Scintilla Tk Porting Project

Brian S Griffin
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070
brian_griffin@mentor.com

Abstract

A couple of years ago we decided that the application window used to display program source file text was inadequate in several ways. The basic problem was with the performance of a few significant features: keyword/syntax coloring, inline annotations, and large file load times. There have also been recurring problems with keeping displayed line numbers in sync with the text body. Rather than continuing to attempt repairs on the current implementation, a decision was made to find a complete replacement. After reviewing and testing a number of applications and widgets, we narrowed the field down to two options: customizing the Tk text widget, or porting Scintilla to Tk. This paper will discuss the requirements and implementation challenges with ScintillaTk.

Introduction

In 1995 when development started on our Tk-based GUI, a simple Tk Text widget was used to display program text. The simple Tcl code implementing the source code viewer was more than adequate to meet the needs at that time. As features were added to the application, the demands on the source view grew. The Supertext[1] widget was added to support undo. In 2004, an effort was made to provide a more IDE-like experience in the application; we decided to incorporate a Tcl/Tk based text editor developed for another product in the company. This editor widget gained widespread use across several products.

Although the editor was a bit buggy, the real problems began when we attempted to implement source code value annotations in 2006. Annotations display values above or below variables that appear in the source code. Implementing this using Tcl/Tk is a challenge. Various techniques were attempted until we finally settled on using double-spaced text and overlaid label widgets below each location. This gave a clean presentation and allowed for easy and efficient update to values as they change,

---

[1] Supertext extension of the Tk text widget: http://wiki.tcl.tk/3387

however, when scrolling the text, the labels had to be removed before the scroll and laid back down after the scroll. This leads to slow scrolling performance and a lot of flashing.

After living with the flashing and buggy editor implementation, a decision was made to start over. We searched for existing source code viewer widgets that provided all the necessary features. All possible solutions were considered, from using an external editor to writing a fully custom widget. The Scintilla widget quickly gained favor primarily because of the annotation, line number, and folding features. In this paper, I will describe the project of porting Scintilla to work with Tcl/Tk as a Tk widget, the Scintilla architecture. I will also describe the challenges faced and solved, or not solved as the case may be. In the conclusion I will offer some suggestions for possible improvements to Tk and Scintilla.

Problems with [text]

To begin, I have to say that the Tk text widget is an awesome implementation. It is amazingly fast compared to all other text viewers we've tested. The feature set is really good, especially the powerful tag system. And the peer[2] feature is pretty cool.

There are some things that are hard to do or hard to do right. One of the most common things done with the text widget when displaying source code is to display line numbers along the left hand column. There are at least 3 techniques to do this:

- Insert the line number with the text in the text widget.
  ```
  $text insert $index "$lineno  $text"
  ```
- Fill a second text widget with line numbers and pack this widget next to the body text widget.
  ```
  while {[$linetext compare end < [$text index end]]} {
      $linetext insert end "${lineno}¥n"
      incr lineno
  }
  ```
- Actively draw line numbers in a canvas packed next to the text widget.
  ```
  set dline [$text dlineinfo ${i}.0]
  set y [lindex $dline 1]
  $canvas create text 0 $y -anchor nw -text $i
  ```

The editor was implemented using the second technique – using a second text widget to contain line numbers as well as other indicators, marks and counters. Inevitably, bugs would creep in over time, causing the contents displayed in the two widgets to be

---

[2] TIP #169: Add Peer Text Widgets (http://www.tcl.tk/cgi-bin/tct/tip/169.html)

out of sync. Most recently, the change in the text widget to support smooth scrolling caused a synchronization problem in the editor.

The rich tag mechanism is a great feature of the text widget. One of the most visible uses is for syntax highlighting of source code. However, a Tcl-based lexer is not the most efficient way to process large text files. Our current editor employed such a lexer and consequently, the load time performance suffered with very large files.

