Chapter 4
Using the Canvas Widget

You can turn an ordinary program into an extraordinary application by using graphics to convey information. After all, a picture is worth a thousand words. And in the realm of graphical user interfaces, an interactive picture is worth a thousand buttons!

Imagine an application that monitors a factory floor for problems on the production line. Suppose it displays a diagram of the factory floor, and it marks problem areas with a flashing red square. That kind of interface lets you see the status of the entire factory at a glance, and it is much more intuitive than a listbox full of status messages.

Tk has something called the *canvas* widget that makes it easy to build such things. You simply create a canvas and add the lines, rectangles, and polygons that make up a drawing. You can even make the drawing come to life by binding actions to certain events on the canvas. When you click on a problem area on the factory floor, for example, the program could display the status for that area. When you drop a wrench icon onto the problem area, the program could dispatch a maintenance crew.

In this chapter, we'll see how you can use the canvas to build interactive displays. We'll start by explaining how the canvas works with some simple examples. Then we'll look at a series of case studies to see how the various features work together:

- We'll build a progress gauge showing the status of a task from 0% to 100% complete.
- We'll build a color selection wheel.
- We'll add some tabs to the notebook that we created in Section 2.1.7, to create a tabbed notebook.
- We'll build a calendar that lets you page through the months and click to select individual days.

- We'll build a simple drawing program that lets you create things like rectangles and circles. We'll add bindings so that you can select these items, resize them, move them and change their color.

You can use the same techniques to build other displays such as the factory floor monitor, a seat assignment chart for airline reservations, or whatever else your application requires.

4.1 Understanding the Canvas Widget

When you create a canvas like this:

```
canvas .c -width 2i -height 1i
```

you get an empty canvas—just a blank area with no default behavior. In this example, the canvas is 2 inches wide and 1 inch high. You create a drawing on the canvas by adding drawing elements called *items*. For example, we can draw a line by creating a line item like this:

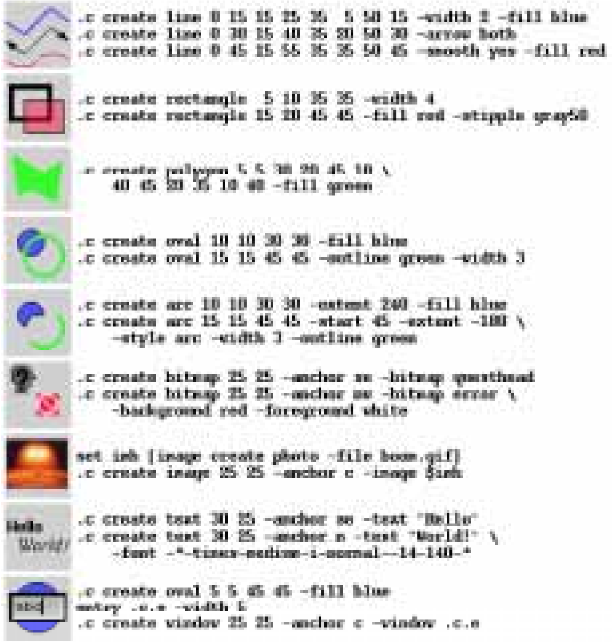
```
.c create line 0 15  15 25  35 5  50 15  -width 2 -fill blue
```

Each pair of numbers represents an (x,y) coordinate for the line. So this line goes from (0,15) to (15,25) to (35,5) to (50,15). All of these coordinates are relative to the origin at (0,0), which is in the upper-left corner of the window. X-coordinates increase toward the right, and y-coordinates increase going down. In this example, the numbers are just integers, so they are treated as pixel coordinates. But you can add a letter after each number to indicate its units. For example, the value `1.5i` is 1 inches, and`10c` is 10 centimeters.

Each item has configuration options that control its appearance. By default, lines are black. But this particular line is blue, and it has a width of 2 pixels.

You can create many different kinds of items on the canvas. Figure 4.1 shows examples of the various types. If you are looking for a detailed description of each item and its configuration options, you can find it on the manual page for the canvas widget. But we'll mention each type briefly below, so you get a feeling for the things that you can create on a canvas:

- **`line`**
  A line has two or more coordinates. You can add arrowheads at the ends. If you turn on smoothing, the line is drawn as a set of Bezier splines.
- **`rectangle`**
  A rectangle has two coordinates representing two opposite corners. You can set a color and a line width for its outline, and you can set a separate color to fill its interior.
- **`polygon`**
  A polygon has three or more coordinates. Like a rectangle, you can set its outline color and its line width, and you can use a separate color to fill its interior. Like a line, you can turn on smoothing, and its outline will be drawn as a set of Bezier splines.



```
.c create line 0 15 15 25 35  5 50 15 -width 2 -fill blue
.c create line 0 30 15 40 35 20 50 30 -arrow both
.c create line 0 45 15 55 35 35 50 45 -smooth yes -fill red

.c create rectangle  5 10 35 35 -width 4
.c create rectangle 15 20 45 45 -fill red -stipple gray50

.c create polygon 5 5 30 20 45 10 \
    40 45 20 35 10 40 -fill green

.c create oval 10 10 30 30 -fill blue
.c create oval 15 15 45 45 -outline green -width 3

.c create arc 10 10 30 30 -extent 240 -fill blue
.c create arc 15 15 45 45 -start 45 -extent -180 \
    -style arc -width 3 -outline green

.c create bitmap 25 25 -anchor se -bitmap questhead
.c create bitmap 25 25 -anchor nw -bitmap error \
    -background red -foreground white

set imh [image create photo -file boom.gif]
.c create image 25 25 -anchor c -image $imh

.c create text 30 25 -anchor se -text "Hello"
.c create text 30 25 -anchor n -text "World!" \
    -font -*-times-medium-i-normal--14-140-*

.c create oval 5 5 45 45 -fill blue
entry .c.e -width 5
.c create window 25 25 -anchor c -window .c.e
```

Figure 4.1. The various items that you can create on a canvas.

- **`oval`**
  An oval has two coordinates representing the rectangle that contains it. Like a rectangle, you can set its outline color and its line width, and you can use a separate color to fill its interior.
- **`arc`**
  An arc is like an oval, but it has a starting angle and an extent that control how much of the oval is drawn. An arc also has a style option. It can be drawn as a line, as a pie wedge, or with a chord connecting the end points.
- **`bitmap`**
  A bitmap has one coordinate representing its anchor point. The bitmap is aligned with this point according to its `-anchor` option. Each bitmap has only two colors: a foreground color which is normally black, and a background color which is normally the same as the canvas widget.
- **`image`**
  An image has one coordinate representing its anchor point. But unlike a bitmap, an

image can have any number colors, and it will dither automatically when it is displayed on a monochromatic screen.

- **`text`**
  A text item has one coordinate representing its anchor point. A single text item can have multiple lines, and you can control their justification. You can also set the color and the font for the text.
- **`window`**
  A window item has one coordinate representing its anchor point. It acts as a placeholder for a widget embedded in the canvas. So you can mix things like buttons and entries with the other graphics on the canvas. For example, in Figure 4.1 we put an entry widget on top of a blue oval.

Many of these items also have an option for their *stipple pattern*, which controls how they are filled in. By default, items are drawn with a solid color. But if you use a bitmap screen like `gray50` as a stipple pattern, you will get a stenciled effect. Where the bitmap is black, the item will be drawn with its fill color, and where it is not, the item will be transparent. One of the rectangles in Figure 4.1, for example, has the stipple pattern `gray50`. Its interior is a red screen that lets the rectangle underneath it show through.

4.1.1 Scrolling

Each canvas has an unlimited drawing area, but the canvas widget itself has a certain size on the screen. By default, the canvas will display as much as it can starting from the (0,0) coordinate in the upper-left corner of the drawing area, as shown in Figure 4.2. The visible area of the canvas is called the *viewport*. You can add items outside of the viewport, but you won’t see them unless you tell the canvas to change its view.

If your drawing extends beyond the viewport, you can attach scrollbars to control the view. For example, we used the following code to create the display in Figure 4.2:

```
canvas .display -width 3i -height 2i -background black \
    -xscrollcommand {.xsbar set} -yscrollcommand {.ysbar set}
scrollbar .xsbar -orient horizontal -command {.display xview}
scrollbar .ysbar -orient vertical -command {.display yview}

.display create line 98.0 298.0 98.0 83.0 -fill green -width 2
.display create line 98.0 83.0 101.0 69.0 -fill green -width 2
.display create line 101.0 69.0 108.0 56.0 -fill green -width 2
...
```

This is the same code that we saw in Section 2.2.1. But at that point, we were only concerned with how scrollbars are positioned next to a canvas. Here, we will explain how scrollbars are attached to control a canvas.

Each scrollbar has a `-command` option with a command to control the canvas view. In this example, when you adjust the horizontal scrollbar it executes the command

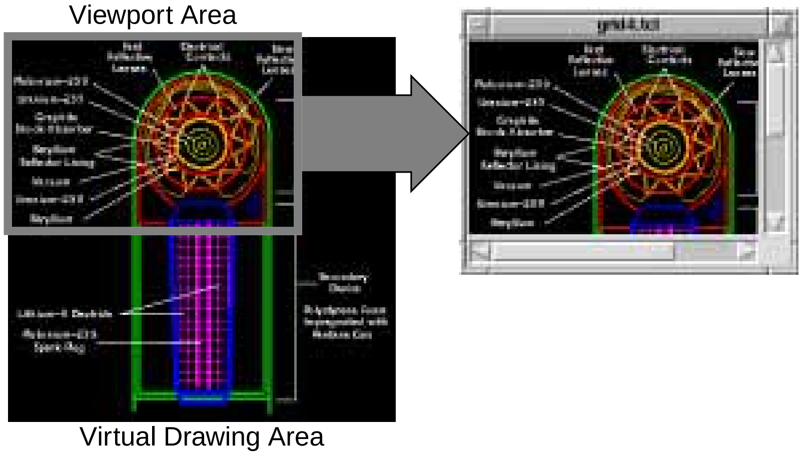


Figure 4.2. The canvas widget acts as a viewport, displaying part of the virtual drawing area that is available. You can add scrollbars to change the view.

`.display xview`, with a few extra arguments to shift the viewport left or right. And when you use the vertical scrollbar, it executes the command `.display yview`, again with suitable arguments, to shift the viewport up or down. Similarly, the canvas has `-xscrollcommand` and `-yscrollcommand` options which it uses to control the scrollbars. In this example, whenever its view changes it executes the commands `.xsbar set` or `.ysbar set` with suitable arguments, to reposition the bubble in the middle of the each scrollbar.

These command options allow the canvas and the scrollbars to communicate with one another and stay in sync, but they aren't enough to make the canvas scroll properly. The canvas also has a `-scrollregion` option that sets the boundaries of the scrolling region. By default, the scrolling region corresponds to the viewport area. So even if your actual drawing is much larger than the viewport, the canvas will think that there is no room for scrolling. To make the canvas scroll properly, you must tell it the size of your drawing by giving it some coordinates, like this:

```
.display configure -scrollregion {0 0 250 375}
```

This says that the upper-left corner of the scrolling region is (0,0), and the lower-right corner is (250,375). The canvas will let the viewport move within this region.

Quite often, you won't know the overall size of your drawing, and guessing at the size is prone to errors. If you make the scrolling region too small, you won't be able to see part of the drawing, and if you make it too big, you will see a lot of empty area. Instead, you should use the following trick to set the scrolling region:

```
.display configure -scrollregion [.display bbox all] \
    -xscrollincrement 0.1i -yscrollincrement 0.1i
```

The command `.display bbox all` automatically computes a bounding box around all of the items currently on the canvas. We use this result to set the area for the scrolling region. You must do this *after* you've created all of the items for your drawing. If you add more items to the drawing, you should recompute the bounding box and set the scrolling region accordingly.

The canvas also has `-xscrollincrement` and `-yscrollincrement` options. You can set these to indicate how much the canvas should move when you press the arrows on either end of a scrollbar. In this example, we have the drawing shift in increments of 0.1 inch.

4.1.2 Display List Model

The canvas remembers each item that you create. At any point, you can move, resize or change the attributes of an item, and the drawing will be updated automatically. For example, suppose we create a canvas with the following code:

```
canvas .c -width 100 -height 110
pack .c

.c create oval 10 10 90 90 -fill yellow -width 2
.c create arc 15 15 85 85 -start 60 -extent 60 -fill black
.c create arc 15 15 85 85 -start 180 -extent 60 -fill black
.c create arc 15 15 85 85 -start 300 -extent 60 -fill black
.c create oval 40 40 60 60 -outline "" -fill yellow
.c create oval 44 44 56 56 -outline "" -fill black
.c create text 50 95 -anchor n -text "Warning"
```

As each item is created on this canvas, it is added to an internal list called the *display list*. When the canvas needs to display itself, it simply draws each item on the display list. It starts at the bottom and works its way toward the top, as shown in Figure 4.3.

Now, suppose we want to change one of the items. Suppose our nuclear reactor goes critical and we want to change the message from “Warning” to “RED ALERT”. We need a way of referring to the text item within the canvas.

When each item is created, the canvas assigns it a unique number called an *item identifier.* You can capture the identifier whenever you create an item, like this:

```
set id [.c create text 50 95 -anchor n -text "Warning"]
```

Here, the variable `id` will contain an item number like `7`. We can use this later on to configure the item:

```
.c itemconfigure $id -text "RED ALERT"
```

This tells the canvas to find item `7` and change its `-text` option to “RED ALERT”.

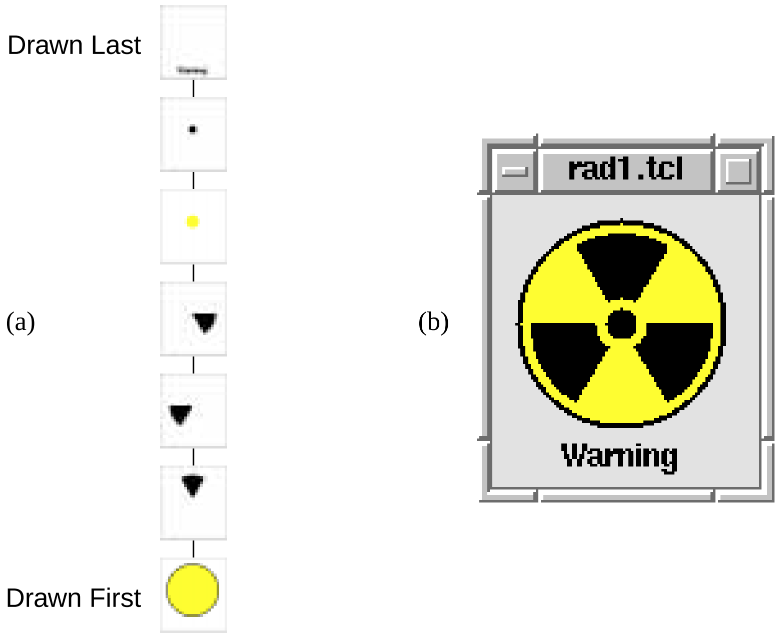


Figure 4.3. (a) The canvas keeps a display list of all its internal items. (b) The resulting picture.

When any item changes, the canvas figures out what portion of the drawing is affected, and it redraws that part of the display. It does so in a highly efficient manner, so the changes appear to be instantaneous. In this example, the area near the text item is regenerated, and the text changes immediately from “Warning” to “RED ALERT”.

The canvas also has search operations that help you find certain items in the display list. For example, its `find enclosed` operation will search for items that are contained within a bounding box. A command like:

```
.c find enclosed 20 20 80 110
```

will find all items in the rectangle from (20,20) to (80,110) and return a list of item identifiers. We could use this in conjunction with the `itemconfigure` operation to highlight these items in red:

```
foreach id [.c find enclosed 20 20 80 110] {
    .c itemconfigure $id -fill red
}
```

This gives the result shown in Figure 4.4.

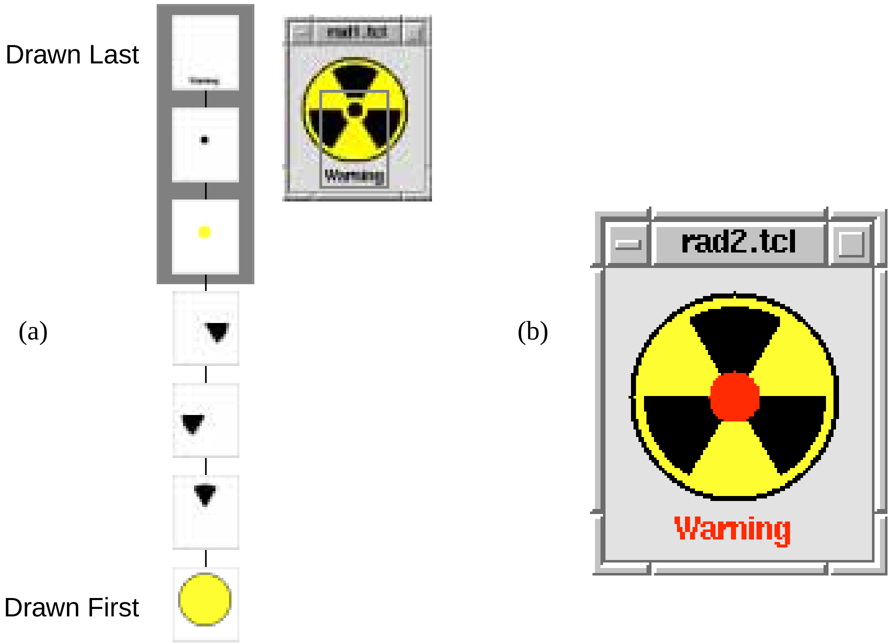


Figure 4.4. (a) The canvas can search its display list for certain items. (b) Items can be configured to change their appearance.

The canvas has a few more search operations. You can find an explanation of each one on the canvas manual page. We mentioned the `find enclosed` operation here simply to show how canvas operations build on one another, and how the item identifiers come into play. We'll see some examples later in this chapter that show the search operations in action.

