In August, the Tclsh Spot article introduced an html viewer for older, curses based email programs like `elm` and `pine`.

By design, the viewer didn't support loading images or hypertext links.

I usually don't want to waste time waiting for images to load while I'm reading my mail, and I seldom want to bounce to a hypertext link. But, sometimes I do want that functionality, so...

This article will describe how to add hypertext links and image display to the htmlview.tcl package. Along the way, we'll briefly examine some of Tcl's support for large-scale projects (the `namespace` and `package` commands), binding events to actions, accessing the web with the `HTTP` package, and the `image` object.

Obviously, I won't be going into a lot of detail on these topics. You can find more details in my book (*Tcl/Tk for Real Programmers*), Steve Ball's book *Web Tcl Complete*, and Mike Doyle and Hattie Schroeder's *Interactive Web Applications in Tcl/Tk*. Will Duquette has written a good discussion of the namespace and package commands at: http://www.cogent.net/~duquette/tcl/namespaces.html.

Two of the Tcl commands that simplify large scale programming projects are the `namespace` and `package` commands. These commands support two software engineering concepts that help construct easily maintained, modular code. The `namespace` command hides a set of commands and data in a private area where they won't interact with other code. The `package` command groups a set of procedures (possibly in several source files) into a single entity that can be accessed by name and optionally selected by revision number.

A namespace is similar in some respects to a C++ or Java class. Like a class, the items within a namespace may be either procedures or data. The data hidden within a namespace is available to all members of that namespace without being visible from the global scope (unless a script specifically requests the item). Tcl namespaces do not support inheritance, but namespaces can be nested.

Items within a namespace are named with a path style naming convention, similar to the way X-11 windows, or directory paths are named. The path separator for Tcl namespaces is the double-colon (::). For example, an item `baz` within a namespace `bar` that's included within a namespace `foo` would be named as `::foo::bar::baz`.

A namespace is created with the `namespace eval` command:

Syntax: namespace eval *namespaceID arg ?args?*

`namespace eval`	Create a namespace, and evaluate the script *arg* in that scope. If more than one *arg* is present, the arguments are concatenated together into a single command to be evaluated.
namespaceID	The identifying name for this namespace.
arg ?args?	The script or scripts to evaluate within namespace *namespaceID*

For example, this code will create a `fastfood` namespace, with internal variables for burgers and fries, and a `burgerSeller` procedure for modifying the number of burgers. By keeping the burger count inside the namespace the customers can only access a

burger by interacting with a `burgerSeller`.

```
namespace eval fastfood {
    variable burgers
    variable fries


    proc burgerSeller {} {
        variable burgers
        incr burger -1
    }
}
```

Selected components within a namespace can be made easily accessible with the `namespace export` and `namespace import` commands, or they can be accessed with the full namespace path identifier.

The HTTP package hides all of its functions within the `http` namespace. Since the modified `htmlview` package will only use the `http` namespace, the simplest way to access the `http` functions will be via their full path. For example, the command `::http::geturl`, evaluates the procedure `geturl` within the `http` namespace. Note that this doesn't invoke the procedure defined within the `http` namespace in the current namespace, it evaluates `geturl` procedure within the `http` namespace.

The `package` command is a librarian, similar in some respects to the Unix `ar` command. It joins a bunch of related procedures so that they can be accessed with a single name. Unlike the `ar` command, the `package` doesn't create a new file for the joined data. The `packge create` command creates a directory file (`pkgIndex.tcl`) that's used to determine which files should be loaded to resolve procedure references at runtime.

A Tcl script uses the `package require` command to declare that it will need to access a previously defined package. The script can also declare whether it requires a particular revision of this package, the latest revision, or the most recent minor revision of a particular major revision.

Syntax: package require ?-exact? *packageName ?versionNum?*

`package require`	Informs the Tcl interpreter that this package may be needed during the execution of this script. The Tcl interpreter will attempt to find the package and be prepared to load it when required.
`-exact`	If this flag is present, then *`versionNum`* must also be present. The Tcl interpreter will only load that version of the package.
`packageName`	The name of the package to load.
`?versionNum?`	The version number to load. If this parameter is provided, but `-exact` is not present, then the interpreter will allow newer versions of the package to be loaded, provided they have the same major version number. If this parameter is not present, any version of `packageName` may be loaded.

So, with these two commands, we can look at how to use the `HTTP` package. The two most used commands in the `http` package are:

`http::geturl`	download data from a URL, and return a token to use to access this data

url	in the future.
http::data *token*	return the data associated with a token.

