Tcl/Tk HTML Tools

Brent Welch and Steve Uhler
Sun Microsystems Laboratories, Inc.

Tcl/Tk HTML Tools

Brent Welch
Steve Uhler
{bwelch,suhler}@eng.sun.com
Sun Microsystems Laboratories
2550 Garcia Ave. MS UMTV29-232
Mountain View, CA 94043

Abstract

This paper describes tools and techniques that support HTML processing with Tcl and Tk. The tools include an HTML parser, a table-driven display engine, and WebEdit, which is a WYSIWYG editor for HTML documents. The parser and display engine are written as a small library that can easily be added to applications that need to display HTML documents. The library has a modular implementation so that applications can customize and extend the library for specialized needs. In particular, WebEdit extends the library to provide an authoring environment. The editor uses the tag and mark facilities of the Tk text widget as the primary data structure for the representation of HTML within the edited document. The paper also describes the performance issues associated with the text widget and one optimization to its implementation.

1 Introduction

HTML is becoming a defacto standard for documents because of its support for formatted text, images, and hypertext links. Not only is it used in the global Internet, but closed intra-nets often use HTML for shared documents and corporate knowledge bases. Many applications are being reimplemented to use HTML, especially forms, for their user interface. This can be awkward, however, and requires complex cgi-bin programming to implement the application in the client-server browser context. An alternative approach to using standard browsers is to embed support for HTML into existing applications.

This paper describes a small script library that makes it easy to support HTML for systems that use Tcl and Tk [Ousterhout94][Welch95]. The library is compact and efficient, and its table-driven implementation makes it easily extensible to support new HTML tags. An application can define new tags, or override the semantics of standard tags, to support direct control of the application from the HTML document.

The fundamental operation of the library is an `Html_Parse` operation that applies a function to each HTML markup tag in a document. Different functions are applied to the HTML to achieve different effects. For example, the `Html_Render` function displays the document in a Tk text widget. The `Html_Validate` function validates URLs in hyperlink and image tags. The library also includes supporting functions to deal with fetching URLs and submitting forms.

Authoring is clearly an important part of an HTML-oriented environment. This paper also describes a WYSIWYG editor for HTML, WebEdit, that is based on the HTML display library. WebEdit supports basic markup, lists, hypertext links, images, imagemaps, and forms. Table support is in progress, and has been demonstrated by other tools [Ball96] that also use our display library. WebEdit is also a browser, so you can roam the web and assemble pages from parts of other pages. This differentiates it from other tools that provide similar features such as Adobe's PageMill™.

The following sections describe our tools in more detail. Particular attention is paid to some interesting Tcl programming techniques that can be applied to a variety of Tcl applications.

- The parser maps data into a Tcl program that is then evaluated to process the data.
- The editor uses the tag and mark facilities of the Tk text widget as its data structure to represent information about the HTML.

2 The HTML Display Library

The HTML display library implements HTML/2.0 in about 1300 lines of Tcl code. This includes support for basic formatting tags, hypertext links, images, forms, and some simple extensions (e.g., centering, colors, font size). A simple browser requires a couple hundred more lines for a decent user interface. A stripped down version of the library that just handles local files and local images (i.e. no forms or HTTP) takes about 600 lines of Tcl.

The library architecture is extensible to make it easy to add support for new HTML tags*. The library is organized into the building blocks described below:

- The heart of the display library is `Html_Parse` that maps a function onto each HTML tag. The caller of `Html_Parse` specifies a function and possibly some initial parameters, and then `Html_Parse` calls that function with some additional parameters for each HTML tag in the document.
- The `Html_Render` procedure uses the Tk text widget to display HTML. It is invoked as the callback function from `Html_Parse`. `Html_Render` is described in more detail later.
- The `Html_Init` and `Html_Reset` procedures configure a Tk text widget for use with `Html_Render`. `Html_Init` is called once to define various Tk text tags, and `Html_Reset` is called before each new page is displayed.
- `Html_Render` maintains several state stacks that control different aspects of the display such as font, spacing, and margins. Each HTML tag can have an entry in a table, `htmlTagMap`, that define which state stacks it affects. Another table, `htmlBreakMap`, defines which HTML tags cause line breaks.
- Each HTML tag can have an associated helper procedure that is called by `Html_Render` for special processing. The names of these procedures include the tag name so `Html_Render` can call them automatically. Examples include `HtmlTag_img` and `HtmlTag_/form`.
- The helper procedures for hyperlinks, images, and forms have an additional layer so that the library can implement their display while allowing the application to provide the semantics for these elements. The callbacks defined by the library are `Html_LinkSetup`, `Html_SetImage`, and `Html_SubmitForm`. There are sample implementations of these callbacks that are suitable for a regular web browser, but an application could redefine them for its own purposes.
- The library includes an HTTP package that uses the Tcl7.5 socket facilities to fetch URLs.