4.1.3 Using Tags

Item identifiers are useful, but since they are just numbers, they are not very meaningful. Your canvas programs will be much easier to understand if you tag important items with symbolic names. An easy way to do this is to set the `-tags` option as each item is created. For example, we could rewrite our code to tag some of the items in our radiation symbol like this:

```
canvas .c -width 100 -height 110
pack .c
```

```
.c create oval 10 10 90 90 -fill yellow -width 2
.c create arc 15 15 85 85 -start 60 -extent 60 \
    -fill black -tags sign
.c create arc 15 15 85 85 -start 180 -extent 60 \
    -fill black -tags sign
.c create arc 15 15 85 85 -start 300 -extent 60 \
    -fill black -tags sign
.c create oval 40 40 60 60 -outline "" -fill yellow
.c create oval 44 44 56 56 -outline "" \
    -fill black -tags sign
.c create text 50 95 -anchor n -text "Warning" -tags message
```

We tagged the “Warning” message with the symbolic name `message`. When we need to change the message later on, we can use the tag name `message` as an item identifier:

```
.c itemconfigure message -text "RED ALERT" -fill red
```

That way, we don’t have to create extra variables to store item identifiers, and the code is easier to follow.

You can also use tag names to identify a group of related items. For example, we tagged all of the black parts of the radiation sign with the symbolic name `sign`. Having done that, we can change the color of all four items at once with a single command:

```
.c itemconfigure sign -fill red
```

The canvas finds all items tagged with the name `sign` and changes their fill color to red, as shown in Figure 4.5. We could have accomplished the same thing by looping through a list of item identifiers and configuring each item individually. But this is more convenient, and it is much more efficient. Remember, the canvas operations are handled in compiled code, so they run much faster than a block of Tcl statements, which are all interpreted.

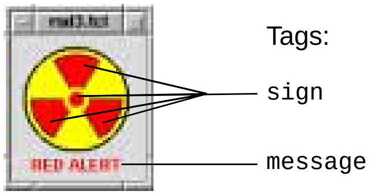


Figure 4.5. Tag names are used to identify items on the canvas.

Each item can have lots of different tag names associated with it, so you can put the same item in different groups. For example, suppose we want to create a group of items called `hilite` that will all light up red at some point. We might include the `message` item and the sign items in this group, so that we could configure them all with a single command, like this:

```
.c itemconfigure hilite -fill red
```

We need to add the tag name hilite to all the items that we want to have in this group. One way to accomplish this is to add the tag when each item is created, like this:

```
.c create text 50 95 -anchor n -text "Warning" \
    -tags {message hilite}
```

As you can see, the -tags option accepts a list of tag names. In this example, the elements message and hilite are treated as separate names that both refer to the text item.

Another way is to add the tag name after the item has been created. The canvas addtag operation supports many different ways of finding and tagging items. For example, the command:

```
.c addtag "hilite" withtag "sign"
```

finds all of the items with the tag name sign and adds the tag hilite to them.

At this point, we can use the name message to refer to the text item, the name sign to refer to the four parts of the radiation sign, or the name hilite to refer to all of these items.

4.1.4 Canvas Bindings

You can add new behaviors to a canvas using the bind command, just as you would for any other Tk widget. For example, suppose we want the radiation symbol to light up red whenever the mouse pointer enters the window, as shown in Figure 4.6. We simply bind to the <Enter> and <Leave> events on the canvas, like this:

```
bind .c <Enter> {
    .c itemconfigure hilite -fill red
}
bind .c <Leave> {
    .c itemconfigure hilite -fill black
}
```

Here, we are leveraging the tag names described in the previous section. When the mouse pointer enters or leaves the canvas, we simply change the fill color for all items tagged with the name hilite.

But unlike the other widgets, the canvas also lets you bind to events on the items within it. For example, suppose that we want the radiation sign to light up only when the mouse pointer is touching one of the hilite items. Instead of detecting <Enter> and <Leave> events on the canvas as a whole, we want to detect <Enter> and <Leave> events on the individual hilite items, as shown in Figure 4.7. So instead of using the usual bind command (which applies to an entire widget), we must use a special bind operation on the canvas (which provides access to items within the canvas). It looks like this:

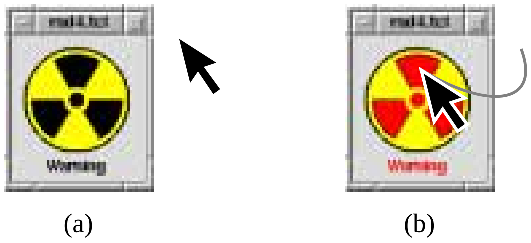


Figure 4.6. Binding to the canvas as a whole. (a) The pointer is outside the canvas. (b) The pointer moves inside the canvas and the items tagged as `hilite` change color.

```
.c bind hilite <Enter> {
    .c itemconfigure hilite -fill red
}
.c bind hilite <Leave> {
    .c itemconfigure hilite -fill black
}
```

These look like ordinary `bind` commands, but the `.c` prefix indicates that they apply only to items within the canvas `.c`.

Bindings can be applied to individual items via their item identifier, or to groups of items using their tag name. In this case, we added bindings to all items with the tag name `hilite`.

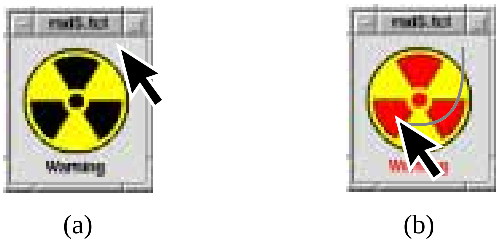


Figure 4.7. Binding to canvas items. (a) The pointer is on the canvas. (b) The pointer moves onto an item tagged `hilite` and all items with this name change color.

Suppose we add another binding for the `message` item. When you click on this item, the text will toggle between “Warning” and “RED ALERT”:

```
.c bind message <ButtonPress-1> {
    if {[.c itemcget message -text] == "Warning"} {
        .c itemconfigure message -text "RED ALERT"
    } else {
        .c itemconfigure message -text "Warning"
    }
}
```

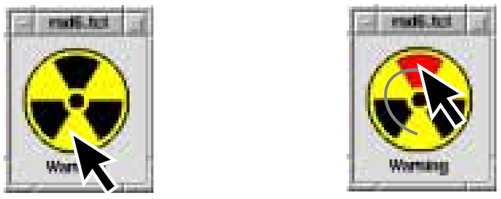
Again, if we had added this binding to the canvas as a whole, the text would change no matter where you click on the canvas. But instead, we added the binding only to the `message` item, so you must click directly on the item for the text to change.

The canvas has one other feature that comes in handy for bindings. Whenever the mouse pointer touches an item, that item is temporarily tagged with the name `current`. This makes it easy to figure out which item is active at any point in time. For example, suppose that we want parts of the radiation sign to light up individually as the mouse pointer touches them, as shown in Figure 4.8. This technique is called *brushing*, and it helps the user realize what parts of your diagram are active.

As before, we bind this behavior to all of the items tagged with the name `hilite`:

```
.c bind hilite <Enter> {
    .c itemconfigure current -fill red
}
.c bind hilite <Leave> {
    .c itemconfigure current -fill black
}
```

But this time, only one item tagged with the name `current` will change color. In effect, we have added the same binding to a group of items named `hilite`, but we set up each item to react individually.



(a) (b)

Figure 4.8. Detecting the current item. (a) The pointer is on the canvas. (b) The pointer moves onto an item with the tag name `hilite`, but this time only the current item changes color.

In the examples that follow, we'll see tags, bindings and other canvas techniques in action. Although there are many different examples, there is really one underlying philosophy for using the canvas: You must focus on the items within it. Decide how to tag items so that you can refer to them later. Tag related items with the same name so you can handle them as a group. In some cases, you may bind to events on the canvas as a whole. But more often than not, you will bind to individual items or groups of items, creating active areas on your display. If you have worked with other graphical toolkits, this may seem like a paradigm shift, and it may take some getting used to. But once you have mastered tags, you have mastered the canvas.

4.2 Scrollable Form

When the canvas is too small to display an entire drawing, you see only a small portion of it, and you can use scrollbars to adjust the view. We saw how this works in Section 4.1.1. But what if the canvas has widget items on it? As long as the canvas knows the size of its scrolling region, we can scroll the widgets in and out of the view.

You can mix widgets and graphics on the canvas to create some fancy displays. But in this example, we'll do something much simpler. We'll use the canvas to scroll through a long form like the one shown in Figure 4.9. This technique is handy when you have a lot of entries that won't fit on the screen all at once.

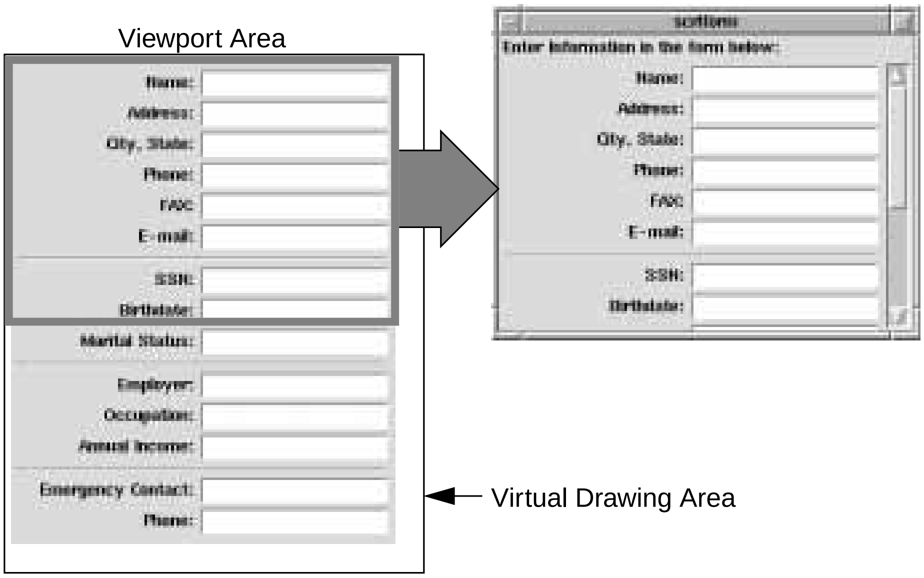


Figure 4.9. You can use the canvas to scroll over a large collection of widgets, in effect creating a scrollable form.

To build a scrollable form, we simply pack (or grid) a large collection of widgets together in a frame, and then position the frame on the canvas. We use its overall size to set the limits on the scrolling region for the canvas. Once the scrollbars are attached, the canvas handles the scrolling, and the scrollable form is complete.

The following procedure makes it easy to create a scrollable form. It creates a canvas and attaches a vertical scrollbar, and then positions an empty frame in the drawing area. You simply pass in a widget name for the whole assembly:

```
proc scrollform_create {win} {
    frame $win -class Scrollform

    scrollbar $win.sbar -command "$win.vport yview"
    pack $win.sbar -side right -fill y

    canvas $win.vport -yscrollcommand "$win.sbar set"
    pack $win.vport -side left -fill both -expand true

    frame $win.vport.form
    $win.vport create window 0 0 -anchor nw -window $win.vport.form

    bind $win.vport.form <Configure> "scrollform_resize $win"
    return $win
}
```

We start by creating a frame to wrap up the canvas and its scrollbar. Inside this frame, we create a canvas and a scrollbar, and we connect them together as discussed in Section 4.1.1.

Next, we create an empty frame that will eventually contain all of the widgets for the form. Notice that the name of this frame (`$win.vport.form`) includes the canvas name (`$win.vport`) as a prefix. This is very important. It makes the frame sit inside the canvas, so that only part of it is visible in the viewport. If instead you use a name like `$win.form`, the frame will sit on top of the canvas. It will hang over the edges of the canvas, and it may even obscure the scrollbar.

We position the empty frame on the canvas by creating a window item. This anchors the northwest corner of the frame at the origin (0,0), which is the upper-left corner of the drawing area.

For the form to scroll properly, we need to set the size of the scrolling region. But there is one problem: We haven't added anything in the form frame yet, and we're not sure how big it will eventually be. Instead of setting the scrolling region directly, we bind to the `<Configure>` event on the form frame. The following procedure will be called automatically to set the scrolling region whenever the form frame changes size:

```
proc scrollform_resize {win} {
    set bbox [$win.vport bbox all]
    set wid [winfo width $win.vport.form]
```

```
        $win.vport configure -width $wid \
            -scrollregion $bbox -yscrollincrement 0.1i
    }
```

It uses the canvas bbox operation to compute the overall size of the drawing. This automatically takes into account the overall size of the frame. We simply set the scrolling region to cover this area, and we set the -yscrollincrement option to scroll in increments of 0.1 inch.

You may have noticed that this scrollable form doesn't have a horizontal scrollbar. We could have added one, but instead we assumed that the entire width of the form would be visible. After all, the user will probably want to see both the label and the entry when he is entering data into the form. We want to make sure that the canvas is just wide enough to display the entire form. So we use the winfo width command to determine the overall width of the form, and we use this result to set the -width option for the canvas.

At this point, the scrollable form is ready to use. We just need to pack some widgets into the form frame $win.vport.form. But instead of documenting this name as part of our scrollform library, we should wrap it up in a procedure. That way, we can make changes to the library later on, without breaking any code that uses the library.

So when you create a scrollable form, you can use the following procedure to get the name of the form frame:

```
proc scrollform_interior {win} {
    return "$win.vport.form"
}
```

We can use our new scrollform procedures to create the example shown in Figure 4.9. First, we create a title with a scrollable form beneath it:

```
label .title -text "Enter Information in the form below:"
pack .title -anchor w

scrollform_create .sform
pack .sform -expand yes -fill both
```

Next, we get the name of the form frame:

```
set form [scrollform_interior .sform]
```

Finally, we pack some widgets into the form frame. We could use a long series of commands to create the entries and their labels and pack them all together. But for a long form, it is much easier create the widgets in a loop like this:

```
set counter 0
foreach field {
    "Name:"       "Address:"         "City, State:"
    "Phone:"      "FAX:"             "E-mail:"
    "-"
    "SSN:"        "Birthdate:"       "Marital Status:"
    "-"
```

```
        "Employer:"   "Occupation:"      "Annual Income:"
        "-"
        "Emergency Contact:"  "Phone:"
    } {
        set line "$form.line[incr counter]"
        if {$field == "-"} {
            frame $line -height 2 -borderwidth 1 -relief sunken
            pack $line -fill x -padx 4 -pady 4
        } else {
            frame $line
            label $line.label -text $field -width 20 -anchor e
            pack $line.label -side left
            entry $line.info
            pack $line.info -fill x
            pack $line -side top -fill x
        }
    }
```

The `foreach` command iterates through a list of names. Each name represents one line in the form. Normally, each line has an entry widget and its associated label. But if the name is “`-`” we create a separator line instead.

In either case, we need a unique widget name for each line. We want these widgets to sit inside the form frame, so the names must start with `$form`, which in this example expands to `.sform.vport.form`. We use the `counter` variable to generate a unique number for each line. During each pass through the loop, therefore, the `line` variable will have values like `.sform.vport.form.line1`, `.sform.vport.form.line2`, *etc*. We use these names to create a frame for a separator line, or a frame containing a label and an entry.

When all of the widgets are packed in position, the form frame will shrink-wrap itself around its contents. Its size change will trigger a `<Configure>` event, and the `scrollform_resize` procedure will be called to adjust the scrolling region. If at some point we add some more lines, the form frame will change size again, triggering another `<Configure>` event and another call to `scrollform_resize`. So we can change the size of the form on the fly, and the scrollbars will adjust automatically.

At this point, our scrollable form is complete. As you adjust the scrollbars, the canvas will shift its view up and down, and the form will scroll.

4.3 Progress Gauge

You can use the canvas to display the status of something like the factory floor that we mentioned earlier. But what if something changes? You don’t have to erase the drawing and start over. You can change certain items on the canvas, and leave the rest of the drawing intact. We’ll see how this works in the following example.

Suppose you need to read a large data file, or download a web page, or change the contrast of an image. Each of these tasks may a while to perform, and while they are running, the user will want to know how much progress has been made. Many applications handle this by displaying a progress gauge like the one shown in Figure 4.10. It displays the status of a task from 0% to 100% complete. As the numbers change in the foreground, a bar creeps from left to right in the background, giving a pictorial view of the progress.

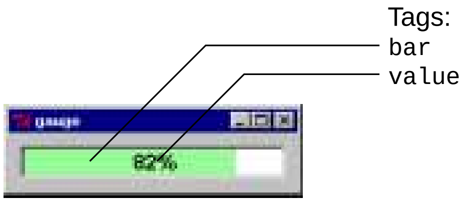


Figure 4.10. A progress gauge built using the canvas.

Tk doesn't have a progress gauge widget, but it is easy to build something like this with the canvas. We simply create a small canvas with two items: a text item for the percentage value, and a rectangle for the bar in the background. We'll tag the text item with the name `value`, and the rectangle with the name `bar`. That will make it easy to refer to these items later on. Whenever our progress changes, we'll update the percentage value in the `value` item, and we'll update the coordinates for the `bar` item.

The following procedure makes it easy to create a progress gauge. You give it a widget name for the whole assembly and an optional color for the bar:

```
proc gauge_create {win {color ""}} {
    frame $win -class Gauge

    set len [option get $win length Length]
    canvas $win.display -borderwidth 0 -background white \
        -highlightthickness 0 -width $len -height 20
    pack $win.display -expand yes

    if {$color == ""} {
        set color [option get $win color Color]
    }
    $win.display create rectangle 0 0 0 20 \
        -outline "" -fill $color -tags bar
    $win.display create text [expr 0.5*$len] 10 \
        -anchor c -text "0%" -tags value
    return $win
}
```

We start off by creating a frame for the whole assembly that we'll call the *hull*. We give it the class name `Gauge`, so that you can add settings to the resource database to customize the gauges in your application. For example, we include the following resources with the gauge code:

```
option add *Gauge.borderWidth 2 widgetDefault
option add *Gauge.relief sunken widgetDefault
option add *Gauge.length 200 widgetDefault
option add *Gauge.color gray widgetDefault
```

By default, all gauges will have a gray bar and an overall sunken appearance. We were careful to give these settings the lowest priority `widgetDefault`. You can override them in applications that use the gauge, and users can override them for their own desktop. On Unix platforms, for example, users can add their own settings to a *.Xdefaults* or *.Xresources* file.

