Fifth International World Wide Web Conference
May 6-10, 1996, Paris, France

WebWriter: A Browser-Based Editor for Constructing Web Applications

Arturo Crespo
Stanford University
email: crespo@cs.stanford.edu

Eric A. Bier
Xerox Palo Alto Research Center
email: bier@parc.xerox.com

Abstract

Constructing server-based Web applications requires creating both Web pages and programs that generate Web pages. This requires a knowledge of the Hypertext Markup Language (HTML), the Common Gateway Interface (CGI) protocol, and a programming language, such as C++, Python, or Perl. While this is not a barrier for programmers, it is for non-programmers. This paper describes *WebWriter*, an integrated system for constructing Web applications that supports the creation of Web pages by non-programmers. WebWriter includes a direct manipulation Web page editor, the *WebWriter Editor*, which runs in a Web browser as a CGI service, and the *WebWriter Page Generator*, which creates new pages as an application runs. As in HyperCard, users create a Web application as a stack of pages, where each page can contain output regions that are filled in at runtime by a script. This paper describes the WebWriter system, issues of server-based authoring tools, and some example applications.

Introduction

There are many ways for programmers to produce interactive applications that run from a Web page. By learning HTML, the Common Gateway Interface (CGI) protocol, and a programming language, such as C++, Python, or Perl, the application designer can create applications that run on the server. Similarly, by learning HTML, and a browser-supported language, such as Java (supported by Netscape Navigator 2.0 [Netscape95b]) or python (supported by the Grail browser [CNRI96]), the designer can produce applications that run on the client (browser). This works fine as a way for programmers to produce heavy-weight applications. However, we believe that many non-programmers will want to write simple applications when this becomes easy.

Using HyperCard [Apple87] as our main model, we have constructed a system, *WebWriter*, that supports the construction of simple interactive applications without the need to learn HTML. In HyperCard, the user builds up an application as a stack of cards, where each card can contain text, images, buttons, and content that is computed on the fly by HyperTalk scripts. Similarly, in WebWriter, the user builds an application as a sequence of Web pages, where each page can contain text, images, HTML forms, and content that is computed on the fly by WebWriter scripts.

The user constructs the layout of each page of an application using the *WebWriter Editor*, a CGI program that runs in any browser that supports tables, such as Netscape Navigator [Netscape95b]. Hence it runs on any hardware/software platform supported by such browsers. While there are already many HTML editors in the world, we created our own; we wanted an editor whose source code we could control, that was easy to extend, and that would run in a Web browser on all of the platforms we use.

The user adds application behavior using the WebWriter Editor by selecting a behavior type and filling in a form for that type. For example, the user can select a file listing behavior and fill in a form to specify the files to list. For more general programming, the user can select a scripting language, such as the Unix shell language, and type a program in that language.

In addition to the Editor, the WebWriter system for non-programmers includes the *WebWriter Page Generator*, a server-based CGI service that creates new pages as an application runs. Because they use the Page Generator, applications produced by WebWriter run as CGI programs on a Web server and hence can be used from many platforms and in many Web browsers.

Finally, for programmers, we provide WebApp, a library of C++ routines for constructing CGI programs that maintain state from page to page.

The WebWriter Editor and Page Generator have been implemented as C++ programs, using the CGI protocol, running on SunOS 4.1.3 on SUN SPARCstations. We are running WebWriter on both CERN and Apache HTTPD servers. All of the figures in this paper show WebWriter running in Netscape 2.0.

The rest of this paper is organized as follows. We review related work. We describe the application model that WebWriter supports--a model that includes five main parts for each application. We show how a user creates these five parts with WebWriter. To illustrate further, we describe two applications that were created with WebWriter. Having described the WebWriter approach, we discuss some of its advantages and disadvantages. We describe its implementation. Finally, we discuss security issues and give our conclusions and plans for future work.

Related Work

WebWriter was inspired by the HyperCard system [Apple87], which succeeded in making it possible for end-users to build customized applications. WebWriter borrows several concepts from HyperCard: applications are built up as an ordered "stack" of documents; the documents can contain text, graphics, and scripts; and the scripts are simple enough to be built by non-programmers. Unlike HyperCard, WebWriter applications are created, modified, and run in a Web browser.

The WebWriter Editor is a server-side HTML editor, like ssed [Glicksman95] and WebEdit [Pier95, Pier96]. Like ssed, WebWriter augments the document with editing handles. Unlike ssed or WebEdit, WebWriter is targeted at users who do not know HTML, and hence shows each part of the document as interpreted HTML (bold text is bold, images are displayed in their actual appearance, etc.) and allows point-and-click editing of regions in clickable maps. In addition, WebWriter supports the creation of pages in an extended version of HTML that adds behavior to pages through a scripting language.

The WebWriter Editor has features that are found in standalone HTML editors, such as HoTMetaL PRO 2.0 [SoftQuad95] or NaviPress [NaviSoft96]. Like HoTMetaL, WebWriter can show HTML tags mixed in with a preview of formatting, including the layout of tables, font changes and so on. In fact, WebWriter's view is more accurate in some respects because it uses the formatting capabilities of an actual browser at all times.

Existing systems that extend HTML to add behavior include MAWL [Ladd95] and JavaScript [Netscape95a]. The MAWL system allows extension tags, including variables that will have a value at runtime, to be mixed in with ordinary HTML in order to allow the static HTML to be checked for correctness at creation time. WebWriter also combines static and computed content, but for a different reason--to allow the HTML to be edited interactively by a user who is not familiar with HTML. JavaScript makes it possible to add behavior to a page that will be executed on the client-side. For example, in page that includes JavaScript, clicking on a button in a form may execute JavaScript code on the client, instead of submitting the form to the server as is done for traditional HTML/CGI pages. Using JavaScript currently requires being comfortable with HTML.

Existing systems that provide direct manipulation creation of clickable images include Adobe PageMill [Adobe95], Microsoft FrontPage [Microsoft96], and MapEdit [Boutell95]. Creation of clickable images in WebWriter is less convenient than in these tools, but runs in a Web browser and hence on a broader variety of platforms.

The W3Kit System [Burchard95], like the WebApp C++ library used in WebEdit, provides facilities for saving and restoring the state of runtime data structures. However, W3Kit works in the less popular Objective-C while WebApp works in C++.

The WebWriter Application Model

WebWriter creates applications that can be expressed as a stack of Web pages, where these pages are visited in a specific sequence, and where each page can contain both static content that is always present and computed content that is generated on the fly. This model is built on the traditional model of server-side CGI script applications, which we characterize as follows: the application generates a page that contains an HTML form; the user fills out the form and presses a button; based on the user's input, the system generates another page with another form; the user fills this form and presses a button; and so on.

The WebWriter model differs from this general model in that the pages that will be successively shown to the user as the application runs are not entirely generated at runtime by a program; the static parts of each page are created in advance by the designer using a text editor. This static content takes the form of a *template page*, which includes the specification of locations within the page, called *dynamic areas*, where computed content should be inserted at runtime to produce the final page that will be displayed to the user. For each dynamic area, the designer specifies a *script* that will be run to provide the computed content for that dynamic area. Each script may make use of *variables*, which describe the input provided by the user on previous pages or the arguments in the URL when the application was called. Finally, for each template page, the designer names the *next template* that should be shown to the user after the user has submitted the form on the page generated from the current template.

