arrive at with WYSIWYG bitmap editors.

As yet another example, suppose we were to create a page which contains a Snoopy and recursively the same page nested within itself like Figure 4, no WYSIWYG graphics editor known to us will be able to realize that by any obvious means. With a programming language



```
36 { 60 0 45 0 360 arc stroke 10 rotate } repeat
```

Figure 3: *Rotated Circles.* A set of rotated circles and the corresponding PostScript code used to generate it. PostScript has a postfix syntax, so the one line code here means iterate a procedure 36 times. In each iteration first a circle centered at $(60, 0)$ with a radius of 45 is drawn (in points), then the coordinate system is rotated by 10 degrees.

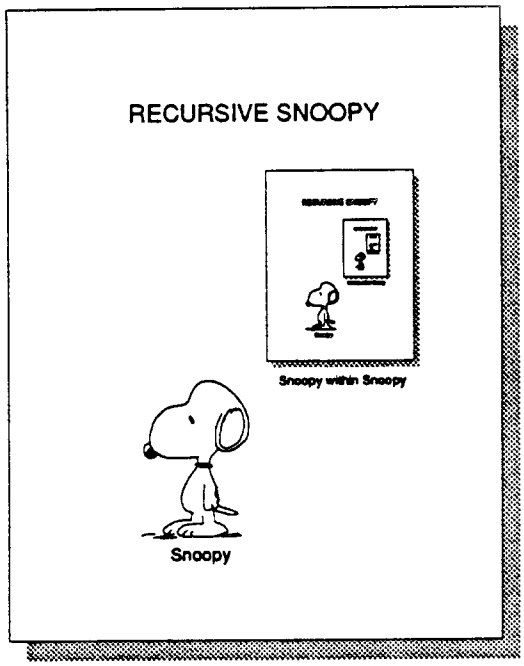


Figure 4: *Recursive Snoopy*. The picture of Snoopy recursively appears within itself. This is created by drawing Snoopy first with the graphics editor *Gremlin* [52], then passing its output through a *Gremlin*-to-PostScript translator (due to John Coker), and adjusting the resulting PostScript code. The idea is borrowed from an example given in [54].

like INTERPRESS [16], POSTSCRIPT [14], or DDL [4], however, such a page can be defined as a procedure which recursively invokes itself to a specified depth. We previously argued that it is easier to create pictures like Snoopy with a WYSIWYG editor and that generating a textual representation for the drawings is not difficult. The ideal approach here is to draw Snoopy by direct manipulation first, generate the corresponding code next, and finally perform some adjustments to realize the recursive invocations.

This is one flavor of the hybrid model which utilizes WYSIWYG as the frontend interface and a textual representation for off-line filing. This has the advantages of being more compact and device independent over filing the bitmap images. *CricketDraw* [3,50], for example, and a host of WYSIWYG graphics editors for the Macintosh can generate a filing representation either in PICT, Macintosh's standard graphics description format,[1] or in POSTSCRIPT. Some adjustments may be done on this textual representation before the drawing is filed or sent off to the printer for hardcopy.

3.2.3 Achieving Accuracy

An important issue arises in graphics specification when certain geometric properties of graphical objects must be satisfied or when their precise placement is required. In direct manipulation editors, the most naive solution to precise placement is to echo the current cursor coordinates on demand. A more commonly used technique is to provide the user with a rectangular grid. Sometimes a gravity facility which automatically attracts the cursor to fixed positions in the grid may be useful. Yet a more powerful paradigm based on the ruler and compass metaphor [17] also increases the desirable accuracy.

The other possibility is to apply a class of techniques known as the *constraint-based approach* which requires the user to specify a set of parameters that satisfies certain algebraic or logical equations. Given the constraints, the system tries to solve the equations simultaneously and returns the corresponding graphical objects. There may be more than one solution to the same set of constraints, hence a mechanism for selecting the desired solution must be provided.

One common criticism against this approach is that defining constraints is often counter-intuitive, which makes it difficult for inexperienced users to add new kinds of constraints to the system. This is actually due to the multiple representation issue inherent in this type of system; no matter what the user interface appears on the surface, there is an underlying source program that realizes the constraints. Switching back and forth from a highly-encapsulated graphical interface to a more primitive textual one is difficult for casual users. Some recent developments have focused on graphical specification of constraints with the goal of closing the gap between the task domains [19].

The constraint-based approach has been incorporated in interactive systems like *Sketchpad* [61], *ThingLab* [20], and *Juno* [49], as well as in textual programming languages like METAFONT [42] and *ideal* [64]. *Juno*, in particular, is interesting because in addition to a WYSIWYG type interface, the underlying constraint definition language is also made explicit

[1] in terms of *QuickDraw* commands, Macintosh's built-in graphics primitives.

to the user — both representations are editable and changes will propagate automatically — although the language capability is somewhat restricted from the programming language point of view.

3.2.4 The Hybrid Approach

Clearly the ideal approach is one that exploits the prompt visual feedback available in direct manipulation as well as the programming capability provided by the source language model. This is exemplified by the *Tweedle* graphics editor [15]. In *Tweedle*, the user is allowed to edit objects in the WYSIWYG manner; also supported is a text editor for editing the underlying procedural language description. Each object in the graphical representation corresponds to a piece of code in the textual representation. Changes made to either representation will be mapped to the other automatically.

This approach differs from the off-line hybrid model mentioned earlier in that the textual representation (program) is manipulated interactively. Any modification to the source program is immediately re-evaluated, which then updates the graphical representation, and vice versa. A great number of language design and user interface issues are involved in creating such a system. Typical problems include variable naming and binding, object sharing and linking, and most importantly, the internal state to be maintained in order to incrementally reevaluate the objects.

The programming side of the hybrid approach can be realized using a visual interface in which program constructs like variables, conditionals, procedures or macros, iterations, recursions, etc. are encapsulated as menu items in the standard WYSIWYG fashion. The user is still required to switch back and forth between editing programs (graphical rather than textual in this case) and editing the actual drawings (results of program evaluation). These graphical programs are equivalent to the textual ones in every respect. So a multiple representation system's emphasis is not so much on having something textual per se, but on explicitly maintaining one representation which is programmable. This is a very important point and will be reiterated later in Section 6.

3.3 Formatting and Layout

Traditional document development systems like the *troff* [53] family (with auxiliary processors *tbl* [46], *eqn* [41], etc.), TEX [43], *Scribe* [56], and *SGML* [32] are largely noninteractive language compilers. A document described in such a language has a textual source representation which contains its content as well as formatting commands. A target representation can be created by passing the source through the formatter. Normally, the task of editing (or simply browsing) either representation is separated from the task of formatting.

By contrast, in direct manipulation systems such as *Bravo* [45], *Star* [1], Interleaf Publishing System [6], or *FrameMaker* [5], formatting is an integral part of document editing. Here, the document is reformatted as it is edited. The distinction between source and target representations is vague or even nonexistent; no textual commands would be used to describe the formatting information.

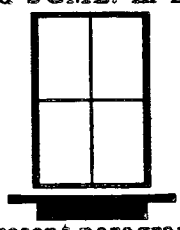
Each of these trains of development has important advantages and disadvantages. By and large, the output quality produced by language-based systems is higher than that by WYSIWYG editors. This is because most such compilers are batch-oriented, which means their formatting strategies can be better optimized. Direct manipulation systems are limited, in this respect, by certain performance requirements in response time. In order to achieve better quality, some WYSIWYG systems like *Cedar*'s *Tioga* [62,63], *Andrew*'s text editor [48], and *MSWord* [13] provide an option which performs some off-line formatting optimizations before the final hardcopy is generated. This formatted version can be previewed on the screen, but not edited. In other words, systems like *Tioga* are essentially WYSIWYG *galley* editors. These systems assure that "what you see is an approximation to what you get" when higher quality formatting is taken into account.

As mentioned earlier, an important advantage of the source language model is the "expressiveness" or "programmability" provided by symbolic languages. Suppose a document processing system is a set of operations defined on a collection of objects. With respect to simple operations, there may be little difference between the two models. The major difference becomes noticeable in cases where higher level abstractions such as macros and conditionals are desired. These are normally available as first-class citizens in a document formatting language. But in most WYSIWYG editors, manipulating complicated cases like these are either impossible or very cumbersome.

On the other hand, the most common criticisms against language-based systems center around (1) the unnecessary overhead they always pay in reprocessing the whole document with only few changes, and (2) the low degree of interaction and poor interface they provide to the user. It is in these areas that the direct manipulation model seems to prevail. As highly interactive systems, WYSIWYG editors are incremental in nature; they only perform the minimal work required to reformat a document and display the new image immediately. This immediate response is crucial to certain operations requiring visual feedback.