The `borderWidth` and `relief` settings apply directly to the hull frame. All frames recognize these resources and handle them automatically. But we invented the resources `length` and `color` expressly for the gauge. These settings will not have any effect unless we query their values and handle them explicitly.

Returning to the `gauge_create` procedure, you can see that we query the `length` resource for the hull frame with an `option get` command. We use the result to set the width of the canvas, and we hard-code the height of the canvas to a reasonable size. This gives us some control over the initial size of a gauge.[1] Also, if the optional bar color was not passed into this procedure, we use the default `color` resource, again determined from the `option get` command.

Finally, we create the two items on the canvas. The rectangle for the bar sits in the background, so we create it first. Rectangles have a black outline by default, but we disable this by setting the outline color to the null string. We set the fill color to the bar color that we determined a moment ago. And of course, we tag the rectangle with the name `bar`, so we can refer to it later.

We place the text item in the middle of the canvas and have it display “0%” as an initial value. We tag it with the name `value`, so we can refer to it later.

As your program works its way through a task, you can call the following procedure to update the gauge:

```
proc gauge_value {win val} {
```

[1] To keep things simple, we will ignore any size changes that might occur, say, when the user expands a window and stretches out a progress gauge. In Section 4.6, we'll see how to make a canvas react to size changes.

```
        if {$val < 0 || $val > 100} {
            error "bad value \"$val\": should be 0-100"
        }
        set msg [format "%3.0f%%" $val]
        $win.display itemconfigure value -text $msg

        set w [expr 0.01*$val*[winfo width $win.display]]
        set h [winfo height $win.display]
        $win.display coords bar 0 0 $w $h

        update
    }
```

You pass in the name of the gauge, and the percentage value for the gauge to display. If the value is out of range, we immediately flag an error. Otherwise, we use the `format` command to neatly format the value, and we display the result in the `value` item.

The `%3.0f` part of the format string prints the floating point value as an integer, rounding it if necessary. The `3` says that the number should take up 3 spaces, and the `0` says that there should be no digits after the decimal point. The extra `%%` at the end of the format string becomes a literal `%` in the display string. So a value like 82.41 would be displayed as “` 82%`”.[2]

Next, we adjust the length of the bar so that it reflects the new value. We use `winfo width` and `winfo height` to get the overall size of the canvas, and we scale the width down according to the percentage value. Once we have computed the size, we use the canvas `coords` operation to change the coordinates of the `bar` rectangle.

Finally, we use the `update` command to flush changes out to the display. The `update` command is very important. The canvas will avoid redrawing itself until the application is idle and has nothing better to do. You won’t see the bar move or the text change unless we use `update` and force the canvas to redraw itself for each new value.

We can use our new gauge library to create the display shown in Figure 4.10. First, we create a gauge and pack it into the main window:

```
gauge_create .g PaleGreen
pack .g -expand yes -fill both -padx 10 -pady 10
```

Instead of relying on the default bar color, we used the value `PaleGreen` in this example. Now, we perform our long-running task:

```
for {set i 0} {$i <= 100} {incr i} {
    after 100
    gauge_value .g $i
}
```

[2] Notice that this string has a leading space, since our field width is 3 digits.

On each pass through the loop, we simply wait 100ms, then call `gauge_value` with the current status and continue on through the loop. But in real applications you would replace the `after` command with some real code. You might read in a file, handle some input on a socket, or process part of a list. As long as you call `gauge_value` from time to time with a progress value, you'll see your progress in the gauge.

4.4 HSB Color Editor

You can use the canvas to build interactive control panels. Certain items can act as knobs or handles that the user can drag around on the screen. We'll see how the canvas `bind` operation supports this in the following example.

Many drawing programs let you dial up your own colors using a color wheel like the one shown in Figure 4.11. You position the dot on the wheel to select a particular color, and you adjust the bar on the right-hand side to make it lighter or darker. In the center of the wheel, the colors are said to be *unsaturated*—they are some shade of gray. As you move outward toward the edge of the wheel, they become *saturated*, or full of color. As you move around the edge of the wheel, the color changes from red to green to blue, and back to red. The position around the rim determines the overall *hue* of the color. And as you adjust the bar on the right-hand side, you control the *brightness* of the color. So as you move all of these controls, you are really adjusting the three components that determine a color: its hue, its saturation, and its brightness. This kind of color selector is referred to as a Hue-Saturation-Brightness (HSB) color editor.

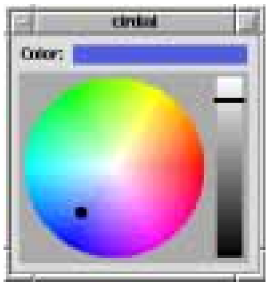


Figure 4.11. An HSB color editor built using the canvas.

An HSB color editor is not included as part of Tk, but it is easy to build one using the canvas. We could draw the various parts of the editor by adding items to the canvas. We could add a text item for the *Color:* label, a rectangle for the current color sample, some arcs for the color wheel, and so on. But for this example, we will use a few tricks to simplify things.

First, we will use the canvas only for the color wheel and the brightness controls. We'll create a separate label widget for the *Color:* label, and a separate frame widget for the color sample, as shown in Figure 4.12. That way, we won't need any tricky canvas code to compute the size of the label and align it with the color sample. We'll see how to align things in the next section. For now, we'll simply use the `pack` command to align the three major elements.

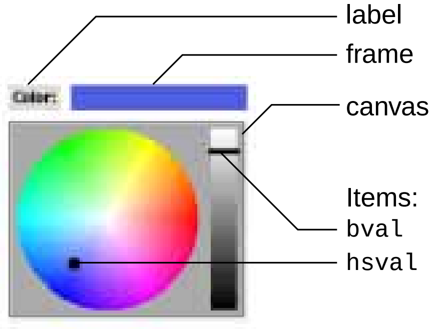


Figure 4.12. The HSB color editor has a label, a frame and a canvas. Two items on the canvas act as markers for the current hue-saturation and brightness values.

Also, we'll use the predefined image shown in Figure 4.13 as the background of the color wheel and brightness controls. We could draw something similar by creating arc and rectangle items, but the result would look chunky. If we created lots of small arcs and rectangles, the picture would look better, but the controls might be sluggish. The canvas performs quite well with tens or hundreds of items, but as the number of items increases, the performance degrades. Quite often, you can use predefined images like this to add detail to a drawing without incurring a performance penalty.

So the color editor boils down to a canvas with three items: an image for the background, an oval tagged with the name `hsval` to mark the current hue-saturation value, and a line tagged with the name `bval` to mark the current brightness value. We'll add bindings to the `hsval` and `bval` items so that you can move them around to adjust the color value.

The following procedure creates a color editor. You simply give it a widget name for the whole assembly:

```
proc colordial_create {win} {
    global env cdInfo
    frame $win -class Colordial
```

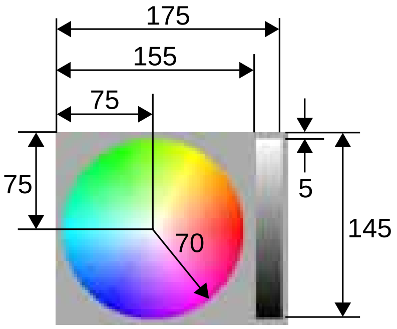


Figure 4.13. The background of the HSB color editor is a color image with fixed dimensions.

```
canvas $win.dial
pack $win.dial -side bottom

label $win.label -text "Color:"
pack $win.label -side left

frame $win.sample -width 15 -height 15
pack $win.sample -expand yes -fill both -padx 4 -pady 4

set fname "$env(EFFTCL_LIBRARY)/images/colors.gif"
set imh [image create photo -file $fname]
$win.dial create image 0 0 -anchor nw -image $imh
$win.dial create oval 0 0 0 0 -fill black -tags hsval
$win.dial create line 0 0 0 0 -width 4 -fill black -tags bval

$win.dial configure -width [image width $imh] \
    -height [image height $imh]

$win.dial bind hsval <B1-Motion> \
    "colordial_set_hs $win %x %y"
$win.dial bind bval <B1-Motion> \
    "colordial_set_b $win %y"

set cdInfo($win-hue) 0
set cdInfo($win-saturation) 0
set cdInfo($win-brightness) 1
```

```
        colordial_refresh $win
        return $win
    }
```

As usual, we start by creating the hull frame which contains the other components. We give it the class name `Colordial`, so you can add resources to the option database to customize all of the colordials in your application.

Next, we create the separate canvas, label and frame widgets, and pack them into the hull. By default, an empty frame has a width and height of 0 pixels, so we must do something to fix its size. We set its requested size to 15 × 15 pixels (so the frame will ask to be at least this big), and we pack it to expand and fill. Since it is packed after the other two widgets, it fills the area to the right of the label, in the upper-right corner of the hull.

Next, we create the three items on the canvas. The image is positioned with its northwest corner at the origin of the canvas. The `hsval` and `bval` items are created with dummy coordinates. We update them later using the `colordial_refresh` command so that they have the right coordinates for the initial color value.

We use the `image create photo` command to load the color wheel image from a file called *colors.gif*. This file must be distributed along with the script file containing the code for our colordial. These files will be installed in some directory like */usr/local/efftcl*. But instead of hard-coding the directory name, we use an environment variable `EFFTCL_LIBRARY` to point to the proper directory. We can write an installation program which sets up this variable when a program is installed, as we'll see in Chapter 8.

The canvas should be just big enough to display the color wheel image. We use the `image width` and `image height` commands to query the overall size of the image, and we configure the canvas to this size.

Next, we add the bindings which allow the `hsval` and `bval` items to be moved around on the canvas. We use the event specification `<B1-Motion>`, which triggers when you hold down the mouse button #1 and drag the pointer around. We added the bindings to the individual `hsval` and `bval` items—not to the canvas as a whole. So you will only get these events when you click directly on one of the two items, and then drag the mouse. Notice that we are careful to enclose the commands for the bindings in `""` so that the value of `$win` is substituted in this context, while it is known. In effect, we are building custom bindings for each colordial that we create.

Next, we set the initial color value. The global `cdInfo` array acts as a data structure, as we described in Section 1.3.2. Each colordial has three slots in this array for the hue, saturation and brightness components of its current color. The slots are parameterized by the name `$win`, so each colordial has its own set of slots. Once the color is set, we call `colordial_refresh` to update the display. This sets the background of the color sample and moves the `hsval` and `bval` items into the appropriate position. We'll see how this is implemented in a moment.

Finally, we return the name of the new colordial as the result of this procedure.

When you click and drag on bval, it will trigger a series of calls to colordial_set_b, with the y-coordinate for each motion point. This procedure is implemented as follows:

```
proc colordial_set_b {win y} {
    global cdInfo
    set bright [expr (145-$y)/140.0]
    if {$bright < 0} {
        set bright 0
    } elseif {$bright > 1} {
        set bright 1
    }
    set cdInfo($win-brightness) $bright
    colordial_refresh $win
}
```

First, we convert the y-coordinate to the appropriate brightness level. The y-coordinates in the brightness area range from 5 to 145, as you can see in Figure 4.13. We scale the y-coordinate, and then limit the brightness value so that it falls in the range from 0 to 1. Finally, we update the brightness component of the data structure for this colordial, and then use colordial_refresh to update the display.

Likewise, when you click and drag on hsval, it will trigger a series of calls to colordial_set_hs with the x- and y-coordinates for each motion point. This procedure is implemented as follows:

```
proc colordial_set_hs {win x y} {
    global cdInfo

    set hs [colordial_xy2hs $x $y]
    set hue [lindex $hs 0]
    set sat [lindex $hs 1]

    if {$sat > 1} {
        set sat 1
    }
    set cdInfo($win-hue) $hue
    set cdInfo($win-saturation) $sat
    colordial_refresh $win
}
```

Again, we convert the (x,y) coordinate to the appropriate hue and saturation values on the color wheel. This is straightforward, though a bit messy, so we have encapsulated the details in a procedure called colordial_xy2hs. It takes an (x,y) coordinate and returns a list containing the corresponding hue and saturation values. We won’t show this procedure here, but you can find it in the file *efftcl/clrdial.tcl* in the source code that accompanies this book.

Again, we limit the saturation to the range from 0 to 1, so that it cannot be pulled beyond the color wheel. We update the hue and saturation components of the data structure, and then use `colordial_refresh` to update the display.

The `colordial_refresh` procedure is implemented like this:

```
proc colordial_refresh {win} {
    global cdInfo

    set angle $cdInfo($win-hue)
    set length $cdInfo($win-saturation)
    set x0 [expr 75 + cos($angle)*$length*70]
    set y0 [expr 75 - sin($angle)*$length*70]
    $win.dial coords hsval \
        [expr $x0-4] [expr $y0-4] \
        [expr $x0+4] [expr $y0+4]

    set bright $cdInfo($win-brightness)
    set y0 [expr 145-$bright*140]
    $win.dial coords bval 154 $y0 176 $y0

    $win.sample configure -background [colordial_get $win]
}
```

We save the current hue and saturation values in the `angle` and `length` variables. This makes the two lines of code that follow a bit easier to read. They are simply the formula for converting a polar coordinate like (length,angle) to a rectangular coordinate like (x,y). An angle of 0° and a length of 1 corresponds to the canvas coordinate (145, 75). As you can see in Figure 4.13, this is the position for pure red, on the right-hand side of the wheel. An angle of 90° and a length of 1 corresponds to (75,5), which is the yellow-green color at the top of the wheel.

Once we have computed the coordinate (`x0`,`y0`), we change the coordinates for the `hsval` marker to center it on this point. Remember, an oval is characterized by the two corners of its bounding box. We set one corner 4 pixels to the left and 4 pixels above the coordinate, and we set the other corner 4 pixels to the right and 4 pixels down. This makes the `hsval` marker a circle 8 × 8 pixels in size.

In a similar manner, we scale the brightness value back to a y-coordinate in the range from 5 to 145, and we update the coordinates of the `bval` line. The brightness scale extends from 155 to 175 along the x-axis, but we have stretched the line from 154 to 176. This makes it overhang by one pixel on each end, so it stands out from the background.

Finally, we change the background of the color sample frame to display the current color. We use the procedure `colordial_get` to convert the current color to a string that Tk can understand.

When you use a colordial in your own applications, you can use the same `colordial_get` procedure to query the current color on the dial. It is implemented like this:

```
proc colordial_get {win} {
    global cdInfo

    set h $cdInfo($win-hue)
    set s $cdInfo($win-saturation)
    set v $cdInfo($win-brightness)
    return [colordial_hsb2rgb $h $s $v]
}
```

We simply pass the current hue, saturation and brightness components to a procedure called `colordial_hsb2rgb`. This converts the color to equivalent red-green-blue (RGB) components. Exactly how this procedure works is outside the scope of this discussion.[3] However we will include its implementation here so that we can show one important trick:

```
proc colordial_hsb2rgb {h s v} {
    if {$s == 0} {
        set v [expr round(65535*$v)]
        set r $v
        set g $v
        set b $v
    } else {
        if {$h >= 6.28318} {set h [expr $h-6.28318]}
        set h [expr $h/1.0472]
        set f [expr $h-floor($h)]
        set p [expr round(65535*$v*(1.0-$s))]
        set q [expr round(65535*$v*(1.0-$s*$f))]
        set t [expr round(65535*$v*(1.0-$s*(1.0-$f)))]
        set v [expr round(65535*$v)]

        switch [expr int($h)] {
            0 {set r $v; set g $t; set b $p}
            1 {set r $q; set g $v; set b $p}
            2 {set r $p; set g $v; set b $t}
            3 {set r $p; set g $q; set b $v}
            4 {set r $t; set g $p; set b $v}
            5 {set r $v; set g $p; set b $q}
        }
    }
    return [format "#%.4x%.4x%.4x" $r $g $b]
}
```

[3] For more details, see J. D. Foley and A. Van Dam, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.

Once we have computed the proper `r`, `g` and `b` values, we use the `format` command to convert them to a hexadecimal representation. The "`#`" sign appears literally in the final string. Each "`%.4x`" tells how to print the `$r`, `$g` and `$b` components. The "`x`" says that they should be printed as hexadecimal numbers, and the "`.4`" says that they should be 4 digits wide, with leading zeros if need be. So overall, this handy statement will produce color values like "`#ffff00000000`" for red, "`#0000ffff0000`" for green, *etc.*

Now that you have these procedures, it is easy to add colordials to your applications. You can create a colordial like this:

```
colordial_create .cd
pack .cd
```

and spend some time adjusting the color. At any point, you can get the current color choice like this:

```
set cval [colordial_get .cd]
```

and then use that value to configure other widgets in your application.

4.5 Tabbed Notebook

The coordinates in a drawing are not always fixed. Sometimes the size of one item depends on another. If you're drawing a bit of text, for instance, you may want to fit a rectangle around it. In this example, we'll see how you can build a display that adjusts its layout according to its contents.

In Section 2.1.7, we built a simple notebook that lets you browse through various pages of widgets. At the time, we used a radiobox to dial up a particular page. Now, we'll use a canvas to decorate the top of each page with a tab, as shown in Figure 4.14. The result looks like the kind of notebook that you might find in an office supply store. We'll call this assembly a tabnotebook.