Thus a WebWriter application has five key elements, as shown in figure 1:

1. a set of template pages written in HTML (numbered 1 in figure 1),
2. one or more dynamic areas on each template page (numbered 2),
3. a set of named form elements used as variables (numbered 3),
4. a script for each dynamic area (numbered 4), and
5. a pointer on each page indicating the page to use next (numbered 5).

Note that a page may refer to itself, allowing an application to show versions of the same page in a loop. We will describe the representation of these elements in the implementation section.

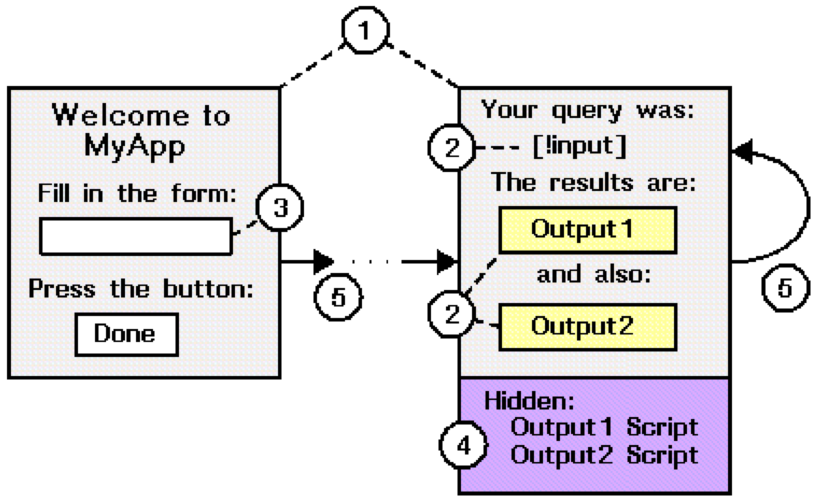

Figure 1. The form of a WebWriter application showing template pages (1), output areas (2), form elements (3), scripts (4), and the next template pointer (5).

Constructing a WebWriter Application

As described in the previous section, WebWriter applications have five key parts. In this section, we describe how WebWriter supports the construction of each of these parts: creating the template pages, adding dynamic areas, creating variables, specifying scripts, and linking a page to the next page. In addition, we describe two final steps of application-building: saving and loading a stack and running the application.

Creating the Template Pages

To create template pages, the designer uses the WebWriter Editor, which is a general-purpose HTML editor that runs in a Web browser. This editor has a two-column interface as shown in figure 2. The left column, called the *document area*, contains the page that is being edited. The right column contains a control panel that has three parts. The upper part provides file commands. The middle part provides commands that can be used at any time. The lower part contains commands that are specific to the currently-selected object. Users start the editor by clicking on a link that jumps to the WebWriter start page.

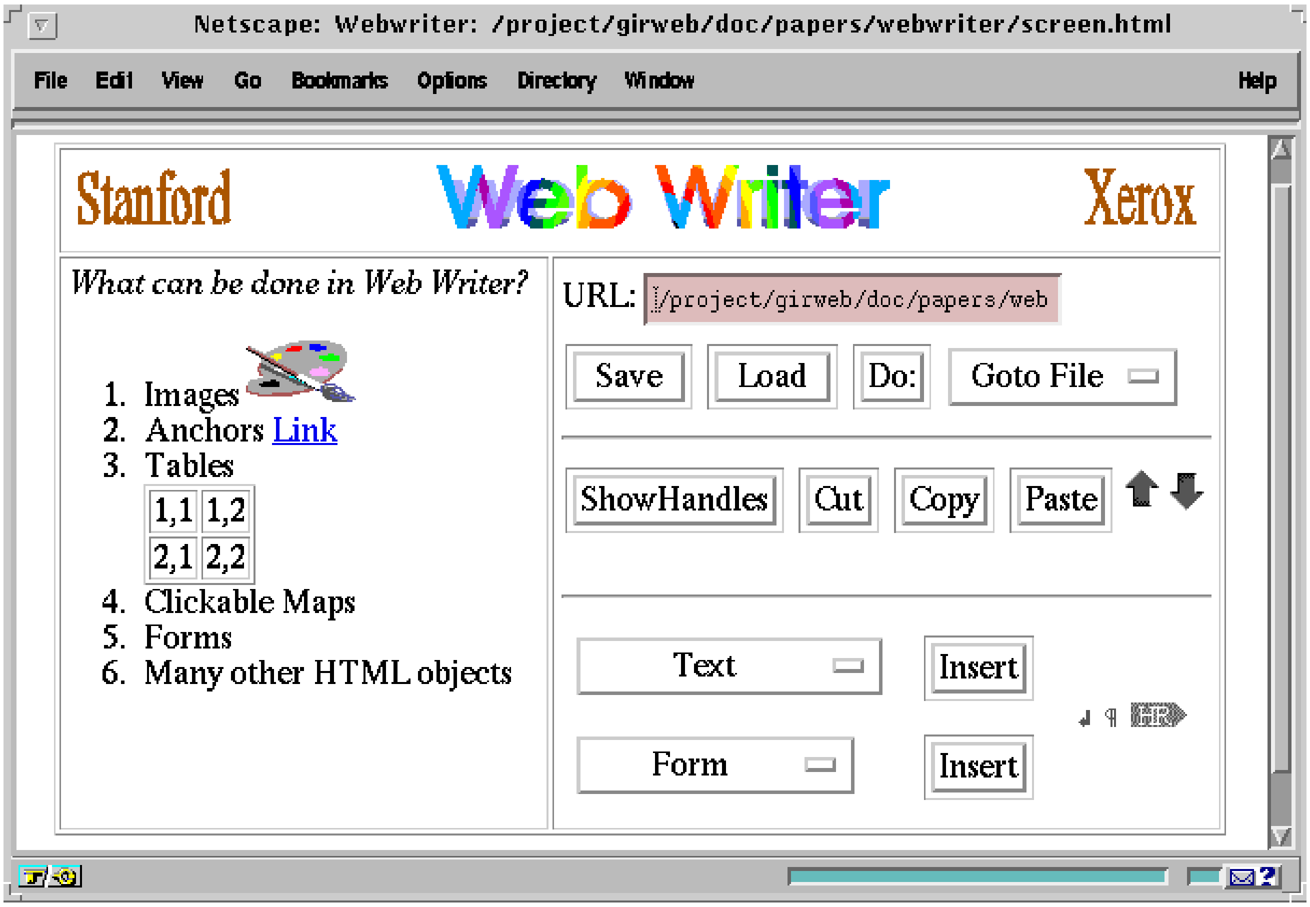

Figure 2. The WebWriter screen has the document in the left column and control panels on the right.

Insertion Points and Handles

In editing mode, WebWriter displays the current page as interpreted HTML together with additional objects, called *insertion points* and *handles*, that aid in the editing process. Figure 3 shows part of the document of figure 2 with handles and insertion points.

