

The Tile Widget Set

Joe English

October 2004

Abstract

The Tile widget set is an experimental reimplementation of the core Tk widgets, built on top of a revised and enhanced version of the TIP #48 theme engine. This paper explains how the engine works, and how the new widgets differ from the standard Tk ones.

Motivation

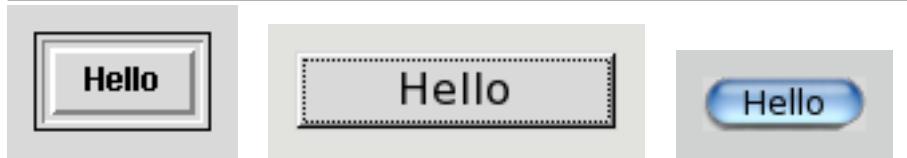
The Tk toolkit was five years ahead of its time — ten years ago. Although the foundation is still sound and it remains one of the easiest ways to build a GUI, the look and feel has failed to keep up with trends and fads in user interfaces. This is most apparent on Windows XP, where the native controls have a radically different appearance; and Tk applications even look out of place on Unix nowadays under modern desktop environments like Gnome and KDE.

The Tile widget set provides a new look for Tk — several new looks, in fact. By separating the code responsible for widgets' appearance from the rest of the implementation, it's possible to completely change the look and feel without having to modify widget internals.

Themes

Figure 1 shows three different buttons. The first is a classic Motif-style button. Notice the outer highlight ring (indicating keyboard focus) and the thick inner border (indicating that this is the “default” button). The second is a Windows-style button: it has

Figure 1 Three buttons



a slightly different border, and the focus indicator is a dashed box drawn inside the border instead of a solid box on the outside. The third button is a different possibility altogether.

Except those aren't really three different buttons: it's actually the same button drawn under three different themes.

The Tile package includes several built-in themes. The default theme on X11 has a new, streamlined look; a **classic** theme implementing Tk's current Motif-like appearance is also available. On Windows XP, the **xpnative** theme uses the Windows "Visual Styles" API to make Tk widgets indistinguishable from native controls. On other versions of Windows, the **winnative** theme matches the Microsoft Windows User Experience guidelines. On Mac OSX, the **aqua** theme uses the Carbon Appearance Manager for (almost-)native appearance on that platform. New themes can be implemented as add-on packages written in Tcl or in C, depending on the level of customization required.

A Tile theme is a collection of *elements*, *layouts*, and *styles*, which are combined with widget options to determine the look and feel.

Elements

An *element* displays a single part of a widget. For example, the Windows-style button has a border, a focus ring, and a label, each of which are distinct elements.

Elements are usually implemented in C, with one function to compute the required size and another to draw the element. Elements can also be defined from Tk images to create "pixmap" themes.

Elements have options, just like widgets. For example the default **border** element has **-borderwidth** and **-relief** options, and the **text** element includes **-text**, **-font**, and **-foreground** options. Element option values are supplied by the containing widget and by the style system, as explained later on.

Layouts

A *layout* defines the collection of elements that make up a widget and how they are arranged. The layout mechanism is based on a simplified version of Tk's **pack** geometry manager: a layout contains a list of elements, along with which **-side** of the cavity to place them on and a **-sticky** specification that determines how to place the element within its allocated parcel. Elements may also have a list of **-children**, which are placed inside the element. See Figure 2 for an example.

The layout engine automatically computes the position and size of each element based on packing options and the widget's overall size. Some widgets further adjust the layout; for example the scrollbar widget updates the position and size of the **thumb** element based on the visible portion of the widget it's linked to.

Figure 2 shows a layout specification for horizontal scrollbars. Notice that this layout

Figure 2 Scrollbar layout specification

```
style layout Horizontal.TScrollbar {  
    Scrollbar.trough -children {  
        Scrollbar.leftarrow -side left  
        Scrollbar.rightarrow -side right  
        Scrollbar.leftarrow -side right  
        Horizontal.Scrollbar.thumb -side left -sticky ew  
    }  
}
```



includes two arrow buttons on the right side, as found in NeXTStep and Mac OSX scrollbars. It's just as easy to build scrollbars with one button on each side, two on the same side, a pair of buttons on both ends, or with no buttons at all. The scrollbar widget itself doesn't care; this is entirely up to the theme engine.

States

Not pictured in Figure 1 is the dynamic appearance. The first button “lights up” when the mouse pointer hovers over it, and the border relief changes from **raised** to **sunken** when the button is pressed. The focus indicator and default ring are only displayed under certain conditions, and the text is grayed out when the button is disabled. This is where states come in.