Annotating the source code with values is a desired feature. We first attempted to make use of tags to implement annotations. We tried to -offset[3] the value text, but without an x (horizontal) offset option, the desired effect could not be achieved. We also tried embedded windows[4] (widgets), but again there is insufficient offset placement control to achieve the desired effect.

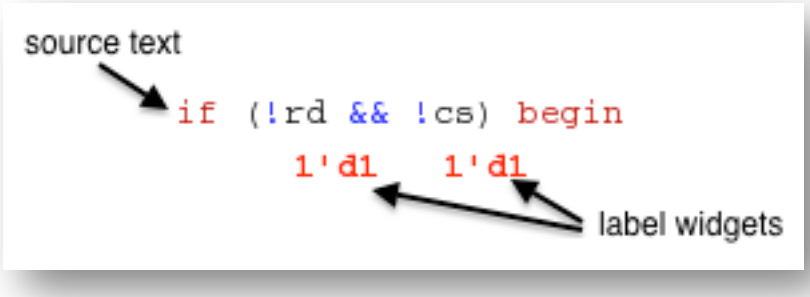

Figure 1. Label widgets placed under the source text to display annotated values. ("1'd" means 1-bit wide, decimal radix followed by the value.)

The solution we settled on was to place label widgets over the double-spaced text widget, using the bounding box information supplied by the text widget. The desired effect was achieved, but with the side effect of flashing when the window is scrolled, since a scrolling operation required removing and replacing the label widgets. In addition, the work involved was too much and delayed the scrolling action. Another undesirable trait of this approach is the double-spacing of the source text, which ends up wasting a lot of screen space, as illustrated in Figure 2a. Ideally, the annotation rendering would be handled directly by the widget so that it is all drawn at one time. Then the annotations would take up space only where they are needed, as shown in Figure 2b.

---

[3] Text widget tag vertical offset option. (http://www.tcl.tk/man/tcl8.5/TkCmd/text.htm#M37)

[4] Text widget embedded window feature. (http://www.tcl.tk/man/tcl8.5/TkCmd/text.htm#M49)

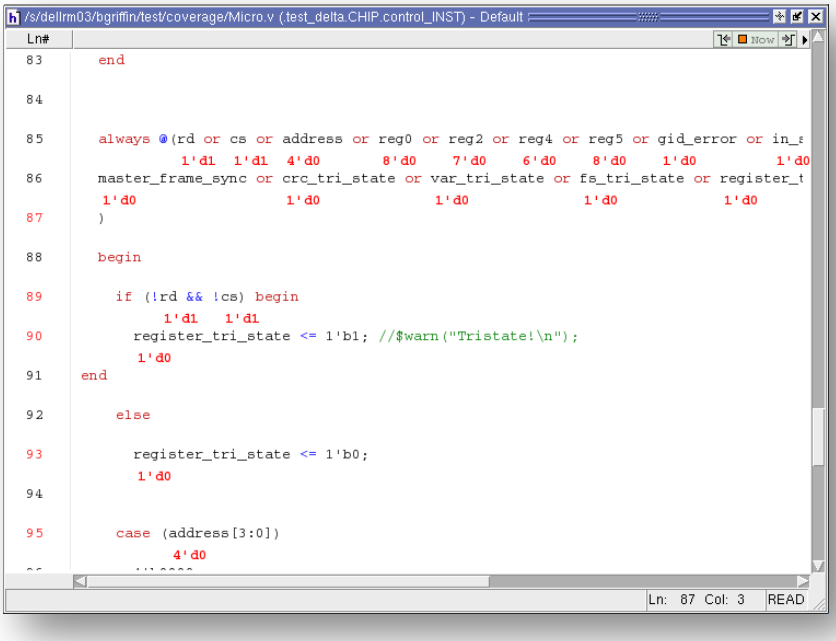

Figure 2a. Double-spaced source text to allow for placement of label widget annotations.

