Part IV:

Tk's C Interfaces

Chapter 36
Introduction

Like Tcl, Tk is a C library package that is linked with applications, and it provides a collection of library procedures that you can invoke from C code in the enclosing application. Although you can do many interesting things with Tk without writing any C code, just by writing Tcl scripts for `wish`, you'll probably find that most large GUI applications require some C code too. The most common reason for using Tk's C interfaces is to build new kinds of widgets. For example, if you write a Tk-based spreadsheet you'll probably need to implement a new widget to display the contents of the spreadsheet; if you write a charting package you'll probably build one or two new widgets to display charts and graphs in various forms; and so on. Some of these widgets could probably be implemented with existing Tk widgets such as canvases or texts, but for big jobs a new widget tailored to the needs of your application can probably do the job more simply and efficiently than any of Tk's general-purpose widgets. Typically you'll build one or two new widget classes to display your application's new objects, then combine your custom widgets with Tk's built-in widgets to create the full user interface of the application.

The main focus of this part of the book is on building new widgets. Most of Tk's library procedures exist for this purpose, and most of the text in this part of the book is oriented towards widget builders. However, you can also use Tk's library procedures to build new geometry managers; this is described in Chapter 43. Or, you may simply need to provide access to some window system feature that isn't supported by the existing Tcl commands, such as the ability to set the border width of a top-level window. In any event, the new features you implement should appear as Tcl commands so that you can use them in scripts. Both the philosophical issues and the library procedures discussed in Part III apply to this part of the book also.

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

36.1 What's in a widget?

All widget classes have the same basic structure, consisting of a widget record and six C procedures that implement the widget's look and feel. More complex widgets may have additional data structures and procedures besides theses, but all widgets have at least these basic components.

A *widget record* is the C data structure that represents the state of a widget. It includes all of the widget's configuration options plus anything else the widget needs for its own internal use. For example, the widget record for a label widget contains the label's text or bitmap, its background and foreground colors, its relief, and so on. Each instance of a widget has its own widget record, but all widgets of the same class have widget records with the same structure. One of the first things you will do when designing a new widget class is to design the widget record for that class.

Of the widget's six core procedures, two are Tcl command procedures. The first of these is called the *create procedure*; it implements the Tcl command that creates widgets of this class. The command's name is the same as the class name, and the command should have the standard syntax described in Section XXX for creating widgets. The command procedure initializes a new widget record, creates the window for the widget, and creates the widget command for the widget. It is described in more detail in Chapters 37 and 38.

The second command procedure is the *widget command procedure*; it implements the widget commands for all widgets of this class. When the widget command is invoked its `clientData` argument points to the widget record for a particular widget; this allows the same C procedure to implement the widget commands for many different widgets (the counter objects described in Section XXX used a similar approach).

The third core procedure for a widget class is its *configure procedure*. Given one or more options in string form, such as "`-background red`", it parses the options and fills in the widget record with corresponding internal representations such as an `XColor` structure. The configure procedure is invoked by the create procedure and the widget command procedure to handle configuration options specified on their command lines. Chapter 38 describes the facilities provided by Tk to make configure procedures easy to write.

The fourth core procedure is the *event procedure*. It is invoked by Tk's event dispatcher and typically handles exposures (part of the window needs to be redrawn), window size changes, focus changes, and the destruction of the window. The event procedure does not normally deal with user interactions such as mouse motions and key presses; these are usually handled with class bindings created with the `bind` command as described in Chapter XXX. Chapter 39 describes the Tk event dispatcher, including its facilities for managing X events plus additional features for timers, event-driven file I/O, and idle callbacks

The fifth core procedure is the *display procedure*. It is invoked to redraw part or all of the widget on the screen. Redisplays can be triggered by many things, including window exposures, changes in configuration options, and changes in the input focus. Chapter 40

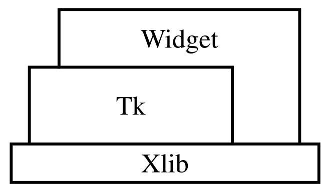


Figure 36.1. Tk hides many of the Xlib interfaces from widgets, but widgets still invoke Xlib directly for a few purposes such as drawing on the screen.

discusses several issues related to redisplay, such as deferred redisplay, double-buffering with pixmaps, and Tk's support for drawing 3-D effects.

The last of a widget's core procedures is its *destroy procedure*. This procedure is called when the widget is destroyed and is responsible for freeing up all of the resources allocated for the widget such as the memory for the widget record and X resources such as colors and pixmaps. Widget destruction is tricky because the widget could be in use at the time it is destroyed; Chapter 41 describes how deferred destruction is used to avoid potential problems.

36.2 Widgets are event-driven

Part II described how the Tcl scripts for Tk applications are event-driven, in that they consist mostly of short responses to user interactions and other events. The C code that implements widgets is also event-driven. Each of the core procedures described in the previous section responds to events of some sort. The create, widget command, and configure procedures all respond to Tcl commands. The event procedure responds to X events, and the display and destroy procedures respond to things that occur either in X or in Tcl scripts.