Many Tk widgets have a **-state** option that can be set to **normal** or **disabled**. Some widgets allow additional state values; for example entry widgets have a **readonly** state and buttons can be in the **active** state.

The Tile widget set generalizes this idea. Every widget has bitmask of mutually independent state flags. State flags include ‘**active**’ (set when the mouse pointer is over the widget), ‘**focus**’ (widget has keyboard focus), ‘**disabled**’ (widget disabled under program control), ‘**pressed**’, ‘**selected**’, and several others.

The **state** and **instate** widget commands modify and query the widget state¹; see Figure 3 for an example.

The <Enter> binding, ‘%w state active’, turns the **active** flag on, and the <Leave> binding ‘%w state !active’ turns it off. That is, the widget is “active” whenever the mouse pointer is inside the widget.

Pressing the mouse button (<ButtonPress-1>) sets the **pressed** flag, which is cleared if the pointer leaves the widget with the mouse button still down (<B1-Leave>), and

¹The **-state** widget option is also supported for compatibility with the core widgets that have one, although in a limited fashion.

Figure 3 Button class bindings (slightly simplified)

```
bind TButton <Enter>           { %w state active }
bind TButton <Leave>            { %w state !active }
bind TButton <ButtonPress-1>    { %w state pressed }
bind TButton <B1-Leave>         { %w state !pressed }
bind TButton <B1-Enter>         { %w state pressed }
bind TButton <ButtonRelease-1>  {
    if {[%w instate pressed]} {
        %w state !pressed
        eval [%w cget -command]
    }
}
```

set again if the pointer re-enters the widget (<B1-Enter>). Releasing the mouse button (<ButtonRelease-1>) evaluates the widget's **-command** and clears the **pressed** flag (but only if the widget was in the **pressed** state to begin with; this way the user can drag the pointer outside the button and release the mouse button without activating the widget).

The attentive reader may have noticed that the background color and border relief haven't been mentioned yet. We'll get to that next section. The widget class bindings simply change the widget state and occasionally perform an action like running the **-command**; all dynamic changes to the appearance are controlled by the associated *style*.

Styles

Unlike class bindings, which are associated with the widget class and shared by all themes, *styles* are defined in and belong to a single theme. Figure 4 shows sample style settings for the button widget. The **style default** command specifies default values for element options. The **style map** command specifies state-dependent values, which override the default when the widget is in a particular state or combination of states.

There can be multiple state specifications for each option; the first matching specification takes precedence. Each specification contains zero or more state names; if more than one is present then all state flags must match. (By implication, an empty state specification always matches). State names can also be prefixed with an exclamation point indicating that the corresponding flag must be off.

For example, to prevent the relief from changing when the button is disabled, you could use:

```
style map TButton -relief {disabled raised pressed sunken}
```

in which case the **disabled** specification would match first, or you could use:

Figure 4 Button style settings

```
style default TButton \
    -background #d9d9d9 \
    -foreground black \
    -relief raised ;
style map TButton \
    -background {active #ececfc} \
    -foreground {disabled #a3a3a3} \
    -relief {disabled flat pressed sunken} ;
```

```
style map TButton -relief {{pressed !disabled} sunken}
```

in which case the **sunken** value would be used only if the button is both pressed and not disabled.

Element option values are initialized from the following places, in order of precedence:

1. State-dependent dynamic settings specified by **style map**, if any match the current state;
2. an option of the same name in the containing widget, if one is present and non-null;
3. default values specified by **style default**; or
4. the built-in fallback value supplied by the element.

Typically, options like **-text** and **-image** are taken from the widget instance while appearance-related options like **-foreground** and **-background** are taken from theme defaults. This way, defaults defined in the theme may be overridden by options set on the widget, which in turn may be overridden by state-specific dynamic settings.

It's also possible to specify, for example, a specific **-font** for an individual label widget to override the theme default, but to fully customize the look and feel of widget instances every Tile widget has a **-style** option. This is used to select a custom style.

Custom Styles

Here's how you make a "toolbar-style" checkbutton in Tk 8.4:

```
checkbutton .toolbar.cb \
    -indicatoron false \
    -relief flat \
    -overrelief raised \
    -offrelief flat \
```

```

-selectcolor {} \
... -text and other options ...

```

Paraphrasing, this says: don't display the check box (**-indicatoron false**); the border is flat by default (**-relief flat**) and when the checkbox is deselected (**-offrelief flat**), but raised when the widget is active (**-overrelief raised**); and don't change the background color when the widget is selected (**-selectcolor {}**).

