he Macintosh's killer application was Aldus PageMaker. PageMaker and the Mac seemed made for each other. Apple needed an application to showcase its graphical user interface innovations. PageMaker was what designers had wanted (though probably not knowingly) all along, and the Macintosh was sufficiently advanced, graphically, that it inspired Aldus to create PageMaker. This was quite lucky for Apple, since without PageMaker, Macintosh (and Apple) probably wouldn't be around today.

Most of the important early Macintosh applications—MacWrite, MacPaint, PageMaker, etc.—were based on the What You See Is What You Get (or WYSIWYG) paradigm. As a result, WYSIWYG is deeply rooted in the Macintosh culture. Mac users expect that the way their documents look on-screen is the way they will look when printed. They expect that they can move elements of their documents around with drag-and-drop. Experimenting is easy and risk-free, and there is immediate feedback. Consequently, Macintosh encourages the user to experiment, tweak, and play. This brings forth perhaps the most important Macintosh ideal: the user, not the machine, should be in control. And this hard-to-describe feeling of not being at the mercy of the machine is the principle reason Mac users are so passionate about their choice of computer.

As with the graphical user interface, the WYSIWYG paradigm has not gotten significantly better since the Macintosh shipped in 1984. The idea of WYSIWYG as a sort of interactive print preview, a way of experimenting with and proofing a layout is sound. I would never want to go back to pre-WYSIWYG computing. Nevertheless, the WYSIWYG paradigm as realized by today's software has inherent flaws—or tradeoffs—that show no sign of being overcome. In this article I hope to argue not why WYSIWYG is bad—for I don't think that at all—but why for many tasks a completely non-WYSIWYG solution is preferable. Mac users, in particular, have little exposure to the non-WYSIWYG world, and while it is perfectly okay to live a sheltered existence, there is also some benefit in understanding the non-WYSIWYG world, even if you never have occasion to spend time in it.

The non-WYSIWYG World

Believe it or not, there is a world of non-WYSIWYG software that is thriving. I'm not talking about typewriterish software like Bank Street Writer and AppleWorks (for the Apple II). Though perfectly good pieces of software, they provide nothing in exchange for forcing their users to work in non-WYSIWYG environments. Other solutions, the most popular of which are probably HTML and TeX, do not provide WYSIWYG (as the base representation), but provide a range of functionality not found in the WYSIWYG world.

HTML: HyperText Markup Language

When you hear WYSIWYG in relation to HTML, the language behind the Web, the first thing that comes to mind is the increasingly popular collection of WYSIWYG HTML editors—like CyberStudio, PageMill, and HomePage. The purpose of these programs is to make the creation of Web pages a WYSIWYG process, even though HTML was never intended to be a WYSIWYG language. HTML was designed as a markup language, which simply means that the commands, or markup, are imbedded directly in the content. HTML tags are for denoting the logical structure of content. In “pure” HTML, one does not specify fonts, styles, and sizes, but rather the meaning behind parts of the document. There are tags to specify different levels of headings, to emphasize text, and to indicate that text is part of a quotation or computer input/output. With HTML 4’s Cascading StyleSheets, page authors can even define their own structural markup, and specify how it should be rendered under different circumstances.

This makes sense for several reasons. HTML was designed to be viewed on many different platforms, from text-only terminals, to Macintoshes, to handheld computers. There’s no way to guarantee which fonts will be available—or even if there will be a choice of fonts at all. The page creator embeds information about the structure of his content; the renderer (browser) determines how best to interpret that structure into a format that its host computer is equipped to display. Usually this means that headings come out large and bold, emphasized text is italic or bold, quotes are indented, etc. But there is no law that says it must work this way. The browser is encouraged to tailor the actual look of the page to the machine it is running on. It is perfectly possible, and reasonable, for the page to render differently on the screen of its reader than it did on the screen of its author. With printed paper documents, this was never an issue; with electronic ones, it is. WYSIWYG programs, designed as an electronic extension to paper, generally do not take this into consideration.