This script downloads and prints the contents of the scriptics homepage:

```
package require http
set token [http::geturl www.scriptics.com]
puts [http::data $token]
```

The htmllib.tcl package has hooks to handle hypertext links. The htmllib.tcl package creates a binding on the text between an <HREF> and </HREF>. When a user clicks on that text, the procedure HMlink_callback will be invoked.

Syntax: HMlink_callback *win href*

HMlink_callback	A procedure to handle a hypertext link.
win	The currently active window.
href	The URL for the hypertext link.

Adding this code to the htmllib.tcl will enable the htmlviewer to access links:

```
package require http

proc HMlink_callback {win href} {

    # Clear the old contents from the window.

    HMreset_win $win

    # Get the url:

    set token [http::geturl $href]

    # Display the new text.

    HMparse_html [http::data $token] "HMrender $win"
}
```

That takes care of links, how about images? Since I still don't want all images to be loaded when I read some spam by accident, I want to be able to click on a blank image and load just that image. (Cynical folks might notice that this selection mechanism lets me avoid banner ads.)

When the htmllib.tcl package sees an <img...> tag, it evaluates the HMset_image procedure with three arguments: the name of the primary text window, the name of the window that marks the location for this image, and the url of the image. When the

image has been loaded, `HMset_image` will invoke `HMgot_image` with the location marker and the new image data.

A normal browser would load the image data in `HMset_image`, and immediately invoke `HMgot_image` to display it. Since we'd rather select the images we're interested in seeing, we'll bind an action to the marker label, and load the image when the label is clicked.

The Tcl `bind` command lets a script define an event that can happen to a graphic object, and define a script to evaluate when that event occurs. The syntax for this is:

Syntax: bind *widgetName eventType script*

bind	Define an action to be executed when an event associated with this widget occurs.
widgetName	The widget to have an action bound to it.
eventType	The event to trigger this action. Events can be defined in several formats:
alphanumeric	A single printable (alphanumeric or punctuation) character defines a KeyPress event for that character.
<modifier-type-detail>	This is similar to the X windows event descriptors that precisely define any event that can occur. Event descriptions like <ButtonPress-1> are most common.
script	The script to evaluate when the event occurs while this window has focus.

If we define a new procedure `HMimage_request` to actually acquire and display the image, we can define `HMset_image` like this to create a binding on the label that will invoke `HMimage_request` when the label is clicked. The `src` variable contains the URL for the image. By calling `HMgot_image` with `$src` variable, the `htmllib.tcl` package will change the text in the label from the generic `image` to the actual URL (making it easier for us to decide if we want to see this image or not.)

```
proc HMset_image {win handle src} {
    bind $handle <ButtonPress-1> "HMimage_request $win $handle $src"

    HMgot_image $handle $src
    return ""
}
```

The Tcl command for creating an image is the `image` command. Tcl can create both bitmap, or full color (photo) images from X11 ibitmaps, gif, or PPM/PGM format data. Several extensions to Tcl add support for other image formats (see Jan Nijtmans'site: `http://www.worldaccess.nl/~nijtmans/`).

The Tcl `image` command can accept photo data as binary data in a file, or as B64 encoded data in a variable. Since converting binary data to B64 and back to binary is a bit silly, we might as well use an intermediate file to hold the binary data.

Syntax: image create *type ?name? ?options?*

image create	Create an image object of the desired type, and return a handle for referencing this object.

type The type of image that will be created. May be:
- `bitmap` a 2-color graphic.
- `photo` a multicolor graphic.

?name? The name for this image.

?options? Options that are specific to the type of image being created.

The `http::geturl` command can be instructed to download the URL data into a channel instead of into memory with the `-channel` option.

Here's the code for `HMimage_request`. It uses the `pid` command to get the task's process ID as a cheap and dirty way to make a (probably) unique file name. Once it has a name, it opens a an output channel to that file, and invokes `http::geturl` to copy the image data from the remote site into that file. Once the image has been created (the data has been read) that file can be deleted with the `file delete` command.

```
proc HMimage_request {win handle url } {

     set tmpFile /tmp/tmp.[pid]

     set outfl [open $tmpFile w]

     http::geturl $url -channel $outfl

     close $outfl

     set fail [catch {image create photo -file $tmpFile} img]

     file delete $tmpFile

     if {$fail} {
         return ""
         }
     }

     HMgot_image $handle $img

     return ""
 }
```

At this point, the `htmlview` program has a most of the functionality of a real browser, with a much smaller footprint than Netscape. Actually writing a full featured browser in Tcl is not quite so trivial as this makes it look. You may want to look at the `plume` browser at `http://plume.browser.org` to see an example of a full featured Tk based browser.

A version of `htmlview.tcl` with support for images and hypertext links is available at `http://www.cflynt.com`.