Realizing the theme dream

Marc Culler

October 14, 2019

Abstract

The integration of the Tile project into Tk 8.5 with TIP #248 had two goals. One was to offer a better way of designing and building widgets while the other was to provide application developers with the ability to write a single script that would provide a user interface that looked good on many platforms. The first goal was largely achieved. Much progress was made on the second but, especially for macOS, it was not complete.This note describes recent progress towards realizing the theme dream for the Aqua theme.

1 Introduction

By introducing a flexible and structured process for creating widgets and controlling their appearance, the Ttk package opened up the possibility of writing a single Tk user interface script which would look like a "native" interface on all of the platforms but contain only a minimal amount of platform-specific code. It also allowed adding new widgets for individual platforms which mimic native widgets that may not exist on other platforms. Significant progress was made on implementing native themes for linux, Windows and macOS. However, especially in the case of macOS, this work was incomplete. And, to complicate matters, OS vendors (in particular Apple, with the "Dark Mode" included in the Mojave release) have begun adding variants to the native theme which are intended to be controlled at the system level by setting user preferences.

With Tk 8.6.10 some progress has been made toward the goal of producing native-looking interfaces for macOSwhich also respond to the user preference setting for "Light Mode" or "Dark Mode". In this note we will summarize the new features that have been added as well as features that still need to be implemented. At the end, we will discuss some of the technical issues that arose in doing this.

The black-and-white format of this document means that it would be pointless to include screen shots which display the changes made to widget colors. So, unfortunately, we will forgo screen shots here and just try to describe in words what has happened.

2 Colors

The difference between “Dark Mode” and the former default now known as “Light Mode” is, as the name suggest, entirely about colors. Since before the beginning of macOS(formerly OS X) Apple has had a color naming scheme for system colors in which they are named for the purpose that the color serves in the user interface. For example, macOS10.0 included such a “semantic” NSColor named windowBackgroundColor which was intended to be used as the background color of an App Window. This concept was leveraged in order to implement “Dark Mode” by allowing the RGB values of a semantic color to be time dependent (as well as being dependent on the OS version) and to change when the user changed from “Light Mode” to “Dark Mode” or back.

The implementation of system colors in Tk predated the release of macOS10.0, and the system colors which were supported by Tk were inherited from the Carbon window manager. These colors continued to be supported by the Cocoa window manager, so they continued to work in Tk. However, those colors had fixed RGB values and hence did not change appearance when a user changed to or from “Dark Mode”. Consequently, one important change needed to allow Tk user interfaces to change color according to the “Mode” was to add several new system colors, including systemTextColor, systemSelectedTextColor, systemLabelColor, systemControlTextColor, systemDisabledControlTextColor, systemSelectedTabTextColor, systemTextBackgroundColor, systemSelectedTextBackgroundColor,
systemControlAccentColor. Some of these did not exist prior to macOS10.14 and had to be simulated. In addition, 7 gray colors named systemWindowBackgroundColor$N$, $N = ...7$, were added to support ttk::notebook and ttk::labelframe widgets.

2.1 App windows

Apple’s Human Interface Guidelines [1] lists five types of windows which are used in macOS: Document Windows, App Windows, Panels, Dialogs and Alerts. Here we are primarily concerned with App Windows, which are the ones which contain arrays of controls used to manage an application, such as the windows in Apple’s Preferences app. When building a TK application, the majority of Tk widgets will appear in what Apple would call an App Window. It is obvious to a macOSuser that all App Windows have the same background color, and in fact since OSX 10.0 there has been a “semantic” system color named windowBackgroundColor. This would have been a

natural choice for the default background color for Tk Windows and Buttons in the Aqua theme. However, since the first port of Tk to Apple's Cocoa window manager the default background color for Windows has been a color named (in Tk) as systemWindowBody which was left over from the earlier Carbon window manager. In plain English that color is named "white" and it never matched the backgrounds of Apple's App Windows [2, Background Colors]. Changing the background color to what is now known in Tk as systemWindowBackgroundColor was a small but significant step towards making it easier to produce Tk applications which look like they belong on a Macintosh. Of course, on Mojave and later systems this color will change from light to dark when "Dark Mode" is selected by a user.

2.2 Buttons

Many (but not all) of the Ttk button classes are closely matched by button types listed in Apple's Human Interface Guidelines [1] and hence can be used to build HIG compliant macOSuser interfaces. Here is a table describing the correspondence.

Appkit	Ttk
Checkbox	ttk::checkbutton
Push Button	ttk::button
Pop-up button	tk_optionmenu
Pull-down button	ttk::combobox -state readonly
Combo Box	ttk::combobox (default state)
Image Button	ttk::button (additional configuration is needed)
Radio Button	ttk::radiobutton
Bevel Button	ttk:button -style Toolbutton

Apple's Disclosure Control, Gradient Button and Help Button do not have direct analogues in Tk, although it should not be difficult to build Tk widgets which correspond to these. This is one direction where future work is needed.