For instance, consider the task of laying out a page of windowed text. In a layout-driven WYSIWYG system like *PageMaker* [11], *Ready-Set-Go!* [12], or *FrameMaker* [5], one would do this simply by dragging the mouse, specifying the text blocks involved, and defining their links for the flow of text, as illustrated in Figure 2. The interface to the task domain is direct and straightforward.

Creating an exotic layout like this is quite difficult, if not impossible, in some rigid language-based systems like *Scribe* and *SGML*. In more flexible systems such as *troff* and TEX, however, such things are possible, but nontrivial. Defining macros which handle the general case of windowed text requires a very high degree of wizardry. In TEX, for instance, "output routines" can be redefined to layout pages differently, but the code required to produce general windowed text is quite involved. But once it has been figured out, generating a paragraph such as the present one is relatively straightforward. Based on the macros defined in [35], producing the present paragraph is reduced to calling a pair of window opening and closing macros at the beginning and end of the paragraph (see Figure 5).



The standard direct manipulation approach to this kind of irregular page layout and the way it is handled in TEX are somewhat different. In the WYSIWYG approach, the

```
\beginwindow\lintel 2\lines \side 2.5in \window 6\lines
   Creating an exotic page layout like this is quite difficult,
   if not impossible, in some rigid language-based systems like ...
   ......
\endwindow
```

Figure 5: *Producing Windowed Text in TEX*. This piece of text shows how one could create a windowed paragraph with a *lintel* (text above the window) of 2 lines tall, a 2.5-inch wide block at each side, and a window of 6 lines tall (whose width is the width of the paragraph minus 2 times the width of the side block). Naturally, the remaining text forms the *sill*. The principal macros involved are \beginwindow which takes these settings as parameters and \endwindow which outputs the formatted text. The actual code corresponding to these two macros is much more complex and is very difficult for a casual user to generate. The example here is based on the work of Alan Hoenig [35].

abstractions of text blocks and their links are general enough to cover a variety of situations. For example, creating a circular window would be based on the same interface used to create a rectangular one as shown in Figure 2, so would a layout whose text first fills up every line on the left side and then the right, instead of running across the window for each line. In TEX, producing a circular window requires modifying \beginwindow to accept a much more complicated parameter passing scheme. As for the other case, a new set of macros must be defined for that purpose specifically. The real issue, however, is a visual approximation to the ultimate page layout. It is clear that direct manipulation is more appealing in this respect.

3.4 Pre- and Post-Processing

There are a number of tasks that must be performed either before or after formatting which we collectively call pre- and post-processing facilities. A common nature of these facilities is that they each requires a stand-alone processor for its intended task. For instance, a spelling checker, a bibliography processor, and an index processor are needed for checking spelling, resolving citations, and permuting index entries, respectively. The traditional approach is, of course, language-based and batch-oriented: an intermediate file is generated as a derivative of the main document; this file is then passed to a designated processor, and the result is incorporated back into the main document. This approach applies not only to standard noninteractive document compilers like *troff* and TEX, but to a number of WYSIWYG environments as well. For example, index processing in *MSWord*, *FrameMaker*, and *Tioga* are all handled by a noninteractive off-line program.

Direct manipulation, from the processing point of view, requires too much overhead for a relatively minimal payoff. For instance, in compliance with the direct manipulation paradigm, an index entry, whenever entered into the document body, must immediately appear in the index section (with its page number) in alphabetical order with respect to other entries already

there. This requires extensive support in the internal data structure, but does contribute toward any significant improvement in generating the final index. The same criticism applies to the processing of similar objects such as bibliography, glossary, table of contents, cross references, etc.

In spite of the need for stand-alone processors, which seem inevitably batch-oriented, there are other aspects of these pre- and post-processing facilities that require more interactive support. These include, among others, correcting misspelled words in the manuscript, browsing the bibliography database to make citations, and placing index commands into the document body in a systematic fashion. All of them require a close integration with the document editor. If these objects are not maintained in the internal representation, a good manuscript level pattern matching mechanism (e.q. regular expression search, query replace, query insert, etc.) is imperative.

Notice that in a direct manipulation system, the tags for marking citations, cross references, and index entries cannot appear directly in the document under manipulation because their original forms may not correspond to any physical appearance. A common solution is to put them in "shadow pages" instead of the WYSIWYG representation. Thus, a shadow document is the original document plus these tags whose markers can be displayed upon request for editing purposes. As a result, users operating under this extended direct manipulation model are actually dealing with dual representations of the document, although the variation between the "source" and "target" is not as significant as that in a true language-based system.

3.5 Imaging and Filing

The on-line imaging mechanism of most direct manipulation systems is based on immediate interpretation of their internal representation. Their off-line filing representation is some textual description of the internal structure, but not necessarily in any real programming language. This textual description can in turn be passed to a device-specific printer driver for a hardcopy. Similarly, most language-based systems generate their output in some generic representation; device drivers are needed for either screen previewing or hardcopies. Typical examples include TEX's DVI format [28] and the *ditroff* format [40]. A common feature of this type of device independent "virtual machine" is that its imaging model is extremely simple-minded; its basic construct resembles low-level assembly code more than a high-level programming language.

Recently, a new breed of programming languages known as *page description languages* (PDLs) has emerged as the preferred representation for imaging as well as filing. There are currently three major players in the arena: INTERPRESS [16], POSTSCRIPT [14],[2] and DDL [4]. Advantages of PDLs in general include high-level program constructs, arbitrary transformations at the imaging level, uniform treatment of graphics and text (fonts), device independence, etc.

Many systems have subscribed to PDLs as the off-line filing representation because PDLs

[2] A comparison between INTERPRESS and POSTSCRIPT can be found in [55].

are supported by an increasing number of printers. The on-line imaging mechanism in WYSIWYG systems, nevertheless, is still based on lower-level descriptions. The result is a discrepancy between the on-line imaging and the potentially more powerful one given by the off-line representation. This problem can be resolved by providing a PDL server on-line and representing the internal structure as the PDL. In reality, a PDL server (interpreter) can be realized as, in ascending degree of generality, a client level application for graphics specification (e.g. PostScript in VorTeX, see Section 6.2), the underlying imager of a window system (e.g. Interpress in *Cedar*), or even the window system server itself (e.g. PostScript in *NeWS* [10]).

Another aspect of off-line filing concerns saving a snapshot of the internal state so that future invocations can take place incrementally. This is analogous to saving object files for a source program. It can also be viewed as a checkpointing mechanism which provides backups as well as a means to support undo operations. The issue here is the standard space/time tradeoff. The simplest approach is to save the entire core image and reload it when a rollback or reinvocation is requested. The penality, of course, is tremendous storage overhead. A more "source-based" approach would take more time abstracting only the essential parts and saving them structurally in a textual format. Again, it takes time to recover the state when a rollback or reinvocation happens, but the filing representation would be much more compact.

3.6 Dynamic Reading and Annotations

So far we have focused on document composition, or the *writing* side of document preparation as a whole. The other half of the story, which has too often been ignored, concerns effective *reading* of a document. This is an area where the language-based model does not seem to carry over. Traditionally reading is a "direct manipulation" process. When references are involved, however, we rely on a somewhat "indirect" approach. For instance, when a bibliography reference is of interest, we need to go to the bibliography section and look up the cross reference information available there.

This static notion of documents still dominates our way of reading even in the era of electronic media. On our favorite document preparation systems, we are still artificially creating bibliographies and indexes for our books. Part of the reason is being able to generate hardcopies consistent with the tradition. But if hardcopy compatibility as an issue is relaxed, then one should think seriously about what exactly are the purposes of references like the bibliography and index. Their foremost function is to allow the reader access relevant information efficiently. Creating separate bibliography and index sections is the best one can do with the static print medium.

In an integrated document development environment, much of this linking information can be stored internally. Therefore, instead of requiring the reader to actually "read" the section which contains the references (indirect accessing), it is possible to access references in a direct and context-sensitive way. For instance, when a citation is of interest, a menu of options would allow the reader to (1) inspect the content of the bibliography entry in a separate window so that the reading of the main document is not hindered, or (2) visit the actual document referenced by the citation (this can occur recursively). If an object of a

different nature is selected, its context gets reflected in the menu immediately.

Operations like these can go beyond "what you see is what you get". With the "shadow document" approach mentioned earlier, for example, annotations can be associated with key concepts and embedded in the document invisibly. Thus the information a document is able to convey is much more than "meets the eye". More elaborate hypertext operations are possible [65]. Some candidates include local features like *filtering* (restricted reading) and *fisheye viewing* (focused reading) [29], or more global issues like document navigation and dissemination, and so on.

Another important aspect of an electronic document is that its presentation is not confined to a single medium of static image. Thus, a document may comprise dynamic pictures (animation) with voice narrations, and more. All of this requires extensive internal support and the user interface must be based on a clever blending of the two models.