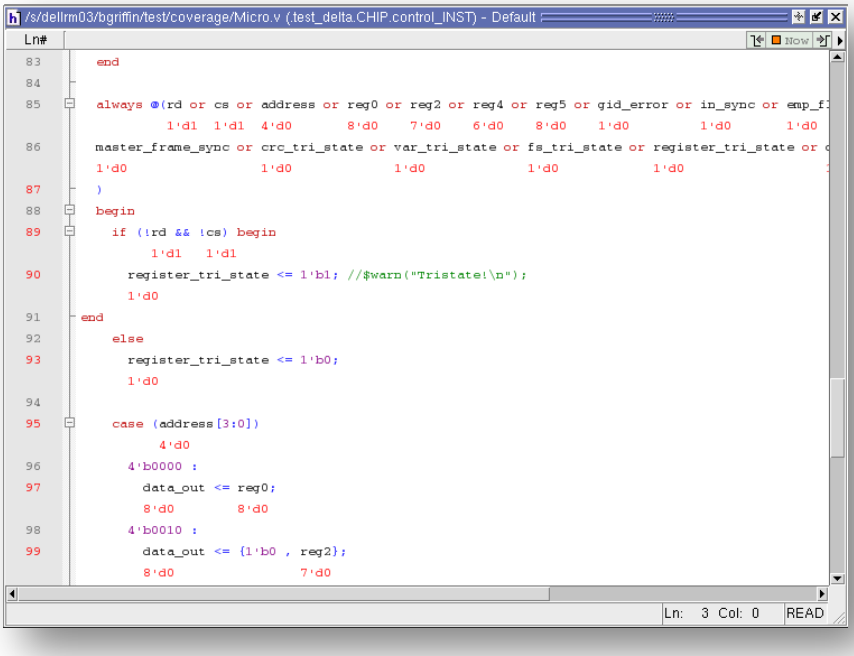

Figure 2b. With vertical space allocated only to annotations, 4 additional lines of source text can be displayed in the same screen space.

Replacement Criteria

In order to improve the user experience with the source code viewer, we identified key requirements that must be addressed.

Fast file load with syntax highlighting

The time to load and display a very large source file with syntax highlighting must be the best possible. A review was conducted of various IDEs and text editors to determine a nominal standard. The review was not exhaustive, but several popular tools were tested.

Embedded Annotations

A highly efficient and flexible mechanism is needed to annotate the source text without getting in the way of display performance or text selection and editing.

Margin Annotations

A mechanism to place marks and other data annotations in a margin to the left side of the text, by line, is required. Some common examples are line numbers, breakpoint symbols, bookmark symbols, and execution counts. At times it may be necessary to have markers overlap and still be recognizable, as shown in Figure 4.

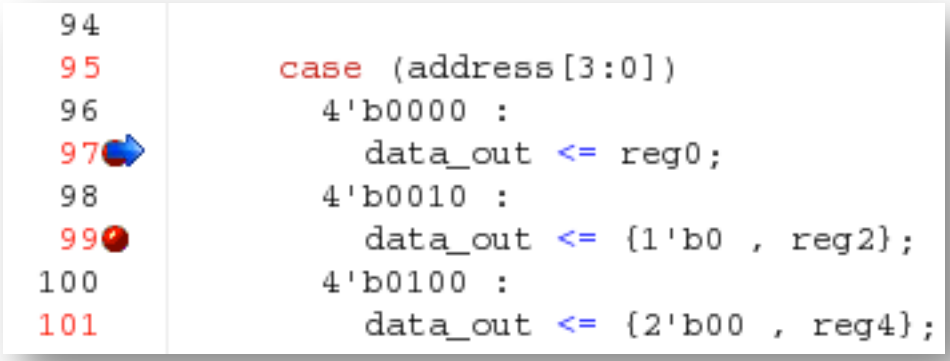

Figure 4. Overlapping markers. Red dots are breakpoints. Blue arrow is the current execution location.

Scope folding

Scope folding is a combination of special margin marks coupled with lexical parsing to identify language scopes. This feature is considered a lower priority, “nice to have” capability.