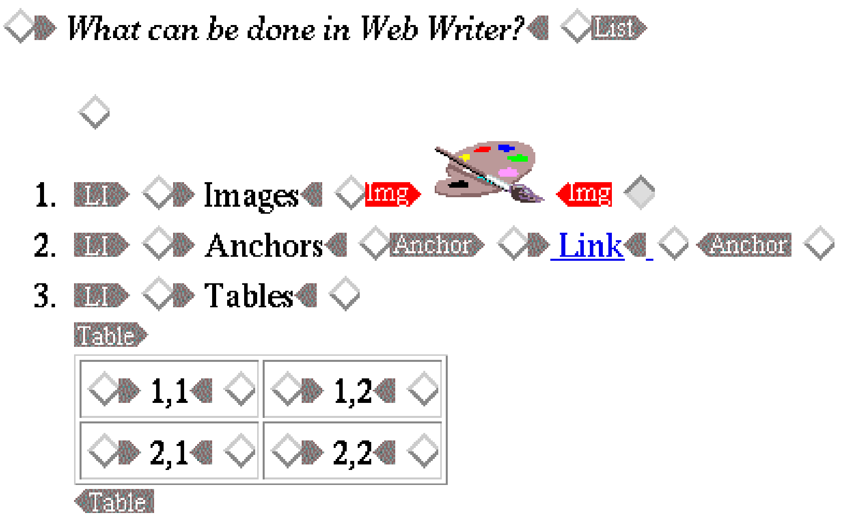

Figure 3. Handles and insertion points are added to the document during editing.

Insertion points, shown as diamonds in figure 3, indicate those points in the page where new content can be added. Only one can be selected at a time. The currently selected insertion point is the gray diamond to the right of the image object.

Handles are clickable shapes with a gray or red background that generally correspond to HTML tags. Each handle or matched pair of handles represents an object in the document hierarchy. For example, the Table handles in figure 3 represent an object that in turn contains four objects, each of them text. When the user clicks on a handle, the corresponding object becomes selected, the handle becomes red, and a form appears in the bottom control panel for editing the selected object. Clicking the HideHandles button causes both handles and insertion points to disappear, allowing the user to preview the final appearance of the page.

Adding and Editing Content

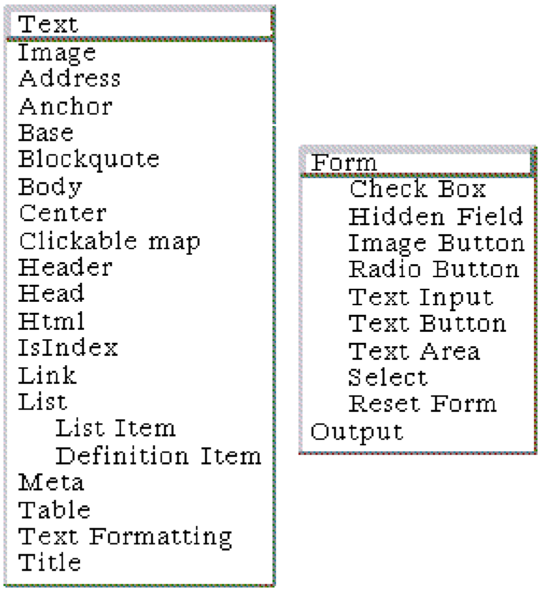

Figure 4. The user can add any of the 32 HTML object types from these two menus.

To add content at any insertion point, the user clicks on that insertion point and then selects an HTML object type from either the Text or the Form menu, which are located as shown in figure 2. These menus include most of the tags available in HTML 2 as extended by Netscape plus such HTML 3 features as tables. The elements of these menus appear in figure 4. The user next clicks on the appropriate Insert button. WebWriter creates an empty object of the selected type and inserts it. At the same time, a form for creating that object type appears in the lower right section of the control panel. The user fills out the form and clicks Done and the new object appears in the document area.

While most objects in WebWriter share a similar style of creation user interface, more general user interfaces for creation are possible in the WebWriter architecture. In particular, WebWriter supports direct manipulation creation of clickable images, as described in appendix A.

Each object type has its own creation form. WebWriter currently has 32 forms, corresponding to the 32 object types of figure 4. Figures 5 (a), 5 (b), and 5 (c) show forms for objects of type text, list, and image. The creation form for text allows specification of face (italics, bold, keyboard, or code). The form for a list provides for several types of lists (bulleted, numbered, definition, directory, and menu). The image creation form allows the URL of the image to be selected from a menu or typed. The user may also specify a text string to show in place of the image in browsers that don’t support images and may specify how to align the image relative to other objects on the same document line.

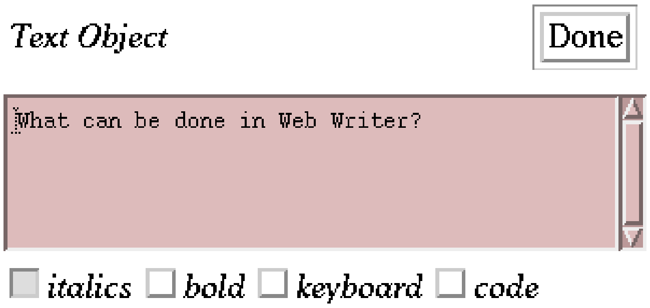

Figure 5 (a) The creation form for text.

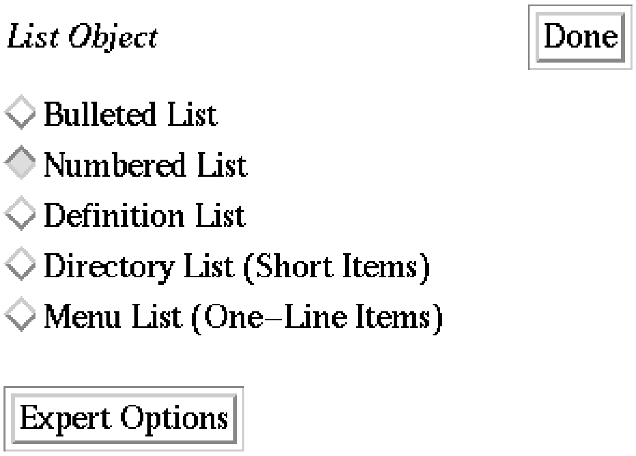

Figure 5 (b) The creation form for a list.

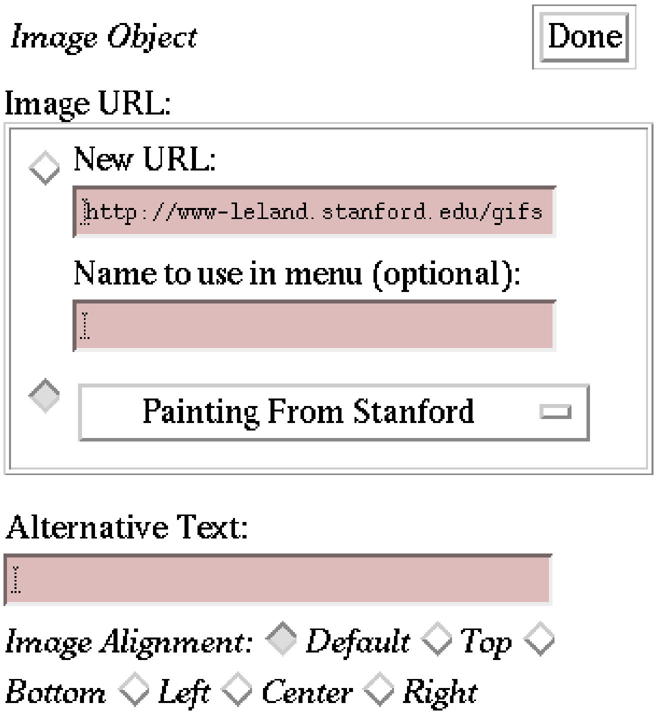

Figure 5 (c) The creation form for an image.

