Cuneiform - Implementing Document Layout with Clay

Presented at:
The 26th Annual Annual Tcl Developer's Conference
(Tcl'2019)
Houston, TX
November 4-8, 2019

Abstract

When composing massive machine generated manuscripts, one is often tempted to fall back on PDF as a format of choice for digital documents. At least until one has to develop content for an international client who uses a Logographic language system. This paper describes an object oriented XML/HTML layout system called Cuneiform. Cuneiform acts as an Tcl-Savvy object model for formats that demand a document object model. At the same time, it can leverage the similarities between XML and HTML to integrate SVG files with text.

Sean Deely Woods
Senior Developer
Test and Evaluation Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 20185
Email: yoda@etoyoc.com
Website: http://www.etoyoc.com

Introduction

Cuneiform is a package written in TclOO to allow the interaction of several complex rule sets to generate professional quality documentation in a variety of formats. It is by no means simple. It provides no training wheels. There is very little in the way of sugar coating. You must understand the fundamentals Tcl, TclOO, XML, HTML5, and CSS for this system to be of any use.

A Brief History

Content development is the most expensive part of any project, be it artwork for a game or an expert system model for a ship. This project, oddly enough, started with the former, and ended up supporting the later.

I had just finished a Tk display engine for a text-based game, and was eager to start developing content. It was the fall of 2018. Time for the leaves to start to changing color, and Apple to release its latest OS. And just before our conference, I discovered that Apple manage to wreck Tk on the Mac. Again. And it got me to thinking about ways to future-proof the content.

As I was pencilling ways to have one set of story elements target a captive browser in either Tk or embedded http server, when I was presented a problem at work. We had a documentation tool that generated volumes of cards used by our US Navy customers, in PDF form.

We had just signed a contract with the Republic of Korea Navy to produce similar documentation for their ships. On the surface, things should have been easy. Unfortunately PDF's ability to render logographic languages (like Hangul) is somewhat limited. I needed to find a new format, and quickly.

The answer was, oddly enough, HTML. With the right structure, HTML is very printable. As a format it is completely UTF-8 savvy. Essentially if a text can be produced on a keyboard, HTML can display it and print it. The format even supports embedding SVG data into documents to integrate text and vector graphics. Which, as it turns out, was fantastic for including little maps inside of the documents without having to create a brittle file system to store images.

But the flexibility has a cost. The layout for a printed document is VERY different from an interactive application. I needed a system that could allow one content engine to support both.

Rules and Exceptions and Exceptional Rules

Producing and interpreting the XML and HTML formats requires reading quite a bit into the content writer's intent. It also requires and understanding of the display engine's implementation. On the XML side, this is resolved by pointing to namespace specifications. Though mostly this involves pointing fingers, as nobody can honestly READ those specifications. For HTML, this is resolved by tomes of commentary on every incarnation of the standard, and how every incarnation is actually interpreted by the myriad browsers in the market.

Tools like TDOM are brilliant at ensuring one's document is grammatically correct. But, TDOM can't tell you if the structure of that document is correct. Essentially your document needs to have a structure ahead of time, then TDOM can enforce it.

Having a structure ahead of time sounds like a straightforward requirement for IT. One has requirements. Those requirements beget rules. Rules infer structure. The problem with my card generator project in particular, and many projects in general, is that rules don't come from one and only one source. Rules can (and often do) come from many sources. Those sources of rules are under no obligation to play well with one another. In the card generator case, the product is governed by 3 different rules sources: Expert System Rules, Visual Language Rules, and finally Naval Culture Rules.

I could spend a paper describing each. So let's summarize the impact as thus: the display engine needed to easily support a lot of custom reports in several formats. So now let us get down to the display engine itself.

Cuneiform

Cuneiform was written as a way to give all of the rules a place to interact before we start cutting a final document. It is network of object model on top of the standard HTML/XML Document Object Model. My personal style is to leverage the power of Tcl whenever possible. Even if that makes code look a little lumpy in spots. There's no language to learn for Cuneiform, but there are a lot of design patterns. The principle design pattern is that all objects in our Document Object Model are, in fact, TclOO Objects. And each of those TclOO objects wants to reduce itself to a string in the final rendering.

Documents

Document objects are the objects that are exposed directly to the outside. They represent the completed state of the output. Most of their methods either spawn subordinate objects or assemble the final deliverable.