Ideally, the on-screen display needn’t completely match the printed output. Although I’m not sure if anyone currently takes advantage of it, HTML has the ability to specify different styles for viewing and printing. For instance, italic text is often very difficult to read on-screen. Color, however, is readily available. Thus, emphasized text could appear in a different color on-screen, but change to italic text that looks great when printed. Similarly, just because a document looks good printed in Times, there’s no reason people shouldn’t see it in their browser using a nice screen font like Geneva, Espy Sans, or Verdana.

A Brief History of TeX

TeX (pronounced “tekh”) is a powerful typesetting language created by Stanford Professor Donald Knuth, author of the seminal work in Computer Science. Knuth designed TeX several years before the Macintosh and Aldus PageMaker came along. Being a mathematician, he made it especially good at typesetting mathematics, for which it to this day it has no peer. Although this makes TeX especially good at typesetting scientific documents, its appeal is by no means limited to them.

TeX is somewhat similar to HTML in that it is a markup language. The user creates a text file with their document and adds TeX commands to format it as desired. TeX is also different from HTML in some important ways. It is completely free, and the source code is available—with extensive documentation. What this means is that every implementation of TeX uses the same rendering code. TeX’s feature set is frozen (except for bug fixes), so one can be

guaranteed that a TeX document you write today will look identical on any current (or newer) implementation of TeX—on any platform. TeX includes its own fonts, so output really is identical on all platforms. Unlike HTML, TeX is a full-fledged programming language. Whereas HTML code is “rendered” with a browser, TeX code is “run” with the TeX program, similar to the way PostScript code is executed in a printer or other PostScript interpreter.

Knuth says that TeX is for producing beautiful documents, and he went to great lengths to build in a lot of typographic know-how. The hyphenation algorithm alone was the subject of a PhD thesis. Since TeX understands more about typography (and especially mathematical typesetting) that most typographers, it takes care of most of the details for you. (You can, of course, override it if you want to achieve a specific look.) It automatically handles ligatures and both horizontal and vertical spacing. It is careful not to leave a heading hanging at the bottom of a page. You don’t have to remember how many spaces to put after a period or how much space to put before and after each type of heading. (This is especially true when using a formatting package like LaTeX.) If you change the base font size of your document, TeX updates all the vertical spacings and margins accordingly. Another example is that according to conventional typographic rules, the first paragraph following a heading should not be indented. In a word processor, you’d likely have to apply some styles to get this effect. With TeX/LaTeX, it happens automatically.

Thus, documents produced with TeX generally look much more professional than those created with a WYSIWYG word processor. And, surprisingly perhaps, they are often much easier to create. The fundamental difference in philosophy is that WYSIWYG word processors try to give you as much control as possible over your document, in an easy-to-use, visual manner. This freedom generally means that you must do most of the formatting work yourself. TeX does as much as possible automatically, generally with better results than if the user had done it. Describing to TeX a format that doesn’t know, is considerably more difficult that in a WYSIWYG word processor, although in the end TeX is vastly more flexible.

Often, TeX can figure out what you want, without your having to specify the details. For instance, it has a “&” command, which is similar to a tab stop in a word processor. Simply inserting ampersands in your text will usually cause TeX to pick the correct alignment. Often there’s no need to specify what you want at a low level, like “right-aligned tab at 6.5 inches from the left margin.”

In other cases, there is no way TeX can tell what you want, so you have to be very specific. For instance, quotation marks must be inserted using either double back-quotes or double apostrophes (`` or ''), depending on whether they are opening or closing (same thing goes for single quotes). On the one hand, this is more work than you generally need to do to insert quotation makers in a WYSIWYG program. On the other hand, the “smart quotes” algorithms in word processors often curl the quotes the wrong way.

Extensions to TeX

While most word processors have macro languages (or AppleScript) for extending their capabilities, TeX is a programming language, so it’s relatively easy to customize and extend it. Many packages for extending TeX can be found at the Comprehensive TeX Archive Network <http://www.ctan.org>, ranging from packages that help with placing graphics, to packages for card players, circuit designers, and more.

One of the most popular such extensions LaTeX. Released in 1985, LaTeX (which stands for Layout TeX or Layman’s TeX depending on who you talk to) is a powerful collection of TeX

macros aimed at simplifying the creation of regularly formatted documents. LaTeX is so popular that it's included in most TeX distributions. If you want to create a document in a standard format (like a letter, report, article, or book) it's probably even easier to do it with LaTeX than with a conventional word processor. LaTeX makes it easy to deal with logical structures such as footnotes, cross references, different levels of headings, lists, quotations, and more. Because of LaTeX's underlying TeX architecture, the quality of the typography is very high.