We'll design the tabnotebook so that you can use it just like the notebook described in Section 2.1.7. You create a tabnotebook like this:

```
tabnotebook_create .tn
pack .tn
```

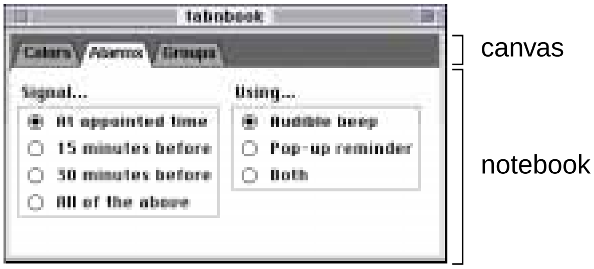
You use the `tabnotebook_page` procedure to create a new page:

```
set p1 [tabnotebook_page .tn "Colors"]
```

This creates an empty frame within the notebook and returns its window name. You can put widgets on this page by creating them as children of the frame. For example,

```
label $p1.mesg -text "Something on Colors page"
pack $p1.mesg -side left -expand yes -pady 8
```

You can select a particular page in the notebook by clicking on its tab, or by calling the `tabnotebook_display` procedure, like this:



Signal...
● At appointed time
○ 15 minutes before
○ 30 minutes before
○ All of the above

Using...
● Audible beep
○ Pop-up reminder
○ Both

Figure 4.14. The tabs in a tabnotebook are drawn on a canvas.

```
tabnotebook_display .tn "Colors"
```

Now that we understand how the tabnotebook works, let’s see how it is implemented. The `tabnotebook_create` procedure looks like this:

```
proc tabnotebook_create {win} {
    global tnInfo

    frame $win -class Tabnotebook
    canvas $win.tabs -highlightthickness 0
    pack $win.tabs -fill x

    notebook_create $win.notebook
    pack $win.notebook -expand yes -fill both

    set tnInfo($win-tabs) ""
    set tnInfo($win-current) ""
    set tnInfo($win-pending) ""
    return $win
}
```

As usual, we start by creating the hull frame which contains the other components. Inside the hull, we create a canvas and a notebook, and we pack them into position. We pack the canvas to fill across the top, so it will get wider as the window gets bigger. We pack the notebook to expand and fill, so it will get wider and taller as the window gets bigger.

We set the `-highlightthickness` option on the canvas to `0`. This removes the focus highlight ring that normally appears around the canvas. The focus highlight ring changes color whenever you type onto the canvas. Since we won’t be letting the user type directly onto the tabs, we don’t need this ring, and we don’t want the extra padding.

We give the hull the class name Tabnotebook, so you can add resources to the option database to customize all of the tabnotebooks in your application. For example, we include the following resources as defaults for the tabnotebook:

```
option add *Tabnotebook.tabs.background #666666 widgetDefault
option add *Tabnotebook.margin 6 widgetDefault
option add *Tabnotebook.tabColor #a6a6a6 widgetDefault
option add *Tabnotebook.activeTabColor #d9d9d9 widgetDefault
option add *Tabnotebook.tabFont \
    -*-helvetica-bold-r-normal--*-120-* widgetDefault
```

The first resource sets the background color for the canvas (named tabs) within our tabnotebook (class Tabnotebook). This is a standard option for the canvas, so Tk will handle it automatically. However, the other resources are names that we invented for the tabnotebook. We'll use the option get command to query their values as we draw the tabs.

We use the global tnInfo array as a data structure. Each tabnotebook has three slots in this array parameterized by $win, the name of the tabnotebook. The slot $win-tabs stores the list of tab names. The slot $win-current stores the tab name for the page that is being displayed. And we'll see in a moment how the $win-pending slot is used.

Once a tabnotebook has been created, you can add pages by calling the tabnotebook_page procedure:

```
proc tabnotebook_page {win name} {
    global tnInfo

    set page [notebook_page $win.notebook $name]
    lappend tnInfo($win-tabs) $name

    if {$tnInfo($win-pending) == ""} {
        set id [after idle [list tabnotebook_refresh $win]]
        set tnInfo($win-pending) $id
    }
    return $page
}
```

You pass in the name of the tabnotebook and the name of the new page. We call the notebook_page procedure defined in Section 2.1.7 to create the actual page, and we return this as the result of the procedure.

We also add the page name to the list of tabs, and we set things up so that the new tab will appear on the canvas. We will use a procedure called tabnotebook_refresh to clear the canvas and draw a new set of tabs. But if we call this procedure directly in tabnotebook_page, the canvas will be redrawn again and again as each new tab is added. That wouldn't be very efficient.

You may encounter this “refresh” problem when you build other canvas applications. You can handle it as follows. Instead of calling tabnotebook_refresh immediately, we use the after idle command to defer the call. When the application is idle and has nothing better to do, it will draw the set of tabs. Each call to `after idle` returns a unique identifier like `after#12`. We save this identifier in the `$win-pending` slot for the tabnotebook. So the next time we add a page, we will know that one refresh call is pending, and we won't generate another.

All of the interesting canvas code appears in the `tabnotebook_refresh` procedure that draws the tab set. It looks like this:

```
proc tabnotebook_refresh {win} {
    global tnInfo

    $win.tabs delete all

    set margin [option get $win margin Margin]
    set color [option get $win tabColor Color]
    set font [option get $win tabFont Font]
    set x 2
    set maxh 0

    foreach name $tnInfo($win-tabs) {
        set id [$win.tabs create text \
            [expr $x+$margin+2] [expr -0.5*$margin] \
            -anchor sw -text $name -font $font -tags [list $name]]

        set bbox [$win.tabs bbox $id]
        set wd [expr [lindex $bbox 2]-[lindex $bbox 0]]
        set ht [expr [lindex $bbox 3]-[lindex $bbox 1]]
        if {$ht > $maxh} {
            set maxh $ht
        }

        $win.tabs create polygon 0 0  $x 0 \
            [expr $x+$margin] [expr -$ht-$margin] \
            [expr $x+$margin+$wd] [expr -$ht-$margin] \
            [expr $x+$wd+2*$margin] 0 \
            2000 0  2000 10  0 10 \
            -outline black -fill $color \
            -tags [list $name tab tab-$name]

        $win.tabs raise $id

        $win.tabs bind $name <ButtonPress-1> \
            [list tabnotebook_display $win $name]

        set x [expr $x+$wd+2*$margin]
    }
    set height [expr $maxh+2*$margin]
```

```
        $win.tabs move all 0 $height

        $win.tabs configure -width $x -height [expr $height+4]

        if {$tnInfo($win-current) != ""} {
            tabnotebook_display $win $tnInfo($win-current)
        } else {
            tabnotebook_display $win [lindex $tnInfo($win-tabs) 0]
        }
        set tnInfo($win-pending) ""
    }
```

There's a lot of code here, but the end result is quite simple: It clears the canvas and draws a tab for each page in the notebook. The position of each tab depends on the tabs before it, so we have to draw the tabs in order from left to right. We know what each tab should look like—it's just a polygon item for the tab, and a text item for the label. But we don't know the exact coordinates for the polygon until we've drawn the text. After all, the tab name can be any length, and the text font can be any size. So we create the text item, draw a polygon around it, create the next text item, draw a polygon around it, and so on.

Let's go through the procedure step by step. Keep in mind that the `win` parameter refers to the entire tabnotebook assembly, so the canvas widget inside is `$win.tabs`.

We start off by telling the canvas to delete all of its items. This clears any tabs that we may have drawn the last time the tabs were refreshed.

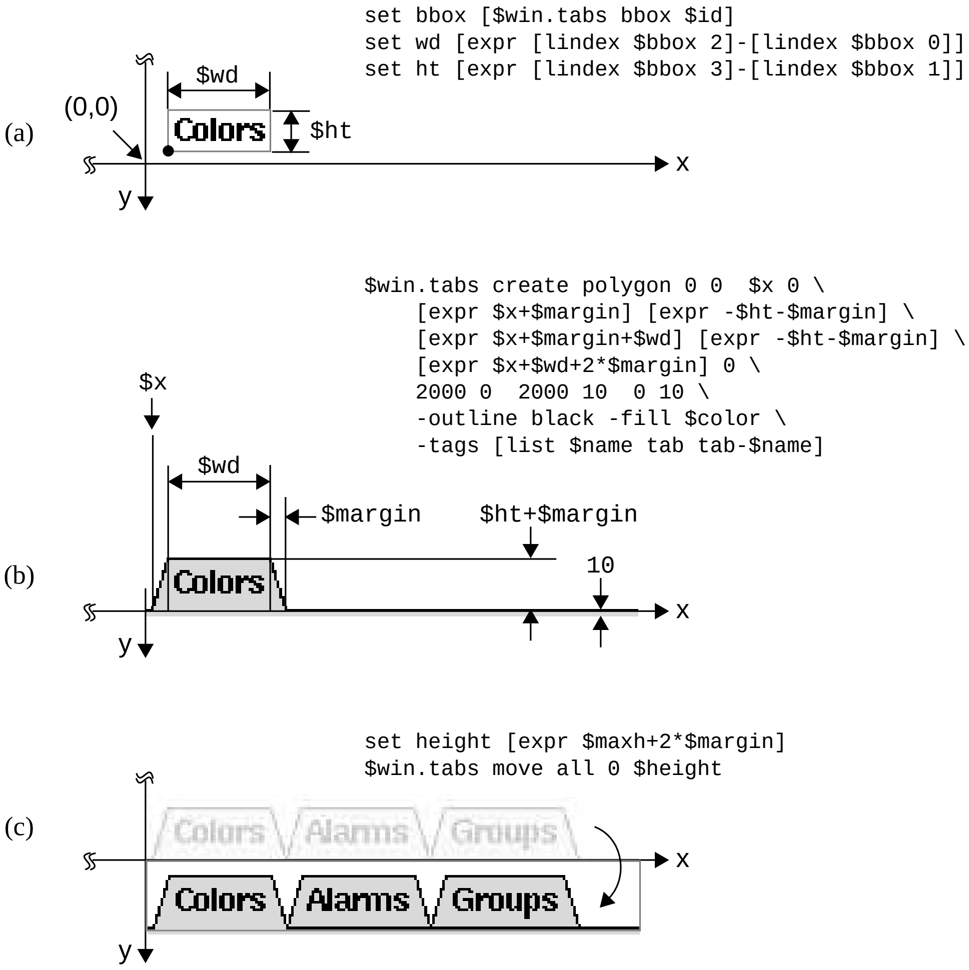
Next, we initialize some variables that control the drawing process. We use the variable `x` to keep track of our position from left to right as we draw each tab. We use the variable `maxh` to store the height of the tallest tab.

The variable `margin` controls the padding around the label on each tab. The variable `color` controls the background color for each tab. The variable `font` controls the font used for the label. Instead of hard-wiring these values into the script, we query them from the option database. That way, you can customize the look of the tabnotebook for different applications.

Next, we iterate through the list of tab names and draw each tab, as shown in Figure 4.15. We can simplify the coordinates a bit if we draw each tab with its baseline at y=0. Of course, if we left the tabs at this position, they would be outside of the normal viewing area. So before we finish, we'll move them back down where they belong. But for now, we'll think of (0,0) as the lower-left corner of the tab set.

As we said earlier, we create the text item for each tab first. We position its southwest corner a little to the right of `$x` and a little above the baseline, as shown in Figure 4.15(a). Remember, the canvas returns a unique number for each item that we create. We'll need this number to refer to the text item, so we save it in a variable called `id`.

We use the canvas `bbox` operation to query the bounding box for the text. It returns a list of four numbers representing the (x,y) coordinate for the upper-left corner, and the (x,y) coordinate for the lower-right corner. We subtract the two x-coordinates to compute the overall width, and we subtract the two y-coordinates for the overall height.



(a)

```
set bbox [$win.tabs bbox $id]
set wd [expr [lindex $bbox 2]-[lindex $bbox 0]]
set ht [expr [lindex $bbox 3]-[lindex $bbox 1]]
```

(b)

```
$win.tabs create polygon 0 0  $x 0 \
    [expr $x+$margin] [expr -$ht-$margin] \
    [expr $x+$margin+$wd] [expr -$ht-$margin] \
    [expr $x+$wd+2*$margin] 0 \
    2000 0  2000 10  0 10 \
    -outline black -fill $color \
    -tags [list $name tab tab-$name]
```

(c)

```
set height [expr $maxh+2*$margin]
$win.tabs move all 0 $height
```

Figure 4.15. A series of commands builds the tab display on the canvas.

Next, we create a polygon for the tab, as shown in Figure 4.15(b). This requires a long list of coordinates. The first point is (`0,0`), the next point is (`$x,0`), and so on. We extend the tab's baseline out to `2000` along the x-axis, which for all practical purposes is infinity. We draw the bottom of the tab down to `10` along the y-axis. When we shift all of the tabs down, this will be low enough that it will disappear off the bottom of the canvas.

As we create each polygon, we tag it with three names. The name tab applies to all of the tab polygons. We'll use this later to reset the background color for the tabs that are not selected. The name tab-$name applies to the polygon for a particular tab. We'll use this to set the background of the selected tab. The name $name applies to both the text and the polygon for each tab. We'll use this whenever we want to refer to the tab as a whole.

Since the polygon is created after the text item, it would normally obscure the text. We fix this by using the canvas raise operation to raise the text item back to the top.

Finally, we bind to the <ButtonPress-1> event so that clicking on a tab will invoke tabnotebook_display, and display its associated page. Notice that we bind to the tag name $name, which applies to both the text and the polygon. So you can click anywhere on a tab, and it will respond. Had we added the binding to only the polygon, you would have to click directly on the polygon to actuate the tab. If you clicked on the text by accident, it would do nothing.

Also, notice that we use the list command to wrap up the call to tabnotebook_display. This keeps $name together as a single argument, even if it has spaces embedded within it. Had we used ""'s, names like “Colors” would work, but names like “Employment History” would generate an error.

We repeat this process for each tab in the display, shifting the position $x toward the right as we go. When all of the tabs are drawn, we compute the overall height of the tab set, and move the tabs down into position, as shown in Figure 4.15(c).

The canvas should be just big enough to display the final tab set, so we configure its width and height accordingly. We expose a few pixels below the baseline, but not the bottom of the tab polygon. This adds a little margin, but still makes it look like the tab is connected to the notebook below it.

If a page has already been selected, it is displayed again. As we'll see in a moment, this raises the tab for that page and changes its color. If there is no current page, then we select the first tab as the current page.

At long last, the refresh operation is done. We must be careful to reset the $win-pending slot in the tabnotebook data structure. If we add another page later on, this will allow us to start the cycle all over. Another call to tabnotebook_refresh will be pending, and when the application is idle, the tab set will be regenerated.

We need one more procedure for the tabnotebook. Clicking on a tab invokes tabnotebook_display, which is implemented like this:

```
proc tabnotebook_display {win name} {
    global tnInfo

    notebook_display $win.notebook $name

    set normal [option get $win tabColor Color]
    $win.tabs itemconfigure tab -fill $normal
```

```
        set active [option get $win activeTabColor Color]
        $win.tabs itemconfigure tab-$name -fill $active
        $win.tabs raise $name

        set tnInfo($win-current) $name
    }
```

It uses `notebook_display` to bring up the actual notebook page, and then highlights the current tab and raises it to the foreground.

Having good tag names makes this easy. We can use the name `tab` to refer to all of the tab polygons, so we can reset them all back to their normal color with a single command. We can use the name `tab-$name` to refer to a particular tab polygon, so we can highlight that tab with the active color. And we can use the name `$name` to refer to both the text and the polygon for the tab, so we can raise both items to the foreground as a group.

4.6 Calendar

The drawing on a canvas has a fixed size. So when you resize a canvas, you'll see more or less of its drawing, and you can use scrollbars to adjust the view. But you may want a drawing to scale with the size of a canvas. For example, suppose the drawing shows the status of the factory floor that we mentioned earlier. When you expand the window, you should see a larger view of the factory—not a larger window with lots of empty space. In the this example, we'll see how you can make a drawing react to size changes in the canvas. We'll also see some new techniques for handling selections and for updating complex displays.

Many business applications require a calendar like the one shown in Figure 4.16. Tk doesn't have a calendar widget, but as you probably know by now, it is easy to build one using the canvas. We'll draw each day as a rectangle item with a text item for the day number. We'll even add an image item in the lower-right corner of each day so that we can add decorations for holidays. We'll use a text item at the top to display the current month, and we'll use window items to position some buttons on either side, so you can move back and forth through the months.

4.6.1 Handling Size Changes

As usual, we'll write a procedure called `calendar_create` to create the calendar. You can call the procedure like this:

```
calendar_create .cal 7/4/97
pack .cal -expand yes -fill both
```

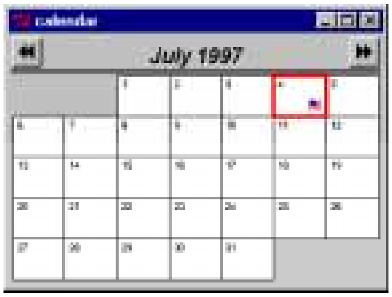


Figure 4.16. The canvas is used to create an interactive calendar.

This creates a calendar named `.cal` and packs it into the main window. When the calendar appears, it will display July 1997, but you'll be able to change the month by pressing the arrow buttons.

Notice that we packed the calendar to expand and fill. So if you expand the window, the canvas will expand too. Whenever it changes size like this, we'll redraw the calendar to cover the new size.

You can see the basic size-handling code in the `calendar_create` procedure, which is implemented like this:

```
proc calendar_create {win {date "now"}} {
    global calInfo env

    if {$date == "now"} {
        set time [clock seconds]
    } else {
        set time [clock scan $date]
    }
    set calInfo($win-time) $time
    set calInfo($win-selected) ""
    set calInfo($win-selectCmd) ""
    set calInfo($win-decorateVar) ""

    frame $win -class Calendar
    canvas $win.cal -width 3i -height 2i
    pack $win.cal -expand yes -fill both

    button $win.cal.back \
        -bitmap @$env(EFFTCL_LIBRARY)/images/back.xbm \
        -command "calendar_change $win -1"
    button $win.cal.fwd \
        -bitmap @$env(EFFTCL_LIBRARY)/images/fwd.xbm \
        -command "calendar_change $win +1"

    bind $win.cal <Configure> "calendar_redraw $win"

    return $win
}
```

Much of this code follows the recipe that we use for creating an assembly of widgets. We use the global variable `calInfo` as a data structure for all calendars. Each calendar has four slots in this array, parameterized by the calendar name `$win`. We'll explain the slots `$win-selected`, `$win-selectCmd` and `$win-decorateVar` later, when we need them. For now, we simply initialize them to the null string.