Solution space

Several possible solutions were considered to satisfy our high priority requirements. We tested several widgets and editors, including the Tk text widget. We even considered writing a text widget from scratch, but that was rejected as too large a project and the performance testing showed that there were several other viable candidates.

Some additional requirements, not listed above: the widget must work with Tk, be easy to implement within our application, and naturally interact with the rest of the GUI. Only the Tk text widget met these criteria. The next best candidate was the Scintilla[5] text widget, an open source code-editing component. Except for not supporting Tk, it meets all of the other required features.

This led us to two possible conclusions:

1. Using the Tk text widget, enhance it to support annotations and margins, C based lexers for syntax highlighting, and scope folding.
2. Port Scintilla to Tk.

Clearly #2 is the easier task as it only has 1 item!

Scintilla Architecture

What makes it even possible to consider porting Scintilla to Tk is the architecture of Scintilla; it was developed with porting in mind. This component has been ported to Gtk, Windows, OSX, ncurses, MorphOS, wxWidgets, and FOX. The developers have done a pretty good job of isolating the platform requirements from the editor. A platform header file provides the definition of a few virtual classes which, when implemented, provide all the facilities needed to support Scintilla editing and presentation.

- Font – allocate fonts, provide font parameters
- Surface – abstract place to draw
- Window – access to window manipulation and properties
- ListBox, Menu – Widgets for menus and dialogs
- ElapsedTime – timer access
- DynamicLibrary – abstract dynamic library loading facility
- Platform – used to retrieve system wide parameters such as double-click speed and chrome color.

---

[5] Scintilla is maintained by Neil Hodgson and hosted on Sourceforge. The web site is http://www.scintilla.org

The key interface classes here are Surface, Window, and Font. These provide for the crux of the rendering engine. The Platform, DynamicLIbrary, and ElapsedTime complete the necessary general facilities. The ListBox and Menu widget interfaces are only necessary if the Scintilla editor is going to manage Auto completion or Popup menus, respectively, otherwise, the platform may use other means to implement these behaviors.

All operations on the component are done through a generic message interface that looks like this:

```
sptr_t sci_cmd(
                sptr_t ptr,
                unsigned int iMessage,
                uptr_t       wParam,
                sptr_t       lParam
);
```

As of this writing, there are 645 different messages.

The component also separates the notion of Document from Window; Documents hold the text contents and styling, and a Window presents a Document on the screen, similar to the Tk text peer feature. From the Scintilla documentation:

> "A Scintilla window and the document that it displays are separate entities. When you create a new window, you also create a new, empty document. Each document has a reference count that is initially set to 1. The document also has a list of the Scintilla windows that are linked to it so when any window changes the document, all other windows in which it appears are notified to cause them to update. The system is arranged in this way so that you can work with many documents in a single Scintilla window and so you can display a single document in multiple windows (for use with splitter windows)."[6]

So, how hard could it be to marry Tcl/Tk, a portable GUI toolkit, with Scintilla, a port capable text-editing component?

Implementation

One of the driving goals in implementing this port was to take maximum advantage of the Tk API in order to maximize portability of the port. In other words, we want the widget to work wherever Tk works without having to deal with OS specific platform

---

[6] Scintilla Documentation, section Multiple Views (http://www.scintilla.org/ScintillaDoc.html#MultipleViews)

issues within the ScintillaTk port code. After all, that is Tcl/Tk's philosophy[7] as well. In the descriptions below you'll see how this is applied and how successful it is.

Surface

The Surface class was probably the most straightforward to implement. This class has methods like FillRectange and DrawText that align closely with Tk C API calls. Almost every method is implemented with just a few lines of code. The one limitation is the rendering with alpha (transparency) channel in colors. Currently the alpha channel is ignored since the current attempted implementation has poor performance. This is a known limitation in Tk.

Fonts

Fonts are implemented as a wrapper class around the Tk_GetFont() API and a bit of code to manage the font references. Scintilla does require the use of Xft in order to support the Font property requests, which include arbitrary float sizes. Older X11 fonts aren't sufficient and produce less than desirable results.