Containers

Containers are objects which are envelopes for other objects. If we think of our document as a file system, containers are the folders. If we are using metaphors from Tk, they are like a frame or a canvas. They can container other containers.

Nodes

Nodes are the otherwise indivisible "producers of output" for our document. Nodes can be a block of text, an image, a function call, or a database query. While the output they produce can be static or dynamic, we know that we can't drill any deeper than a node.

Structural Symmetry

Every abstraction can still perform the same interactions as a lower level component. Every document is a container. Every container is a node. They may have to tailor out certain interactions because they no longer make sense, but they do so in a polite and consistent way. This allows a script to use a one-size-fits-all approach to interaction with nodes, containers, and documents. Likewise, lower level nodes have the same methods as the higher level abstractions. A node can answer a query to "list of children" with a simple empty list. The structure of every component is Cuneiform is identical, we distinguish types by altering their behavior.

Example - Basic HTML

Let us say we want to generate an HTML document that emits "Hello World!"

```
package require cuneiform

cuneiform::document.html create HTML html         ; # Create an object HTML
HTML cuneiform_structure                          ; # Build the "skeleton" of our document
HTML eval {
  set title {Hello World!}
  title $title                                    ; # Replace the title in HTML headers
  my tag h1 content $title                        ; # Add an H1 block with the title string
                                                  ; # Generate a paragraph of text
  para {
This is a demonstration of the [emph power] of
[link http://www.etoyoc.com/fossil/clay Cuneiform]
}
} ; # End eval
puts [HTML html_output]
```

We get back:

```
<!DOCTYPE HTML>
<HTML>
<HEAD>
<title>Hello World!</title>
 <META charset="UTF-8">
 <style media="screen" type="text/css"></style>
 <style media="print" type="text/css"></style>
 </HEAD>
<BODY>
<header id="header"></header>
 <DIV id="top"></DIV>
 <DIV id="output"><h1>Hello World!</h1>
 <P>This is a demonstration of the <I>power</I> of
<A HREF="http://www.etoyoc.com/fossil/clay">Cuneiform</A></P>
 </DIV>
 <DIV id="sideimg"></DIV>
 <DIV id="bottom"></DIV>
 <footer id="footer"></footer>
 </BODY>
</HTML>
```

We can clearly see that Cuneiform leverages the power of Tcl as a control language. The "eval" method, for instance, allows us to execute a block of code inside of the namespace of the HTML object's content. This environment provides us full access to the HTML object's methods, it's children, and commands embedded in its namespace to simplify common interactions. `title`, for instance, replaces the content of the `<TITLE>` tag inside of the `<HEADER>`. `para` formats a `<P>` block, and passes the incoming string through a subst. `emph` wraps the text in `<I>`. `link` formats a hyperlink.

HTML Tables

Developers who are familiar with HTML document may suddenly cry shenanigans. Normally the HEADER tag is laid down before the BODY. And, indeed, with Cuneiform this is also the case. But that process doesn't happen until we invoke `html_output`. Right up until that point, any element of our HTML document can be appended, replaced, or deleted. Thus, despite the `<HEADER>` and `<TITLE>` being laid down during our call to `cuneiform_structure` we can modify their contents.

This seems quite trivial so far, but let us consider the case of a complex table:

Toplevel	System	Type	Unit
Firemain	SWS	VALVE	FM123 (1-122-1)
		VALVE	FM331 (1-124-4)
	AFF	VALVE	AFF33 (1-140-3)
Power	60HZ	POWERPANEL	Power Panel (1-124-3)

In HTML we can achieve this effect with the `ROWSPAN` field. But, if we are retrieving our content from a database query returning a stream of records, we may not know the number of rows we are merging until the very end.

```
set TABLE [my tag table width 100%]                    ; # Create object for <TABLE> tag
set toplevel NULL
set toplevelColumn ::noop
set system NULL
set systekColumn ::noop
set tsystem 0
db eval {
select * from devices where isolated_compartment=:compartmentid order by toplevel,system, name
} record {
  set row [$TABLE row]
  if {$record(toplevel) ne $toplevel} {
    set toplevelColumn [$row column class header content $record(toplevel)]
    set toplevel $record(toplevel)
    set tcount 0
  }
  $toplevelColumn configure rowspan [incr tcount]
  if {$record(system) ne $system} {
    set systemColumn [$row column class header content $record(system)]
    set system $record(system)
    set tsystem 0
  }
  $systemColumn configure rowspan [incr tcount]
  $row column content $record(type)
  # Newly created tags are themselves objects capable of spawning new tags
  [$row column] tag a href /equipment/$record(eqptid) content "$record(name) ($record(location))"
}
```