*. The term “tag” is used in two contexts within this paper: HTML markup tags that appear in HTML documents(e.g., <img src=foo.gif>, and Tk text widget tags that are symbolic names for ranges of text within the Tk text widget. The qualified terms “HTML tag” and “Tk text tag” are used to differentiate the two uses of “tag”.

3 Implementation of the Display Library

The library uses two basic techniques to get a small and efficient implementation: table-driven programming and dynamic code generation. The table-driven techniques are accomplished by defining Tcl arrays that are indexed by the name of the HTML tag, and by using procedure names that are derived from the name of the HTML tag. The dynamic code generation involves transforming input data into Tcl programs and then using the `subst` or `eval` commands to process the result.

3.1 HTML Parsing

The `Html_Parse` procedure rewrites its HTML input into a Tcl program. The basic idea is that `regsub` is used to look for HTML tags that are delimited by < and >, and these are converted into Tcl procedure calls. For example:

```
<tag param=value>some text</tag>more
  text.
```

gets rewritten into the following Tcl script:

```
Html_Render $win hmstart {} {} {}
Html_Render $win tag {} {param=value}
  {some text}
Html_Render $win tag {/} {} {more text}
Html_Render $win hmstart {/} {} {}
```

The caller of `Html_Parse` specifies the name for the procedure in the generated code and optionally some parameters (e.g `Html_Render $win`). The additional arguments to the procedure are:

- `tag`, an HTML tag.
- `not`, either “” or “/”.
- `param`, the parameters from the HTML tag.
- `text`, the text up to the next HTML tag.

The first and last commands of the generated code make the callback with a pseudo-tag (e.g., `hmstart`), as if there were an extra `<hmstart>` and `</hmstart>` around the whole piece of HTML. As described later, application-specific initialization can be associated with the pseudo-tag. A call to `Html_Parse` looks like this:

```
Html_Parse $html \
```

```
[list Html_Render $tkwin] hmstart
```

The basic strategy of `Html_Parse` is to first protect any Tcl special characters that are in the input data. It is important that these do not interfere with the `eval` done later. Next, a regular expression substitution re-writes the input data into a series of Tcl commands. The input data is passed as arguments to the commands. Finally, `eval` is used to run the dynamically generated code. Here is its implementation.

```
proc Html_Parse {html \
      {cmd HhtmlTestParse} \
      {start hmstart}} {
   # Convert Tcl specials to HTML entities
   regsub -all \{ $html {\&ob;} html
   regsub -all \} $html {\&cb;} html
   regsub -all {\\} $html {\&bsl;} html
   set w “ \t\r\n”;# white space
   # Expression to match HTML tags
   set exp <(/?)(\[^$w>]+)\[$w]*(\[^>]*)>
   # Re-write pattern:
   # \1 is either / or the empty string
   # \2 is the HTML tag
   # \3 is the parameters to the tag
   set sub “\}\n$cmd {\\2} {\\1} {\\3} \{“
   regsub -all $exp $html $sub html
   eval “$cmd {$start} {} {} \{$html\}”
   eval “$cmd {$start} / {} {}”
}
```

There are five passes through the HTML input: four global `regsubs` and one `eval`. The first three `regsub` commands replace all curly braces and backslashes with entities (i.e., `&ob; &cb;` and `&bsl;`) so they do not interfere with `eval`. Backslashing these characters won’t help because things are grouped with braces in the generated code. The entity encoding is used because `Html_Render` already has to decode entities for HTML special characters like ‘>’ and ‘<‘. The decoder for these entities is described shortly. The fourth `regsub` command picks out the HTML tags and their parameters, and does the rewriting.