The slot `$win-time` stores an integer value from the system clock. It measures time as the number of seconds that have elapsed since January 1, 1970. When you call `calendar_create` with a date like `7/4/97`, we use the `clock scan` command to convert this to a time value, and we store it in the array. But the date is optional; if you don't specify a value, it gets the default value `now`, and we use the `clock seconds` command to query the current system time. One way or the other, we get a time value, and we store it away so that when we draw the calendar later on, we'll display the month that contains that time.

We create a hull frame with the class name `Calendar` to act as a container for the assembly. That way, we can add resource settings to the options database, to customize all of the calendars in an application. We create a canvas for the calendar and pack it to expand and fill in the hull. And we create the two buttons that let you page forward and backward through the months. Both of these buttons have bitmap labels with names like `@`*fileName*, so the bitmaps are loaded from files. But notice that neither of these buttons are packed into the hull. Instead, we'll position them on the canvas by creating window items when we draw the calendar.

All of the size-handling code boils down to a single `bind` command. Whenever a widget changes size, it receives a `<Configure>` event. So we bind to this event on the canvas. Whenever the canvas changes size, we call `calendar_redraw` to erase the canvas and draw the calendar at the new size. We enclose the `calendar_redraw` command in `""`'s instead of `{}`'s so that the value of `$win` is substituted now while it is known, instead of later when the event occurs.

Notice that we don't have to call `calendar_redraw` explicitly to draw the first month. Instead, when the canvas window appears on the desktop, it will get a finite size, so it will get a `<Configure>` event, and `calendar_redraw` will be called automatically to handle the size change.

Having a "redraw" procedure like this is useful for other reasons too. Suppose you click on one of the arrow buttons to change the month. We simply adjust the calendar's

`$win-time` slot forward or backward by a month, and then call `calendar_redraw` to display the new month. So the `calendar_change` procedure is implemented like this:

```
proc calendar_change {win delta} {
    global calInfo

    set dir [expr ($delta > 0) ? 1 : -1]
    set month [clock format $calInfo($win-time) -format "%m"]
    set month [string trimleft $month 0]
    set year [clock format $calInfo($win-time) -format "%Y"]

    for {set i 0} {$i < abs($delta)} {incr i} {
        incr month $dir
        if {$month < 1} {
            set month 12
            incr year -1
        } elseif {$month > 12} {
            set month 1
            incr year 1
        }
    }
    set calInfo($win-time) [clock scan "$month/1/$year"]

    calendar_redraw $win
}
```

We use the `clock format` command to extract the month and the year from the time value. The `%m` field is replaced with the month number. We use `string trimleft` to remove any leading `0`'s from this number, so it is not misinterpreted as an octal value when we change it later on. The `%Y` field is replaced with a year like `1997`. It won't have leading `0`'s, so it doesn't require the `string trimleft` step.

You would normally call this procedure with a `delta` value of ±`1` to move forward or backward by one month. But you can use a larger `delta` value to skip over several months at a time. We simply step forward or backward one month at a time, looking for changes to the year. When the month is decremented below January, we wrap around to Decemeber of the previous year. Or when the month is incremented beyond December, we wrap around to January of the next year. When we have arrived at a new month and year, we use the `clock scan` command to convert that date back into a time value for the calendar. We store it in the `$win-time` slot for the calendar, and then use `calendar_redraw` to display the month containing that time.

The `calendar_redraw` procedure erases the canvas, and then creates the items to draw a particular month. It is fairly long, so we have simplified it here by removing some code. We'll show the missing code later, as we continue to develop the example, and you can find a complete listing in the file *efftcl/calendar.tcl* in the source code that accompanies this book. But the simplified version looks like this:

```
proc calendar_redraw {win} {
    global calInfo
    ...

    $win.cal delete all

    set time $calInfo($win-time)
    set wmax [winfo width $win.cal]
    set hmax [winfo height $win.cal]

    $win.cal create window 3 3 -anchor nw \
        -window $win.cal.back
    $win.cal create window [expr $wmax-3] 3 -anchor ne \
        -window $win.cal.fwd
    set bottom [lindex [$win.cal bbox all] 3]

    set font [option get $win titleFont Font]
    set title [clock format $time -format "%B %Y"]
    $win.cal create text [expr $wmax/2] $bottom -anchor s \
        -text $title -font $font

    incr bottom 3
    $win.cal create line 0 $bottom $wmax $bottom -width 2
    incr bottom 3

    set font [option get $win dateFont Font]
    set bg [option get $win dateBackground Background]
    set fg [option get $win dateForeground Foreground]
    ...

    set layout [calendar_layout $time]
    set weeks [expr [lindex $layout end]+1]

    foreach {day date dcol wrow} $layout {
        set x0 [expr $dcol*($wmax-7)/7+3]
        set y0 [expr $wrow*($hmax-$bottom-4)/$weeks+$bottom]
        set x1 [expr ($dcol+1)*($wmax-7)/7+3]
        set y1 [expr ($wrow+1)*($hmax-$bottom-4)/$weeks+$bottom]
        ...
        $win.cal create rectangle $x0 $y0 $x1 $y1 \
            -outline $fg -fill $bg

        $win.cal create text [expr $x0+4] [expr $y0+2] \
            -anchor nw -text "$day" -fill $fg -font $font
        $win.cal create image [expr $x1-2] [expr $y1-2] \
            -anchor se -tags [list $date-image]
        ...
    }
    ...
}
```

We start by deleting all items on the canvas. That will erase any drawing that might exist from the last time we called this procedure. We use `winfo width` and `winfo height` to determine the overall size of the canvas. Our x-coordinates will run from `0` to `$wmax`, and y-coordinates, from `0` to `$hmax`.

We position the arrow buttons by creating window items on the canvas, as shown in Figure 4.17(a). We place one button anchored on its northwest corner in the upper-left corner of the canvas, and the other, anchored on its northeast corner in the upper-right corner of the canvas. These window items are merely placeholders for the actual buttons that we created in `calendar_create`. So we can delete the window items and make the buttons disappear without actually destroying the buttons themselves. Deleting a window item is analogous to the `pack forget` operation described in Section 2.1.7.

Next, we use the canvas `bbox` operation to get the y-coordinate for the bottom of the buttons. We use this as a baseline for the title that displays the current month, as shown in Figure 4.17(b). We use the `clock format` command to extract the title from the calendar's time value. The `%B` field gets replaced with a full month name like `July`, and the `%Y` field gets replaced with the full year like `1997`. We create a text item centered on the width, anchored with its south side on the baseline.

Instead of hard-coding the title font, we query the `titleFont` resource from the option database. We query some other resources too, including the `dateBackground`, `dateForeground` and `dateFont`. That way, you can customize the look of the calendar for different applications. For example, we include the following resources as defaults for the calendar:

```
option add *Calendar.dateBackground white widgetDefault
option add *Calendar.dateForeground black widgetDefault
option add *Calendar.selectColor red widgetDefault
option add *Calendar.selectThickness 3 widgetDefault

option add *Calendar.titleFont \
    -*-helvetica-bold-o-normal--*-180-* widgetDefault
option add *Calendar.dateFont \
    -*-helvetica-medium-r-normal--*-100-* widgetDefault
```

Once we have created the title, we move down 3 pixels, add a line item, and move down another 3 pixels. The variable `bottom` contains our final position. We'll fit the date squares into the remaining height.

We use another procedure called `calendar_layout` to determine where the date squares fall on the calendar. We won't show you how this is implemented. If you're curi-

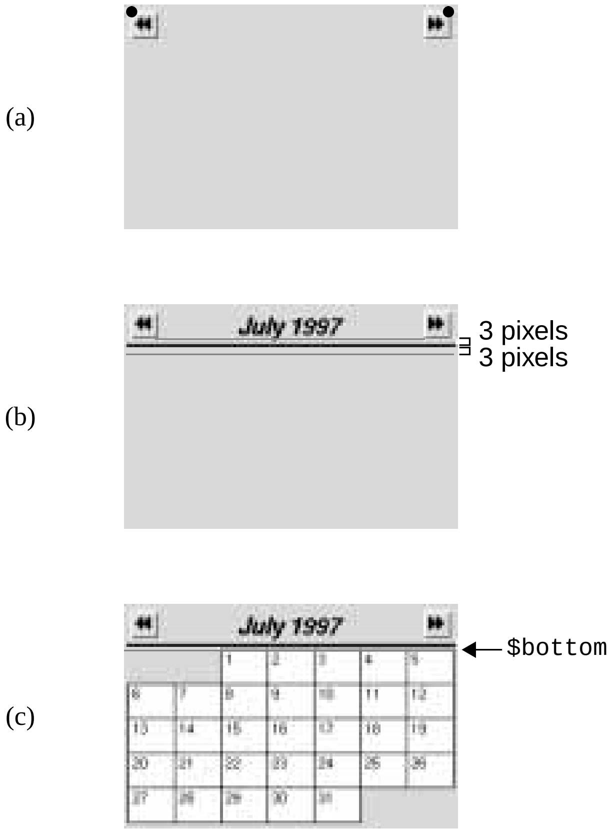


Figure 4.17. A calendar is drawn by creating items on the canvas.

ous, you can look at the file *efftcl/calendar.tcl*. But it returns a list of values that looks like this:

```
    1 07/01/1997 2 0   2 07/02/1997 3 0   3 07/03/1997 4 0 ...
```

The first four elements represent the first day on the calendar; the next four elements, the next day, and so on. Of the four elements, the first is the number of the day, the second is the actual date (including the month and the year), the third is the column for the day of the week, and the last element is the row for the week. The last group of four elements represents the last day on the calendar. So the very last element in the list is the index for the very last row of weeks. We use this to determine the total number of weeks, so we can divide up the remaining space on the canvas.

Normally, you use a simple foreach command to iterate through the values in a list. But what do you do when the values have a sequence that repeats itself like this? You give the foreach command a list of variables that represents your sequence. For example, we can iterate through our list with a command like:

```
set layout [calendar_layout $time]
foreach {day date dcol wrow} $layout {
    ...
}
```

The foreach command extracts the first four values from the list $layout, assigns them to the variables day, date, dcol and wrow, and then executes the body of the loop. On the next pass, it extracts the next four values, and again executes the body of the loop. So you can use the foreach command not only to iterate through a list, but also to dissect it.

As we scan through our list of layout information, we create the items to represent each date square, as shown in Figure 4.17(c). We use the row and column numbers for each date to compute the coordinates (x0,y0) and (x1,y1) for its background rectangle. We create the rectangle, and add a text item in its upper-left corner to display the day number. We also add a blank image in the lower-right corner. We'll use this later to add decorations for important holidays.

At this point, the calendar is complete. When it appears on the desktop, the canvas will get a <Configure> event, triggering a call to calendar_redraw to draw the calendar. If you expand the window, the canvas will get another <Configure> event, triggering another call to calendar_redraw. All items from the last calendar will be deleted, and new items will be created to fill the canvas at its new size.

4.6.2 Sensors and Callbacks

Suppose you use this calendar as part of an application for handling appointments. When you click on a particular day, we could highlight that day and bring up a list of appointments. Many canvas drawings have "hot spots" or items within them that can be selected. In this section, we'll see how you can support selections in a generic way. We'll add support for a selection command that lets you customize how the calendar will react in different applications.

By now you probably know that if we want an item to respond when you click on it, we simply bind to its <ButtonPress-1> event. Each date square on the calendar is composed of three items: the background rectangle, the day number, and an image that might be used for decorations. So if we want the date square to respond when you click on it, we could simply bind to <ButtonPress-1> on all three items. Or, if we tag all three items with the same group name, we could simply bind to the group as a whole. This is how we handled the tabs in the tabnotebook described in Section 4.5.

But there is another technique for handling selections that is sometimes easier to use. Instead of binding to all three items in each date square, we simply cover the square with an invisible rectangle, and bind to it. The canvas will let you use `""` as a color name, so you have a way of suppressing the outline color or the fill color of an item. If you suppress both colors, you will get an invisible item. But the item will still react to events on the canvas, as if it were filled with a solid color. We'll call this kind of item a *sensor.*

We can add some code to `calendar_redraw` to create a sensor over each date square, like this:

```
proc calendar_redraw {win} {
    ...

    foreach {day date dcol wrow} $layout {
        ...
        $win.cal create rectangle $x0 $y0 $x1 $y1 \
            -outline $fg -fill $bg

        $win.cal create text [expr $x0+4] [expr $y0+2] \
            -anchor nw -text "$day" -fill $fg -font $font

        $win.cal create image [expr $x1-2] [expr $y1-2] \
            -anchor se -tags [list $date-image]


        ...
        $win.cal create rectangle $x0 $y0 $x1 $y1 \
            -outline "" -fill "" \
            -tags [list $date-sensor all-sensor]

        $win.cal bind $date-sensor <ButtonPress-1> \
            "calendar_select $win $date"
    }
    ...
}
```

Again, we've left out some code to avoid repeating what was shown in the last section.

Notice that we create the sensor after the other three items, so it will cover them on the canvas. Had we done it the other way around, the sensor itself would be covered, and it would not get any events. If for some reason we had to create the sensor earlier in the procedure, we could fix this problem by using the canvas `raise` or `lower` operations to achieve the proper stacking order.

We tag each sensor with the name `$date-sensor`, so the sensor for July 1, 1997 has the tag `07/01/1997-sensor`. Also, we tag all sensors with the name `all-sensor`, so we can handle them as a group.

Finally, we bind to the `<ButtonPress-1>` event on each sensor, so clicking on it triggers a call to `calendar_select` to select that date. Again, we enclose the calendar_select command in ""'s instead of {}'s so that the values for $win and $date are substituted in. In effect, each sensor has its own custom binding that tells the calendar to select its date.

The calendar_select procedure highlights a particular date on the calendar. It is implemented like this:

```
proc calendar_select {win date} {
    global calInfo
    set time [clock scan $date]
    set date [clock format $time -format "%m/%d/%Y"]

    set calInfo($win-selected) $date

    set current [clock format $calInfo($win-time) \
        -format "%m %Y"]
    set selected [clock format $time -format "%m %Y"]

    if {$current == $selected} {
        set fg [option get $win dateForeground Foreground]
        $win.cal itemconfigure all-sensor \
            -outline "" -width 1

        set color [option get $win selectColor Foreground]
        set width [option get $win selectThickness Thickness]
        $win.cal itemconfigure $date-sensor \
            -outline $color -width $width
        $win.cal raise $date-sensor
    } else {
        set calInfo($win-time) $time
        calendar_redraw $win
    }

    if {[string trim $calInfo($win-selectCmd)] != ""} {
        set cmd $calInfo($win-selectCmd)
        set cmd [percent_subst %d $cmd $date]
        uplevel #0 $cmd
    }
}
```

The first two set commands look a bit strange, but they do something important. They normalize the date argument to a standard format by converting the date to a system time value, and then back again. So you can use a command like:

```
calendar_select .cal "July 1, 1997"
```

and the date argument will be normalized to 07/01/1997. As long as the date is in this format, we can access the sensor using the tag name $date-sensor.

Next, we save the selected date in the $win-selected slot of the calendar data structure. Later on, if you need to know what date is currently selected, you can call the following procedure:

```
proc calendar_get {win} {
    global calInfo
    return $calInfo($win-selected)
}
```

It simply looks into the data structure and returns the selected date.

Getting back to the calendar_select procedure, we need to determine whether or not the selected date is displayed on the current calendar. Normally, we will enter this procedure when you click on a date. So normally, the selected date is indeed displayed. But you could call this procedure from elsewhere in an application, and in that case, you could select any date. To check for this, we use the clock format command to build two strings. One represents the month and year that is currently displayed on the calendar. The other represents the month and year of the selected date. If the two are different, then we redraw the calendar to display the selected date.

Otherwise, we highlight the selected date by changing the outline of the sensor rectangle, making it visible. We also raise the sensor so that its outline is not obscured by any other items on the canvas. Instead of hard-coding the color and thickness of the selection highlight, we query them from the option database using the option get command.

Notice how our tag names simplify this operation. We remove the highlight from all sensors using the tag name all-sensor. Then we add the highlight to the selected date using the tag name $date-sensor.

Finally, we want the canvas to react to the date selection by bringing up a list of appointments, or inserting the date in an entry widget, or something like that. Instead of hard-coding the behavior in this procedure, we set things up so you can customize the calendar with your own callback command. So just as you would configure the -command option of each button, you can add a command to each calendar. You can add this same feature to other libraries that you create, if you follow the recipe shown here.

We use the $win-selectCmd slot of the calendar data structure to store the selection callback for each calendar. If there is any string in this slot, we invoke it as a command. First, we substitute the selected date into any %d field in the command, using the percent_subst procedure described in Section 7.6.7.3. So you can have a callback command like this:

```
puts "selected: %d"
.entry delete 0 end
.entry insert 0 "%d"
```

and when you select the date 07/01/1997, it would execute this:

```
puts "selected: 07/01/1997"
.entry delete 0 end
.entry insert 0 "07/01/1997"
```

This mimics the way that the bind command works.

When all the substitutions are in place, we invoke the callback command using `uplevel #0`. This forces it to execute outside of our procedure, in the global scope. So if the callback sets any variables, they will be treated as global variables.

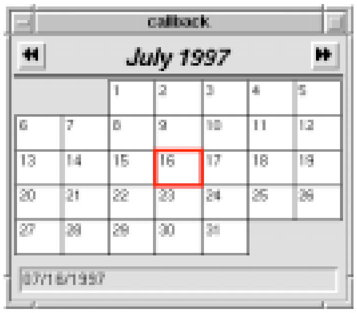
You can use the following procedure to set the callback command for a calendar:

```
proc calendar_select_cmd {win cmd} {
    global calInfo

    if {![info exists calInfo($win-selectCmd)]} {
        error "bad calendar name \"$win\""
    }
    set calInfo($win-selectCmd) $cmd
}
```

It simply checks to make sure that the slot `$win-selectCmd` exists, and then assigns a code fragment to it.