All of the Ttk buttons above now change appearance when "Dark Mode" is selected.

2.3 Scrollbars

Starting from the first port of Aqua to the Cocoa window manager the Tk scrollbars have been problematic. The ttk::scrollbar was not implemented for macOSprior to Tk 8.6.10. Instead code in the Tcl library attempted to simply replace the ttk::scrollbar constructor with the Tk scrollbar constructor; this did not work correctly since the two constructor's have different signatures [2, Scrollbars]. More importantly, the scrollbars were drawn by the HIToolbox and the parameters accepted by the HITools drawing routine did not map to the size and location of the scrollbar on the screen in any obvious way. Consequently the scrollbar thumb was never drawn in the location where

Tk expected it to be, so mouse clicks in the thumb were sometimes interpreted as being clicks in the trough and vice-versa. Using the scrollbars could be a quite frustrating effort.

One important advantage of using HIToolbox to draw widgets is that it automatically produces widget appearances to match the style of the OS version where it is being run. This means that major appearance changes need not require any change to working code. However, while Apple continued to update the library with new OS releases, they did not provide documentation after the release of Cocoa. Moreover, Apple decided not to provide support for "Dark Mode" in the HIToolbox [3].

As it happens, there was a major change in the appearance of Apple's scrollbars between macOS10.6 and 10.7 but no change since then. As a less than perfect compromise, HIToolbox is now used to draw scrollbars (which behave erratically due to the long standing mismatch in dimensions) in macOS10.8 and earlier, but in newer versions the scrollbars are drawn directly. This allows them to respond to changes between "Dark Mode" and "Light Mode" and also for the thumb to be drawn with the size and location which Tk expects, so it works correctly. Also, the ttk::scrollbar was implemented in the same way, eliminating artifacts caused by the different function signatures.

2.4 LabelFrames and Notebooks

In the Aqua theme the ttk::labelframe and ttk::notebook widgets are realized by using the Apple native Group Box and Tab View respectively. These Apple interface elements are displayed similarly. There is a frame around a group of related controls or a notebook pane. A Group Box has an optional label above it and a Tab View has the tabs used to select notebook pages arranged in a bar centered on the top of the frame. In macOX 10.6 this frame was a raised line, making it similar to the X11 labelframe, but with the label moved upward. But in macOS10.7 the appearance changed to a recessed rectangle with rounded corners and a slightly darker background than the containing window. Moreover, when one of these grouping widgets was nested inside another the inner widget would have an even darker background. This was essential in order to create a contrast with its container after the recess had become so subtle as to be almost unnoticeable. And, to add further complications, in "Dark Mode" the contrast was reversed so that the inner regions were lighter than their surroundings.

The rounded corners and the recess, which grew more and more subtle with successive releases of macOSwere drawn by the window manager and so did not cause any trouble for Tk. But the darker background was problematic, due to the fact that the abstract model used by Ttk assumes that each widget has a fixed background color no matter where it is placed on the screen. A button inside of a ttk::labelframe or ttk::notebook page is expected to have the same background color as the same type of button placed outside. Since this problem was not addressed in previous releases or

ttk::labelframe or ttk::notebook page would have a dark region surrounding a lighter rectangle containing the group of widgets inside of the frame or page.

One might have hoped that Apple would have exposed the logic that they use to determine these contrasting backgrounds by having a "semantic" color which was aware of how deeply nested a widget is. But this does not seem to be the case. For Ttk widgets this problem was addressed by adding logic which determines nesting depth and sets a background color which matches the one that Apple uses. In addition it was necessary to deal with the possibility that a ttk::labelframe or ttk::notebook might contain a standard Tk widget. Unlike Ttk widgets, most Tk widgets have a configurable background color, and a developer will usually know how deeply nested the container of a Tk widget will be. So the solution here is to synthesize a family of "semantic" colors that behave the way that Apple's might have behaved. That is, in "Light Mode" successive colors in the family become darker while in "Dark Mode" they become lighter. In Tk 8.6.10 these are the colors named systemWindowBackgroundColor$N$, for $N = 1 \ldots 7$, allowing for nesting of depth up to seven. (Note, however, that Apple recommends that nesting Group Boxes should be avoided if possible.)

2.5 Other changes

There were a number of other less prominent changes which we mention here. In previous releases of Tk the column header on the ttk::treeview was realized by Apple's List Header Button, as drawn by HIToolbox. However, the appearance of the List Header Button has been several times with new OS releases, and the HIToolbox stopped updating the header appearance with the release of macOS10.9. Now HIToolbox is only used in 10.8 and earlier while in the newer OS versions the header is drawn directly so that it matches the Apple header and can change color appropriately when the user switches between "Light Mode" and "Dark Mode".

