Introduction

A few years ago, in the Stone Age of the World Wide Web, editing of HTML documents was seen as a significant problem due to the nature of the web and the lack of SGML compliant tools. In the intervening time, there have arisen a number of systems designed for editing HTML documents, some based on non–HTML or SGML markup systems. In this time, there has also become an increasing use of active documents which are not statically edited but are generated on the fly. In this paper, we discuss the problems which exist with editing HTML documents and possible ways around them. publishing, objects, ATK.

History

Rumour has it that the original web project had an editor which worked under the NeXTstep operating system. This editor never saw the public light of day and many of the original web documents were written by simple attacking at files with 'vi' or 'emacs'. Since the HTML language is fairly simple, this was not too great a hardship. However, persuading non–technical people that they would need to learn a new language, albeit a simple one, did not work too well. To address this problem, several solutions sprang up, mostly involving conversion between one editing format and HTML. However conversion between different documents is a procedure which can often cause errors. A more practical solution was to produce an editor which could read and write "native" HTML. 'htmltext' was an editing package I wrote using the Andrew Toolkit which addressed this concern.

HTML Editors

The original 'htmltext' editor was written after a persusal of the public domain marketplace showed very little software in the area of HTML editors. During the initial implementation and distribution of htmltext, there were no other available editors (tkWWW was available in a pre–release kind of stage, but was extremely tricky to install and get working). The htmltext package was implemented to meet the goal of being an easy–to–use WYSIWYG style editor. The andrew toolkit sprung to mind as the ideal platform with which to implement this package due to its highly developed text object which provided almost all of the desired functionality. Since that time, the htmltext package has been heavily used at a large number of sites and bundled with the AUIS distribution. At a later time, the ATK developers within IBM Rochester decided to independently implement an HTML editing tool, aiming at providing an HTML 3.0 (then called HTML+) compliant editor. Building on either of the two versions of HTML editing provided by these two groups, Nathaniel Borenstein created a fully functional web browser: the initial version of this browser was implemented with only a few hundred lines of code!

Both of these HTML editing packages were based on the concept of WYSIWYG: this is an underlying feature of the text object provided by ATK. Another approach, as exhibited by *HoTMetaL*, is to provide an editor which is oriented towards the HTML language, providing a structured view of the document. To see the conceptual difference in design expressed by these implementations, we need to look a bit further into the specification of HTML and the markup which it provides: both physical and logical.

Logical and Physical Markup

Logical markup allows the author to describe the structure of a document and the semantic meaning of segments of text. By using such markup, the document becomes portable to different viewing methods and becomes more easily program parseable. Physical markup, on the other hand, allows the author to describe the appearance of a document, with no reference to the meaning behind the appearance. The author can place styles across sections of text dictating their appearance under a particular viewing model. Physical markup allows the author to really achieve WYSIWYG editing: the final document viewed by anyone is guaranteed to look the same as it was intended by the author.

There are problems with both logical and physical markup and they address very different uses of text. Logical markup is geared more towards distribution of a document between different viewing models and providing semantically analysable documents. As such, the author has little control over the final appearance of the document and it may appear in a radically different form to that the author intended. Physical markup allows the author to express their view of how a document should appear, but limits the presentation of the document to the viewing model with which the document was generated.

Because of the advantages in portability, HTML was based upon a logical markup model, specified with SGML. This allowed documents to be viewed via simple text based telnet connections, from complex graphical browsers or even through different media such as speech. The semantic information available within logical markup was also of vital interest, allowing programs to receive documents and be able to interpret useful information (as an example, think of browsers which can automatically determine the author or maintainer of a document).

Hypertext Structure

Beyond the logical structure of a document, there is also a structure implied between a group of documents. HTML allows some of this to be specified by placing relation types on links. The entire problem of designing a well structured web is complex. It is further complicated by the problem that not all documents may be viewed (for example dynamically generated documents).

As an experiment, we created a file−browser style interface for viewing the links of documents. This allowed the user to see at any time all of the links coming from a document, although the only form which the prototype allowed was a tree based structure. Even in its prototype form this interface, biased towards the links rather than the content, was extremely useful for "digging−down" through links to locate a particular document, and seeing the range of subjects which could be accessed. This style of interface should be provided to the author as a tool for viewing the structure of a web

Logical & Physical Editing

Physical style editing is a fairly trivial matter: many editors exist which ably perform this function. Logical style editing is more rare, and often disliked by authors. It is far too easy to fall into the trap of playing with styles in order to achieve the desired look of a document, while losing the semantic information contained within the logical styles. To provide a system which edits logical styles, there are really two approaches:

1. Provide a WYSIWYG style interface and then map operations into the logical styles during a save. This is how editors such as tkWWW, htmltext and Word for Windows (with the HTML extensions) operate.
2. Or alternatively, provide an interface which allows the author to specify the logical styles and then map the document structure into a WYSIWYG appearance. This approach is exhibited by HoTMetaL, and other editors which are based upon the SGML specification.