4 Procedurality vs. Descriptiveness

Document processing systems can also be classified according to their "degree of procedurality" which refers to the granularity of control the user is allowed to possess over a specific task. Equivalently, one can think of this as the amount of information a system must know *a priori* in terms of the document's style and structure. Consider formatting for example, at one end of the spectrum we have what may be called the pure *procedural* scheme which requires the user to specify exactly *how* the formatting ought to be carried out at the *physical* layout level. At the other end is the pure *descriptive* (or *declarative*) scheme in which the user specifies just *what* a document should be at the *logical* structure level; formatting details associated with various document styles are hidden from the user.

The notion of procedurality is orthogonal to the two models mentioned previously; both procedural and descriptive schemes exist in both models. For instance, in the source language model, *troff* and `initex` (i.e. TEX with no macro packages preloaded) are pure procedural languages. *Scribe* and *SGML*, on the other hand, are pure descriptive systems. On the direct manipulation side, *MacWrite* [9] is tilted toward the procedural approach in that restrictions on the user's control over the document appearance are minimal. Systems like *Mint* [34] and *Lara* [33] are simply WYSIWYG versions of *Scribe* and thus may be classified as descriptive.

There are systems with a combined flavor of both schemes. Pure procedural languages can usually be shifted toward the descriptive end using their abstraction mechanisms. For example, the `ms` macro package [47] for *troff* and the `plain` macro package for TEX both provide some higher order structures on top of their primitives. Although the low level control is still accessible, they are essentially dialects of the original languages with a higher degree of "descriptiveness".

Yet a better example is LATEX [44] which is an attempt to emulate *Scribe* in the world of TEX. Like *Scribe*, LATEX recognizes a number of predefined document styles. Based on these styles, the user specifies a document by invoking generic entities such as chapters and sections. Exactly how these chapters and sections will be formatted is hidden from the user. But unlike *Scribe*, LATEX also allows some low level functions which do not contradict with

its style definitions to be imported from TEX. It is reasonable to classify a system like LATEX as standing half way between the two extremes.

There are pros and cons for both schemes. The first consideration is the issue of device independence. Some *SGML* advocates argue that by associating different formatting functions for different devices to the same document style, a descriptive system yields better device independence than a procedural system does [31]. Depending on the definition of device independence, this may or may not be true. Rather than maintaining it explicitly at the source level, a procedural system like TEX achieves device independence by generating its output in some generic format (DVI) which can then be translated to a variety of device (printer or screen) languages. Since device independence is achievable in both schemes with a comparable amount of support, it is not obvious which is superior in this respect.

The true merit of the descriptive scheme is that it relieves the user from dealing with details of formatting. The system is normally more compact and thus easier to implement. The major limitation, however, is with its rigidness. In particular, manipulating document styles requires mastering a set of functions different from the one known to the user. Hence it is generally difficult for a casual user to perform fine tuning if the formatted result is unsatisfactory. A direct manipulation approach is more appealing in this respect. Instead of programming in a style definition meta-language, all descriptive attributes can be encapsulated in property sheets with an obvious form-based user interface. Then the question becomes whether or not every bit of detail in controlling the formatting information can be parameterized descriptively.

In contrast, it is in the issue of fine control when a procedural system demonstrates its strength. Another unique advantage of a procedural system like TEX is its extensibility: macros can be used to define high level structures or even emulate descriptive properties (e.g. LATEX). On the other hand, emulating procedural properties in a descriptive system like *SGML* is very difficult. The tradeoff here is "power" versus "ease of use". The two schemes seem to complement each other in many respects. In any event, the degree of procedurality serves as a basis for evaluating different approaches of the hybrid model in document processing.

5 Design Methodology

We have raised a number of issues; some are orthogonal and others are somewhat contradictory. The most essential question concerns the relationships of the two models, the task domain, the various representations, their transformations, and the notion of procedurality. This section tries to answer this question and to establish a general framework for analyzing and designing multiple representation systems. The framework differs from some more complex models like Sandewall's *Theory of IMS* [57] in that our framework (1) is less complex and therefore much easier to follow, and (2) addresses more properly the multiple representation aspect of document preparation, with possible extensions to similar software environments.

The basic structure of multiple representation document development systems, or the *representation domain*, is illustrated in Figure 6. As shown in the figure, it includes four generic representations:

1. S: a source representation supporting high-level programming constructs such as abstraction mechanisms (e.g. macros, procedures, or variables), control structures (e.g. conditionals, iterations, or recursions), etc. A document in TEX or *troff* is a representation of this type.

2. O: a structural view of the basic objects involved in the system. This representation may be one with built-in descriptive logical components such as a document in *SGML*, or one with object-based input/output such as a drawing under *MacDraw*, or one with a hierarchical structure like the internal representation of VORTEX.

3. T: a representation corresponding to the objects' physical structure after processing. This representation is usually device-independent.

4. D: the actual device-dependent image representation.

This basic structure may be augmented to include derivatives of the four generic representations as required by a particular task. It must be pointed out that S is not the exclusive representation of the source language model mentioned earlier. For instance, *SGML*, a language-based system, is classified as having a primary representation of O rather than S. The distinction here stems from the availability of program constructs.

The basic structure also has an abstraction for various types of user interfaces (U); possible distinctions are keyboard/command-based versus mouse/menu-driven versus their combination, textual versus graphical, etc. Figure 6 shows U as a single entity, but it may be refined to reflect these distinctions or be specified according to more sophisticated guidelines such as those described in [51].

There are several important aspects of this structure which underscore and unify all the issues in question:

- Whether or not a representation (solid box) exists.
- Whether or not an existing representation is made explicit to the user (i.e. if there is a dashed line connecting the solid box and the dashed box); if so, whether or not the relation is bidirectional.
- Whether or not a transformation (solid line) exists between two existing representations; if so, whether or not such a transformation is bidirectional.

More precisely, an instance of the fundamental structure (call it Ω), or the *representation instantiation*, is described by a quadtuple

$$\Omega = (\Pi, \Theta, \Gamma, \Delta)$$

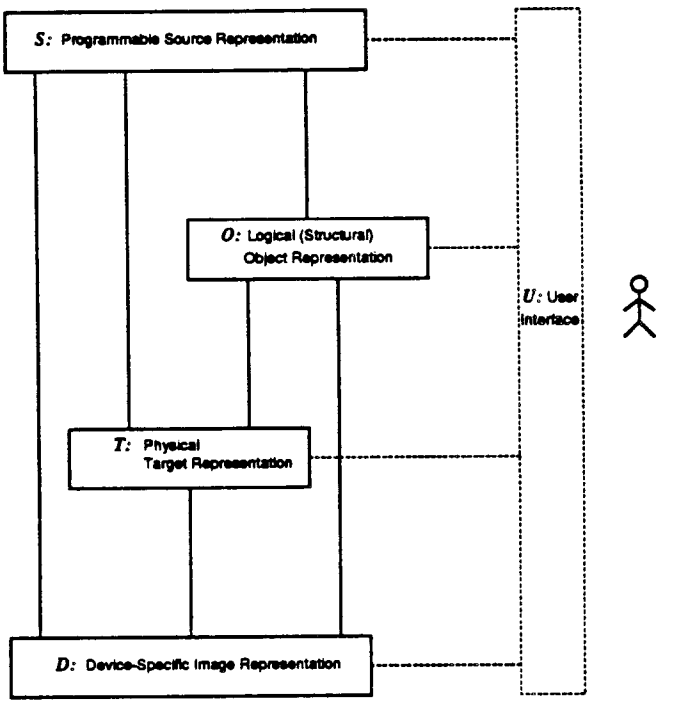


Figure 6: *Fundamental Structure of Multiple Representation Systems.* The legend is the following: the four *solid boxes* are the various representations in question, *solid lines* connecting these boxes each refers to a transformation between the two representations, the *dashed box* on the right stands for the user interface abstraction, and finally, the *dashed lines* each indicates a link between the various representations and the user interface. The figure shows that all four are connected among themselves; they are also connected with the user interface. The real situation, however, is that the connections are directional and for certain systems some of the nodes and edges in the graph may be absent.

where

$\Pi = \Pi_S \cup \Pi_O \cup \Pi_T \cup \Pi_D$, set of one or more representations,
$\Theta = \{U_1, U_2, \cdots\}$, set of user interface abstractions,
$\Gamma = \{\pi_1 \rightarrow \pi_2 \mid \pi_1, \pi_2 \in \Pi\}$, set of interrepresentational transformations,
$\Delta = \{\pi \rightarrow \theta \text{ or } \theta \rightarrow \pi \mid \pi \in \Pi, \theta \in \Theta\}$, set of user interface relations.

and

$\Pi_S = \{S_1, S_2, \cdots\}$, one or more programmable source representations,
$\Pi_O = \{O_1, O_2, \cdots\}$, one or more structural object representations,
$\Pi_T = \{T_1, T_2, \cdots\}$, one or more physical target representations,
$\Pi_D = \{D_1, D_2, \cdots\}$, one or more device image representations,

Intuitively, $\pi_1 \rightarrow \pi_2$ ($\pi_1, \pi_2 \in \Pi$) means representation π_1 can be directly transformed to representation π_2. This can be illustrated graphically by an arrowhead at the end of the solid line connecting π_1 and π_2. Similarly, $\pi \rightarrow \theta$ means representation $\pi \in \Pi$ can be explicitly viewed by the user with interface $\theta \in \Theta$, and $\theta \rightarrow \pi$ means representation $\pi \in \Pi$ can be accessed or manipulated by the user through interface $\theta \in \Theta$. For convenience, $\pi_1 \rightarrow \pi_2$, $\pi_2 \rightarrow \pi_1$ can be abbreviated as $\pi_1 \leftrightarrow \pi_2$, and $\pi_1 \rightarrow \pi_2$, $\pi_2 \rightarrow \pi_3$ as $\pi_1 \rightarrow \pi_2 \rightarrow \pi_3$.[3]

The design of a multiple representation system, therefore, is to derive a representation instantiation (Ω) *for each member of the task domain* as shown in Figure 7. Defining an Ω requires identifying (1) the representations to be maintained (Π), (2) the specification of a user interface abstraction (Θ), (3) the set of inter-representational transformations among members of Π, and finally (4) the set of user interface relations from Π to Θ and vice versa.