To modify an existing object, the user clicks on one of its handles. At this point, the creation form for that object appears, filled in with the current parameters of that object. The user can then edit these parameters and click Done. The modified version of the object appears in the document area. The user can also cut and paste an object by clicking on its handle, clicking the Cut button, selecting an insertion point and clicking Paste.

Placing Dynamic Areas

Dynamic areas are inserted into a template page to indicate regions of the page that will be filled in at runtime by a program. There are two types of dynamic area supported by WebWriter: *variable areas* and *output areas*. To create a variable area, the user simply types a name preceded by "[!" and followed by "]" in any text element of the document. For example, the user might add a string such as "Files in directory: [!directory]". The meaning of this string is that, at runtime, the string in square brackets will be replaced by the value of the variable "directory". Variables will be described in the next section.

An output area is a special WebWriter object (the only special tag that WebWriter adds to standard HTML). The user creates an output area by selecting Output from the Form menu (recall figure 4) and clicking Insert. WebWriter then adds a placeholder for the output area at the current insertion point. The output area object will be replaced at runtime by the output of a script, formatted as HTML and inserted at that point in the document. The creation form for each output area asks the user to specify a script to run. The contents of the output area (in between the Output and Output handles in the document area) specifies how the output of the script should be formatted into HTML. Both scripts and formatting are discussed in more detail below.

Defining Variables

By default, each page of a WebWriter application is an HTML form. As a form, each page can contain user interface elements, such as buttons, check boxes, radio buttons, and type-in fields. The creation form for each of these elements asks the user to specify a name. By naming a user interface element, the user automatically creates a variable of that name. The value of this variable is then available to subsequent pages of the application.

The value of a variable is a string based on the state of the corresponding form element. For a textinput, textarea, or hidden field the value is the textual contents of that element. For check boxes, radio buttons, and select menus the value is the name of the selected box, button, or menu item respectively. If multiple selections are allowed, the value is a space-separated list of the names of all selected elements.

For example, if the user creates a type-in field (e.g., an HTML <input type="text"> field) named "directory", then subsequent pages of the application can refer to the value of this field as [!directory] in any text string.

Scripts and Formatting for Output Areas

Scripts

The user specifies a script by filling in the creation form for an output area. The user selects from a small-set of built-in modules that perform functions such as listing all of the files in a directory, performing arithmetic calculations, or executing a Unix shell command. Each of these modules has a form to fill in to specify its behavior. For example, the file-listing module asks for a file name pattern, and the arithmetic calculator asks for an expression, such as (4.3*12.2)/17.0. For the Unix shell module, the user types a Unix shell command, such as "ls ~bier/*.html".

In any of the module type-in fields, the user can include one or more [!variable] expressions. Each such expression is replaced by its value before the module is run. For example, to produce a Fahrenheit to Celsius converter, the user could select the calculator module and give it the expression

([!temp]-32)*5/9.

Formatting

In order to display the output of a module WebWriter must know how to convert it to HTML. While each module could perform a default conversion to HTML, giving the user control over the conversion allows each module to be used in a larger number of contexts.

In the WebWriter system, formatting is specified by example and is done a line at a time. The user indicates how each line of output should appear by constructing a document fragment, between the handles of an output area, that contains the string [!LINE] in one or more places. At run-time, the WebWriter Page Generator copies the formatting string in place once for each line of output. All instances of the variable [!LINE] in the first copy of the string are then replaced by the first line of output, all instances of [!LINE] in the second document fragment are replaced by the second line of output, and so on.

As an example of scripting and formatting, consider the file listing application of figure 6, which was built using WebWriter. When it is run, this application behaves as follows: The user types a file name into the text input area of the first page, as shown in figure 6 (a). The user types a carriage return. A second page appears showing the list of files in the requested directory, as shown in figure 6 (b).

Figure 6 (a). Displaying page 1 of the file lister in a Web browser.

Listing of /project/girweb/figs/coins/:
coins.htm
dime.gif
nickel.gif
penny.gif
quarter.gif

Figure 6 (b). After typing Return in the text input of figure 6 (a).

Using WebWriter, a designer can implement this file lister as follows: The first page is constructed as shown in figure 7 (a). Notice that the text input field, into which the user will type the file name, has the label "files." As a result, the variable [!files] will have the file name as its value when the next page is generated. The second page is constructed as shown in figure 7 (b). Notice that the variable [!files] both appears in a text object at the top of the page, and as part of the Unix shell command that will generate the contents of the output area.

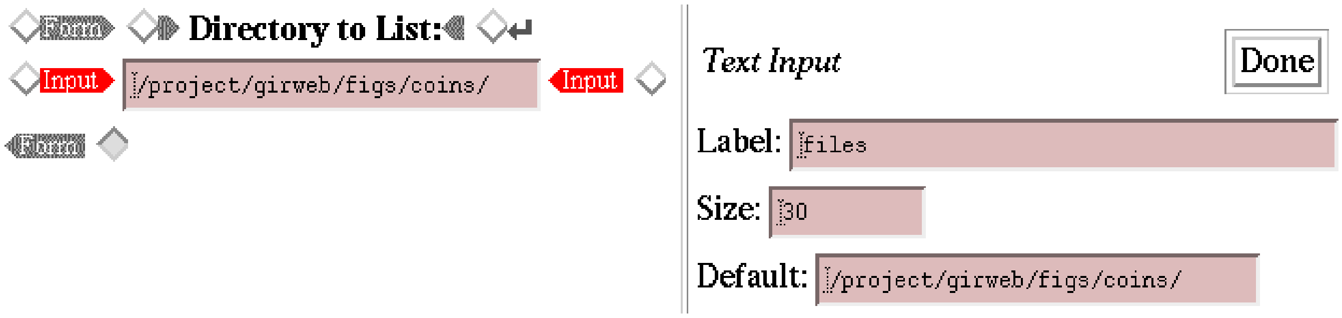

Figure 7 (a). File lister page 1. Adding a text input, named "files".

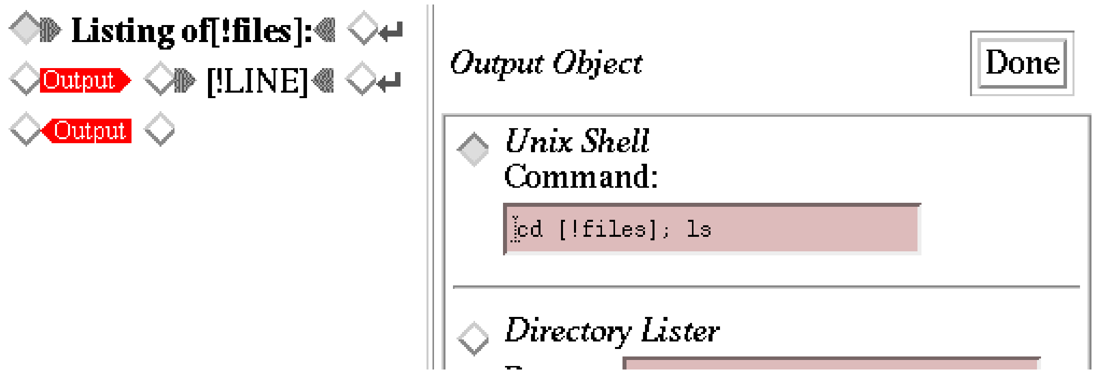

Figure 7 (b). File lister page 2. Writing the script for a new output area.