36.3 Tk vs. Xlib

Xlib is the C library package that provides the lowest level of access to the X Window System. Tk is implemented using Xlib but it hides most of the Xlib procedures from the C code in widgets, as shown in Figure 36.1. For example, Xlib provides a procedure `XCreateWindow` to create a new windows, but you should not use it; instead, call `Tk_CreateWindowFromPath` or one of the other procedures provided by Tk for this purpose. Tk's procedures call the Xlib procedures but also do additional things such as associating a textual name with the window. Similarly, you shouldn't normally call Xlib procedures like `XAllocColor` to allocate colors and other resources; call the corresponding Tk procedures like `Tk_GetColor` instead. In the case of colors, Tk calls Xlib to allocate the color, but it also remembers the colors that are allocated; if you use the same color in many different places, Tk will only communicate with the X server once.

However, Tk does not totally hide Xlib from you. When widgets redisplay themselves they make direct calls to Xlib procedures such as `XDrawLine` and `XDrawString`. Furthermore, many of the structures manipulated by Tk are the same as the structures provided by Xlib, such as graphics contexts and window attributes. Thus you'll need to know quite a bit about Xlib in order to write new widgets with Tk. This book assumes that you are familiar with the following concepts from Xlib:

- Window attributes such as `background_pixel`, which are stored in `XSetWindowAttributes` structures.
- Resources related to graphics, such as pixmaps, colors, graphics contexts, and fonts.
- Procedures for redisplaying, such as `XDrawLine` and `XDrawString`.
- Event types and the `XEvent` structure.

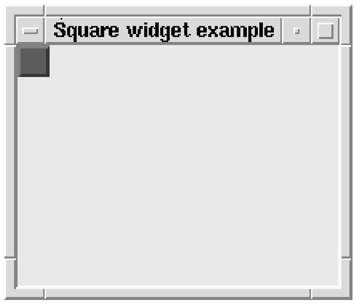
You'll probably find it useful to keep a book on Xlib nearby when reading this book and to refer to the Xlib documentation for specifics about the Xlib structures and procedures. If you haven't used Xlib before I'd suggest waiting to read about Xlib until you need the information. That way you can focus on just the information you need and avoid learning about the parts of Xlib that are hidden by Tk.

Besides Xlib, you shouldn't need to know anything about any other X toolkit or library. For example, Tk is completely independent from the Xt toolkit so you don't need to know anything about Xt. For that matter, if you're using Tk you *can't* use Xt: their widgets are incompatible and can't be mixed together.

36.4 Square: an example widget

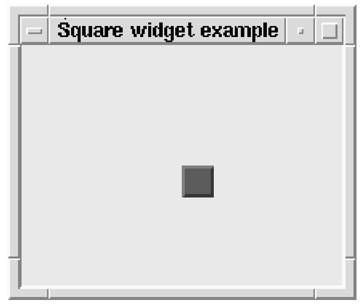
I'll use a simple widget called "square" for examples throughout Part IV. The square widget displays a colored square on a background as shown in Figure 36.2. The widget supports several configuration options, such as colors for the background and for the square, a relief for the widget, and a border width used for both the widget and the square. It also provides three widget commands: `configure`, which is used in the standard way to query and change options; `position`, which sets the position of the square's upper-left corner relative to the upper-left corner of the window, and `size`, which sets the square's size. Figure 36.2 illustrates the `position` and `size` commands.

Given these simple commands many other features can be written as Tcl scripts. For example, the following script arranges for the square to center itself over the mouse cursor on Button-1 presses and to track the mouse as long as Button-1 is held down. It assumes that the square widget is named "`.s`".



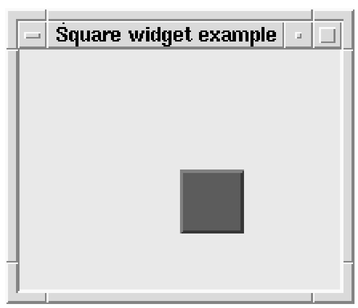

```
square .s
pack .s
wm title .s "Square widget example"
```

(a)




```
.s position 100 75
```

(b)




```
.s size 40
```

(c)

Figure 36.2. A sequence of scripts and the displays that they produce. Figure (a) creates a square widget, Figure (b) invokes the `position` widget command to move the square within its widget, and Figure (c) changes the size of the square.

```
proc center {x y} {
    set a [.s size]
    .s position [expr $x-($a/2)] [expr $y-($a/2)]
}
bind .s <1> {center %x %y}
bind .s <B1-Motion> {center %x %y}
```

Note: *For this particular widget it would probably make more sense to use configuration options instead of the* `position` *and* `size` *commands; I made them widget commands just to illustrate how to write widget commands.*

The implementation of the square widget requires about 320 lines of C code excluding comments, or about 750 lines in a copiously-commented version. The square widget doesn't use all of the features of Tk but it illustrates the basic things you must do to create a new widget. For examples of more complex widgets you can look at the source code for some of Tk's widgets; they have the same basic structure as the square widget and they use the same library procedures that you'll read about in the chapters that follow.