The problem with performing the translation at save time is in determining what to do with incorrect HTML. The SGML specification of the HTML language defines which style tags are allowed, and within what context different tags are allowed. One of the most common mistakes is to place tags outside of their legal environments. For example, placing an underline environment around a heading environment. Structurally, this makes no sense as a heading is a top−level construct which cannot be contained within another environment. However this sort of mistake may be allowed during editing, especially when using a system such as a macro package for Word for Windows which provides no intelligence during the edit stage.

htmltext

The htmltext editing package is built upon a text object which provides physical editing. For this reason, a structural interface could not easily be implemented (although tools such as the *content* object, which provides a view of the table of contents, work surprisingly well). However the approach taken allowed a very simple−to−use editor to be built: an editor with a user interface following the traditional model of physical editing.

The htmltext package provides a natural interface to editing HTML. All HTML markers are hidden from view and interpreted into changing the appearance of the text during editing. Characters which might be interpreted as special by HTML (such as '&', '<' and '>') are automatically translated into their HTML "safe" forms during the save operation without any intervention required by the author. Any inline images are displayed within the editor. In effect, it presents a fully WYSIWYG interface onto editing HTML.

By using standard Andrew Toolkit mechanisms, the editor provides for a simplistic overall view of the structure of a single document: showing a view of all titles within a document and an indication to their level. Using similar mechanisms, the editor could also provide spell checks and preview facilities.

More importantly, because of the mechanism used by the Andrew Toolkit to map styles into a physical representation on−screen (*templates*), the system allows any user to customise the appearance of any and all styles. The htmltext package was shipped with a template which made the editor's appearance similar to that produced by NCSA Mosaic.

To implement the HTML styles, the htmltext code constructed a complex set of functions based around the 'environment' and 'style' objects of ATK, which allow arbitrary styles to be placed on regions of text. This was a poor design decision and the method taken by the html package developed by IBM was far better: subclass the environment object to provide the html specific information.

Another choice which badly affected the htmltext package was to avoid using the SGML side of HTML. Since a physical model was being developed, it seemed unneccessary and overly complex to integrate an SGML library into the code. Instead a simple table was used which mapped between HTML style names and ATK style names. Of course, this loses all of the information about structure which the SGML definition provides and hence allows the user to place any style at any location. It was up to the routine which wrote out the databaobject to ensure legal HTML.

With an SGML parser built in, the system could have used the active flag feature of the ATK menus to indicate which logical or physical styles were permissable on any given selection, removing the need to decode what the author *might* have intended during the write operation. The design currently being developed by the author is to create a general SGML object, subclassed from the text object, which provides the control of menus and textual environments based on a combination stylesheets and SGML definition files. HTML would then only need to be a subclass of the SGML object: all of the basic HTML operations provided by the SGML definition and the HTML object concentrating on only the advanced functions such as controlling links between objects and inlining images.

Publishing

Creating a document for use in the World Wide Web involves more than just editing a file containing HTML. To actually "publish" the document − make it visible within the current Web, possibly indexed and integrated into other documents − requires a more integrated solution than simply providing an editor. It would also be preferable to provide more support for editing Web documents than that provided by simple file system permissions: the Web itself is a network distributed system so why limit its construction to local system usage? It may be desirable to allow particular authors to update and edit documents, even if they are at a site remote from the server and filesystem in question.

The original specification of the World Wide Web protocol, HTTP, provided mechanisms for retrieving files from the Web "locked" for editing, and for storing the files after the modifications had been made. These were the CHECKIN and CHECKOUT methods of the protocol. Unfortunately, very few Web servers implemented these methods and so there was very little point in adding support into the HTML editors.

However there are many advantages to a publication mechanism for managing document updates and any such mechanism would ideally be integrated into the editor being used.

The most obvious advantage is providing proper document maintenance control over all of the documents within the Web. Editorial control may be applied to areas of the web automatically, whilst allowing other areas to be freely updated by authors anywhere on the network.

More interestingly, by having a system which knows when any document changes, it becomes trivial to create automatic systems to update search indexes, and to notify any relevant parties (e.g. caches) about updates to documents.