This file listing example uses a very simple formatting string consisting of "[!LINE]" and the line break symbol, so it just shows the file names separated by line breaks. However, we can easily make a more interesting application by using a richer formatting string. For example, to produce an application that displays a name, a GIF image, and a check box for every file in directory [!files], we can use the formatting pattern shown in figure 8, where the check box has the name [!LINE] and the GIF image specifies a URL of the form "http://servername[!files][!LINE]". Note that any variable name may appear in a formatting pattern; [!LINE] is special only in that it has a different value for each copy of the pattern.

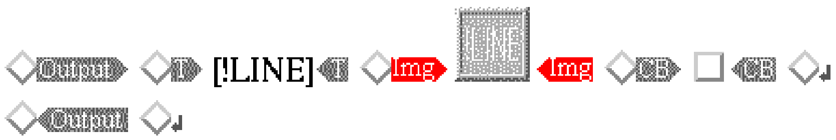

Figure 8. A document fragment for formatting a file listing.

Flow of Control from Page to Page

Most WebWriter applications contain one or more HTML forms. Like other HTML elements, form objects (such as Form itself, check box, text input, and buttons) are added to the page by selecting the object type from a menu and clicking Insert.

When adding a Form object, the user must decide which Web service will be called when one of the form's buttons is pressed. WebWriter allows the user to choose any service. However, by default, the form calls the WebWriter Page Generator, which allows the user to create a custom Web service based on the WebWriter application model. If the Page Generator is selected, the user is also asked to name a next template. This is the template that WebWriter will display, at runtime, after the user has submitted the form on the current template. By default, the next template is set to be the next template in the stack, or the current template if there is no next template. If necessary, the user can enter a complete URL to specify the next template, allowing control to jump to any template in any stack.

Saving and Loading the Stack

The user can save an individual page to disk by typing its file name into a field and clicking the Save button. The user can group a set of pages into a stack by creating a page that contains links to all of pages that are to be part of the stack. WebWriter provides a special command, LoadStack, which reads such a stack page, extracts all of the URLs from its links and then adds a menu of these URLs to the persistent control panel. The user can use this menu to jump from page to page in the stack. WebWriter also provides NextPage and BackPage buttons that make it easy to move forward and backward in the stack. Note that any HTML page can be used as a stack page, so LoadStack can also be used to make it easy to edit all of the child pages of an arbitrary page.

Once a stack is loaded, WebWriter can load individual template pages for viewing and editing. These pages may or may not have been created by WebWriter. While the WebWriter HTML parser is much like other HTML parsers in most respects, it is different in that it understands WebWriter-specific tags, such as <OUTPUT>, and creates corresponding WebWriter objects. Also, the WebWriter parser can collapse several HTML tags into a single WebWriter object, where that is appropriate. For example, the string <i>Bold-Italic</i> can be collapsed into a single text string object that has both its bold and italic properties set to true. Collapsing in this way reduces the number of editing handles that the user needs to look at and simplifies editing.

Running the Application

Once a WebWriter application is created and saved to disk, it can be run in one of two ways. If the first page of the application has no dynamic areas (i.e., no [!variable] strings and no <OUTPUT> regions), then the application can be started just by going to the URL of its first page with the Web browser. WebWriter provides a GoTo File command so that jumping to a page can be done in a single click.

On the other hand, if the first page of the application is active (i.e., does contain [!variable] strings or <OUTPUT> regions), it must be called via the URL of the WebWriter Page Generator. This URL must specify the value of any variables in the page. For example, to run the second page of the file listing application described above on files /a/*.gif, without requiring the user to type this file name pattern, we could call:

> http://server.xerox.com/cgi-bin/wwpagegen?
> template=http://server.xerox.com/apps/list-2.html&files=/a/*.gif

Other Example Applications

Above, we described an application that displays all of the GIF images in a directory. This section describes two other WebWriter applications: a file directory editor and a distributed chat program.

WebNeptune File Directory Editor

WebNeptune is a file directory editor, inspired by the Neptune [Xerox79] directory editor used on the Xerox Alto computer in the late 70's. Using HTML tables to achieve two-column formatting, it displays the files in one directory in its left column and the files in another directory in its right column. Each file name has a check box next to it. When the user checks one or more files and clicks DoIt, all of the checked files are moved to the directory listed in the other column.

WebNeptune is a two-page WebWriter stack. The first page, shown in figure 9 (a) allows the user to type file paths for two directories and click Display. The second page, shown in figure 9 (b), has two visible output areas, one for each file listing. When the user clicks the DoIt button, this second page calls the Page Generator on its own template, resulting in a new listing showing which files have moved.

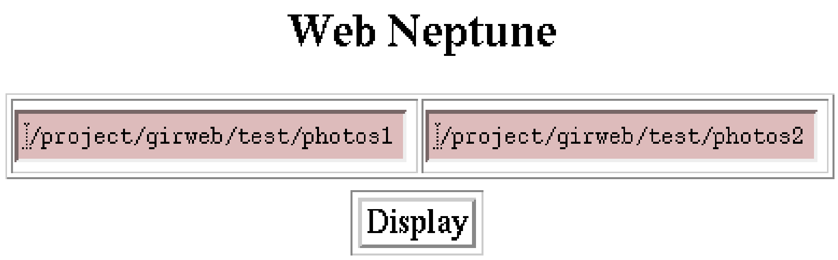

Figure 9 (a) The first page of WebNeptune.

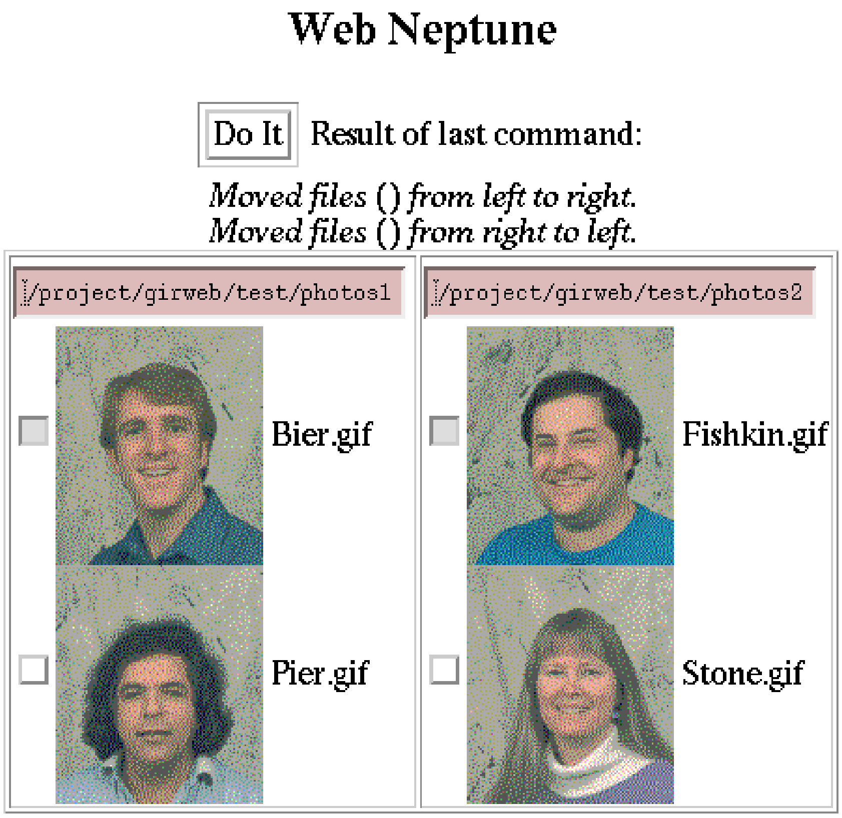

Figure 9 (b) The second page of WebNeptune.