Images

The Scintilla interface for images is based on a raw 4-plane image data array: rgb color + alpha channel. This format is compatible with Tk_GetImage, however, there is no Tk API to directly draw a raw image to a drawable. To draw the image it must first be put into a Tk_Image. This presents a challenge in managing images from Tcl/Tk since they pass through Scintilla as an unmanaged array. There isn't any way to associate a Scintilla image draw request with a particular Tk_Image.

There are two reasons why I believe Scintilla has a single DrawImage request using raw image data: 1) some of the images drawn are internally created by Scintilla, not supplied by the client or platform, and 2) Scintilla can optimize a redraw by requesting a partial image redraw. This single interface keeps things simple for the components implementation.

It would be helpful if Scintilla defined a class or opaque handle for images and then let the platform implementation manage the resource. It would also be helpful if Tk exposed the TkPutImage() as a public function.

Encodings

Scintilla supports both UNICODE and UTF-8 encodings internally as a runtime configuration option. For the Tk platform, keeping Scintilla set to UTF-8 encoding

---

[7] John K. Ousterhout, Ken Jones, [et al], 2010, *Tcl and the Tk Toolkit – Second Edition*, Addison-Wesley, page xxxii.

makes it simple to interface text between Tcl and the Scintilla Document since Tcl is all UTF-8 internally.

Loadable Lexers

One of Scintilla's features is that it comes with a number of programming language lexers used to perform syntax highlighting. It can also be configured to make lexers loadable dynamically. It is up to the platform implementation to provide the loading mechanism. This was simple to implement using Tcl's Tcl_LoadFile API, however, the set of symbol entry points that Scintilla needs are not the same pair of entry points for which LoadFile was designed. The platform DynamicLibrary class also requires a FindFunction method. We decided to make use of TclpFindSymbol() in the implementation. This is currently a problem since this function is not easily exposed.

Tcl command interface

To complete the porting effort, the Scintilla component needs to act and feel like any other Tk widget. This means integrating the Scintilla component into the Tcl/Tk command & widget infrastructure. The code here constitutes about 80% of the implementation. For this part of the project we set the goal to match the Tk text widget subcommands and options as much as possible within the limits of what Scintilla can support. For example, we matched the scrollbar interface, but since Scintilla does not support embedded windows, that part of the Tk interface is left out. In some cases, code from the Tk text widget was borrowed in order to match the behavior. This was done for the "search" subcommand, for example.

All of the important text widget subcommands have been implemented for managing the widgets' content and display. For the most part, the Scintilla widget is compatible with the Tk text widget. Figure 5 lists the widget subcommands. Those that are underlined in the figure have been added to support Scintilla specific features. Tk text widget subcommands that have not been implemented are shown in the figure printed in grey italic.

A couple of items cannot be supported in Scintilla; they are embedded images and windows. Scintilla simply has no mechanism to duplicate these Tk text widget features, therefore, these subcommands have not been implemented. The tag subcommand implements all the same Tk text widget methods, but a tag's options are different for Scintilla since the styling of text, margins, and annotations diverge from Tk text widgets' design.

annotate	*dlineinfo*	keywords	scisearch	xview
bbox	dump	loadlexer	search	yview
cget	edit	margin	see	zoomin
compare	fold	mark	style	zoomout
configure	get	marker	tag	
count	*image*	*peer*	textwidth	
debug	index	property	*window*	
delete	insert	*replace*	version	

Figure 5. Scintilla widget subcommands. Underlined subcommands are added for Scintilla. Greyed out italic subcommands have not been implemented.

Style vs Tag

In Tk's text widget, formatting of text can be performed by applying a tag to one or more arbitrary ranges of characters. A tag's definition holds the formatting properties to be applied to the text. These properties include color, font, and shading (stippling), but there are also additional controls over things like spacing, margin, justification, offset, and even event binding, a very flexible and powerful mechanism. Scintilla, on the other hand, has the notion of Styles that are applied on a character-by-character basis. Style only affects the look of the character, color, font, weight, etc., and not any other rendering considerations. There are a few predefined styles, and the number of styles is limited to 256.