Limitations of WYSIWYG

While I certainly do not find the current WYSIWYG word processors ideal, I also think that the WYSIWYG paradigm has inherent problems that mean it will never be as good as non-WYSIWYG for certain things.

A WYSIWYG word processor will always be slower than editing in a text editor. My dad still likes to use Bank Street Writer because it takes less time to boot the Apple IIGS and Bank Street Writer than it does to launch Word 98 on his PowerMac. The IIGS is more responsive, too. I use BBEdit for much of my writing because I'm impatient. BBEdit never makes me wait.

Since a WYSIWYG interface has to be provide commands for formatting and layout in its menus and toolbars, there is no way it can be as optimized for text processing as a text editor is. This is unfortunate, since probably only 10% of the time it takes to compose a document is spent on formatting—maybe less. Yet much of a word processor's interface is cluttered with infrequently used commands. I think the Twiddle command (from BBEdit) for swapping the positions of letters or words is far more useful to have readily available than a Drop Cap command, for instance. Yet word processors have a fancy commands like Drop Cap —buttons for them, even—and lack basic text processing commands.

When I'm composing a document, I don't want to be concerned with the way it looks. Just because the actual document uses a small, hard-to-read-on-screen script font, doesn't mean I should not be able to edit with a large font designed for on-screen use. When doing layout, it's nice to see where page breaks fall and how columns and margins look. When I'm writing, they just take up screen real estate would be better-used for letting me see more of the document.

In principle, one could use stylesheets and macros to work around the display vs. printing font issue. Most word processors have commands for hiding margins and page breaks. Still, these seem like clunky solutions to a problem that wouldn't exist if the composition and preview environments were separate.

Some kinds of things are just plain hard to do with WYSIWYG. For instance, if you are printing a document with facing pages, you often want the bottom line of text on each of the pages to align exactly (vertically). There's no way to tell a WYSIWG program stuff like this, unless its designers specifically thought of the command you want and included it in the user interface. TeX can make short work of this problem, as well as more complicated ones like making the ratio of height/width of each page equal to the golden ratio—simply because you can tell it exactly what you want. Here are some effects that would be difficult to do quite as nicely in a WYSIWYG word processor or page-layout program.

lthough WYSIWYG equation editors are very easy to use, they are decidedly underpowered and inefficient compared to a text-based approach like TeX's. It's very inconvenient to locate every single symbol in a menu or palette; you must constantly switch from the keyboard to the mouse, and back. Then you have to find a way to imbed the equation in you word processor. Most of the time, the equation editor is separate from the word processor, which means that it's a pain to use it for lots of small equations. Since the equations behave like graphics boxes, they often throw off the line spacing. Furthermore, the output of WYSIWYG equation editors is usually greatly inferior to TeX (although in principle this need not be the case).

Deferring Layout Decisions

Probably the biggest criticism of WYSIWYG is that it really means "What you see is what you've got." Once you've created something in a WYSIWYG environment, it's generally difficult to change how it looks. While formatting (fonts, sizes, styles, rulers) decisions can easily be changed with suitably defined stylesheets, it is very difficult to defer layout decisions. The structure and overall look are frozen, unless you change them all by hand.

Say you've created a list of definitions in your document. Maybe you've applied a term style (like bold) to each term you've defined and a definition style (like plain) to each definition. You manually entered a colon between each term and its corresponding definition.

WYSIWYG: An acronym for What You See Is What You Get

After entering 500 definitions, you change your mind. It would be much better if the term and definition were separated by a blank line. You want to (for whatever reason) put the definition in parentheses, make it italic, and remove the colon. You also want horizontal lines above and below each definition to visually separate them in the list, and the definition

should be right-aligned at a tab stop (which you hadn't created before). Oh, and because your document makes extensive use of obscure definitions, you want each term marked so it can be cross-referenced.