The implementation of WebNeptune is similar to the file lister described above, except that WebNeptune lists two directories and provides for moving files from one directory to another. Page 2 of WebNeptune does most of the important work. This page contains three output areas, one hidden and two visible. The hidden output area, positioned at the top of the page, produces no actual output. Its script contains the Unix commands "cd [!files1]; mv [!choice1] [!files2]/; cd [!files2]; mv [!choice2] [!files1]/". These commands move files from one directory to another. The variable [!choice1] is a space-separated list of those files that the user selected. If the user hit DoIt in figure 9 (b), [!choice1] would be "Bier.gif". The second OUTPUT area displays the left hand file listing, using the command "cd [!files1]; ls" and formatting like that of figure 8. The third OUTPUT area is just like the second, but applies to the second directory, [!files2]. Page 2 of WebNeptune refers to itself as next template, so the program runs in a loop, allowing the user to move files repeatedly.

Chat Program

The Chat Program is a stack with two pages. The first page, shown in figure 10 (a), allows the user to send messages. The second page, shown in figure 10 (b), allows the user to read the messages sent from the first page. The user can move from one page to the other at any time.

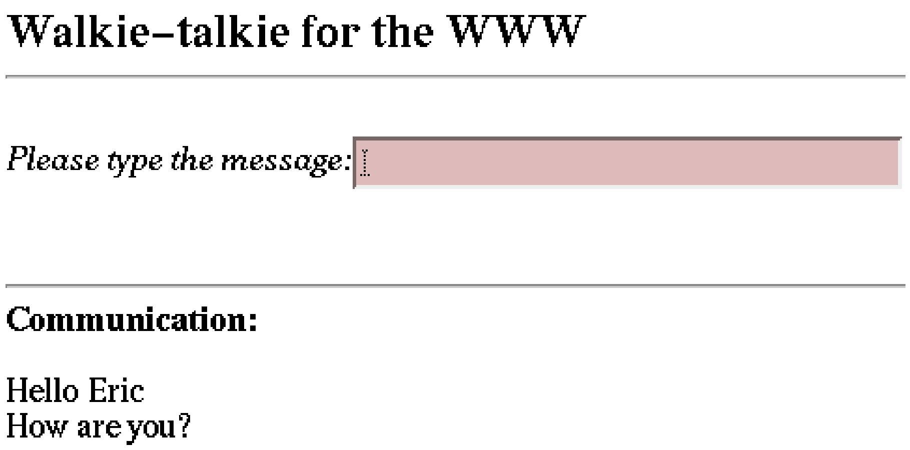

Figure 10 (a) The transmitter of the Walkie-Talkie application.

Figure 10 (b) The receiver of the Walkie-Talkie application.

The walkie-talkie works by storing all messages in a file, /usr/tmp/wt-file accessible from the computer where the Page Generator is running. The first page of Chat simply prepends the message typed by the user to that file. The second page displays the file in an HTML page and uses client-pull to allow automatic updates of the list of messages. Figure 11 shows the template for the receiver page of chat. The script of the selected output area is the Unix shell command "cat /usr/tmp/wt-file". Note that the output area turns the result into HTML by adding a line break,
, after each line of output.

Figure 11. The template page of the Walkie-Talkie receiver.

This application requires the special meta tag

<META HTTP-EQUIV=REFRESH CONTENT="1; URL=http://machine/wt.html">

at the top of the file. While this tag would not make much sense to a non-programmer, this example shows that WebWriter allows users to get at more of the power of HTML as they become more advanced. Also, a simpler version of this program, without automatic updates, can be built without the use of the meta object.

Discussion of the WebWriter Approach

We made several important design decisions for WebWriter: We implemented a new HTML editor, instead of using an existing one. We built an editor that runs in a Web browser as a Web service, instead of as a stand-alone editor or client-side applet. We display the document and control panel side by side in the WebWriter Editor. Finally, we hide HTML from the user. This section describes some of the advantages and disadvantages of the resulting system. Some of the advantages are:

1. *Platform independence*. Because WebWriter runs in a browser, it runs on many platforms and so is accessible to a large set of end-users. Currently, not all browsers support HTML tables, but this situation is improving.
2. *Upgrading as HTML evolves*. As new tags are added to HTML, the Editor user interface can easily track the changes by adding a creation form and handle image for new tags. New HTML forms can be generated by a small amount of code.
3. *Little HTML knowledge needed*. Many people don't want to learn HTML. WebWriter documents are created by filling in forms, not by typing angle brackets. However, because WebWriter displays handles around each object type, users will end up knowing about the structure of HTML after spending some time with WebWriter.
4. *Output appearance constructed interactively*. While programmers are able to write programs that generate HTML code, such programs generally have to include quoting characters interspersed with the HTML, making it even less readable than ordinary HTML. For example, in C++, a programmer must write awkward strings such as `cout << "<b>She said \"I'm tired of these quotes!\"</b>" << endl;`. In WebWriter, the static parts of an output page are placed in a separate template file, generated using the WebWriter Editor.

Some of the disadvantages of the WebWriter approach are that the user can't edit text in place, that WebWriter doesn't work well on long pages, that handles and insertion points can be distracting, and that adding new elements to a page may sometimes take a lot of steps. Below, we describe these problems and suggest some remedies.

1. *Edit text in place*. While WebWriter displays the text of a document in the document view area, this text cannot be edited in place. Instead, the user must click on a handle and then edit a copy of the text in a text creation form. This is less direct and more time consuming than editing in place. We plan to address this problem by reimplementing some or all of WebWriter as a Java applet.
2. *Long pages*. When the user is editing at the bottom of a long page, the WebWriter control panel is no longer visible, because it has scrolled out of view. We plan to address this problem using

frames, like those currently supported in Netscape 2.0. The control panel will appear in one frame while the document appears in a separately-scrollable frame, so the control panel will always be visible.

3. *Handles, insertion points, and white space*. When WebWriter displays a page, it adds handles, insertion points, and line breaks, which disrupts the appearance of the page. As described above, the WebWriter HTML parser partially addresses this problem by collapsing multiple tags into one tag where possible. We are also discussing a technique that would display the handles and insertion points within an object only when the user clicks on it, as in ssed [Glicksman95].
4. *Too many steps*. Adding a new object to a page can take several steps. For example, to add a text anchor, the user must select the anchor type, click Insert, select or type a URL, click Done, select the text type, click Insert, type the text, and click Done, for a total of 8 steps. By providing an accelerator button for anchors and allowing text to be typed in the anchor object, this will be reduced to 4 steps: select anchor, select or type a URL, type the text, and click Done.

Implementation

The WebWriter Editor

The WebWriter Editor is a server-based CGI service written in C++. Each time it is called, it generates a two-column page, like the one shown in figure 2. Each page contains a form and buttons which, when pressed, call the WebWriter Editor again to generate the next page.