Putting all of this together, we can create a simple application like the one shown in Figure 4.18. When you select any date on the calendar, it is automatically typed into the entry widget below the calendar. You might use this as part of a dialog box for selecting dates. That way, the user can either browse through the calendar, or enter a date by hand.



		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

07/16/1997

Figure 4.18. Selecting a date on the calendar fills in the entry below it.

This requires just a few lines of code:

```
calendar_create .cal 7/4/1997
pack .cal -expand yes -fill both

entry .entry
pack .entry -fill x -padx 4 -pady 4
```

```
calendar_select_cmd .cal {
    puts "selected: %d"
    .entry delete 0 end
    .entry insert 0 "%d"
}
```

By adding a selection callback, we've added a great deal of power to our simple calendar library. It now provides a convenient way of selecting dates for many different applications.

4.6.3 Monitoring Variables

Suppose we use the canvas to display the status of something, like the factory floor example that we mentioned earlier. We might use global variables to keep track of things like the output of the production line, the inventory at each station, and so on. But we need to know when something changes, so we can update the canvas. In this example, we'll see how you can monitor global variables in your application. When a variable changes, you can have the canvas update itself automatically to display the change.

Returning to our calendar example, we want to mark holidays and other important dates with an image in the lower-right corner, like the flag shown in Figure 4.16. Each date already has a blank image in the lower-right corner, so we can simply configure the image item on a certain date to display a particular image.

But how do we know which dates to decorate, and what images to use? We can use an array variable to store all of the important dates for the calendar. We might initialize the array like this:

```
set holidays(07/04/1997) [image create photo -file flag.gif]
set holidays(12/25/1997) [image create photo -file bell.gif]
...
```

The image in the file *flag.gif* is associated with the Fourth of July holiday; the image in *bell.gif* is associated with Christmas, and so on.

We need to make the calendar aware of this array, so it will consult the array as it draws each month. When you call a procedure like this, for example:

```
calendar_decorate_with .cal holidays
```

it will tell a calendar named `.cal` to use an array named `holidays` for decorations. This procedure is implemented as follows:

```
proc calendar_decorate_with {win decorateVar} {
    global calInfo

    if {![info exists calInfo($win-decorateVar)]} {
        error "bad calendar name \"$win\""
    }
    set calInfo($win-decorateVar) $decorateVar
    calendar_redraw $win
```

```
        global $decorateVar
        trace variable $decorateVar wu "calendar_decorate $win"
    }
```

First, we check to make sure that `$win` refers to a calendar. We look for the `$win-decorateVar` slot in the calendar data structure, and if it doesn't exist, we report an error. Otherwise, we store the array name—something like `holidays`—in the `$win-decorateVar` slot, so we can refer to it later. With the new information in place, we call `calendar_redraw` to update the calendar. As it draws each date, it will look for an entry in the array of decorations, and display the appropriate image.

We add the following code to `calendar_redraw`, to handle the decorations:

```
    proc calendar_redraw {win} {
        global calInfo

        if {$calInfo($win-decorateVar) != ""} {
            upvar #0 $calInfo($win-decorateVar) decorate
        }
        ...
        foreach {day date dcol wrow} $layout {
            ...
            $win.cal create image [expr $x1-2] [expr $y1-2] \
                -anchor se -tags [list $date-image]

            if {[info exists decorate($date)]} {
                $win.cal itemconfigure $date-image \
                    -image $decorate($date)
            }
            ...
        }
        ...
    }
```

Again, we've left out some code to avoid repeating what was shown in previous sections.

The slot `calInfo($win-decorateVar)` contains a name like `holidays`, which is the variable that we will consult for decorations. So we have one variable acting like a pointer to another variable. We can access the other variable quite easily if we connect to it using `upvar`. The `upvar` command says that we have a variable named `$calInfo($win-decorateVar)` (e.g., `holidays`) in another context, but in this procedure we'll call it `decorate`. Normally, `upvar` looks up to the calling procedure for the variable that you're trying to access. But in this case, we included the `#0` argument, telling `upvar` to look for a global variable (at level #0 in the call stack). From this point on, the name `decorate` is an alias for the variable that we're trying to access. When we query `decorate`, for example, we are really querying `holidays`. If we set `decorate`, we are really setting `holidays`.

As we draw each day on the calendar, we create a blank image tagged with the name `$date-image`. This will make it easy to refer to the image later on. The image item for a date like July 20, 1997, will have the name `07/20/1997-image`. After we create each image, we use `info exists` to look for an entry in the decorations array. If we find one, then we modify the image item to display the image for that date.

Now suppose that sometime later in the application, we add an entry to the `holidays` array. For example, we might let the user enter birthdays in a dialog. If one of those birthdays falls on the current calendar, its image should change immediately. A birthday cake icon should appear in the lower-right corner to mark the holiday.

We could say that whenever you update the `holidays` array like this, you must call `calendar_redraw` to see the changes. But there might be lots of places in the application where we modify the holidays array. If we forget to redraw after any one of them, users will report it as a bug.

There is a better way to handle this. We can monitor changes to any variable by putting a trace on it. In the last two lines of the `calendar_decorate_with` procedure, we put a trace on the decorations array. First, we declare it as a global variable. Otherwise, it would be treated as a local variable, and the trace would be forgotten when we exit the procedure. Next, we use the `trace variable` command to add the trace. The argument `wu` says that we want to be notified when the variable is written to (`w`) or unset (`u`). When this happens, it will trigger a call to `calendar_decorate`, to update the calendar.

Notice that in these two lines of code, we refer to the decorations array as `$decorateVar`. If we had used the name `decorateVar`, we would have attached the trace to the `decorateVar` variable in this procedure. Again, the variable `decorateVar` contains a name of another variable like `holidays`, which is the variable that we really want to trace. So `decorateVar` is acting like a pointer to another variable. When we use `$decorateVar` as a variable name, it is like dereferencing the pointer.

When we add a holiday, like this:

```
set holidays(07/20/1997) [image create photo -file birthday.gif]
```

or remove a holiday, like this:

```
unset holidays(07/20/1997)
```

it will trigger a call to the trace procedure, `calendar_decorate`. It is implemented as follows:

```
proc calendar_decorate {win name1 name2 op} {
    upvar #0 $name1 decorate

    if {[info exists decorate($name2)]} {
        set imh $decorate($name2)
    } else {
        set imh ""
    }
    $win.cal itemconfigure $name2-image -image $imh
}
```

Notice that it takes four arguments. We included the win argument when we added the trace in calendar_decorate_with. The other three arguments are added automatically on each call by the trace facility. The variable name1 contains the name of the variable that is being traced. So in our case, name1 will contain the name holidays. If this variable is an array, then the variable name2 indicates what slot is being modified. In our case, it will have a value like 07/20/1997. And the variable op contains the operation (w or u) that is currently being traced.

Once again, the variable name1 is acting like a pointer to another variable. So once again, we use upvar #0 to connect to it. We'll refer to the global variable called $name1 using a local variable called decorate.

We use the info exists command to look for a decoration in the slot decorate($name2). If it exists, then the trace is telling us that it was just modified, so we look up the new image value. If it doesn't, then the trace is saying that it was just deleted, so we clear out the image. In either case, we tell the canvas to look for an item named $name2-image and change its image. So when you modify a slot like holidays(07/20/1997), the canvas will look for an item called 07/20/1997-image. If it finds that item, then it updates the item. Otherwise, it ignores the request.

You can see the power of the trace facility in the following simple example:

```
image create photo flag -file $env(EFFTCL_LIBRARY)/images/flag.gif

calendar_create .cal
calendar_decorate_with .cal flags
calendar_select_cmd .cal {set flags(%d) flag}

pack .cal -expand yes -fill both
```

We create a calendar and tell it to use an array called flags for decorations. Then, we tell it to set the flags array whenever you select a date. If you click on a date like December 22, 1992, it will store the name flag in the slot flags(12/22/92). This will trigger a call to calendar_decorate, and a flag will appear on that day. In fact, a flag will appear on each day that you select. As application programmers, we never worry about redrawing the calendar. When we add dates to the flags array, the calendar reacts automatically.

4.7 Simple Drawing Package

The canvas lets you create items, change their colors, raise them, lower them, move them, resize them, and so on. This may seem familiar. Many commercial drawing packages work in almost the same way. In fact, you can use the canvas to build a commercial drawing package. In this section, we'll build the simple drawing program shown in Figure

4.19. Along the way, we'll see some new techniques, such as editing text on the canvas, saving the contents of a canvas, and generating PostScript output for a printer.

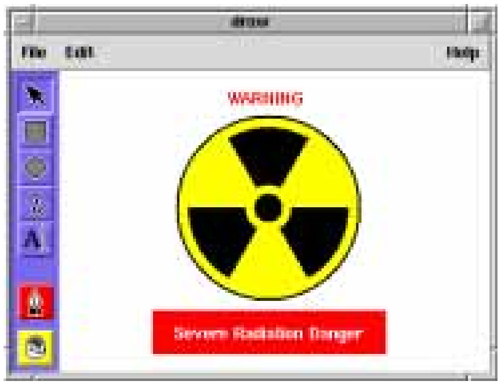


Figure 4.19. The canvas is used to create an interactive drawing program.

We can create most of the drawing program using components that we developed in other chapters. Figure 4.20 shows the interface broken down into its major components:

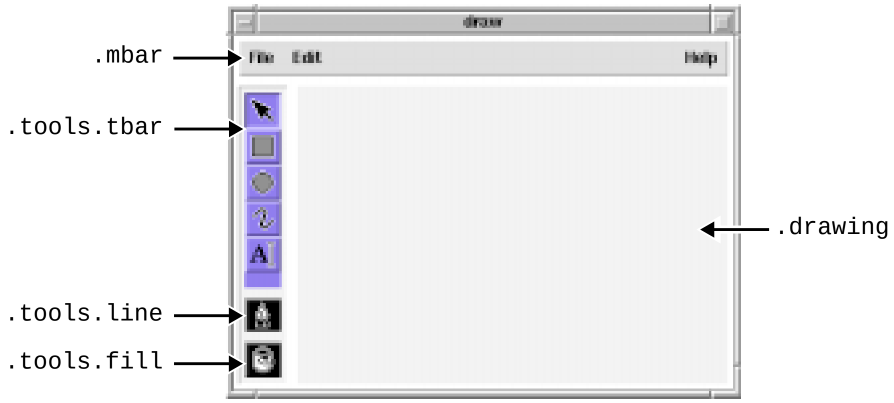


Figure 4.20. Major components in the drawing program.

- Along the top, we have a menu bar named `.mbar`. This is simply a frame with a few menubuttons packed inside it.
- Along the side, we have a toolbar named `.tools.tbar`. This is created with the toolbar library that we developed in Section 3.4.3. It includes the following tools, shown in order from top to botom: the selection tool, the rectangle tool, the circle tool, the spline tool, and the text tool. When you select a tool, it activates the appropriate set of bindings for the canvas. For example, if the rectangle tool is selected, the canvas has a set of bindings for the click, drag and release operations used to create a rectangle. We use bind tags to switch between the different bindings for each tool, as described in Section 3.5.4.
- Under the toolbar, we have the color selectors `.tools.line` and `.tools.fill`. They set the outline color and the fill color for new items. When you click on them, you get a short menu of color choices, as we saw earlier in Section 1.3.3.
- Most of the window is covered by the drawing canvas named `.drawing`. It is packed to expand and fill, so if you stretch out the window, the canvas will get bigger.

The code that creates these components is not particularly interesting. Similar code appears in the chapters that we've just mentioned. If you would like to see the details, you'll find the code in the file *apps/draw*, in the source code that accompanies this book. In the rest of this section, we'll assume that all of these components exist, and we'll focus on the canvas and its drawing operations.

4.7.1 Drawing Items

In Section 3.4.2, we saw how you can bind to the click, drag, and release events to create ovals on a canvas. We'll repeat this briefly here, so we can emphasize the role of the canvas.

When you select the rectangle tool, it activates the following bindings on the canvas:

```
bind rect <ButtonPress-1> {
    canvas_shape_create %W rectangle %x %y
}

bind rect <B1-Motion> {
    canvas_shape_drag %W %x %y
}

bind rect <ButtonRelease-1> {
    canvas_shape_end %W %x %y
}
```

Remember, the `bind` command automatically substitutes values into the `%` fields. The `%W` field will contain the name of the canvas, which in our program is `.drawing`. The `%x` and `%y` fields will contain the coordinates of the mouse pointer (relative to the upper-left corner of the canvas) at the time of the event.

When you click on the canvas, the canvas_shape_create procedure creates a new rectangle item with both corners at the click point. It is implemented like this:

```
proc canvas_shape_create {win shape x y} {
    $win create $shape \
        $x $y $x $y -outline black -width 2 \
        -tags "rubbershape"
}
```

We simply create an item of type $shape, which in this case is rectangle, and we tag it with the name rubbershape. This makes it easy to change the item in the following steps.

As you hold down the mouse button and drag the pointer, the canvas_shape_drag procedure moves the lower-right corner of the rectangle. It is implemented like this:

```
proc canvas_shape_drag {win x y} {
    set coords [$win coords "rubbershape"]
    set coords [lreplace $coords 2 3 $x $y]
    eval $win coords "rubbershape" $coords
}
```

We use the canvas coords operation to get the coordinates of the item called rubbershape. This returns a list of four numbers, representing the upper-left and lower-right corners of the rectangle. We substitute the current drag point into the list using lreplace, then assign the new coordinates back to the item. Notice that we use the eval command in this last step. The canvas coords operation requires individual coordinate numbers like this:

```
$win coords "rubbershape" 12 42 36 54
```

If we give it a list of coordinates like this:

```
$win coords "rubbershape" $coords    ;# error!
```

then it will see only one argument that it will interpret as a rather strange looking number, and it will report an error. The eval command joins its arguments together, and then interprets the result. In doing so, it strips off the quotes that normally delimit arguments, like this:

```
  eval .drawing coords "rubbershape" {12 42 36 54}
➥ .drawing coords rubbershape 12 42 36 54
```

So the eval command causes the numbers in $coords to be treated as separate arguments on the command line.

When you release the mouse button, the canvas_shape_end drops the rectangle and applies the colors from .tools.line and .tools.fill. It is implemented like this:

```
proc canvas_shape_end {win x y} {
    global canvInfo

    canvas_shape_drag $win $x $y
```

```
        $win itemconfigure "rubbershape" \
            -outline $canvInfo($win-penColor) \
            -fill $canvInfo($win-fillColor)

        $win dtag "rubbershape"
    }
```

We call `canvas_shape_drag` to update the coordinates for this last point, and then set the proper outline and fill colors. The colormenus store their values in the `$win-penColor` and `$win-fillColor` slots of the canvas data structure. We'll see how this works later in this section, but for now we simply use these colors. At this point, we "drop" the rectangle by using the canvas `dtag` operation to delete the `rubbershape` tag. The rectangle will remain, but we won't be able to refer to it using the name `rubbershape`. So the next time we create an item called `rubbershape` and drag its corner around, this rectangle won't be affected.

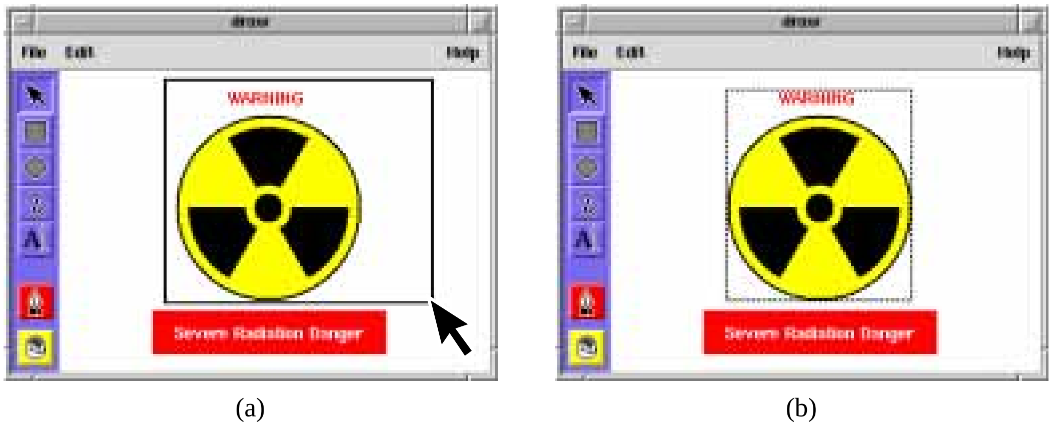
4.7.2 Selecting Items

When the selection tool is active, you can select items and change their characteristics. First, you drag out a *selection rectangle* as shown in Figure 4.21(a). When you release the mouse button, the selection rectangle disappears, and the selected items are enclosed in a dashed rectangle that we'll call the *marker rectangle*, as shown in Figure 4.21(b). We'll animate the dashes on this rectangle, so that they move or *shimmer* as a function of time. This makes it obvious that the marker rectangle is part of the selection process, and not a new item in the drawing.

When you enter the selection mode, we add the following bindings to the canvas:

```
bind select <ButtonPress-1> {
    canvas_shape_create %W rectangle %x %y
}
bind select <B1-Motion> {
    canvas_shape_drag %W %x %y
}
bind select <ButtonRelease-1> {
    canvas_select_end %W %x %y
}
bind select <Shift-ButtonRelease-1> {
    canvas_select_end %W %x %y add
}
```

We handle the click and drag events just as we described in the last section. So when you click on the canvas, we create a rectangle, and as you drag the mouse pointer, we stretch out the rectangle. But we handle the release event differently. Instead of leaving the rectangle as a new item on the canvas, we call `canvas_select_end` to handle the selection process. We look for items within the rectangle, mark them as "selected", and then delete the selection rectangle.



(a) (b)

Figure 4.21. (a) You select items by dragging out a selection rectangle. (b) Selected items are highlighted by a marker rectangle with a shimmering pattern.

There are two flavors of the release event. Normally, the items within the selection rectangle become selected, and any items that were previously selected are forgotten. But if you hold down the *Shift* key while making the selection, the items in the selection rectangle are *added* to the items that were previously selected. So you can build up a selection by holding down the *Shift* key as you select more and more items.