Note that each rewrite begins with a close brace and ends with an open brace. This groups the unmatched text between the HTML tags. The first `eval` command supplies the balancing braces. Profiling measurements indicate that page display time is dominated by the Tcl parser and the Tk text widget, not by the regular expression substitutions.

There are two bugs in this version of the parser. The first is that HTML comments are not handled reliably because they can contain ‘>’ characters. Similarly, the parameter values in HTML tags could also contain ‘>’ characters. Comments can be correctly handled with another pass that maps their syntax into something compatible with the matching done last. For display the comments can be completely deleted. For editing, we divert the values of the comments into an array, and replace the comments with HTML tags that reference the array.

For maximum speed and reliability we will soon rewrite the `Html_Parse` procedure in C. It will remain relatively simple, however, and retain the basic strategy of calling a Tcl procedure to handle each HTML tag. The ability to map different functions over the HTML has proven to be quite useful in the editor and other HTML document management tools.

3.2 An Overview of Html_Render

The `Html_Render` procedure displays HTML in a Tk text widget. It is called from `Html_Parse` for each HTML tag in the document. It maintains a state machine to determine how text is formatted. The state machine is table-driven, and any special cases are handled by special per-tag helper routines.

The basic steps to `Html_Render` are shown below. The parameters (e.g., `tag`, `not`, `param` and `text`) were described in the previous section. The following subsections describe some of these steps in more detail.

- Decode any entities in `text`.
- Push or pop state associated with `tag`.
- Decide if a line break should occur, and deal with white space.
- Call a per-tag helper procedure, `HtmlTag_$not$tag`, for special processing, if any.
- Compute a set of Tk text tags to apply to `text`.
- Insert `text` into `win` with the current set of Tk text tags.

3.3 Decoding HTML entities

Decoding HTML entities provides another example of dynamically generating Tcl code to process data.

An entity encodes characters like ‘<‘ and ‘>’ that are special to HTML. If they are to appear in the document as literal ‘<‘ and ‘>’ characters, they need to be encoded so they are not interpreted as HTML markup. Characters that have their high-order bit set (e.g., ‘©’ and ‘â’) are also encoded. The encodings are keywords or decimal values that are enclosed with ‘&’ and ‘;’, like these:

```
Copyright &169; less than &lt; greater
  than &gt;
```

The basic idea of the decoder is that it first replaces entities with a `format` command that will generate the real character. The `subst` command is then used to replace the `format` commands with the special character. Here is the code for the entity decoder:

```
proc HtmlDecodeEntity {text} {
  if {![regexp & $text]} {return $text}
  regsub -all {([][$\\])} $text \
      {\\\1} new
  regsub -all {&#([0-9][0-9]?[0-9]?);?} \
    $new {[format %c \
        [scan \1 %d tmp;set tmp]]} new
  regsub -all {&([a-zA-Z]+);?} $new \
      {[HtmlMapEntity \1]} new
  return [subst $new]
}
```

The first `regexp` just checks to see if any work really needs to be done. The next `regsub` is a prepass to quote all the Tcl special characters. This is necessary so that `subst` doesn’t interpret the wrong things. The next `regsub` command replaces the decimal-valued entities with a `format` command. The `format` command uses `scan` to interpret the decimal values to avoid cases like “`&09;`” that are otherwise incorrectly interpreted as invalid octal numbers by `format`.[*]

The named entities require a table that maps from the entity name to a character code value. Access to the table is done by the procedure `HtmlMapEntity` to make error handling easier. A subset of `htmlEntityMap` is shown below.

```
array set htmlEntityMap {
  lt < gt > amp & quot \" copy \xa9
  ob \x7b cb \x7d bsl \\
}
proc HtmlMapEntity {text {unknown ?}} {
  global htmlEntityMap
  set result $unknown
  catch {
    set result $htmlEntityMap($text)
  }
    return $result
  }
```

Note the difference between using `eval` in `Html_Parse` and using `subst` in `HtmlDecodeEntity`. `Html_Parse` has to handle every character, even those not matched by the `regsub` pattern. The clever placement of curly braces groups *unmatched* text into a command argument. Eval is necessary to pass that argument to a procedure. In `HtmlDecodeEntity`, only the matched text has to be processed, and the unmatched text should not be modified. The `subst` only affects text matched by regsub. It’s also easier to quote Tcl special characters because `subst` is less sensitive to curly braces.