We have taken a complex HTML layout, and fit it into the flow of a database reading a stream of records. The ::noop command is a handy fake procedure that accepts any input and has no body:

```
proc ::noop args {}
```

The rest of the loop is detecting if the "toplevel" or "system" fields have changed, and perform the HTML wizardry to inject a column in the right place. If you've ever implemented a similar structure while also streaming out the HTML you know that there are some, shall we say, complexity to it all. Very often this requires splaying one's query into an intermediary form, such as a dict or array, and then counting the elements before generating the HTML. Or simply living with layouts that look like a spreadsheet.

HTML Forms

Forms are another area where the quoting rules for Tcl and HTML/XML gets… nasty. But in Cuneiform, the code looks quite appetizing. Here is an example taken straight from some production code:

```
clay::define  ::paphmis::content {
  method SearchForm object {
    set result {}
    set form [$object tag form action /search method post]
    set sel [$form tag select name type]
    set formdata [my FormData]
    $sel tag option value any content Any
    dict for {type info} [::paphmis::content.search clay get search] {
      $sel tag option value $type content [dict get $info desc] \
        selected [expr {$type eq [dict getnull formdata type]}]
    }
    $form tag input type text name searchstring value [dict getnull $formdata searchstring]
    $form tag input type submit name go value "&#x1F50D\;"
  }
}
```

The code is run inside of a dynamic content generator implemented in the httpd module from Tcllib. Some things to note:

- The place in the document to insert the form is given as the argument *object*
- The method does not return a value

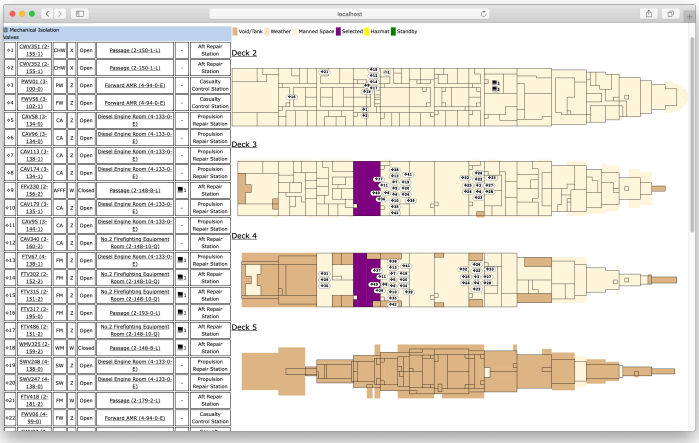
- The only HTML specific code is the Unicode escape for the string representation for the "go" button.
- Values are coming in from the http request via the `httpd::reply` method `FormData`
- We don't have to worry about closing tags!

Embedded SVG

The power of Cuneiform extends beyond straight HTML. Children for tags can be from a different XML namespace.

```
set SVG [my tag svg xmlns http://www.w3.org/2000/svg viewBox {0 0 100 100}]
set A [$SVG tag a href http://www.ilovecircles.com]
set G [$A tag g id mygroup css:fill red]
$G tag add circle cx 50 cy 50 r 40 stroke black stroke-width 3
```

Basically, we create the <SVG> tag, and add and configure elements under it just like we would any other tag in HTML. This despite the fact that SVG is actually straight XML with slightly different rules than HTML. And, just like the rest of Cuneiform, we don't have to deal with quoting hell, and the tags close themselves.



I realize this example is trivial. Now imagine instead we had to produce an document on the right.

For the hyperlinks to work properly in all browsers, the SVG data needs to be embedded *in* the html. An `<img>` tag inclusion won't do. And we don't just have one SVG image. We have one image for each of several decks of the ship. The lines for the walls are pulled from one table in a ship description file. The shapes for the clickable polygons are pulled from another table. The overlay of equipment is pulled from yet a third.

Foundations of Cuneiform

The overall theme of Cuneiform is a prototyping system. Everything starts of life as a generic "object', and decisions made during that object's lifespan mix in specialized behaviors. But before we get to the dessert, we have to eat our vegetables first.