Dynamic Document Publication

The editing and publishing mechanisms discussed are fine for static, HTML documents. But one of the more powerful advantages of the Web is the ability to create dynamic documents: documents which are generated at the time of the network access, produced by executing a program such as a CGI script.

Any solution which is given for managing a Web of documents must be able to cater to these two different aspects of document generation. If the Web maintainer needs to switch between different editors dependent on how a document is constructed, then the system in use does not provide a complete solution. Editors such as "ez" or "emacs" can switch into whichever mode is appropriate for the document at hand however there is currently no facility within the HTTP protocol for determining the source of the dynamic document, or even for determining if a document has been generated.

An Object Based Web

A powerful view of the web is as a model of linked distributed objects. Allowing generic objects, rather than limited markup documents allows complicated ideas to be expressed within the web: for example allowing "documents" to be generated on the fly, or even for documents to contain active code which is executed within the browser. It is only after content and quality negotiation between a server and client that the final form of an object is ever resolved. As an example, a typical HTTP

URL"/home" could resolve to an HTML document describing the organisation within which the browser or client is currently operating. The same URL could resolve to an image, or a VRML model of the organisation, perhaps a model of the building in which you work. The resolution depends entirely on the preferences of the user, and represented to the web server by the browser in use.

An object–oriented programming model fits this viewpoint well, especially if the implementation allows dynamic object resolution. For this reason, the Andrew Toolkit is an excellent development tool for a World Wide Web interface. For similar reasons, Hot–Java from Sun is a good platform (although Java concentrates more on the benefits of active documents containing executable code).

Mapping Web Objects into ATK

ATK provides the potential for a browser which allows editing of any part of a compound document, or a document of any media type. For example, raster editing of images, audio editing of sound bytes and plain textual editing of HTML (or other markup) documents.

The HTML text itself is obviously mapped to a class such as "text" or the more advanced "html", or "htmltext" classes. ATK also provides image and audio insets which can be used for such media types. A table inset can be used (and has been used, by the IBM rochester version of the html classes) to provide HTML3.0 tables. It is also feasible to create a class, similar to the ATK *ness* object, which interprets Java code

Currently, all of these classes read and write streams in Andrew Datastream format only, with an extra routine to print in either troff code or postscript. Ideally, these routines would all be integrated into a generic input/output system which can utilise different data formats. This would allow the editor to immediately integrate tables and any other ATK inset without the trouble of creating a new class which understands the new data format.

Despite this drawback, the object–orientedness of ATK already provides a system which allows easy integration: html objects may be loaded into the mailreader automatically, when HTML documents are sent by mail and other classes which operate on ATK objects can be easily applied to the HTML editor. As an example, the gesture recognition package, contributed with the Andrew Toolkit distribution, can be used as is with the HTML editor.

A Publishing Environment

By placing full HTTP support within the ATK toolset, it becomes possible to use the Andrew user applications as part of a publishing environment. By simple opening a URL in the same way as a file is currently opened, the system could perform a retrieval of the document, marked as read–only. When the user wishes to edit the document, the HTTP interface could automatically checkout the document, obtaining any authorisation as neccessary. The user would not need to know anything about the underlying HTTP operations, or the "true" location of the file within the UNIX filesystem.

By adapting such tools as the current Andrew directory browser, it also becomes possible to browse the structure of the Web documents, viewing the hypertext connectivity between developing documents. From an overall hypertext structural view, it should be possible to push into any type of object and either view or edit it, just as the directory browser currently operates.

Furthermore, by using the current Andrew dynamic loading system, it is possible to add new media types to the web without causing undue complexity in adding support for such a media type in the editing system. This would require that ATK classname to object instantiation be enhanced to use MIME typenames, rather than arbitrary filename extensions.

Obstacles to a Publishing System

The Andrew User Interface is not an incredibly popular interface and the size of the distribution required to effectively use AUIS is formidable. Terry Gliedt in the Linux community has provided an effective solution to this problem, providing pre−packed subsets of AUIS which work fresh out of the wrapping paper.

The interface is not the only area which needs significant development, but the server technology and the protocol need to be examined extensively to provide the best server−side handling of the system with minimal, if any, changes to any of the server protocols.

Conclusion

ATK Provides a mature and extensive base on which to develop a system for viewing, editing and managing of web documents. It allows for extension to new media types as they become available and already supports a vast majority of the currently available media objects.

Unfortunately, the current implementation of some of the inner workings of ATK force such a system to be fairly clumsily implemented and requires a fair amount of development to achieve its true potential.

Despite these difficulties, we hope that a truly integrated Web solution built with ATK will be shortly developed, building on some of the previous work mentioned in this paper.