3.4 Table-Driven Display State

One of the more interesting table-driven techniques concerns the display state. There are several orthogonal properties that combine to affect the way formatted text is displayed. These properties include the margins, line spacing, and the font. The font itself is determined by several properties including the font family (i.e. typeface), size, style (e.g. bold, italic, underline), and color. `Html_Render` keeps a separate state stack for each of these individual properties, and the `htmlTagMap` table defines how a given HTML tag affects the stacks. A subset of `htmlTagMap` is shown below:

```
array set htmlTagMap {
  b {weight bold}
  code {size 12 family courier}
  em {style i}
  h1 {size 24 weight bold
      Tspace hspacebig}
  h3 {size 16 weight bold
      Tspace hspacemid}
  ol {indent 1}
  u {Tunderline underline}
  pre {fill 0 family courier size 12
      Tnowrap nowrap}
}
```

The key to the map is the name of the HTML tag (e.g. `h1`). The value is a list of name-value pairs. The name identifies a stack, and the value is pushed onto the stack when the HTML tag appears. When the corresponding close tag appears (e.g., `/h1`), the values are popped. The effects of most of the HTML tags can be completely defined by their entries in the `htmlTagMap`, which means there is no need for a per-tag helper procedure.

---

*. If the HTML specification for entities used hexadecimal, the format command could be written simply like this:

```
format %c 0x\1
```

The HtmlStack and HtmlStack/ procedures push and pop values from the stacks, respectively. The pop routine has a funny name, HtmlStack/, and the same arguments as the push routine, HtmlStack. This lets Html_Render do the pushing and popping for the current tag with a single Tcl command shown below. The catch is necessary because there may not be an htmlTagMap entry for all tags.

```
catch {
    HtmlStack$not $win $htmlTagMap($tag)
}
```

3.5 The Helper Procedures

Some HTML tags require special processing. This include tags in the HTML header (e.g., <title>), list-related tags (e.g., <ol> and <li>), link tags, image tags, and form-related tags. The special processing is implemented by Tcl procedures that include the name of the HTML tag in their name. This makes it easy to extend the library to support new tags.

Html_Render attempts to call a per-tag handler as shown below. Again, catch is used because there may not be a tag handler. The text parameter is passed by name, not by value, so the tag handler can side-effect the text before Html_Render inserts it into the text widget.

```
catch {
  HtmlTag_$not$tag $win $tag $param text
}
```

The tag handler is illustrated with an example that defines a color tag. This is a nonstandard HTML tag that lets you specify the color for text. It takes one parameter that specifies the color. Specifying red text would look like this:

```
<color value=red>This is red.</color>
  This is not.
```

The job of the HtmlTag_color procedure is to get the value and manipulate the state stacks so that the following text is red. The job of the HtmlTag_/color procedure is to undo the effect by popping the stack. A new stack named Tcolor is introduced to support this. Here is the code:

```
proc HtmlTag_color {win param textVar} {
  set value bad_color
  HtmlExtractParam $param value
  HtmlStack $win "Tcolor $value"
  $win tag configure $value \
      -foreground $value
}
proc HMtag_/color {win param textVar} {
  HtmlStack/ $win "Tcolor {}"
}
```

The HtmlExtractParam procedure picks out values from the Name=Value syntax used in HTML tags. This uses regular expressions, too. While it is not possible to parse all the parameters at once with regular expressions, it is possible to pick out a single parameter at a time.

The second argument to HtmlExtractParam, in this case value, specifies the name of the parameter. If it exists in the parameter list, a Tcl variable by that name is initialized to the value (e.g. red). The variable is not defined if the parameter isn't specified in the HTML tag. In this case the color would remain bad_value and the tag configure would fail. Html_Render ignores errors from tag helper procedures, except during debugging.