Clay Framework

The implementation builds on the Clay framework, which has been discussed in earlier papers[1] and for which a manual is now available online[2]. Clay is my personal shorthand for a concepts and design patterns that I find myself re-implementing with every library I write. I'll try to keep the specifics of clay to a minimum, but if you see a language in Camel Case, odds are it's a clay framework keyword.

Class ::clay::yggdrasil

`clay::yggdrasil` is a design pattern for tree structures, built on top of clay. It consists of a handful of methods for managing links between objects. The most common link being between parent and child. Yggdrasil also allows objects to understand the difference between attributes of the object itself, attributes expressed as XML values, and attributes managed by cascading style sheets (CSS). The main methods provided are *child*, *config*, *css*, *eval*, *link*, *source*, and *uuid*. Several of those public methods have a herd of private methods (*Child_**, *Config_**, *CSS_**, *Link_**) to support them. For specifics, see the manual[3].

Optimizing in C

Producing SVG images from external sources requires quite a bit of mathematical manipulation. To support this project, I had to add several C accelerated routines to transform generic vectors into the myriad of forms that SVG expects. These functions also allow the objects to perform bounding box checks as if they were cast onto a virtual canvas. May utilize a bounding box Tcl_Obj type that is created by Odielib[4].

Command	Arguments	Description
::sprite::svg::bbox_overlap	*bbox bbox*	Evaluate if two bounding boxes values overlap
::sprite::svg::coords_to_bbox	*x1 y1 x2 y2*	Produce a bounding box value from canvas style notation.
::sprite::svg::rect_to_bbox	*minx miny sizex sizey*	Generate a bounding box value from information known about an SVG Rect tag.
::sprite::svg::arc_to_path	*start end x1 y1 x2 y2*	Convert information about a canvas arc into an SVG path
::sprite::svg::ellipse_to_bbox	*cx cy sizex sizey*	Convert information from an SVG ellipse tag into a bounding box value.
::sprite::svg::coords_to_ellipse	*x1 y1 x2 y2*	Convert canvas style coordinates into an SVG ellipse argument string
::sprite::svg::coords_to_ellipse_dict	*x1 y1 x2 y2*	Convert canvas style coordinates into a dictionary suitable for feeding to Cuneiform
::sprite::svg::bbox_to_ellipse	*bbox*	Convert a bounding box value into an SVG ellipse argument string
::sprite::svg::bbox_to_ellipse_dict	*bbox*	Convert a bounding box value into a dictionary suitable for feeding to Cuneiform
::sprite::svg::coords_to_rect	*x1 y1 x2 y2*	Convert canvas style coordinates into an SVG rect argument string
::sprite::svg::coords_to_rect_dict	*x1 y1 x2 y2*	Convert canvas style coordinates into a dictionary suitable for feeding to Cuneiform
::sprite::svg::bbox_to_rect	*bbox*	Convert a bounding box value into an SVG rect argument string
::sprite::svg::bbox_to_rect_dict	*bbox*	Convert a bounding box value into a dictionary suitable for feeding to Cuneiform

The canvas coordinate transformations are to allow a shape library inside of the Integrated Recoverability Model[5] to be rendered in SVG. Note: These conversion routines are used inside of my test application, but are not invoked by the Cuneiform library itself. Cuneiform just cares about the XML attributes.