Each page generated by the Editor contains HTML code that represents the appearance of the current document in the left column, the user interface in the right column, and a generally long hidden tag, containing an encoded version of the C++ data structures of the WebWriter Editor program. This encoded information represents the current state of the editing session. The hidden tag looks like

```
<input name="STATE" type="hidden" value="[0][0][1][][-1][-1][-1][][0][0]">
```

where the square brackets and numbers represent C++ data structures, as described below. The encoded information is needed because CGI services are stateless. Every time the CGI service is called a new copy of the program is started without any knowledge of previous interactions. The encoded information allows the CGI service to restore the state of all relevant data structures, giving the illusion that the program is running continuously. The encoding of the state of the data structures is done automatically by the Web Application Development Environment (WebApp) which is described below in its own section.

The main structure encoded in the hidden tag is the content tree--a representation of the content of the page being edited and indexes into it that define the selected object and the insertion point. Additionally, the hidden tag includes the cut buffer (a second content tree, which supports Cut, Copy and Paste) and global settings of the editor (such as the URL of the page being edited).

The WebWriter Editor defines a C++ class for most HTML tags. However, some tags are joined together to reduce the complexity of the Editor’s user interface. For example, the tags <b> and <i> are available as the bold and italics options, respectively, of the WebWriter text object. The content of each page is represented by a tree of WebWriter objects, where compound objects, such as List, Table, and Center are internal nodes of the tree, and simple objects, such as List Item, Image, and Text are leaves.

After executing each user-specified action, WebWriter generates the next page by traversing the content tree and asking all objects to print themselves in a format appropriate for the document area of the editor. Then WebWriter prints the general control panels and, if an object has just been selected (or inserted), asks that object to print its creation form. Finally, the state of all relevant data structures is put in the hidden field, State, which is added to the end of the new page, and the WebWriter program returns.

The next time WebWriter is called (after the user clicks on a handle or control panel button) the editor restores the state of all data structures from the information in the hidden tag. Next, the WebWriter performs the action requested by the user. There are three types of actions: changes to the document state, such as insert, select, modify, cut, copy, and paste; file operations such as save and load; and changes to the document view, such as hiding handles or showing handles.

We describe the implementation of these three action types by considering an example of each: Insert, Save, and HideHandles. When the user clicks on Insert, WebWriter inserts a new object of the requested class into the content tree data structure at the position designated by the insertion point and sets this new object to be the selected object. Then WebWriter generates a new page as described above; the new page will show the new object and its creation form. To Save, WebWriter asks each object to write an HTML version of itself to a file stream; WebWriter generates a new page including a message saying if the save succeeded or failed. To HideHandles, WebWriter changes the value of a Boolean variable to indicate that handles should be suppressed when each object generates its view in the new page; it then generates a new page without handles.

The WebWriter Page Generator

When a WebWriter application is running, each new page is assembled by the WebWriter Page Generator, another server-based CGI C++ program that knows how to read a template page, run the scripts specified in that template page and create a new page for display to the user. This Page Generator takes the place of the hand-written CGI program that would otherwise have to be produced to create a server-side Web application.

The Page Generator takes two arguments: the URL of a template page, and a list of variable=value pairs that represent the application variables as described in the section Constructing a WebWriter Application. The template page has been written in a version of HTML that has been extended in two ways: First, the template page may contain tags of the form:

<OUTPUT SCRIPT="...">*formatting string*</OUTPUT>

where each OUTPUT tag represents a WebWriter output area. At runtime, the entire output area is replaced by the HTML string that results from executing the program specified by the SCRIPT argument, and then converting it to HTML using the information in the formatting string. Second, the template page may contain strings of the form [!variable]. Such strings will not appear literally in the newly generated page. Instead they will be replaced by the value of variable, expressed textually, where the value of each "variable" is passed as part of the URL as described above.

The Page Generator generates a new page on the fly as it scans through the template page. As it reads, it copies each character to its output stream, until it encounters a [!variable] string or an <OUTPUT> tag.

When it encounters a [!variable] string, it looks up the value of the variable from the variable=value pairs that it received. If it finds a match, it replaces [!variable] with the value as a string. If it finds no match, it generates an error page.

The SCRIPT field of an output tag is a representation of a program name and arguments to that program. Formatting strings were described in previous sections. When WebWriter encounters an OUTPUT tag, it extracts the SCRIPT field and formatting string, runs the named program on its arguments, formats the results using the formatting string and writes the results to its output. The program is run in a separate Unix process so that it can crash without crashing WebWriter. If a crash occurs, WebWriter emits error messages in place of the program output. WebWriter continues reading the template page until it ends.

The Web Application Development Environment (WebApp)

The Web Application Development Environment, or WebApp, is a set of C++ classes that keeps the state of applications that use the WWW as a user interface and provides a library of commonly-needed functions. The WebWriter Editor relies on most of the facilities of WebApp.

WebApp maintains state by asking every C++ object to save its relevant variables by *pickling* (representing as ASCII text) its data structures and saving them in a hidden field of a form. To restore the state, WebApp takes the pickled values saved by an object, converts them back into C++ data structures and makes them available to a newly created object of the same class as the original object.

WebApp can pickle integers, strings and lists. To avoid having problems with different architectures and with special HTML characters, we limit the set of characters used in the pickled form of objects to the numbers 1 to 9, the letters A to F and the minus sign, open bracket, and close bracket characters. Integers are transformed into their ASCII representation. Strings are treated as a list of characters, where each character is represented by its ASCII number in hexadecimal. Lists are represented by enclosing the elements of the list, which may include other lists, in brackets. For example, the STATE value described at the top of this Implementation section was produced by the Pickler.

The supporting libraries of WebApp include: clients for the HTTP and FTP protocols, HTML Widgets, an HTML parser, management of CGI variables, and the Pickler.

Security Issues

There are two security issues with WebWriter, related to saving files and generating new pages.

The WebWriter Editor can save files to directories that are writable by the server on which it runs. Thus, access to WebWriter must be limited to those who have write permission on those directories. At our installation, we achieve this security by requiring the user to produce a user name and password when the WebWriter Editor is first started in a browser. This is achieved by placing an access file (called .htaccess by default) in the directory (often called cgi-bin) from which WebWriter runs; the access file specifies which groups of users may log on. Access files are supported by many HTTPD servers, including the W3C httpd [W3C96] and Apache [Apache96] servers that we use.

Similarly, the WebWriter Page Generator can run a number of programs on the server. This set of programs currently includes the Unix shell interpreter, which among other things, can be used to move and delete files. Thus access to the full-strength WebWriter Page Generator must be restricted to those who already have access permission to use such commands in the domain of our server. Again, we use an access file to require a password the first time a user calls any application that uses the Page Generator. Because the browser caches the access cookie once a correct password has been given, subsequent uses of WebWriter require no further passwords until a new browser is created.

We are producing a version of the Page Generator that implements only a safe subset of scripts. This version will be more appropriate for distributing outside of firewalls and can be used without requiring a password.

Future Work

WebWriter is already useful for editing small HTML pages and applications. However, there are many improvements that would lead to a better tool. As described in the Discussion section, we plan to add in-place editing of text, use HTML frames to support long pages, reduce the number of handles and insertion points and reduce the number of user interface steps. In addition, we will improve security by providing a safe version of the Page Generator. We are augmenting the WebWriter application model to allow each page template to have multiple possible next templates. We plan to test WebWriter on a larger number of users to see how well it works in practice and how it can be improved.