3.6 Formatting the Text

The state stacks cause different Tk text tags to be applied to the text. In the simplest case, the value on the top of the stack is used as the name of a Tk text tag to apply to the text. This is done with any stack that is named with a leading T. The Tcolor stack used for the color HTML tag is an example. The values on the Tcolor stack are color names that have been configured as Tk text tags that have that foreground color. As another example, the h1 tag pushes the value hspacebig onto the Tspace stack. The following initialization code in Html_Init configures the hspacebig tag to have certain interline spacing:

```
$win tag configure hspacebig \
    -spacing1 10p -spacing3 6p
```

This tag configuration and the entry in htmlTagMap are all that is needed to support the h1 HTML tag.

Other uses of the state stacks are a little more complicated. The current indentation level is determined by the size of the indent stack, for example, not its top value. The indent level selects from a set of Tk text tags that are configured to have different indents and tab stops. The font is determined by the combination of the top-of-stack values from the weight, family, size, and style stacks.

3.7 Managing Instance Data

The library must keep state information for each text widget that is being used to display HTML.

(The browser and editor display more than one page at a time.) The name of the text widget is passed into the library, which uses `upvar` to map that into the name of a Tcl array. Procedures contain this statement:

```
upvar #0 HM$win var
```

The rest of the code references `var`. For example, all the simple state stacks are found automatically in this `foreach` loop:

```
foreach stack [array names var T*] {
  # Look at the top-of-stack value
  set top [lindex $var($stack) end]
}
```

4 WebEdit, a WYSIWYG HTML Editor

WebEdit* provides a WYSIWYG editing environment where the user is shielded from direct manipulation of the HTML tags. Instead, the user performs logical operations such as making text bold or changing the paragraph type to a heading, and the editor manages the HTML tags. The page is continuously displayed in the format it would be viewed in a browser. It is possible to view the underlying HTML tags from within the editor, but even in this mode the user is prevented from manipulating the HTML tags directly. This may constrain some HTML wizards from achieving bizarre effects, but it also ensures that the page contains valid HTML.

4.1 Cut, Paste, and URLs

The editor is a client of the display library. Cut and paste is done by generating HTML during copy or cut, and rendering HTML during paste. The display library handles paste, and the editor has an output module for cut, copy, and saving HTML to a file. When a range of text is copied or cut, its HTML representation is computed. For example, a selection containing **bold** text would be returned as the following string:

```
this is <b>bold</b>
```

Pasting is just a matter of using the display library to render the HTML markup. This allows cut and paste between pages. It also allows interoperability between plain text tools such as text editors and email readers. If a user selects HTML markup in their text editor, when they paste that into WebEdit it is automatically rendered as formatted text.

*. This name will change because there is an existing product with this obvious name.

Other edit operations are built on top of cut and paste. For example, when the user selects a range of text and makes it bold, the editor cuts the affected region, wraps it in `<strong>` and `</strong>` HTML tags, and then uses the display engine to redisplay the result.

The editor is also a browser, so you can easily cruise the net and copy links and images from other pages. Copying a URL creates an interesting problem. Suppose a page contains a hypertext link with a relative URL (images have the same problem):

```
<a href=file.html>
```

When this link is copied into another page, the user expects it to work. This may require resolution of the URL into an absolute reference:

```
<a href=http://somwhere.com/file.html>
```

Or, if the page is on the same server but in a different directory, a new relative name may need to be computed:

```
<a href=../otherdir/file.html>
```

The editor does some extra work during copy and paste to transform URLs in this way. First, when it generates a selection that contains a relative URL, a base tag is also emitted as part of the selection:

```
<base href=http://somewhere.com>
<a href=file.html>
```

During a paste operation, base tags are used to compute the “best” representation of a relative URL in the destination page. (The base tag is not copied into the destination page.) As shown above, the new URL may be an absolute URL or a different relative URL.

4.2 Representing HTML

The default behavior of the display library throws away the information about HTML markup. It just derives enough information to compute the display. `Html_Render` was modified to call into the editor so it can add additional tags and marks to the text widget to represent HTML tags. For example, the display library will use three Tk text tags to display an `h1` heading; one for the line spacing, one for the font, and one for the indent level. The editor adds another text tag, `H:h1`, to all the text in the heading. By querying the current tags whose names begin with `H:`, the editor can detect what HTML tags are in effect in order to reinitialize the state of the display library. Using Tk text tags and marks to represent HTML tags is a natural design decision for the editor because the text widget maintains the tags and marks as the user inserts and deletes text.