The `canvas_select_end` procedure is implemented like this:

```
proc canvas_select_end {win x y {op "clear"}} {
    global env canvInfo

    canvas_shape_drag $win $x $y

    set coords [$win coords "rubbershape"]
    foreach {x0 y0 x1 y1} $coords {}
    $win delete "rubbershape"

    canvas_select_done $win $op

    if {abs($x1-$x0) < 2 && abs($y1-$y0) < 2} {
        set items [$win find overlapping $x0 $y0 $x0 $y0]
        $win addtag "selected" withtag [lindex $items end]
    } else {
        eval $win addtag "selected" enclosed $coords
    }
```

```
            set coords [$win bbox "selected"]
            if {$coords != ""} {
                foreach {x0 y0 x1 y1} $coords {}

                $win create line \
                    $x0 $y0 $x1 $y0 $x1 $y1 $x0 $y1 $x0 $y0 \
                    -fill black -width 2 -tags "marker"

                set imagedir $env(EFFTCL_LIBRARY)/images
                set images {
                    stripes.xbm stripes2.xbm stripes3.xbm stripes4.xbm
                }
                set cmd "$win itemconfigure marker -stipple @$imagedir/%v"
                set canvInfo($win-shimmer) [animate_start 200 $images $cmd]
            }
        }
```

We start off by calling `canvas_shape_drag` to stretch the rectangle to its final position, and we use the canvas `coords` operation to query its final coordinates. This returns a list of four numbers defining the upper-left and lower-right corners of the selection rectangle. We could use the `lindex` command to pick apart the list like this:

```
set x0 [lindex $coords 0]
set y0 [lindex $coords 1]
set x1 [lindex $coords 2]
set y1 [lindex $coords 3]
```

But we can accomplish the same thing more compactly with a `foreach` command, like this:

```
foreach {x0 y0 x1 y1} $coords {}
```

This picks out the first four elements of the `$coords` list, assigns them to the four variables, and then executes the body, which does nothing. Since there are only four elements in the `$coords` list, the loop terminates immediately. As you can see, the `foreach` command provides a handy way of dissecting lists.

At this point, we know the coordinates of the selection rectangle, and we are done with the rectangle itself, so we delete it.

We are about to find the items within the selection rectangle and select them somehow. But what do we do with the items that are currently selected? If the `op` argument has the default value `clear`, we'll clear the current selection. But if it has the value `add`, we'll add the items to the current selection. In either case, we need to delete the marker rectangle that marks the current selection. We handle all of this by calling `canvas_select_done`. We'll see how this is implemented in a moment.

But first, we'll finish the select operation. We can keep track of selected items by tagging them with the name `selected`. The following command adds this tag to all of the items within the current selection rectangle:

```
eval $win addtag "selected" enclosed $coords
```

The `enclosed $coords` part says that the tag should be added only to the items that are completely enclosed in the rectangle defined by `$coords`. Once again, the four numbers within `$coords` must appear as separate arguments on the command line, so we use the `eval` command to break them out.

If you click and release on the same point, it will look as though you've drawn a tiny selection rectangle. We can detect this by checking for a small difference in the x- and y-coordinates. When we see this, we'll assume that you were clicking on a particular item, and we'll select that one item. We can use the following command to get a list of all items under the click point:

```
set items [$win find overlapping $x0 $y0 $x0 $y0]
```

This tells the canvas to find all of the items that touch a tiny rectangle from (*x0*,*y0*) to (*x0*,*y0*). The item number for the top-most item will be at the end of this list. We can tag that one item with the name `selected`, like this:

```
$win addtag "selected" withtag [lindex $items end]
```

Near the bottom of `canvas_select_end`, we create the marker rectangle that surrounds the selected items. We use the canvas `bbox` operation to get the coordinates of a bounding box that contains the selected items. If we get back a null string, then no items are selected, and we don't need the marker. This would occur, for example, when you select a blank area of the canvas.

Otherwise, we use the `foreach` command to pick apart the coordinate list, and we create a line item tagged with the name `marker` to represent the marker rectangle. We use a line item instead of a rectangle item so that we can add the dash pattern to the boundary. We set the dash pattern like this:

```
$win itemconfigure marker -stipple @$imagedir/stripes.xbm
```

This tells the canvas to draw the line using a bitmap pattern in the file *stripes.xbm*, which is a pattern of diagonal stripes. Where the bits are set, the line will be drawn in black, and where they are not, the line will be invisible.

We add the shimmering effect by changing the pattern as a function of time. The four files in the `images` list form a sequence of stripe patterns, each shifted by one pixel from the one before it. We use the `animate_start` procedure developed in Section 3.8.2 to cycle through the patterns. Every 200ms, a new pattern name is substituted into the `%v` field of the `cmd` string, and the command is executed, assigning a new stipple pattern to the marker. We save the result from `animate_start` in the `$win-shimmer` slot of the canvas data structure, so that we can stop the animation later on, when we delete the marker rectangle.

At this point, the selection operation is complete. The selected items are tagged with the name `selected`, and we can manipulate them with commands like:

```
.drawing raise "selected"
```

and

```
.drawing lower "selected"
```

These commands handle the *Bring to Front* and *Send to Back* operations found in many commercial drawing packages. We can add them to the *Edit* menu with some code like this:

```
.mbar.edit.m add command -label "Bring to Front" -command {
    .drawing raise "selected"
}
.mbar.edit.m add command -label "Send to Back" -command {
    .drawing lower "selected"
}
```

In the remaining sections, we'll see how to implement other operations like move, delete, resize, and so on.

But what happens when you choose a new tool from the toolbar? The selection mode ends, so we need to forget about the selected items. The toolbar takes care of this by calling `canvas_select_done`, which is implemented as follows:

```
proc canvas_select_done {win {op clear}} {
    global canvInfo

    $win delete "marker"
    if {[info exists canvInfo($win-shimmer)]} {
        animate_stop $canvInfo($win-shimmer)
        unset canvInfo($win-shimmer)
    }
    if {$op == "clear"} {
        $win dtag "selected"
    }
}
```

First, we delete the shimmering marker rectangle. We delete the rectangle itself with a simple canvas `delete` operation, but we must do something more to stop the shimmering. If we find a `$win-shimmer` slot in the canvas data structure, then it contains an identifier for the shimmer animation. We call `animate_stop` to stop the animation cycle, and we delete the `$win-shimmer` slot from the data structure.

Next, we remove the `selected` tag using the canvas `dtag` operation. This removes the tag from all items on the canvas, but leaves the items themselves intact. In effect, this clears the current selection.

Notice that we remove the tag only when the `op` argument has the value `clear`, which it gets by default. If we pass in a value like `add`, the selected items will remain selected. So we can use this procedure in `canvas_select_end` to clear the previous selection before defining a new one. When you hold down the *Shift* key, you'll get the `add` argument, and the new items will be added to the previous selection.

4.7.3 Moving and Deleting Items

Once you've selected some items, it is trivial to move them and delete them. We can add the following keyboard bindings to handle these operations:

```
bind select <KeyPress-BackSpace> {
    canvas_select_delete %W
}
bind select <KeyPress-Delete> {
    canvas_select_delete %W
}
bind select <KeyPress-Up> {
    canvas_select_move %W 0 -2
}
bind select <KeyPress-Down> {
    canvas_select_move %W 0 2
}
bind select <KeyPress-Left> {
    canvas_select_move %W -2 0
}
bind select <KeyPress-Right> {
    canvas_select_move %W 2 0
}
```

Notice that these bindings belong to the `select` bind tag. So like the bindings in the last section, they are active when the selection tool is active.

You can delete the selected items by pressing the *BackSpace* or *Delete* keys. This triggers a call to `canvas_select_delete`, which is implemented like this:

```
proc canvas_select_delete {win} {
    $win delete "selected"
    canvas_select_done $win
}
```

We simply delete all items tagged with the name `selected`, and then call `canvas_select_done` to clear the marker rectangle.

You can move the selected items by pressing the arrow keys on the keyboard. Many drawing packages call this the *nudge* operation. Each key press triggers a call to `canvas_select_move` with a 2-pixel offset in either the x- or the y-direction. The move operation is handled like this:

```
proc canvas_select_move {win dx dy} {
    $win move "selected" $dx $dy
    $win move "marker" $dx $dy
}
```

We simply move all of the items tagged as `selected` by an amount `$dx` in the x-direction, and an amount `$dy` in the y-direction. Of course, we update the marker rectangle by the same amount, so that it follows the selected items.

4.7.4 Configuring Items

At any point, you can change the drawing colors using the .tools.line and .tools.fill color menus. We can detect the color change by assigning a callback to each menu, like this:

```
colormenu_action .tools.line {canvas_pen .drawing "%c"}
colormenu_action .tools.fill {canvas_fill .drawing "%c"}
```

Whenever you choose a new color, the colormenu substitutes the color name in the %c field, and executes its command. So if you choose red as the line color, it will trigger a command like:

```
canvas_pen .drawing "red"
```

The canvas_pen procedure changes the line color of any items that are currently selected, and then makes note of the new line color. It is implemented like this:

```
proc canvas_pen {win color} {
    global canvInfo

    foreach item [$win find withtag "selected"] {
        switch [$win type $item] {
            rectangle - polygon - oval - arc {
                $win itemconfigure $item -outline $color
            }
            line - text {
                $win itemconfigure $item -fill $color
            }
            bitmap {
                $win itemconfigure $item -foreground $color
            }
        }
    }
    set canvInfo($win-penColor) $color
}
```

The canvas itemconfigure operation lets you change the characteristics of one or more items. If the selected items were all rectangles, for example, we could change their line color with a single command, like this:

```
$win itemconfigure $item -outline $color
```

But not all items have a -outline option. For example, the color of a line item or a text item is determined by its -fill option.

So we need to scan through the list of selected items and handle each one according to its type. We use the foreach command to iterate through a list of items tagged with the name selected, and we use the canvas type operation to determine the type of each item. The - character between labels in the switch statement acts as an “or” operator. So if an item is a rectangle, a polygon, an oval or an arc, we set its -outline color. If it’s a line or a text item, we set its -fill color. And if it’s a bitmap, we set its -foreground color.

Finally, we save the new line color in the slot `$win-penColor` in the canvas data structure. We use this color whenever we create an item, as we saw earlier in Section 4.7.1.

A similar thing happens when you choose a new fill color. If you choose green as the fill color, for example, it will trigger a command like:

```
canvas_fill .drawing "green"
```

The `canvas_fill` procedure looks a lot like the `canvas_pen` procedure, but it sets the fill color instead of the line color. It is implemented like this:

```
proc canvas_fill {win color} {
    global canvInfo

    foreach item [$win find withtag "selected"] {
        switch [$win type $item] {
            rectangle - polygon - oval - arc {
                $win itemconfigure $item -fill $color
            }
            bitmap {
                $win itemconfigure $item -background $color
            }
        }
    }
    set canvInfo($win-fillColor) $color
}
```

Again, we use `foreach` to scan through the items tagged with the name `selected`, and we check the type of each item. If an item is a rectangle, a polygon, an oval or an arc, we sets its `-fill` color. If it's a bitmap, we set its `-background` color. Otherwise, it has no fill color and we simply ignore it.

And again, we save the new fill color in the slot `$win-fillColor` in the canvas data structure. Any new items will automatically be filled with this color.

4.7.5 Resizing Items

Many drawing packages add little, black squares called *handles* around the edges of the marker rectangle. You can click and drag on the handles to adjust the size of the item. We could add handles to our drawing program as well, but it would complicate the code quite a bit. Instead, we'll provide a simple-minded way[4] to resize items that illustrates the core of the resize operation.

We'll add two items to the *Edit* menu, like this:

[4] Translation: brain-dead scheme

```
.mbar.edit.m add command -label "Enlarge" -command {
    canvas_select_scale .drawing 1.1 1.1
}
.mbar.edit.m add command -label "Reduce" -command {
    canvas_select_scale .drawing 0.9 0.9
}
```

When you select the *Enlarge* operation, it will enlarge the selected items by 10%, making them 1.1 times their current size. When you select *Reduce*, it will reduce them by 10%, making them 0.9 times their current size.

In either case, we call `canvas_select_scale`, which is implemented as follows:

```
proc canvas_select_scale {win sx sy} {
    foreach {x0 y0 x1 y1} [$win bbox "selected"] {}

    set xm [expr 0.5*($x0+$x1)]
    set ym [expr 0.5*($y0+$y1)]
    $win scale "selected" $xm $ym $sx $sy
    $win scale "marker" $xm $ym $sx $sy
}
```

The canvas `scale` operation scales the coordinates of one or more items by a certain amount in the x- and y-directions. For example, you can scale all of the items on a canvas to 1.5 times their current size, like this:

```
.drawing scale "all" 0 0 1.5 1.5
```

The `0 0` arguments give an (x,y) coordinate for the center of the scaling operation. So in this case, we enlarged the items around the origin in the upper-left corner of the canvas. All of the items spread out toward the right and toward the bottom, making the overall drawing larger.

In the `canvas_select_scale` procedure, we don't want the items to move as they spread out, so we scale them around their midpoint. That way, they get larger, but they stay in the same place on the drawing. We use the canvas `bbox` operation to get a bounding box for the selected items. We need to do some arithmetic on these coordinates, so we use the `foreach` command break up the list. It assigns the values to the variables `x0`, `y0`, `x1` and `y1`, and then does nothing in the body of the loop. This is a handy trick. It is more compact and convenient than an equivalent series of `lindex` commands.

Once we have the coordinates, we compute the midpoint by averaging the x and y values. We use the `scale` operation to scale the selected items by `$sx` in the x-direction, and by `$sy` in the y-direction. Of course, we scale the marker rectangle by the same amount, so that it follows the selected items.

4.7.6 Entering Text

The canvas will support text entry just like an entry widget or a text widget, but it is not automatically turned on. You have to add the right bindings to the canvas to enable this feature. In this section, we'll see how to do this in the context of our drawing program.

The text tool lets you add text annotations to the drawing. You simply click on the canvas to get a text insertion cursor, and type in the appropriate text. If you click on an existing text item, you'll edit that item. Otherwise, you'll get a new text item.

We can handle the click event with a binding like this:

```
bind text <ButtonPress-1> {
    canvas_text_select %W %x %y
}
```

Since this binding belongs to the `text` bind tag, the toolbar adds it to the canvas whenever the text tool is active.

When you click on the canvas, it triggers a call to `canvas_text_select` with the name of the canvas and the coordinates of the click point. This procedure is implemented as follows:

```
proc canvas_text_select {win x y} {
    global canvInfo

    canvas_text_done $win

    if {[$win type current] == "text"} {
        $win addtag "editText" withtag current
    } else {
        $win create text $x $y \
            -fill $canvInfo($win-penColor) \
            -anchor w -justify left -tags "editText"
    }

    focus $win
    $win focus "editText"
    $win icursor "editText" @$x,$y
}
```

First, we call `canvas_text_done` to close the editing mode for any other text item that we might be editing. We'll see how this procedure is implemented in a moment.

Next, we look for an existing text item at the click point. Remember, the canvas recognizes the name `current` as the item under the mouse pointer. So we can look for a text item by querying the type of the `current` item. If we find a text item, we tag it with the name `editText`. Otherwise, we create a new text item at the click point (`$x`,`$y`), and tag it with the name `editText`. As you type characters at the keyboard, we'll simply add them to the item called `editText`.

Now that we've selected a text item, we must direct the keyboard input to it. We do this by setting the keyboard focus, as described in Section 3.2.3. We use one command to set the widget focus for the program:

```
focus $win
```

This directs all keyboard events to the canvas widget $win, so they will trigger any <KeyPress> bindings that we have on the canvas. We use another command to set the focus to a particular item within the canvas widget:

```
$win focus "editText"
```

This directs all keyboard events on the canvas to the item called editText, so they will trigger any additional <KeyPress> bindings that we have added to that item. This also enables the insertion cursor for the editText item, so we can edit the text interactively.

The insertion cursor appears as a blinking line on the canvas. Exactly where it appears depends on its current position. We set its position using the command:

```
$win icursor "editText" @$x,$y
```

This tells the canvas to put the cursor for the editText item near the click point ($x,$y) on the canvas. So if you click in the middle of a text item, the insertion cursor will appear at that point.

Now that the text is selected, we need some bindings to handle key press events. We could use the bind command to add these bindings to the canvas as a whole. Or we could use the canvas bind operation to add the bindings to individual items within the canvas. Or we could use a mixture of bindings. In this example, we can handle all key press events in exactly the same way—by applying them to the item called editText. So in this example, we will bind to the canvas as a whole.

We add the following bindings to handle key press events:

```
bind text <KeyPress> {
    canvas_text_edit_add %W %A
}
bind text <KeyPress-Return> {
    canvas_text_edit_add %W "\n"
}
bind text <KeyPress-BackSpace> {
    canvas_text_edit_backsp %W
}
bind text <KeyPress-Delete> {
    canvas_text_edit_backsp %W
}
```

Most of the keys that you type are handled by the generic <KeyPress> binding. After each key stroke, the bind facility replaces %W with the name of the canvas, and %A with the ASCII code for the key, and then calls canvas_text_edit_add. This procedure adds the new character at the position of the insertion cursor, like this:

```
proc canvas_text_edit_add {win str} {
    $win insert "editText" insert $str
}
```

The first insert keyword tells the canvas to insert some text into one or more text items. The next word editText is the tag name indicating what text items to modify. The next word tells the canvas what character position to use when inserting the text. In this case, the keyword insert says that the text should be added just before the insertion cursor.

Finally, we give the canvas the character `$str` to insert. When the character is added, the insertion cursor shifts over automatically to make room for it.

The more specific key press bindings handle some special cases. We bind to `<KeyPress-Return>` to handle the *Return* key properly. The ASCII code for the *Return* key is the carriage return character “`\r`”. But we want the Return key to add a newline character “`\n`” so that we can force text entry onto another line.