In my example image you see a lot icon overlays. This is accomplished with the following proc:

```
proc ::paphmis::ComptOverlay {SVG deckid unit xscale yscale iconsize obj flags} {
  if {![llength $flags]} return                         ; # Nothing to display? End early
  set fontsize [expr {$iconsize*0.75}]
  set bbox [$obj clay get bbox]                         ; # The polygon shape stored the bounding box extents
  set center [::odie::bbox::center $bbox]               ; # Calculate the center of the bounding box
  set iconrad [expr {$iconsize*.6}]
  foreach item $flags {
    set info [dict create fill violet opacity 1.0]
    switch $item {
      C {
        dict set info fill tomato
      }
      M {
        dict set info fill lightgreen
      }
      F {
        dict set info fill paleturquoise
      }
      E {
        dict set info fill khaki
      }
      V {
        dict set info fill skyblue
      }
      default {
        dict set info fill white
        dict set info stroke black
        dict set info stroke-width 0.5
      }
    }
    # Calculate the size of our icon
    set ry [set rx [expr {$iconsize*0.5}]]
    if {[string index $item 0] eq "&"} {
      set i [string first "\;" $item]
      set len [expr {[string length $item]-$i}]
    } else {
      set len [string length $item]
    }
    if {$len==1} {
      set sizex $iconsize
    } else {
      set sizex [expr {$fontsize*$len}]
    }
    set sizey $iconsize
    # Starting from the center, fan out in progressively larger hexagons to find an empty space large enough
    # for this icon
    while 1 {
      incr idx
      set point [::vectorxy::hex_tile_center $idx $iconrad $center]
      ::vectorxy::assign $point thisx thisy
      set thisbbox [::sprite::svg::ellipse_to_bbox $thisx $thisy $sizex $sizey]
      set overlap 0
      foreach itembbox $items {
        if {[::sprite::svg::bbox_overlap $itembbox $thisbbox]} {
          set overlap 1
          break
        }
      }
      if {!$overlap} break
    }
    lappend items $thisbbox
    $SVG tag rect {*}[::sprite::svg::bbox_to_rect_dict $thisbbox] \
      rx $rx ry $ry \
      css $info
    $SVG tag text x $thisx y $thisy \
      content $item \
      css [list stroke black font-size $fontsize \
         font-family courier dominant-baseline middle \
         text-anchor middle opacity 1.0] \
  }
}
```

Content Management System

Another application I've found for Cuneiform is as a system to maintain by blog. Blogging software needs to combine both the content for the page, maintain a file system (for complex entries), and also provide annotations for the content index.

I'm a lazy programmer, so I also have a requirement that synchronizing my local copy (which I invariably end up running so I can make sure what I coded is what I get) is as simple as invoking Rsync. The system is a little quirky, but it works for my purposes.

```
Title: {Story Crafting}
Class: {blog}
Date: {Sat Jul 27 09:31:30 EDT 2019}
Content-Type: {html}
Format: {clay}
date: {Sat Jul 27 09:31:30 EDT 2019}
owner: 619eb03b-0f7d-490f-a2ac-9eb72e4c789d
--- BEGIN CONTENT ---
para {
I've spent the last few weeks learning the finer arts of
[link {https://www.youtube.com/user/lindybeige} Psychology],
[link {https://www.youtube.com/channel/UCFQMO-YL87u-6Rt8hIVsRjA} {Story Telling}],,
and [link {https://www.youtube.com/user/Drachinifel} {Naval History}]
from YouTube. Feel free to judge me.}

para {On one hand, time I could have spent on story lines was spent listening to
talking heads talk about craft. On the other, those hours learning the craft have saved me
man years of effort re-learning what they learned the hard way.
}

para {
The point here is that activity is not progress. I know I can write. I know I can
write a lot in a short amount of time. (At least if this blog is any evidence.)
If I could monetize writing the first three chapters
of a book, I'd be rich. I have had many a brilliant idea devolve into a degenerate mess.
This time around, I knew I had to do something different. I had to actually learn, not
just do.
}
```

The format it is a half-baked MIME using a dictionary instead of a sane notation. (The advantage being the parser in Tcl is simpler.) A delineation marker separates the headers from the content. You can also see that the format includes several commands in the object's namespace to make common patterns simple. The `[link]` command is simply a macro for `my tag link href $link content $string.`

What draws me to the format is that, if I steer clear of raw HTML, I can run that set of expressions to render onto a Tk text widget, a canvas, a plain text file, or a man page. The DOM can also do styling in the background. At least where the notation doesn't provide specific guidance.

Conclusions

Cuneiform uses a stylized form of TclOO to represent a document object model in a form that is comfortable for a Tcl programmer. It is designed to allow an investment in content development to survive technology changes. Using Tcl as an expression engine allows content writers to represent complex ideas without resorting to wonky syntax. (Well, at least no wonkier than Tcl.)

References and Citations:

[1] "Clay - A Minimalist Framework for TclOO Libraries", Sean Woods
https://tcl.tk/community/tcl2018/assets/talk132/Paper.pdf

[2] "Clay Framework Reference Manual", Sean Woods
http://fossil.etoyoc.com/fossil/clay/doc/trunk/htdocs/clay.html

[3] "Clay Yggdrasil Reference Manual", Sean Woods
http://fossil.etoyoc.com/fossil/clay/doc/trunk/htdocs/clay-yggdrasil.html

[4] See Odielibc:
http://fossil.etoyoc.com/fossil/odielib

[5] For information about the Integrated Recoverability Model, see:
http://www.tnesolutions.com