It is tempting to try and combine the tags that represent the HTML markup with tags that affect the display of the text. Unfortunately, the effects of combining HTML tags make reverse engineering the HTML from the display tags difficult. The `strong` tag, for example, implies a different font when applied within a regular paragraph than when it is applied within a heading. There are also different HTML tags that produce the same visual effect (e.g., `b` and `strong`, or `em` and `var`).

Not all HTML tags can be represented by Tk text tags. HTML tags like `img`, `li`, and `input` do not ordinarily have matching close markers (e.g., there is never a `/img` tag.) WebEdit classifies these as singletons and uses text marks to represent them. One minor issue with marks is that they don't get deleted when the surrounding text is deleted. WebEdit must find and delete the marks explicitly.

The editor uses a set of tables to classify HTML tags into classes. It differentiates between styles (e.g., `strong`), paragraphs (e.g., `h1`), lists (e.g., `ol`), list items (e.g., `dt`), structure elements (e.g., `form`), and singletons (e.g., `img`). These classifications affect editing operations. For example, only one paragraph type can be in effect at once, while multiple style types can be in force.

Each HTML tag also has a set of parameters that are valid for it. These are also defined in a table, and the editor has a general property sheet dialog that is used to set these parameters. Currently you have to edit the WebEdit source to update the tables. We plan to add a user interface to add new tags and add new parameters to tags so that WebEdit can be used to author HTML for custom applications.

The tag classifications are also used during output. The main trick to output is sorting HTML tags that occur at the same text index. For example, a heading that is also a hypertext link will have an `<h1>` and `<a href=...>` HTML tag that both start at the same position in the Tk text widget. During output, the editor must decide which HTML tag comes first. The tag classifications are used to define a sorting order among classes, and tags within the same class just sort alphabetically. The sorting order is listed below from highest (i.e., earliest in output) to lowest:

```
close-style
close-paragraph
close-list
close-structure
open-structure
open-list
list-item
open-paragraph
open-style
singleton
```

The HTML produced for the hypertext link in a heading would be:

```
<h1><a href=...>The text</a></h1>
```

The file output routine uses another table to define how to format tags so the HTML file is legible.

4.3 New Features for the Text Widget

The editor's output module that is layered on a new `dump` operation in the Tk text widget. This is used when saving to a file or when generating a selection. The `dump` operation either returns information about text widget segments, or it can call a Tcl command with information about each segment. The segments include text, tag transitions (i.e., tagon and tagoff), marks, and embedded windows. The information identifies the type of the segment, its value, and its index within the widget. Each text segment represents a range of text that is not split by a mark, tag transition, embedded window, or newline. The segments are a direct reflection of the data structure used internally by the text widget.

The output module uses the callback form of the `dump` operation. It looks for the tag transitions and text marks that represent HTML. These are accumulated until a text segment is encountered. The accumulated tags are then sorted and output before the following text.

The editor often needs to know the current range of a tag. That is, given that `tag names` reports a tag in effect at a given index, where does that tag range start and end? This is used when changing a paragraph type and when editing hypertext links. There used to be no practical way to find out the current range; you had to step through every range of the tag with `tag nextrange`. We added the `tag prevrange` operation that is the complement to `tag nextrange`. The two range operations are used together to find the current range. The procedure is shown below. The new `foreach` command is abused to set multiple variables from a command that returns a list.

```
proc Edit_CurrentRange { win tag mark } {
  foreach {start end} \
    [$win tag prevrange $tag $mark] {}
  if {$end == "" || \
      [$win compare $end < $mark]} {
    foreach {start end} \
      [$win tag nextrange $tag $mark] {}
    if {$start == ""|| \
        [$win compare $start] > $mark]} {
      return {}
    }
  }
  return [list $start $end]
}
```

4.4 Performance Issues

We found quadratic behavior in the implementation of text tags that caused a slowdown with pages that had lots of unique tags. The cost of adding a new tag to the text widget was proportional to the number of other unique tags already in the widget, so adding N unique tags in the process of loading a page was $O(N^2)$ in processing time.

Describing the source of the problem requires an explanation of how the text widget represents its contents. The text widget uses a BTree to represent all the lines of the text widget. An example is shown in Figure 1.There are interior nodes, line nodes, and line segments. The BTree keeps the number of children of the interior nodes balanced. This includes the number of lines under each level 0 node. More levels are added to the tree as needed to maintain the balance. Each line has a list of all the segments of that line: marks, tag transitions, embedded windows, and text segments.