Some or all of WebWriter will be re-implemented in Java. WebWriter will be extended to allow scripts written in JavaScript, which in turn will provide support for client-side applications in addition to the server-side applications currently supported. We plan to produce a version of WebWriter that will be useful to programmers as well as non-programmers.

In WebApp, we plan to further automate the generation of code that saves object state. The programmer will specify in a high level language which objects and which data members of those objects are relevant from interaction to interaction and WebApp will generate all the necessary code for keeping application state.

Conclusions

We have built WebWriter, a prototype Web application builder that includes an HTML Editor and a script-based page generator. Both tools run as Web services in any browser with table support and hence are widely available. Together these tools allow construction of simple Web applications without requiring users to understand the angle-bracket syntax of HTML or a full-fledged programming language.

The WebWriter Editor displays elements of a Web page in a formatted style even during editing. To achieve this behavior in a Web-based editor, we have given up a few features that would be present in a stand-alone editor. In particular, we restrict the user to insert new material only at "insertion points" that occur at HTML object boundaries, and objects can only be selected by clicking on selection handles that occur before and after the object. In addition, text cannot be edited in-place, and editing commands may take longer than in a stand-alone editor if delays in the browser, network, or server result in slow generation of new Web pages. What we have gained is an editor that is available across many platforms, that can easily be extended as HTML changes, and that can show a correct preview of the current

HTML page without the distraction of popping up an additional browser window.

The WebWriter Page Generator produces a new page based on a static page template and the output of one or more scripts. This approach allows the static part of each application page to be designed interactively, using the WebWriter Editor, assuring that this part of the page will be in correct HTML. Even the format of the program output is designed interactively. By packaging WebWriter with different scripting languages or built-in applications, the designer can strike a balance between a powerful system that can build many applications, as when the Unix shell interpreter is used, and one that is safe for a broader base of users, as when a simple set of forms-based applications is provided.

Acknowledgments

The authors are grateful to Bay-Wei Chang for his modifications to WebWriter that led to an improved system and improved figures and for his comments that helped us improve this paper. We thank Bill York for building an improved HTML parser. We thank all of the members of the Information Sciences and Technologies Lab at Xerox PARC who tested WebWriter for their comments and suggestions. Finally, we thank the Stanford Digital Library Project, and the Xerox Corporation for supporting this project.

About the Authors

Arturo Crespo
[http://www.stanford.edu/~crespo]
crespo@cs.stanford.edu
Stanford University
Computer Science Department
Gates Bldg. B12
Stanford, CA 94305
Eric A. Bier
[http://www.parc.xerox.com/istl/members/bier/]
bier@parc.xerox.com
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

References

[Adobe95] Adobe Systems Incorporated. Adobe PageMill: The Easiest Way to Create Pages for the World Wide Web. At URL http://www.adobe.com/Apps/PageMill/.

[Apache96] The Apache HTTP Server Project. Welcome to the Apache HTTP Server Project. At URL http://www.apache.org/.

[Apple87] Apple Computer, Inc. Hypercard User’s Guide. Apple Computer, Inc., 1987, 20525 Mariani Ave., Cupertino, CA 95014

[Boutell94] Thomas Boutell. Gd 1.2, A graphics library for fast GIF creation. At URL http://www.boutell.com/gd/.

[Boutell95] Thomas Boutell. MapEdit imagemap editing software. At URL http://www.boutell.com/mapedit/.

[Burchard95] Paul Burchard. W3Kit 2.2. At URL http://www.geom.umn.edu/docs/W3Kit/W3Kit.html.

[CNRI96] Corporation for National Research Initiatives. CNRI Presents Grail. At URL http://monty.cnri.reston.va.us/grail/.

[Glicksman95] Jay Glicksman. A server-side editor for HTML. In *Poster proceedings of the 4th International World Wide Web Conference* (Boston, December), O'Reilly & Associates, Inc., pages 86-87.

[Ladd95] David A. Ladd, J. Christopher Ramming. Programming the Web: An application-oriented language for hypermedia service programming. In *Proceedings of the 4th International World Wide Web Conference* (Boston, December), O'Reilly & Associates, Inc., pages 567-586. Also at URL http://www.w3.org/pub/Conferences/WWW4/Papers/251/.

[Microsoft96] Microsoft Corporation. Microsoft FrontPage Product Information. At URL http://www.microsoft.com/msoffice/frontpage/intro.htm.

[NaviSoft96] NaviSoft Inc. Welcome to NaviPress 1.1. At URL http://www.naviservice.com/register/prstrial.htm.

[Netscape95a] Netscape Communications Corporation. Netscape Javascript. At URL: http://home.netscape.com/comprod/products/navigator/version_2.0/script/.

[Netscape95b] Netscape Communications Corporation. Netscape Navigator. At URL: http://www.netscape.com/comprod/products/navigator/index.html.

[Pier95] Ken Pier, Eric A. Bier, Ken Fishkin, and Maureen Stone. WebEdit: Shared Editing in a Web Browser. In *Poster proceedings of the 4th International World Wide Web Conference* (Boston, December), O'Reilly & Associates, Inc., pages 66-67.

[Pier96] Ken Pier, Eric A. Bier, Ken Fishkin, and Maureen Stone. WebEdit: Shared Text Editing in a Web Browser. At URL http://www.parc.xerox.com/istl/groups/gir/doc/webedit/webedext.htm.

[SoftQuad95] SoftQuad Inc. SoftQuad HoTMetaL PRO 2.0 for Microsoft Windows. SoftQuad Inc., Toronto, Canada. At URL http://www.sq.com/.

[W3C96] World Wide Web Consortium. W3C httpd. At URL http://www.w3.org/hypertext/WWW/Daemon/Status.html.

[Xerox79] Xerox Corporation. The Alto User's Handbook. Xerox Palo Alto Research Center. 1979.

Appendix A. Adding a Clickable Image

Clickable images allow the user to jump to different URLs depending on the area of the image that is clicked. Because clickable images are part of many interactive Web applications, it is important to support them in an application-building tool.

To create a clickable image in WebWriter, the user specifies the URL of the image that will be clickable, a file in which to save the coordinates of the clickable areas and a default URL that is used when the user of the page clicks outside the defined areas. After that, the image is inserted in the document and the user can define rectangular, circular and polygonal clickable areas by clicking directly on the image in the document column. Clickable images that are displayed in the document view are active when they are not selected, allowing the user to test them in place.

To edit a clickable image, the user selects it. At this point, as shown in figure A.1, the boundary of the clickable areas are displayed over the image, and the user can edit or remove one of areas by clicking inside it. A creation form appears allowing the URL to be changed or a new region to be defined.

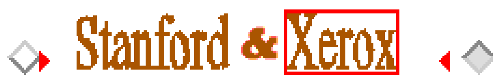

Figure A.1 An image with a clickable area.

Clickable images demonstrate some of the user interface flexibility of WebWriter. Most WebWriter objects inherit the standard user interface behavior, but an object is free to implement a completely different behavior. For example, clickable images extend the concept of the creation form to the image itself, allowing the user to click directly on the image to define the clickable areas. Also, when a clickable image is selected, WebWriter modifies the original GIF image to display its clickable areas, using the GD library [Boutell94] (a C++ graphics library that produces GIF images) to add the area boundaries on top of the image.