Since Tk text widget tags are effectively unlimited, this presents a challenge to implementing tags in the Scintilla widget. In order to present the appearance of an unlimited number of tags in Scintilla, a Tag Manger is implemented to keep track of tag names and ranges, and to manage the allocation of styles as needed. One of the heavier uses of tags in the text widget tends to be as meta-data on the text. This use model does not require consuming any style resources since no formatting changes are applied to the tag. So far we have not seen any practical case of hitting the limit on styles. Styles can also be applied to margin text and annotation text. Luckily, these styles can be configured to be a different set from those used in the document body.

Margins, Annotations, Indicators, Markers

Scintilla provides a number of different ways to augment the document body with additional information: margins, indicators, and annotation.

- **Margins and markers**: Scintilla provides for up to five margins. There are three types of margins: symbol, number, or text. The margins are configured using the margin subcommand. Symbols can be set in the margins using either the built-in Scintilla markers, or using Tk images. Markers are configured using the markers subcommand.
- **Indicators**: Indicators are drawn on the text in the document body. An indicator is specified via the tag subcommand.
- **Annotations**: These are read-only lines of text that appear under editable lines of text in the document. The text is styled independently from the document text. Annotations are configured via the annotation subcommand.

Lexers

Scintilla provides a number of built-in lexers for a variety of programming languages. Additional lexers can be dynamically loaded using the loadlexer subcommand. The configure option, -language, is used to set the name of the lexer language to use. The lexer is then configured using the "keywords" and "property" subcommands. Each lexer defines the number of keyword sets, and the names and types of properties. In order to use these subcommands correctly one must become familiar with the particular lexer in use.

Searching

Scintilla provides a search function, as does Tk. Because there are differences in the options and results of these two functions, we decided to provide both search subcommands in the widget. The search method uses the Tk text widget algorithm to perform the search. The code for this was copied from Tk. The second method, scisearch, calls the Scintilla function to perform the search. The differences come about because they use different regular expression engines, and they have different definitions of a word. Rather than trying to accommodate or justify merging the behaviors, providing two subcommands was just easier.

Textwidth and Zoom

Other subcommands provide direct access to Scintilla.

The textwidth subcommand is similar to the [font measure] command in Tk, except that Scintilla performs the calculation.

The zoomin and zoomout methods provide a capability specific to Scintilla.

Not all specialized calls that Scintilla provides have been implemented; to date, only the calls that have been needed have been implemented in the widget interface.

Conclusions

Scintilla has a good foundational architecture that made the porting effort relatively easy. The work here is not complete; there is plenty to do to evolve this widget extension into a powerful text component for Tk.

Some of the remaining work needed to complete the port:

- Complete all Tk text widget configure options and subcommands to improve compatibility.
- Implement event binding for tags.
- Create a custom subset of Tk text widget tests to verify compatibility.
- Complete access to all Scintilla API calls and options.
- The configure script(s) for all Tk supported platforms. The current configure script has only been tested on some Linux machines, and only works on Windows under specific conditions.
- Full html documentation.

There are a few things that could be done in each project that would benefit the integration:

- It would probably be better if all lexers were dynamically loaded as a Tk package, one per language.
- Create TIP's to expose the few internal functions needed to complete the Tk port and keep the ScintillaTk portable and isolated as much as possible.
- I would like to see ScintillaTk brought into the Scintilla project directly since most of the implementation variations will come from Scintilla itself. The platform port intentionally uses Tk's public API (with already noted exceptions), which keeps the port isolated from the Tcl/Tk releases.

Acknowledgments

I'd like to thank the developers who did the porting work: Joe Mueller, John Vella, and Ron Wold. It's their "boots on the ground" that are getting this job done. Thanks, guys!