Tags are represented by tagon and tagoff segments. These segments are at the leaves of the BTree, but

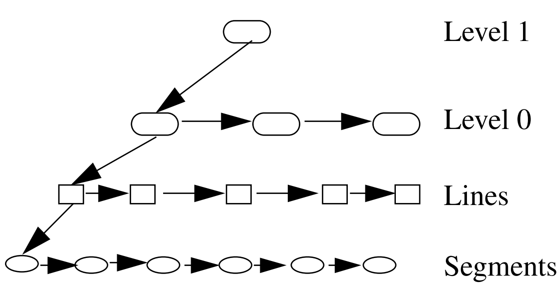

Figure 1. The Text BTree Representation

information about the tag segments is propagated up the tree in the form of summary counts. At each interior node, there is a count of all the tagon and tagoff segments for each tag in all the lines below that node.

By starting at the root it is easy to walk the BTree and only visit nodes that have summary information about a particular tag. This is used for the various tag range operations (i.e., `ranges`, `nextrange`, and `prevrange`).

By starting at a segment and walking up the BTree, you can count up the summary information to determine what tags are active at a particular node. If the count is odd it means the last transition is a tagon and the tag is active.

One problem with this scheme is that there is summary information about every tag in the root node of the BTree. Adding a tag range requires searching the summary list at each node going up the tree from the tag transition segments. The cost of this is proportional to the number of unique tags in the text widget. Therefore adding N unique tags suffers this cost N times, for an overall cost proportional to the square of the number of unique tags. Figure 2 plots the cost of adding and deleting N tags to a text widget. The $O(N^2)$ behavior of the old tag implementation is clear.

We optimized the storage of summary information to prune the tag information out of the interior nodes of the BTree. The information is only necessary below the interior node that covers all the ranges for a given tag. In the best case, a tag that is only in effect on one line has no information at all in the interior nodes. In the worst case, a tag that has an active range at the beginning and end of the widget will have information that does propagate all the way to the root. For applications that have lots of unique tags that are not widely used (like tags that represent URLs in hypertext links), the optimization is very effective. Figure 2 shows that the cost of adding N unique tags that each have a single range is $O(N)$ instead of $O(N^2)$.

The figure shows that the cost of deleting a tag is still proportional to the number of other tags in the text widget. Deleting a tag removes all information about the tag. When this happens, the complete set of tags is searched in order to adjust tag priorities. The priority determines which tag is in effect if