Given this framework, it becomes natural to analyze systems belonging to either language-based or direct manipulation camp, or to discriminate procedural systems from descriptive ones. For instance, a language-based batch system implies the existence of a representation S or O and some unidirectional relations from this representation to T or D.[4] A direct manipulation system, on the other hand, will be based on an instance of Ω having $\pi \leftrightarrow \theta$ in Γ ($\pi \in \Pi$ and $\theta \in \Theta$), with the added criterion that feedback from the system be immediate in order to create the sensation of directness. Furthermore, the property of procedurality usually means that either $S \rightarrow T$, $S \rightarrow D$, or $T \rightarrow D$ is in Γ, and that either $U \rightarrow S$ or $U \rightarrow T$ is in Δ (i.e. either S or T is user manipulable). Finally in a descriptive system, Π_S will normally be empty and Π_O will be the only set of manipulable representations.

Based on these observations, it is interesting to compare WYSIWYG graphics editors such as *MacPaint*, *MacDraw*, and *Illustrator*. *MacPaint* can be described by

$$O \rightarrow D,\ U \rightarrow O,\ D \leftrightarrow U$$

because the user creates drawings with some object-level menus ($U \rightarrow O$), but once specified, objects are transformed into a device-dependent image ($O \rightarrow D$) which can be viewed and

[3] This is solely for the convenience of notational abbreviations. No transitivity is implied in $\pi_1 \rightarrow \pi_2 \rightarrow \pi_3$ (i.e. it does not imply $\pi_1 \rightarrow \pi_3$).

[4] Our notational conventions are obvious here; we assume $S \in \Pi_S$, $O \in \Pi_O$, $T \in \Pi_T$, $D \in \Pi_D$, $U \in \Theta$, and similarly with subscripted ones used later.

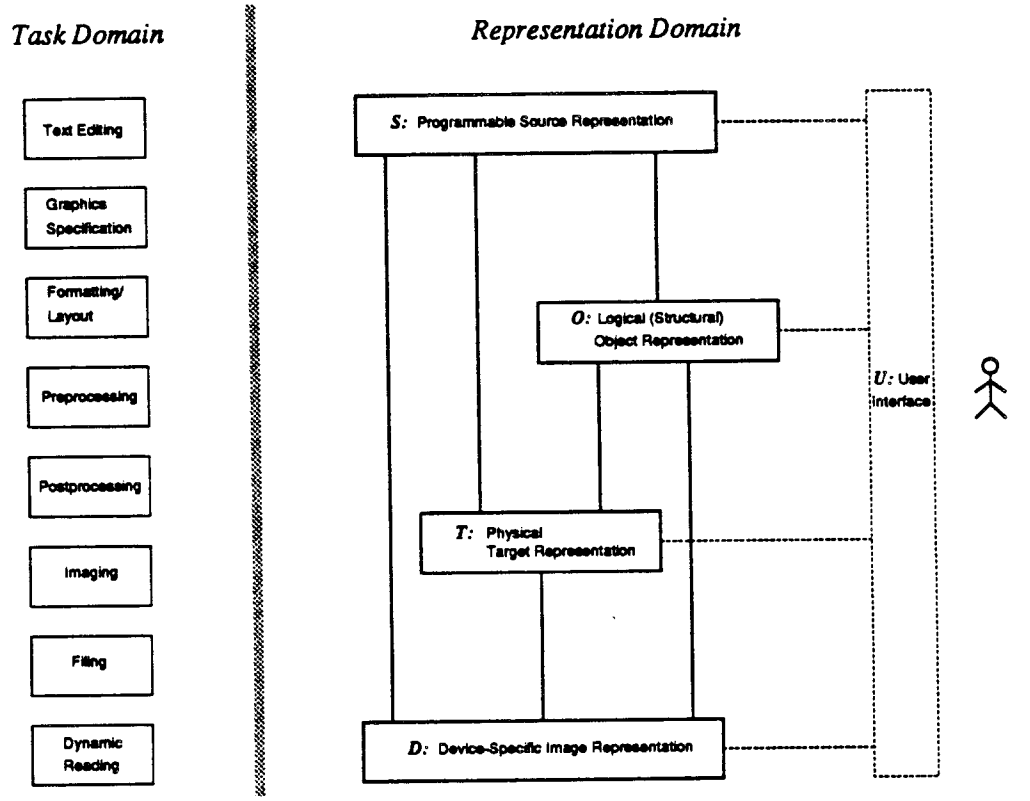


Figure 7: *Task and Representation Domains*. The gray vertical bar represents the boundary between the task domain on the left and the representation domain on the right. A multiple representation system is a mapping from the task domain to the representation domain.

manipulated by the user ($D \leftrightarrow U$). By contrast, *MacDraw* corresponds to

$$O \leftrightarrow T,\ T \leftrightarrow D,\ O \leftrightarrow U.$$

There are two major differences here: (1) in *MacDraw* the user views and manipulates drawings at the object level, and (2) drawings in *MacDraw* are device-independent due to the presence of a target representation. Finally, *Illustrator* is close to

$$O \leftrightarrow T,\ T \leftrightarrow D,\ D \rightarrow O,\ O \leftrightarrow U,\ U \rightarrow D.$$

The crucial difference between *Illustrator* and *MacDraw* is $U \rightarrow D \rightarrow O$ which underscores *Illustrator*'s capability for the user to unravel geometric objects from bitmaps by tracing the image.

More importantly, this framework facilitates the analysis of existing multiple representation systems and the design of new ones. The basic criterion here is that at least two members of Π must be manipulable. *Emacs*, for example, can be viewed as a system having Ω_{Emacs} for text editing, with the majority of remaining members in the task domain mapped to the empty set, where Ω_{Emacs} is defined as follows:

$$\begin{aligned}
&\Pi_{Emacs} = \Pi_S \cup \Pi_O \cup \Pi_T \cup \Pi_D = \{S\} \cup \{O\} \cup \{T_1, T_2\} \cup \{D\},\\
&\Theta_{Emacs} = \{\, U \,\},\\
&\Gamma_{Emacs} = \{\, S \rightarrow S,\ O \rightarrow S,\ S \leftrightarrow T_1,\ T_1 \leftrightarrow T_2,\ T_2 \leftrightarrow D,\ \},\\
&\Delta_{Emacs} = \{\, U \leftrightarrow S,\ U \rightarrow O,\ U \rightarrow T_1,\ U \leftrightarrow T_2,\ \}.
\end{aligned}$$

To interpret this specification, one can think of S as the Lisp code under which *Emacs* operates, O as the collection of user-level objects such as characters, words, lines, regions, etc., T_1 as the one-dimensional text stream (where linefeed is just an ordinary ASCII character), and T_2 as the corresponding two-dimensional text array (where linefeed causes a line break). The combination of T_1 and T_2 forms the overall target representation T.

Thus Ω_{Emacs} says that the user can view both source and two-dimensional target representations ($S \rightarrow U$ and $T_2 \rightarrow U$) and manipulate either the one-dimensional text stream ($U \rightarrow T_1$), the two-dimensional text array ($U \rightarrow T_2$), the objects ($U \rightarrow O$), or the program ($U \rightarrow S$). Operations on objects get transformed into code ($O \rightarrow S$) which is then evaluated and represented in the one-dimensional text stream ($S \rightarrow T_1$). Both O are T_1 are implicit since they are not explicitly exposed to the user (i.e. no $O \rightarrow U$ or $T_1 \rightarrow U$).