To do the same thing with the Tile checkbutton takes a bit more work to set up, but once that's done you can just say:

```
ttk::checkbutton .toolbar.cb -style Toolbar.TCheckbutton ...
```

Here's how you set it up: To achieve the effect of **-indicatoron false**, create a new layout that doesn't have an indicator:

```

style layout Toolbar.TCheckbutton {
    Toolbutton.border -children {
        Toolbutton.padding -children {
            Toolbutton.label
        }
    }
}

```

Then use **style map** and **style default** to control the border appearance:

```

style default Toolbar.TCheckbutton \
    -relief flat
style map Toolbar.TCheckbutton -relief {
    disabled     flat
    selected    sunken
    pressed     sunken
    active      raised
}

```

We can provide additional feedback by changing the foreground and background colors in different states:

```

style default Toolbar.TCheckbutton \
    -background "#dcdad5" -foreground #000000
style map Toolbar.TCheckbutton -background {
    disabled "#dcdad5"
    pressed "#bab5ab"
    active "#4a6984"
} -foreground {
    disabled "#cccccc"
    active "#ffffff"
}

```

Notice that the above uses a slightly darker background when the widget is pressed — something that the standard Tk checkbutton can't do without substantial changes to the internals.

The Tile widgets also have a **-class** option, which can be specified at creation time to set the widget's class. This affects the initial bindtags, option database lookups, and other things; it's used to change the behavior of an individual widget²

Fallback Rules

The theme engine uses a number of inheritance rules to make it easier to define new themes and custom styles,

Every theme has a *parent theme*, which is consulted for elements and layouts that aren't defined in the current theme. For example the **xpnative** theme inherits some elements from the **winnative** theme, which in turn inherits some from the **alt** theme (a cross-platform implementation of the MSUE guidelines). The **default** theme is at the root of the hierarchy; it defines the default layout for all widget classes and fallback element implementations.

Elements use a hierarchical naming convention, with components separated by periods much like widget path names. If an element is not found, the initial component of the name is removed to yield a generic name which is checked next. The process continues until no components are left, in which case the search continues with the original name in the parent theme.

By convention, layouts refer to generic names like **Button.border** and **Entry.border**. This way a theme can, for example, define a generic **border** element that will be used by most widgets and provide a specialized **Button.border** element just for buttons. Or it can not define a **border** element at all and inherit it from the parent theme.

Layouts use the same hierarchical naming scheme as elements, falling back to the parent theme if no match is found. Styles also use hierarchical names, but with a different fallback chain. The search for maps and defaults proceeds along the name hierarchy, but terminates at the “root” style named “.”. The parent theme is not consulted. Theme-wide default settings like **-background** and **-font** can be set on the root style, overriding them if necessary on individual widget-specific styles.

Differences

The Tile widgets are *mostly* compatible with the corresponding Tk widgets. They support the same set of widget commands and the same essential options. However, most of the appearance-related options have been removed — more precisely, they've been *moved* — out of the widgets themselves and into the style system. For example,

²For example, to make an autorepeating pushbutton.

Figure 5 Tk 8.4 button options vs. Tile button options

Tk 8.4 button options				
-activebackground	-activeforeground	-anchor	-background	
-bd	-bg	-bitmap	-borderwidth	
-command*	-compound*	-cursor*	-default*	
-disabledforeground	-fg	-font	-foreground	
-height	-highlightbackground	-highlightcolor	-highlightthickness	
-image*	-justify	-overrelief	-padx	
-pady	-relief	-repeatdelay	-repeatinterval	
-state*	-takefocus*	-text*	-textvariable*	
-underline*	-width*	-wraplength		

Tile <code>ttk::button</code> options				
-class	-command*	-compound*	-cursor*	-default*
-image*	-padding	-state*	-style	-takefocus*
-text*	-textvariable*	-underline*	-width*	

Options common to both widgets indicated with an asterisk.

the Tk 8.4 button widget has 35 options; the Tile button only has 14³; see Figure 5 for a comparison.

It is hoped that many applications will require only minor modifications to switch to the Tile widgets. In particular, those that rely on toolkit defaults and those that use the option database for customization should be easy to adapt.

Other Stuff

The Tile widget set also includes combobox, notebook, and progress bar widgets; a multi-column hierarchical tree widget is in the works.

The Tile project was originally intended as an experiment, to test out new ideas for the next generation of Tk. The experiment has worked out pretty well so far, and it might be included in Tk 8.5 as an option for applications that wish to take advantage of the new features.

In the meantime, the Tile package is available as a loadable extension, compatible with Tk 8.4, under the **tktable** project at SourceForge:

<http://tktable.sourceforge.net/tile/>

³... at the time of this writing. There may be more, or fewer, in the 1.0 release.