We bind to both `<KeyPress-BackSpace>` and `<KeyPress-Delete>`, so hitting either of these keys triggers a call to `canvas_text_edit_backsp` to erase a character. This is implemented as follows:

```
proc canvas_text_edit_backsp {win} {
    set pos [expr [$win index "editText" insert] - 1]
    if {$pos >= 0} {
        $win dchars "editText" $pos
    }
}
```

We use the canvas `dchars` operation to delete a character in the text item called `_editText`. But we have to tell it which character to delete. The insertion cursor identifies the character to the right of it. So if we told `dchar` to delete at the insertion cursor, like this:

```
$win dchars "editText" insert
```

then it would delete the character after the insertion cursor, instead of the one before it.

Instead, we use the canvas `index` operation to find the position of the insertion cursor in the `editText` item. This returns a number, representing the character position. Characters are numbered from 0, so a number like 5 refers to the 6th character in the string. We subtract 1 to get the character before the insertion cursor, and then we delete that character.

If you change tools, or if you select another text item, we need to end the current editing operation. We handle this in `canvas_text_done`:

```
proc canvas_text_done {win} {
    set mesg [$win itemcget "editText" -text]
    if {[string length [string trim $mesg]] == 0} {
        $win delete "editText"
    }

    $win dtag "editText"
    $win focus ""
}
```

The first part eliminates any blank items. This might happen, for example, if you erase all of the text in an item, or if you click to create a text item, then change your mind and select another tool. We use the `itemcget` operation to get the contents of the `editText` item, and if it is blank, we delete it.

Next, we remove the `editText` tag from the current text item. If we didn’t do this, you would notice a problem when you edit another text item—both items would be tagged with the name `editText`, so each key stroke would apply to both items![5]

Finally, we reset the focus within the canvas so that no specific item has focus. This hides the text insertion cursor, indicating that the editing operation is done.

4.7.7 Printing a Drawing

Adding a *Print* feature to our drawing program is remarkably simple. The canvas has a `postscript` operation that you can use to generate PostScript output for the current drawing. For example, the canvas in our drawing program is named `.drawing`, so the command:

```
.drawing postscript
```

returns a very long string containing a PostScript representation of the canvas.

By default, the canvas will generate PostScript for only the portion of the drawing that is visible. Normally, the entire drawing is visible in our drawing program. But suppose that we compress the window to a smaller size, and then try to print. Or suppose that at some point we add scrollbars to the drawing program. In either case, we want to print the entire drawing—not just the part that is showing on the screen.

We need to pass the overall size of the drawing to the `postscript` operation, so we write a short procedure to handle this:

```
proc draw_print {} {
    foreach {x0 y0 x1 y1} [.drawing bbox all] {}
    set w [expr $x1-$x0]
    set h [expr $y1-$y0]
    return [.drawing postscript -x $x0 -y $y0 -width $w -height $h]
}
```

We use the canvas `bbox` operation to get a bounding box that surrounds all of the items, and we pick out the coordinate numbers `x0`, `y0`, `x1` and `y1`. We compute the overall width by subtracting the x-coordinates, and the overall height by subtracting the y-coordinates. Then, we tell the canvas to generate PostScript for a region of width `$w` and height `$h`, with its upper-left corner at the coordinate (`$x0`,`$y0`). We return the PostScript as the result of `draw_print`.

Most Unix systems have a command called `lpr` that routes output to a printer. We can use this command to print the canvas, like this:

```
exec lpr << [draw_print]
```

The `exec` command executes `lpr` as another process, feeding the PostScript output to its standard input. This queues up a print request, and a moment later, the printer produces the drawing.

Of course, that simple `lpr` command sends output to a default printer. In a real application, you should be able to send output to different printers, or perhaps save it in a file.

[5] Actually, this could be a bug or a feature, depending on how you write the user's guide.

We'll develop a printer dialog to handle all of this in Section 6.6.3. Let's assume that we've done the work and use it here.

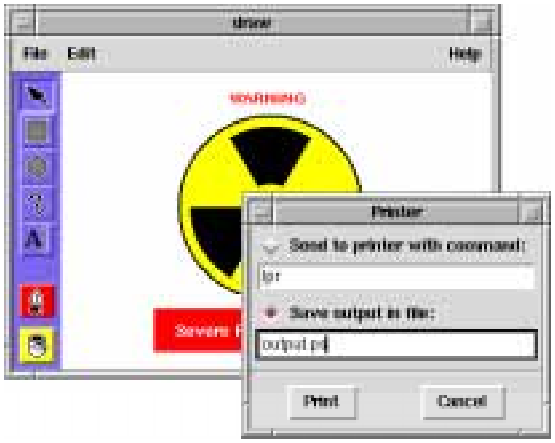
We can create a printer dialog like this:

```
printer_create .print
```

and add a *Print...* entry to the *File* menu to handle printing:

```
.mbar.file.m add command -label "Print..." -command {
    printer_print .print draw_print
}
```

When you select the *Print...* option, the printer dialog will appear, as shown in Figure 4.22. You can change the printer command to send output to a specific printer, or you can specify a file name for the output. When you press the *Print* button, the dialog uses the command argument `draw_print` to generate output, and then it routes the output accordingly.



○ Send to printer with command:
lpr
● Save output in file:
output.ps

Figure 4.22. The printer dialog routes PostScript output to a printer.

4.7.8 Saving a Drawing

Our drawing program should be able to save a drawing in a file, so you can load it again later to make changes. To accomplish this, we simply ask the canvas for a list of its items, and then examine each item and write a description of it out to a file.

There are many different ways to format the output file. Depending on what format you choose, your job of loading the file can be easy or hard. For example, suppose we use one line to represent each item on the canvas. The first thing on the line could be a number representing the item type—`1` for rectangle, `2` for oval, and so on. The next thing could be a list of coordinates. The rest of the line might depend on the type of the item. The description of a rectangle would have its outline color, its fill color, and its line width, while the description of a text item would have the text message, its fill color, and its anchor point. After we settle a few more details, we might end up with a drawing file that looks something like this:

```
1 {79.0 24.0 256.0 196.0} #da18d6efdd41 black 2
3 {78.0 212.0} {HAZARDOUS ATOMIC WASTE} #23b397c5291d w
2 {145.4 87.45 188.6 129.75} black #23b397c5291d 2
2 {152.6 92.85 171.5 112.65} "" green 2
```

There are a few problems with this format. For one thing, we need to write a lot of code to load this data back into our application. Handling the various item types and their specialized argument lists is rather tedious and error prone. Also, it's not obvious what the various fields in this format represent. Which is first, the outline color or the fill color? What exactly is item type `3`? What does the `w` represent at the end of that line? If you try to maintain this code over a period of several months, you may find yourself asking these same questions again and again.

Instead, let's write out the same information, but format it to look like a series of Tcl commands. For example:

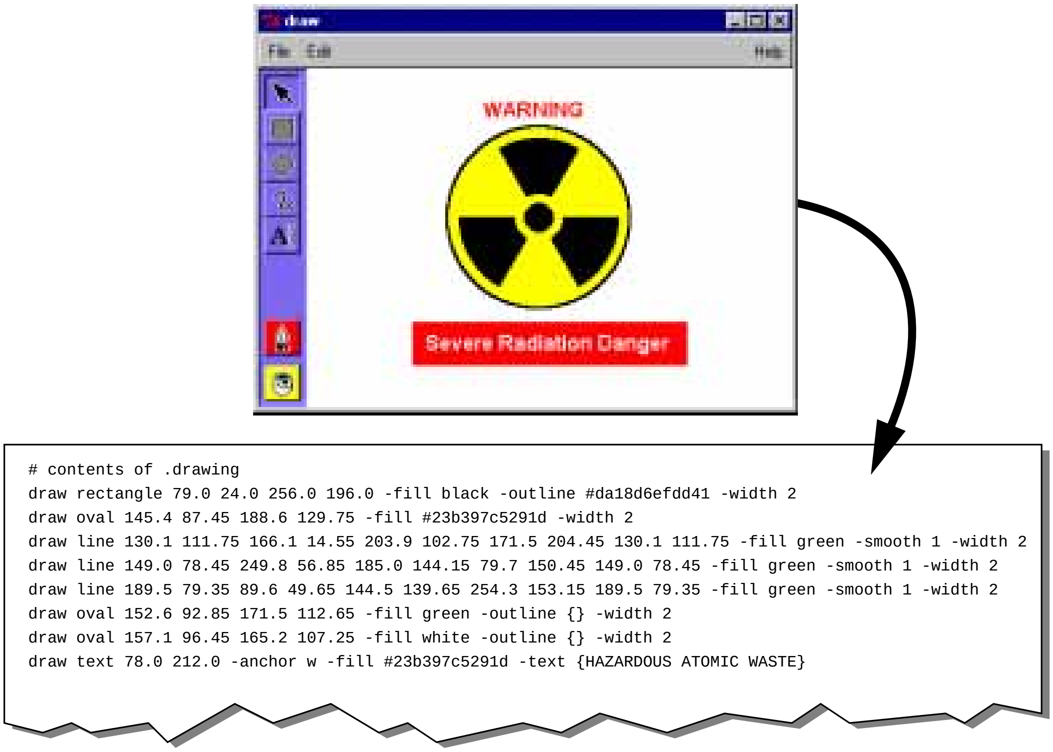
```
.drawing create rectangle 79.0 24.0 256.0 196.0 \
    -outline #da18d6efdd41 -fill black -width 2
.drawing create text 78.0 212.0 \
    -text {HAZARDOUS ATOMIC WASTE} -fill #23b397c5291d -anchor w
.drawing create oval 145.4 87.45 188.6 129.75 \
    -outline black -fill #23b397c5291d -width 2
.drawing create oval 152.6 92.85 171.5 112.65 \
    -outline {} -fill green -width 2
```

This format is quite expressive—it is obvious which is the outline color and which is the fill color. And this format is already well documented. If you forget what a parameter like `-anchor` means, you can always look it up in the manual page for the `canvas` command. But more importantly, this format is extremely easy to load. You can simply execute the drawing file using something like the `source` command, as we'll see in the next section.

We can improve this format a bit by removing the reference to a specific widget like `.drawing`. After all, if we ever decided to rename the `.drawing` widget, all of the drawing files would be useless. Let's replace `.drawing create` with a generic command called `draw`, as shown in Figure 4.23. This decouples the format from our current program, so we could use it for lots of different applications.

This example teaches an important lesson:

Tcl is not just a command language, it is also a data language.



```
# contents of .drawing
draw rectangle 79.0 24.0 256.0 196.0 -fill black -outline #da18d6efdd41 -width 2
draw oval 145.4 87.45 188.6 129.75 -fill #23b397c5291d -width 2
draw line 130.1 111.75 166.1 14.55 203.9 102.75 171.5 204.45 130.1 111.75 -fill green -smooth 1 -width 2
draw line 149.0 78.45 249.8 56.85 185.0 144.15 79.7 150.45 149.0 78.45 -fill green -smooth 1 -width 2
draw line 189.5 79.35 89.6 49.65 144.5 139.65 254.3 153.15 189.5 79.35 -fill green -smooth 1 -width 2
draw oval 152.6 92.85 171.5 112.65 -fill green -outline {} -width 2
draw oval 157.1 96.45 165.2 107.25 -fill white -outline {} -width 2
draw text 78.0 212.0 -anchor w -fill #23b397c5291d -text {HAZARDOUS ATOMIC WASTE}
```

Figure 4.23. The contents of a drawing canvas can be saved as a series of Tcl commands.

Whenever you find yourself inventing a text-based file format, think about saving the information as a series of Tcl commands. You can load the data simply by executing it in a Tcl interpreter where the commands are defined.[6]

Now, let's look at the code needed to save a canvas to a drawing file. First, we'll write a procedure called `canvas_save` that takes the name of a canvas, and returns a script full of `draw` commands. This procedure might be useful for many different canvas applications. It is implemented like this:

[6] Note that this same idea has revolutionized the output format for most printers. Today, almost all documents are formatted as a series of commands in the PostScript language. A PostScript printer simply treats each job as a program, and executes it to draw the pages in the document.

```
proc canvas_save {win} {
    set script "# contents of $win\n"
    foreach item [$win find all] {
        set tags [$win gettags $item]
        if {[lsearch $tags "canvas_save_ignore"] < 0} {
            set type   [$win type $item]
            set coords [$win coords $item]
            set opts ""
            foreach desc [$win itemconfigure $item] {
                set name [lindex $desc 0]
                set init [lindex $desc 3]
                set val  [lindex $desc 4]
                if {$val != $init} {
                    lappend opts $name $val
                }
            }
            append script "draw $type $coords $opts\n"
        }
    }
    return $script
}
```

We start by initializing the `script` variable to contain a comment line, describing the lines that follow. Notice that this line, and every other line that we add to the script, is terminated by "`\n`", the newline character.

We use the command `$win find all` to query a list of all items on the canvas. We build a description of each item, including its type, its coordinates, and its configuration options, and then append a `draw` command for it onto the script.

We query the information for each item directly from the canvas. The command `$win type $item` returns the item type—`rectangle`, `oval`, `text`, *etc*. The command `$win coords $item` returns the coordinates for each item. Unfortunately, there is no simple command that returns the configuration options for each item in the format that we need, so we loop through the options and build our own. The command `$win itemconfigure $item` returns a list of all configuration options. Each element in this list has five values: the option name, two null strings (which we ignore), the initial value, and the current value. We pick out the initial value and the current value, and then check to see if they're the same. If not, then we append the option name and its current value onto a list of option settings in the `opts` variable. When we're done, this variable will contain only the options that differ from their default value.

Notice that we skip any items tagged with the name `canvas_save_ignore`. This is a handy feature. Suppose you have just selected some items in the drawing, so the canvas contains a selection rectangle, as shown in Figure 4.21(b). This rectangle isn't really part of the drawing, so we don't want it to be included in the output from `canvas_save`. Before we call this procedure, we can simply tag transient items like this with the name `canvas_save_ignore`, and they won't be saved.

Now that we have the `canvas_save` procedure, we can use it to save a drawing in a file. We'll add a *Save As...* command to the *File* menu, like this:

```
.mbar.file.m add command -label "Save As..." -command draw_save
```

When you select the *Save As...* command, it calls the following procedure to save the drawing:

```
proc draw_save {} {
    global env
    set file [tk_getSaveFile]
    if {$file != ""} {
        .drawing addtag "canvas_save_ignore" withtag "marker"
        set selected [.drawing find withtag "selected"]
        .drawing dtag "selected"

        set cmd {
            set fid [open $file w]
            puts $fid [canvas_save .drawing]
            close $fid
        }
        if {[catch $cmd err] != 0} {
            notice_show "Cannot save drawing:\n$err" error
        }

        .drawing dtag "canvas_save_ignore"
        foreach item $selected {
            .drawing addtag "selected" withtag $item
        }
    }
}
```

This procedure uses `tk_getSaveFile` to get the name of the output file. This pops up a file selection dialog, letting you navigate the file system and select a file. If you press the *Cancel* button, then `tk_getSaveFile` returns a null string, and the `draw_save` procedure does nothing. Otherwise, `tk_getSaveFile` returns the name that you selected for the save file.

To save the drawing, we need to open the file, write out the script from `canvas_save`, and close the file. Any of these operations could fail, so we wrap them all up as a small script, and execute that script via the `catch` command. If we catch any error, we use the `notice_show` procedure developed in Section 6.4 to display the error message in a notice dialog.

Before we call `canvas_save`, we do two important things:

- We add the tag `canvas_save_ignore` to all of the items tagged with the name `marker`. This prevents the selection rectangle and other transient items from being included in the output.

- We remove the tag `selected` from any selected items. We do this by deleting the tag via the command `.drawing dtag "selected"`. If we didn't do this, the `selected` tag would appear in the drawing output. It would be listed in the `-tags` option for selected items, and this would lead to a subtle bug. The next time the drawing was loaded, these items would act as though they were selected—you could use the arrow keys to shift them about—but no selection rectangle would be visible.

After we call `canvas_save`, we put things back the way they were:

- We remove the tag `canvas_save_ignore` from the marker items by deleting the tag.
- We add the tag `selected` to all of the selected items.

4.7.9 Loading a Drawing

In the last section, we saved a drawing as a series of Tcl commands. We can load the drawing simply by executing it as a Tcl program. In our drawing program, for example, we might load a drawing by adding some code like this:

```
proc draw {args} {
    eval .drawing create $args
}
source $file
```

First, we define a `draw` procedure to handle all of the `draw` commands in the drawing file. Then we use the `source` command to execute a particular drawing file, whose name is stored in the `file` variable. As this file is executed, each of the `draw` commands adds an item to the canvas called `.drawing`.

The `draw` procedure uses the special `args` argument, so it will take any number of arguments. It simply passes these arguments on to the `.drawing create` command, which actually creates the item on the canvas. For example, suppose the drawing file contains a command like this:

```
draw rectangle 79.0 24.0 256.0 196.0 -fill red
```

When it is executed, this command will pass its arguments on to the canvas, like this:

```
.drawing create rectangle 79.0 24.0 256.0 196.0 -fill red
```

The `eval` command in the `draw` procedure is needed to make this work properly. It treats the elements within `$args` as separate words on the command line. Without the `eval` command, `$args` would be treated as a single word, like this:

```
.drawing create {rectangle 79.0 24.0 256.0 196.0 -fill red}
```

The canvas widget would think we're trying to create an item with a strange type name `"rectangle 79.0 24.0 256.0 196.0 -fill red"`, and it would complain that this is an invalid type. Adding the `eval` command avoids this error.

There is a problem with the way that we have loaded this drawing. Suppose some nefarious user gives you a drawing file that looks like this:

```
draw rectangle 79.0 24.0 256.0 196.0 -fill red
```

```
exec rm -rf *
```

When you source in this file, the first command will create a rectangle on the canvas, and the second will erase all of the files on your file system! Using Tcl as a data language is extremely powerful, but it is also extremely dangerous.

Fortunately, there is a simple solution for this problem. Whenever you want to execute commands from an untrusted source, you must do so in a protected context called a *safe interpreter.* We'll talk about safe interpreters and show you exactly how to use them when we look at client/server applications in Section 7.6.6.