T is split into T_1 and T_2 because the user operates on the one-dimensional text stream as well as the two-dimensional text array, although the actual screen appearance is two-dimensional. Normally, `next-line`, `previous-line`, and a host of common operations are two-dimensional. But things like `forward-char` and `backward-char` are one-dimensional; at the boundary of current line they move to the adjacent boundary of the next or previous line linearly. Furthermore, the actual internal representation is one-dimensional because a character is addressed by an offset relative to the beginning of buffer instead of by a two-dimensional coordinate.

One interesting point is that a recursion can be observed in Ω_{Emacs} — although its is not explicitly shown — the editing of S is essentially Ω_{Emacs}. In other words, S can be expanded

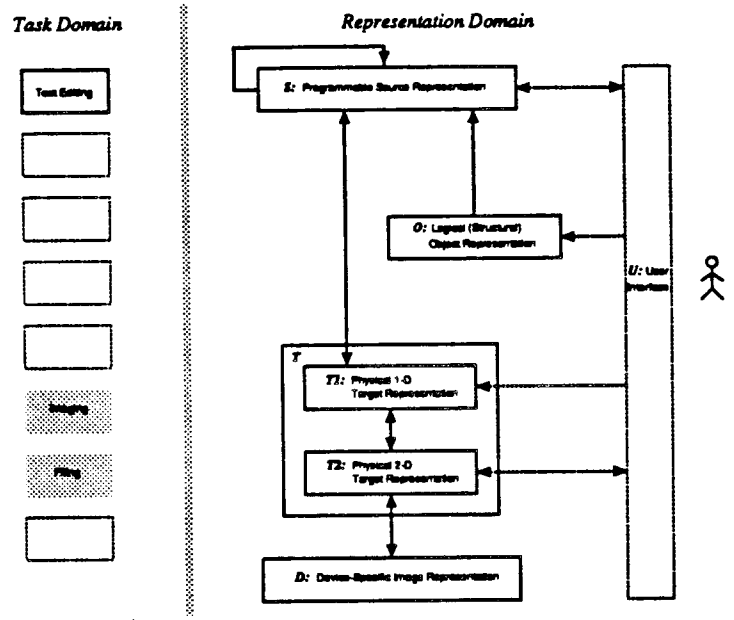


Figure 8: *Graphical Specification of* Ω_{Emacs}. The task of text editing has been singled out by the darkened box on the left. The shaded boxes represent other tasks involved in the overall system which are not the focus of current instantiation. Empty boxes are those tasks which play no roles in the system.

to a secondary Π_{Emacs}, $S \rightarrow S$ is equivalent to the composite of the rest of Γ_{Emacs}, and $S \leftrightarrow U$ is, in effect, the composite of the rest of Δ_{Emacs}.

An Ω may be specified graphically. Figure 8 shows Ω_{Emacs}'s corresponding graphical specification. Figure 9 is an instance of a design reflecting some major features of *Tweedle* mentioned earlier in Section 3.2.4. Because *Tweedle* supports both a textual form of procedural language as well as an object level graphical representation, the task domain includes primarily text editing and graphics specification. The text editing side of the story is identical to that of *Emacs* discussed above. Figure 9 illustrates an Ω corresponding to its graphics specification task only.

The representations and transformations involved are quite self-explanatory except that there is no transformation from O to T because no object level evaluation is available in *Tweedle* — graphical objects always get transformed into code which is then evaluated. One can normally expect low level primitives in terms of registering cursor positions or mouse clicks be provided by the underlying window manager. It must be pointed out that the editing of S is another instance of Ω_{Emacs}. This is an example which shows, as an integration mechanism, how one Ω may be plugged in as a component of another Ω.

This framework is by no means complete or precise, but it does establishes a good approximation of what is to be accomplished by a multiple representation system. We envision a top-down methodology based on this framework can be of use to designers. The design process starts with identifying the task domain. For each element in the task domain, the

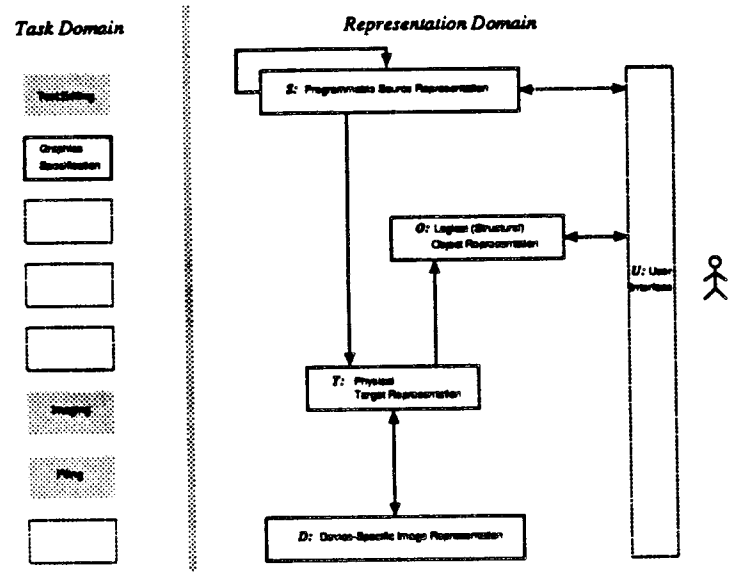


Figure 9: *Representation Instantiation of Tweedle's Graphics Specification.*

representation domain is instantiated with the specification of an Ω. Within each Ω, finer issues are then sorted out, and that may go down deep as stepwise refinement requires.

From a more global perspective, the collection of these Ω's may be thought of as a multi-dimensional space with a similar overall structure in each dimension. Interrelationships and commonality among these "dimensions" in terms of issues as high level as user interfaces or as low as code sharing can be extracted and exploited. The final step of the design process is *integration* which reflects conceptually how these "dimensions" are integrated. In practice, integration is realized by means of sharing subsets of the Ω's involved as connection stubs in the global space. These stubs may be certain representations ($\Pi \cup \Theta$) or transformations ($\Gamma \cup \Delta$). For instance, one Ω may be a member of another Ω's Γ or Δ, and many of the representations (Π) may be shared among various tasks.

6 Case Study

This section discusses the principal properties of VORTEX, a document development environment based on the multiple representation paradigm, as a case study of the methodology introduced previously. Also included in the discussion as comparisons are some key ideas from the *tnt* editor/formatter [30] and two other on-going projects *Lilac* [21] and *Quill* [22], both of which focus on the same set of issues as VORTEX does. Here, we concentrate on properties insofar as specifying Ω is concerned; issues of finer granularity are deliberately omitted for clarity.

Based on the top-down methodology, we start identifying the task domain as containing everything listed in Section 2 plus a few derivatives (to be described later). We then have to define an Ω for each member of the task domain. Before we give the specifics of these

Ω's and their interrelationships, we need to mention that VORTEX's source language for formatting and layout is TEX. Since graphics in TEX is virtually undefined, we have chosen POSTSCRIPT as our graphics specification language. Both of these tasks are maintained in multiple representations.

6.1 Text Editing

Text editing in VORTEX is *Emacs*-based. Despite some differences in the fine points, Ω_{Emacs} given in the last section would suffice in describing VORTEX's multiple representational view of text editing. Like *Emacs*, language-specific modes will be available for editing code in TEX or POSTSCRIPT. The underlying Lisp subsystem is not confined to the the task of editing; it also serves as the basis of system integration and a host of computation-related jobs, as it will become clear later on.

6.2 Graphics Specification

VORTEX's graphics is based on POSTSCRIPT. Like *Tweedle*, a program representation as well as a graphical view of the objects are implicitly maintained by the system and explicitly manipulated by the user. Therefore, the Ω defined in Figure 9 also describes VORTEX's graphics subsystem. In detail, however, the actual representations are distinct due to the differences between POSTSCRIPT and *Tweedle*'s underlying language *Dee*.

A POSTSCRIPT imaging server (interpreter) is available for rendering images in the graphics editor. The same server also interacts with VORTEX's main document display module. When a picture is encountered in the displayer, the corresponding code is transmitted to the POSTSCRIPT server. It in turn hands back the graphics as a raster image, which is then incorporated into the document's device representation.

6.3 Formatting/Layout

For the task of specifying a document's textual content in general and its formatting and layout information in particular, VORTEX provides a source level program (in TEX) as well as a target level view to the user. Operations performed on one representation will be propagated to the other automatically. The idea is to take advantage of the "expressiveness" of a source programming language and also the immediate visual response given by a direct manipulation user interface to the target representation. There are two base editors, one for editing the source representation, the other for the target, both of which are based on an extension to the *Emacs* editing paradigm.