Some other minor changes were to add drawing code which can be used to draw a macOSSlider directly in an appropriate color for "Dark Mode". Apple actually has two different Slider widgets and the appearance depends on whether it is being displayed with tickmarks. While the ttk::scale widget does not provide an option for tickmarks, it was modified to respond to the Ttk alternate state by changing to the form of Slider that Apple uses with tickmarks. (This works in older systems where the slider is drawn with HIToolbox as well as in newer systems where the slider has to be drawn directly.)

Finally, the various entry widgets, spin-boxes, and combo-boxes were all modified so that they change color appropriately when "Dark Mode" is selected. Also, in the Ttk versions of the spin-box the arrow buttons had not been implemented in a way which matched the native appearance. The individual arrow buttons did not respond when clicked. This was corrected as well.

3 Technical issues

In this final section we would like to discuss the profound impact that the changes which Apple's upgrade from macOS10.13 to 10.14 had on Tk. Given that Tk seems to have survived, and now builds and runs on all versions of macOSfrom 10.6 to 10.16, the reason that these changes are of interest is that they affect some of the core assumptions made in the design of Tcl and Tk.

The initial symptom of these changes was that after upgrading to macOS10.14 and building Tcl and Tk, all Tk windows were completely black. No drawing was taking place. This was not true when running Tk built on 10.13 on a 10.14 system, but in that case there were other artifacts ranging from discolored window title bars to the inability to respond in any way to switching between "Light Mode" and "Dark Mode".

Unlike Tk, a typical macOSapplication does not handle any window events itself. Instead the developer provides callbacks for events and runs an event loop which is part of Apple's proprietary AppKit library. Any drawing which such an application wishes to do must be implemented in the [NSView drawRect:dirtyRect] method. This method is empty by default and must be overloaded by an macOSapplication. The event loop is virtually undocumented and can change with new releases of the OS. However, since Tk handles its own events, and needs to use the NSApplication object as an event source, it cannot use the proprietary event loop and must implement its own replacement. This is fairly tricky since there is no explicit documentation about how the event loop works. The sketchy implicit documentation does explain that the event loop is responsible for managing autoReleasePools for the application, which at least indicates that there are many pitfalls lurking in the design of the event loop. And Apple recommends against attempting your own implementation.

The event loop changed significantly between macOS10.13 and 10.14, presumably as part of making "semantic" colors able to change their RGB value depending on whether the system was using "Dark Mode". Experimentation indicated one of these changes to the event loop was that a CGContext object was only valid while the [NSView drawRect] method was being executed. Prior to this it was possible to draw to a window at any time, and the Tk Drawing Procedures, which are always run as idle tasks in the Tcl event loop, never did any drawing during within the drawRect method. In the design of Tk for macOSthe idea was to implement the drawRect method so that all it would do is to generate expose events for all of the widgets that meet the dirty rectangle. Then those widgets would generate idle tasks to draw themselves and those idle tasks would be run within the Tcl event loop.

This forced a redesign of Tk for macOS. All idle tasks which draw to the screen needed to be run within the drawRect method. This was handled by creating a secondary event loop inside of the drawRect method which would generate expose events, handle the expose events, and then handle all of the idle tasks which they cause to be created. A flag was created to allow drawing procedures to check whether they are being run

within the drawRect method before trying to create a graphics context. If not, rather then making a futile attempt at drawing, they mark the containing NSView as needing display so that the drawing will take place later when it is possible to draw.

Another aspect of AppKit which emerged during the experiments is that the drawRect method is run asynchronously. It appears to be the case that the drawRect method is run in a way which is similar to the way that an interrupt handler is run in a typical UNIX system. It seems to run in the same thread and to share the stack, but it can be entered at any time that the Cocoa window manager decides to call it. Presumably, though one can be fairly certain that it won't be run unless the NSView has been marked as needing display. A consequence of this is that every function which gets called while the NSView is marked as needing display should be reentrant. Moreover, every such function should only make calls to Appkit functions which can be run within drawRect, whatever they may be.

While the reentrancy problems seem to be fairly rare, there are some clear situations where the restrictions on what can be done inside drawRect are problematic. One thing which guarantees a crash is to create a new NSWindow inside of drawRect. And, unfortunately, that is exactly what happens if an error occurs which requires posting an error dialog. This issue was worked around to some extent by modifying the bgerror command to open the error dialog within an idle task.

To conclude, we have seen considerable progress towards making Tk apps look as though they belong on an Apple desktop. At the same time, Tk is in a precarious and vulnerable state every year when a new release of macOSoccurs. However, Tk has proved to be robust and resilient so far. There is every reason to expect that to continue, bringing with it the ability to hugely simplify the task of generating complex macOSuser interfaces.

References

[1] Apple Inc., Human Interface Guidelines (macOS), `https://developer.apple.com/design/human-interface-guidelines/macos/overview/themes/`.

[2] Brad Lanam et al., Tk differences on Mac OS X, `https://wiki.tcl-lang.org/page/Tk+differences+on+Mac+OS+X/`.

[3] dkulp, OSX Mojave Dark Mode support, `https://trac.wxwidgets.org/ticket/18146`.