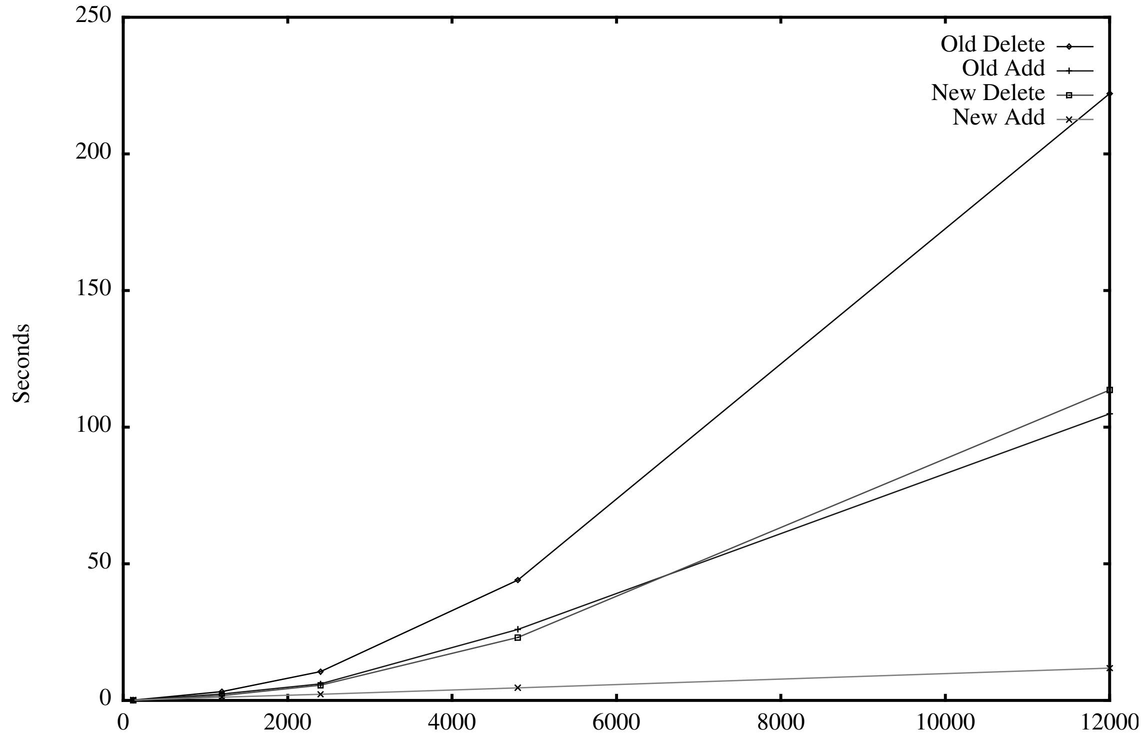

Figure 2. Tag add and delete vs. number of unique tags

more than one tag contributes a display property or a binding.

Note that deleting a tag is different than removing the tag from all ranges of text. The text widget keeps information about a tag even if it has no active ranges. This means that it is good practice to configure your display and binding tags once at the beginning of your application, and then never remove them.

One performance problem remains with the text widget. Complex lines that have lots of segments such as marks and tag transitions are expensive. The BTree does not help to balance this information because each leaf node holds information about one line. A line is defined to end with a newline character, so a long line that wraps many times is still one line. In the limit, if you have no newline characters in your text, it is represented as a linear list of segments, and the BTree is of no value. The tree structure needs to eliminate the distinction between interior nodes and line nodes, and allow the segments for one line to be balanced across nodes.

5 Conclusions and Future Directions

This paper describes a simple and extensible HTML viewer that can easily be added to existing applications. The combination of the generic stack processing and the use of per-tag helper procedures makes it very clean to add support for new HTML tags. In addition, the table-driven programming and dynamic code generation techniques we describe can be used in a variety of Tcl applications.

The WebEdit authoring tool is a natural complement to the HTML display library. The paper describes how WebEdit represents HTML in the Tk text widget, and it describes performance optimizations we have made to the Tk text widget.

The current version of WebEdit is focused on basic HTML generation. This is just a starting point, and we envision a set of related applications that we plan to address in the future:

- Generation of a family of pages from building blocks such as standard headers and images. Any well-managed web site takes this approach in order to provide a consistent look to their pages with a minimum of effort.

• Support for cgi-bin programs that are the standard associated with form processing on the web. WebEdit could create templates and a specification for how the templates get filled with data. The Navisoft servers already let clients embed Tcl in their HTML for this purpose. [Navisoft96]

• Support for client-side functionality in the form of hooks that call functions in the browser or embedding application. This has lots of potential for extending basic HTML with application-specific behaviors. Netscape's Javascript ™[Javascript] is a small step in this direction, but it is currently quite limited. The Netscape browser is closed, so it is not possible to embed it into applications and then use Javascript™ to control the application. In addition, there is no security model for Javascript™, which makes it risky in open environments[LoVerso96]. The use of Safe-Tcl and untrusted interpreters makes it possible to download safe client-side behaviors in open environments.

Availability

The HTML library is available via FTP as: ftp://ftp.sunlabs.com/pub/tcl/html_library-0.3.tar.gz. The version number may change.

WebEdit will be released shortly. Please visit its page at http://www.sunlabs.com/people/brent.welch/editor/intro.html for more information.

References

Ball96	The SurfIt! web browser. http://pastime.anu.edu.au/SurfIt!/
Javascript	http://www.c2.org/~andreww/javascript/
LoVerso96	http://www.osf.org/~loverso/javascript/
Navisoft96	http://www.navisoft.com/servdoc/extend/
Ousterhout94	J. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, April 1994, ISBN 0-201-63337-X
Welch95a	B. Welch, *Practical Programming in Tcl and Tk*. Prentice Hall, May 1995. ISBN 0-13-182007-9