Figure 10 shows the representation instantiation (Ω) of VORTEX's formatting and layout. It says that the document's source representation (S) is transformed into an internal object structure (O), which then becomes the physical layout (T) of the document after formatting. This target representation can be interpreted by a displayer on-line, or translated off-line into a file format such as DVI or a program in certain printer language like POSTSCRIPT. Both S and T may be manipulated by the user. The bidirectional transformation between S and

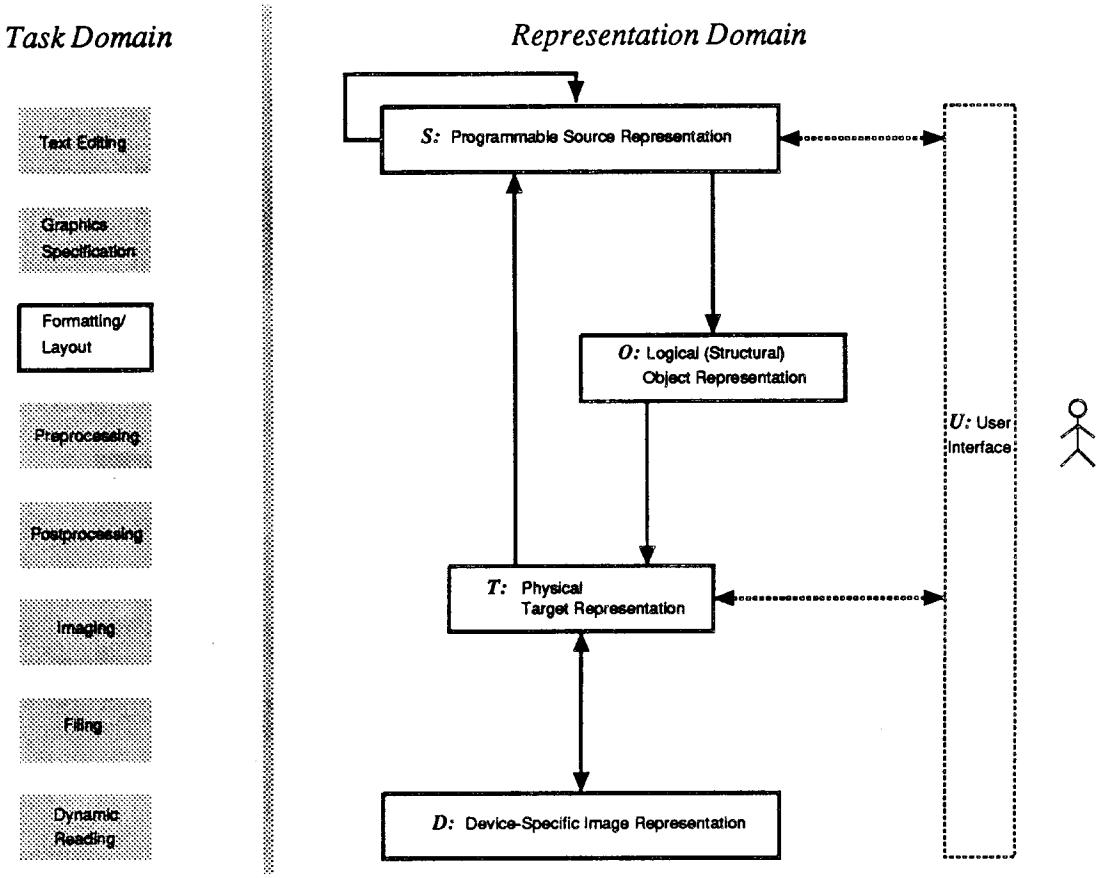


Figure 11: *Representation Instantiation of VORTEX's Formatting and Layout.*

U is an extended version of Ω_{Emacs}. Changes to S are reflected in itself directly and are propagated to T through O. Changes to T, however, are first propagated to S and finally go through the $S \rightarrow O \rightarrow T$ cycle to be reflected back to itself.

Propagating changes from source to target is straightforward in concept because that is exactly what TEX does. The subtlety here is that instead of a batch-oriented implementation, VORTEX needs to be incremental, which generates a number of interesting issues not encountered in the batch version. The fact that TEX is macro-based complicates this problem even more.

In VORTEX, a close relationship is maintained between the source representation (S) and the two internal representations (O and T). Incremental formatting is based on marking and sweeping dirty nodes in S and O and on comparing newly generated code with what is stored in T. The strategy here is, in principle, similar to IBM's *Interactive Composition and Editing Facility, Version 2 (ICEF2)* [24].

```
(defun create-windowed-par (begin end lintel-ht side-wd window-ht)
   (goto-char (target-to-source begin))
   (insert "\\beginwindow"
           "\\lintel " lintel-ht "\\lines "
           "\\side " side-wd "in"
           "\\window " window-ht "\\lines\n")
   (goto-char (target-to-char end))
   (insert "\\endwindow\n"))
```

Figure 11: *Reverse Mapping of Page Layout.* This is the Lisp function to be triggered when a paragraph like Figure 2 is laid out in the target editor. This function operates in the source representation. The first two arguments, given in target positions, must be translated into source positions via internal representation accessing before used. The next three arguments represent, respectively, lintel height, side block width, and window height, as required by the windowed text environment shown in Figure 5.

6.4 Reverse Mapping

The next major issue concerns identifying the set of WYSIWYG operations visible to T and realizing the reverse mapping mechanism which propagates side effects back to S. We believe *page layout*, *object placement*, *attribute update*, and similar operations would benefit most from prompt visual feedback and are therefore reasonable candidates to be incorporated in the WYSIWYG interface to T. For instance, a page layout specified at the target level in direct manipulation like Figure 2 would correspond to the TEX source code of Figure 5 by the reverse mapping facility.

The question is how to carry out reverse mappings systematically. In VORTEX this is realized by associating each target level operation having any side effects with a Lisp function at the source editor. Whenever such an operation is executed in the target editor, the corresponding Lisp function gets invoked and evaluated by the source editor. The user interface is quite flexible in that it can be either command-driven (with the standard *Emacs* keyboard binding scheme), menu-driven (with mouse as the primary input mechanism), or a combination. It is also extensible; new instances of reverse mapping can be added to the system by the user which will turn out consistent with the overall interface structure.

All of these are made possible with the support of a Lisp programming subsystem within the environment. Reverse mapping is programmed on top of the system's editing primitives for source level pattern matching and some extended functionality for internal representation accessing. Thus, to lay out something as exotic as Figure 2 in the middle of a page, the corresponding Lisp function may look like what is shown in Figure 11. In the code, `begin` and `end` represent the beginning and end, in target positions, of the paragraph to which a window is to be opened. The function `goto-char` positions the cursor to the point given as argument in the source editor, where the positions are translated by `target-to-source` from target to source via internal data structure accessing. Finally, inserting the text for opening

and closing the windowed paragraph is straightforward.

The reverse mapping of page layout is relatively trivial compared to things related to *macro unraveling*. A macro and its arguments in the source representation (S) may not have a one-to-one correspondence with the expanded text which ultimately appears in the target representation (T). Typically there are three cases in a macro expansion: (i) text as arguments of a macro in S gets carried over to T, (ii) text in S is consumed by the expansion and therefore disappears in T, (iii) new text originally not in S gets introduced in T by the expansion. When the expanded text is selected in T, what are the semantics of target-level operations using the selected text as an operand?

As a premise, the selection mechanism must be able to tell if the text is part of an expanded macro. Since internal representation accessing primitives are available to the Lisp subsystem and since the object and target representations (O and T) are tightly coupled, one can easily identify if any selected text is in the proper scope of a macro. To handle the semantics, we can first eliminate case (ii) because disappeared text will not be selected in T anyhow. Depending on the user's intention, there are three possibilities:

1. The interest is in plain text only regardless of how the macro is expanded. Thus, including the text introduced by a macro, all "characters" seen by the user can be used as the operand, but no other attributes (e.g. typeface, size, etc.) will be associated with it. Operations of this type must be non-destructive with respect to the selected text itself. A plausible operation belonging to this group is *copy*.

2. If the selected text is carried over from S, destructive operations such as *insert*, *delete*, *move*, and so on are legitimate. Side effects are first reflected in S (the cursor will be "warped" to the source editor window) and eventually get reflected in T through the $S \rightarrow O \rightarrow T$ cycle.

3. If the text is introduced, such destructive operations will be disabled with some warning messages. One step beyond this will be a query asking the user if the intention is to modify the definition of the macro in question. If so, the macro unraveling Lisp code can scroll to the most recent spot in context where the macro is defined and let the user do the modification at the source level. A more elaborate approach is to incorporate certain rules which correlate encapsulated operators and operands in T with the underlying TEX code to be inserted to the macro definition in S.

Reverse mapping on the basis of per target level operation is somewhat special to VORTEX. By contrast, in *Quill* [22], the underlying source language is the fully descriptive *SGML*. There are two levels of internal representations (O and T) maintained in *Quill* as in VORTEX. Unlike VORTEX, however, *Quill*'s external source representation is hidden during editing (i.e. no connections between S and U). The only role *SGML* plays is off-line filing and document interchange. In other words, reverse mapping becomes unnecessary in *Quill*. Its logical object representation (O) is a mirror of an *SGML* document; each node in O corresponds to an *SGML* markup tag. Thus, when the document is to be filed, all that is needed is to traverse O and the corresponding file in *SGML* can be generated.