WYSIWYG (An acronym for What You See Is What You Get)

With a WYSIWYG word processor, one would have to find a way to select all the definitions and apply some kind of macro to make the above transformations. Although it would probably be doable, it would be a lot of work that you shouldn't have to do. Most of the time would be spent recognizing the different parts of the definition, munging text, and locating the definitions—which might be scattered throughout the document (or multiple documents!).

TeX approaches this problem differently. An experienced TeX user would probably create a definition construct. Perhaps he would enter the definitions like this: (Notice that the term and definition each are arguments to the \def macro.)

```
\defWYSIWYGAn acronym for What You See Is What You Get
```

Changing the layout, font, styles, and any other aspect of the definition format is as simple as changing the the way \def is defined. This can be particularly useful for articles that may be published in different journals. Each journal can have its own way of formatting certain mathematical constructs, and all the different formats can be generated from the same TeX file.

Customizability

Many of LaTeX's features, such as automatic formatting of lists, are now available in conventional word processors (this was not true when LaTeX was released). Even so, I generally find LaTeX easier to deal with than the WYSIWYG approach, mainly because the latter requires you to give logical instructions to the word processor using WYSIWYG actions. For instance, in Word 98's list mode, pressing <return> creates a new list item. It is not at all obvious how, then, one should create a single list item consisting of multiple paragraphs. Hitting <return> twice takes you out of list mode. This is confusing!

If you want to change an item's symbol from a • to a + or - (perhaps you are making a pros/cons list), it requires seven mouse clicks in Word 98 (including summoning a contextual menu and navigating two nested dialog boxes). In LaTeX, you would simply change: `\item` to `\item[+]` or `\item[-]`. (Square brackets denote an optional argument to a macro. If you leave them out of `\item`, LaTeX figures out which symbol to use.)

Portability and Versatility

WYSIWYG software tends to not be very portable. ClarisWorks and Microsoft Office run on Macintosh and Windows, WordPerfect has a completely different version for each, and there are many single-platform word processors. There is no WYSIWYG word processor that runs on Mac, Windows, Unix, Linux, NeXT, and Amiga. But HTML and TeX do. Further, since they're both plain-text formats the files are 100% compatible—no file translation or encoding

necessary. Since the renderer and editor are separate, it's easy for browser manufacturers and providers of TeX implementations to add value for their customers without sacrificing compatibility or changing the file format.

Using plain text as a file format has other advantages. Dynamic data-driven Web sites are prevalent today because webmasters can use an ensemble of text-processing tools to create HTML files on-demand. It's unlikely that such a variety of tools for this would exist if HTML were a proprietary WYSIWYG format. And although their pages aren't served up electronically, TeX users enjoy much they same flexibility. It's far easier to write scripts that assemble documents from data using TeX than a word processor, primarily because of the former's markup-like nature.

The future of the Web seems to lie in XML, eXtensible Markup Language. XML provides an extensible means of adding logical markup to documents. For example, an XML version of Hamlet might have tags for speeches and stage directions; the browser/client, if it understood these, could treat them more specially than if it simply received formatting tags. In addition, XML should allow for much more intelligent Web searching, since it provides a means for the computer to understand the semantics of the documents it indexes.

Conclusion

Open standard non-WYSIWYG languages like HTML and TeX bring us some of the promise of OpenDoc. Although they don't free us from the application software paradigm, they break the paradigm of having a one-to-one correspondence between file types and applications. In each case, a multitude of specialized tools can be applied towards document creation, rather than requiring a single monolithic application that is a jack of all trades but master of none.

Non-WYSIWYG systems also have their problems. They are more difficult to learn, and since you usually have to work through a browser or previewer, experimentation is not encouraged. Even coupled with an environment like Textures (see the review in this issue) I would not want to use TeX for an irregularly formatted document like a newsletter. It is just too painful for doing creative work.

WYSIWYG and non-WYSIWYG environments each have their uses, and neither is likely to replace the other. Each paradigm makes a different tradeoff between what sort of work the computer and user need to do. In non-WYSIWYG systems, many of the typographic details are handled by the computer, but it is more difficult to do layout tasks. WYSIWYG systems, on the other hand, give ultimate freedom—provided your software supports the specific feature you want and you are willing to take care of details that the computer could do better. In the end, the most important thing is that we have a choice.