As was argued in Section 4, the tradeoff boils down to complexity versus flexibility. Compared to *Quill*, VORTEX's overall architecture is more complex due to TEX's low degree of descriptiveness and its macro-based abstraction mechanism. On the other hand, VORTEX is more flexible; to create a WYSIWYG type page layout like Figure 2 and be able to map it back to the source is simply beyond *Quill*'s model. Imposing logical document structure is also possible in VORTEX. Although O does not carry any logical meaning in VORTEX, document structure and style like those defined in LATEX can be realized by the reverse mapping facility which operates at the source level. Since the user interface is customizable, one can effectively hide the procedural aspects of TEX in VORTEX.

6.5 Pre- and Post-Processing

The pre- and post-processing facilities by and large follow a trilogy of (1) placing task-specific markup tags (commands) in the document body, (2) processing an auxiliary file containing information related to these tags, and (3) incorporating the results back to the main document. In many cases, these tags do not appear in the target representation; instead, they create links between different objects. These links frequently destroy the strict top-down hierarchy of the document's internal logical structure (O).

In VORTEX, all three steps are again built on top of the Lisp programming subsystem. Since a source representation is explicitly maintained, there is no need to hide these tags in the "shadow". The advantage of operating at the source level is that the internal representation does not have to increase its structural complexity. Tags such as citations retrieved from a bibliography database are directly inserted into the document source. The programming layer also has control over external processors. Thus, when the off-line processing is finished, the result can be interactively incorporated back to the source representation by the top-level of a Lisp program which initiated the processing.

6.6 Imaging and Filing

VORTEX's on-line imaging mechanism is based on direct interpretation of the target representation. Both its source in TEX and a translation of T (e.g. in DVI or POSTSCRIPT) can be filed as the off-line representation. It is also possible to base the on-line displayer completely on a PDL like POSTSCRIPT because such a server is already available for rendering graphics. The Ω for on-line imaging has been covered in Section 6.3; the one for off-line filing and imaging is a straightforward batch approach.

6.7 Dynamic Reading

Given a full-blown PDL as the graphics image server (POSTSCRIPT in this case), VORTEX is able to present pictures dynamically. This may happen in one of two modes: *playback* and *synthetic*. In playback mode, the document displayer constantly gets notifications from the POSTSCRIPT server with new raster images of the same picture. In processing each notification, the old picture is erased and replaced by a new image. If this happens frequently

enough, it becomes, in effect, animation. In synthetic mode, the displayer simply executes a PostScript program which takes care of itself in terms of any dynamics involved. The premise, however, is that the document displayer be the PostScript server itself. From the multiple representation's viewpoint, the playback mode is closer to direct manipulation because scenes as raster images are the basic manipulable objects, while the synthetic mode is more like language-based due to its programming aspects.

Furthermore, given the Lisp programming subsystem, the ability to access internal structure through some lower level primitives, and an extensible user interface, VORTEX is capable of providing the user with some dynamic viewing functionality. This is tightly coupled with the pre- and post-processing facilities mentioned earlier. For instance, one can select a reference and have the content of the reference displayed in a separate window. This type of context-sensitive browsing applies to objects like citations, cross references, indexes, and the like. What is special here is that no hard links are built into the internal representations for browsing purposes. Each operation is realized as a user-level function which performs primarily pattern matching in the source manuscript with the aid of internal representation accessing primitives.

6.8 Integration

Conceptually the Ω's described above can be thought of as a multi-dimensional space which reflects the structure of the overall system. In reality, the system is integrated by means of sharing certain representations ($\Pi \cup \Theta$) or transformations ($\Gamma \cup \Delta$). For instance, Ω_{Emacs} is essentially $S \rightarrow S \leftrightarrow U$ in the Ω of both graphics specification and formatting/layout; the Ω corresponding to dynamic reading just mentioned constitutes part of $T \rightarrow U$ in formatting/layout. Also, the internal representations of formatting/layout are shared by tasks such as reverse mapping, pre- and post-processing, and so on.

In particular, text and graphics integration in VORTEX employs a "cut-and-paste" model. The manipulation of text and graphics each operates under a distinct context. The integration is based on the PostScript imaging server. From the document formatter and displayer's point of view, graphics is just a piece of raster image. Therefore, text within graphics will not be formatted the way regular text is; it all depends on how the graphics imager treats text and fonts.

Quill represents a fundamentally different model in which arbitrary nesting of text and graphics is permitted and their processing is uniform. The uniformity is achieved by sharing a common object representation (O) between graphics specification and formatting. Like text, graphics nodes in O will eventually be mapped to their *SGML* counterparts [23]. These nodes can be arbitrarily nested, a context sensitive menu will be displayed when a node of a particular type is selected. The integration mechanism here is based on representation sharing rather than a transformation plug-in as is in VORTEX.

VORTEX's Lisp programming subsystem provides the essential glue for integrating the bulk of tasks together in a coherent manner. This includes reverse mapping, the many phases of pre- and post-processing, dynamic reading, and so forth. Most importantly, it also serves as the backbone behind job control and user interface customization. For a complex environment

like VORTEX, a user-level programmable source representation like the Lisp substrate reduces a great deal of complexity from the system's internal representations as well as its overall integration mechanism which may otherwise be ubiquitous and difficult to manage.

7 Conclusions

We have reviewed a large number of document development systems for text and graphics alike from the perspectives of both language-based and direct manipulation models. The central theme is that program constructs and visual feedback are complementary to each other and that a hybrid approach would be most desirable. Yet our concern goes beyond superficial feature level comparison and focuses on devising a systematic analysis for the general notion of multiple representations.

A complete document development environment involves multiple tasks, multiple representations internally and externally, and multiple transformations among the tasks and representations. As familiar as it may sound, what does it mean to be WYSIWYG, given this intrinsic complexity? What does it mean to be language-based? What about procedurality, descriptiveness, and other important concepts, all of which are confusing? In this paper we have established some ground work for analyzing all these issues. With the Ω framework, the basic structure of a multiple representation system becomes readily crystallized.

The design methodology based on the Ω framework is simple but robust. It requires the designer to first define the task domain. For each member of the domain, an Ω is then instantiated. Finally all these Ω's are integrated by sharing certain representations or transformations. Although this approach is not meant to be a formal mathematical model and an Ω itself is insufficient to describe every detail of the task, it does provide a good deal of insight into the structuring of complex document development environments. We are currently working on enhancing this framework to convey more information without sacrificing its simplicity. Being pursued in parallel is the implementation of VORTEX which will ultimately offer a proper validation for the design methodology discussed here.

8 Acknowledgements

We gratefully acknowledge crucial contributions in both the design and implementation of VORTEX by other members of the team: John Coker, Jeff McCarrell, Ikuo Minakata, Ethan Munson, and Steve Procter. Michael Van De Vanter's comments on the framework have been most helpful. Thanks also go to Don Chamberlin for the description of *Quill*, to Ken Brooks for the discussion on *Lilac*, to Paul Asente for the information on *Tweedle*, and to Doug Terry for giving us a tour of *Cedar/Tioga*.

References

[1] *8010 STAR Information System Reference Library, Release 4.2.* Xerox Office Systems, El Segundo, California, 1984.

[2] *The Adobe Illustrator Manual.* Adobe Systems, Inc., Palo Alto, California, 1987.

[3] *CricketDraw Manual.* Cricket Software, Inc., Philadelphia, Pennsylvania, 1987.

[4] *Document Description Language (Revision 1.1): Reference Manual and Tutorial.* Imagen Corporation, Santa Clara, California, November 1986.

[5] *Frame Maker Reference Manual, Version 1.0.* Frame Technology Corporation, San Jose, California, February 1987.

[6] *Interleaf Publishing Systems Reference Manual, Release 2.0, Vol. 1: Editing and Vol. 2: Management.* Interleaf, Inc., Cambridge, Massachusetts, June 1985.

[7] *MacDraw Manual.* Apple Computers, Inc., Cupertino, California, 1984.

[8] *MacPaint Manual.* Apple Computers, Inc., Cupertino, California, 1984.

[9] *MacWrite Manual.* Apple Computers, Inc., Cupertino, California, 1984.

[10] *NeWS Reference Manual, Version 1.0.* Sun Microsystems, Mountain View, California, April 1987.

[11] *PageMaker User Manual, Version 3.0.* Aldus Corporation, Seattle, Washington, January 1986.

[12] *Ready,Set,Go! User Manual.* Manhattan Graphics Corporation, Valhalla, New York, November 1986.

[13] *Reference to Microsoft Word, Word Processing Program for the Apple Macintosh, Version 3.0.* Microsoft Corporation, Seattle, Washington, January 1987.

[14] Adobe Systems, Inc. *PostScript Language Manual.* Addison-Wesley Publishing Company, 1985.

[15] Paul Asente. *Editing Graphical Objects Using Procedural Representations.* PhD thesis, Computer Science Department, Stanford University, Stanford, California, in preparation.

[16] Abhay Bhushan and Michael Plass. The Interpress page and document description language. *IEEE Computer*, 19(6):72–77, June 1986.

[17] Eric Bier and Maureen Stone. Snap-Dragging. *ACM Computer Graphics*, 20(4):233–240, August 1986.

[18] Lawrence Bohn and David Weinberger. Why not have it all. *UNIX Review*, 5(7):29–34, July 1987.

[19] Alan Borning. Defining constraints graphically. In *Proc. of ACM SIGCHI'86 Conference*, pages 137–143, Boston, Massachusetts, April 1986.

[20] Alan Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, October 1981.

[21] Kenneth P. Brooks. *Lilac: A Two View Document Preparation System.* PhD thesis, Computer Science Department, Stanford University, Stanford, California, in preparation.

[22] Donald D. Chamberlin. The Quill system. This is a new document preparation environment being developed at IBM Almaden Research Center. No reports have yet been published. The information discussed in this paper is based on personal communication.

[23] Donald D. Chamberlin and Charles F. Goldfarb. *Graphic Applications of the Standard Generalized Markup Language (SGML).* Technical Report RJ 5440 (55569), IBM Almaden Research Center, San Jose, California, December 1986.

[24] Pehong Chen. *A Multiple Representation Paradigm for Document Preparation.* PhD thesis, Computer Science Division, University of California, Berkeley, California, in preparation.

[25] Pehong Chen, John L. Coker, Michael A. Harrison, Jeffrey W. McCarrell, and Steven J. Procter. The VORTEX document preparation environment. In *Proc. of the second European Conference on TEX for Scientific Documentation*, pages 32–24, Strasbourg, France, June 19–21 1986. Published as *Lecture Notes in Computer Science No. 236* by Springer-Verlag, 1986.

[26] Pehong Chen and Michael A. Harrison. *Integrating Noninteractive Document Processors into an Interactive Environment.* Technical Report 87/349, Computer Science Division, University of California, Berkeley, California, April 1987. Submitted for publication.

[27] Robert C. Eckbardt. Illustrator: the tracer's edge. *MacWorld*, 4(6):117–121, June 1987.

[28] David Fuchs. Device independent file format. *TUGBoat*, 3(2):14–19, October 1982.

[29] George W. Furnas. Generalized fisheye views. In *Proc. of ACM SIGCHI'86 Conference*, pages 16–23, Boston, Massachusetts, April 1986.

[30] Richard K. Furuta. *An Integrated, but not Exact-Representation, Editor/Formatter.* PhD thesis, Computer Science Department, University of Washington, Seattle, Washington, September 1986. Published as Technical Report No. 86-09-08. A condensed version appears in *Proc. of International Conference on Text Processing and Document Manipulation* (Cambridge University Press), University of Nottingham, England, April 1986, pp. 246–259.

[31] Charles F. Goldfarb. A generalized approach to document markup. In *Proc. of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 68–73, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[32] Charles F. Goldfarb, editor. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML).* International Organization for Standardization, Geneva, Switzerland, 1986. International Standard ISO 8879.

[33] J. Gutknecht. Concepts of the text editor Lara. *Communications of the ACM*, 28(9):942–960, September 1985.

[34] Peter Hibbard. *User Manual for MINT: The Spice Document Preparation System, Version 2a(21).* Spice Project, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, April 1983.

[35] Alan Hoenig. TEX does windows — the conclusion. *TUGBoat*, 8(2), 1987. To appear.

[36] Gerard H. Holzman. Pico — a picture editor. *AT&T Technical Journal*, 66(2):2–13, March/April 1987.

[37] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. Direct manipulation interfaces. In D. A. Norman and S. W. Draper, editors, *User-Centered System Design*, pages 87–124 (Chapter 5), Lawrence Erlbaum Associates, Inc., Hillsdale, New Jersey, 1986.

[38] William Joy. *An Introduction to Display Editing with Vi.* 1980. Appears in UNIX 4.2BSD User's Manual.

[39] Brian W. Kernighan. PIC — a language for typesetting graphics. In *Proc. of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 92–98, Portland, Oregan, June 8–10 1981. A similar version under the same title appears in *Software: Experience and Practice*, 12(1), pp. 1–20, January 1982.

[40] Brian W. Kernighan. *A Typesetter-Independent TROFF.* Computer Science Technical Report 97, AT&T Bell Laboratories, Murray Hill, New Jersey, March 1982.

[41] Brian W. Kernighan and Lorinda L. Cherry. A system for typesetting mathematics. *Communications of the ACM*, 18(3):151–157, March 1975. Also available as Computer Science Technical Report 17, Bell Laboratories, Murray Hill, New Jersey.

[42] Donald E. Knuth. *The* METAFONT *Book.* Volume C of *Computers & Typesetting*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[43] Donald E. Knuth. *The TEX Book.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1984. Reprinted as Vol. A of *Computers & Typesetting*, 1986.

[44] Leslie Lamport. *LaTeX: A Document Preparation System. User's Guide and Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[45] Butler W. Lampson. *Bravo Manual.* Xerox Palo Alto Research Center, Palo Alto, California, 1978. Appears in *Alto User's Handbook*, Butler W. Lampson and Edward A. Taft (eds.).

[46] Michael E. Lesk. *Tbl—A Program to Format Tables.* Computer Science Techincal Report 49, AT&T Bell Laboratories, Murray Hill, New Jersey, September 1976. Also available in UNIX User's Manual.

[47] Michael E. Lesk. *Typesetting Documents on the UNIX System: Using the -ms Macros with Troff and Nroff.* Computer Science Internal Memo, AT&T Bell Laboratories, Murray Hill, New Jersey, November 1978. Also available in UNIX User's Manual.

[48] James H. Morris, Mahadev Satyanarayanan, Michael H. Corner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. ANDREW: a distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[49] Greg Nelson. Juno, a constraint-based graphics system. *ACM Computer Graphics*, 17(3):235–243, July 1983.

[50] Erfert Nielson. Mac graphics tools. *MacWorld*, 4(8):217–225, August 1987.

[51] Dan R. Olsen, editor. *Proc. of ACM SIGGRAPH Workshop on Software Tools for Uner Interface Management*, Seattle, Washington, November 1986. Available in *ACM Computer Graphics*, 21(2):71–147, April 1987.

[52] Mark Opperman, James Thomson, and Yih-Farn Chen. *A GREMLIN Tutorial.* Technical Report 87/322, Computer Science Division, University of California, Berkeley, California, December 1986.

[53] Joseph F. Ossanna. *Nroff/Troff User's Manual.* Computer Science Technical Report No. 54, AT&T Bell Laboratories, Murray Hill, New Jersey, October 1976. Also available in UNIX User's Manual.

[54] Brian K. Reid. Procedural page description languages. In J. C. van Vliet, editor, *Proc. of the International Conference on Text Processing and Document Manipulation*, pages 214–223, University of Nottingham, April 14–16 1986. Published by the Cambridge University Press. Reprinted in *ACM SIGGRAPH'86 Course Notes on Documentation Graphics*.

[55] Brian K. Reid. POSTSCRIPT and INTERPRESS. In *ACM SIGGRAPH'86 Course Notes on Documentation Graphics*, pages 49–63, Dallas, Texas, August 18–22 1986. Originally appeared on ARPANET Laser-Lovers mailing list.

[56] Brian K. Reid. *Scribe: A document specification language and its compiler.* PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, October 1980. Available as technical report CMU-CS-81-100.

[57] Erik Sandewall. *Theory of Information Management Systems.* Technical Report LITH-IDA-R-83-03, Department of Computer and Information Science, Linköping University, Linköping, Sweden, September 1983.

[58] Ben Shneiderman. Direct manipulation: a step beyond programming languages. *IEEE Computer*, 16(8):57–69, August 1983.

[59] Richard M. Stallman. EMACS: the extensible, customizable self-documenting display editor. In *Proc. of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 147–156, Portland, Oregan, June 8–10 1981. A somewhat extended version appears in *Interactive Programming Environments*, Barstow et al. (eds.), McGraw-Hill Book Company, 1984, pp. 300–325.

[60] Richard M. Stallman. *GNU Emacs Manual, Fifth Edition, Version 18.* Free Software Foundation, Cambridge, Massachusetts, December 1986.

[61] Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 1963.

[62] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.

[63] Warren Teitelman. A tour through Cedar. *IEEE Software*, 1(2):44–73, April 1984.

[64] Christopher J. Van Wyk. A graphics typesetting language. In *Proc. of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 99–107, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[65] Nicole Yankelovich, Morman Meyrowitz, and Andries van Dam. Reading and writing the electronic book. *IEEE Computer*, 18(10):15–30, October 1985.