36.5 Design for re-usability

When building a new widget, try to make it as flexible and general-purpose as possible. If you do this then it may be possible for you or someone else to use your widget in new ways that you didn't foresee when you created it. Here are a few specific things to think about:

1. Store all the information about the widget in its widget record. If you use static or global variables to hold widget state then it may not be possible to have more than one instance of the widget in any given application. Even if you don't envision using more than one instance per application, don't do anything to rule this out.
2. Make sure that all of the primitive operations on your widget are available through its widget command. Don't hard-wire the widget's behavior in C. Instead, define the behavior as a set of class bindings using the `bind` command. This will make it easy to change the widget's behavior.
3. Provide escapes to Tcl. Think about interesting ways that you can embed Tcl commands in your widget and invoke them in response to various events. For example, the actions for button widgets and menu items are stored as a Tcl commands that are evaluated when the widgets are invoked, and canvases and texts allow you to associate Tcl commands with their internal objects in order to give them behaviors.
4. Organize the code for your widget in one or a few files that can easily be linked into other applications besides the one you're writing.

Chapter 37
Creating Windows

This chapter presents Tk's basic library procedures for creating windows. It describes the `Tk_Window` type, which is used as a token for windows, then introduces the Tk procedures for creating and deleting windows. Tk provides several macros for retrieving information about windows, which are introduced next. Then the chapter discusses what should be in the create procedure for a widget, using the square widget as an example. The chapter closes with a discussion of delayed window creation. See Table 37.1 for a summary of the procedures discussed in the chapter.

37.1 Tk_Window structures

Tk uses a token of type `Tk_Window` to represent each window. When you create a new window Tk returns a `Tk_Window` token, and you must pass this token back to Tk when invoking procedures to manipulate the window. A `Tk_Window` is actually a pointer to a record containing information about the window, such as its name and current size, but Tk hides the contents of this structure and you may not read or write its fields directly. The only way you can manipulate a `Tk_Window` is to invoke procedures and macros provided by Tk.

37.2 Creating Tk_Windows

Tk applications typically use two procedures for creating windows: `Tk_CreateMainWindow` and `Tk_CreateWindowFromPath`. `Tk_CreateMainWindow` creates a

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

```
Tk_Window Tk_CreateMainWindow(Tcl_Interp *interp,
    char *screenName, char *appName)
```
Creates a new application and returns a token for the application's main window. `ScreenName` gives the screen on which to create the main window (if `NULL` then Tk picks default), and `appName` gives a base name for the application. If an error occurs, returns `NULL` and stores an error message in `interp->result`.

```
Tk_Window Tk_CreateWindowFromPath(Tcl_Interp *interp,
    Tk_Window tkwin, char *pathName, char *screenName)
```
Creates a new window in `tkwin`'s application whose path name is `pathName`. If `screenName` is `NULL` the new window will be an internal window; otherwise it will be a top-level window on `screenName`. Returns a token for the new window. If an error occurs, returns `NULL` and stores an error message in `interp->result`.

```
Tk_SetClass(Tk_Window tkwin, char *class)
```
Sets `tkwin`'s class to `class`.

```
Tk_DestroyWindow(TkWindow tkwin)
```
Destroy `tkwin` and all of its descendants in the window hierarchy.

```
Tk_Window Tk_NameToWindow(Tcl_Interp *interp, char *pathName,
    Tk_Window tkwin)
```
Returns the token for the window whose path name is `pathName` in the same application as `tkwin`. If no such name exists then returns `NULL` and stores an error message in `interp->result`.

```
Tk_MakeWindowExist(TkWindow tkwin)
```
Force the creation of the X window for `tkwin`, if it didn't already exist.

Table 37.1. A summary of basic procedures for window creation and deletion.

new application; it's usually invoked in the main program of an application. Before invoking `Tk_CreateMainWindow` you should create a Tcl interpreter to use for the application. `Tk_CreateMainWindow` takes three arguments, consisting of the interpreter plus two strings:

```
Tk_Window Tk_CreateMainWindow(Tcl_Interp *interp,
    char *screenName, char *appName)
```

The `screenName` argument gives the name of the screen on which to create the main window. It can have any form acceptable to your X server. For example, on most UNIX-like systems "`unix:0`" selects the default screen of display 0 on the local machine, or "`ginger.cs.berkeley.edu:0.0`" selects screen 0 of display 0 on the machine whose network address is "`ginger.cs.berkeley.edu`". `ScreenName` may be specified as `NULL`, in which case Tk picks a default server. On UNIX-like systems the default server is normally determined by the `DISPLAY` environment variable.

DRAFT (7/10/93): Distribution Restricted

The last argument to `Tk_CreateMainWindow` is a name to use for the application, such as "`clock`" for a clock program or "`mx foo.c`" for an editor named `mx` editing a file named `foo.c`. This is the name that other applications will use to send commands to the new application. Each application must have a unique name; if `appName` is already in use by some other application then Tk adds a suffix like " `#2`" to make the name unique. Thus the actual name of the application may be something like "`clock #3`" or "`mx foo.c #4`". You can find out the actual name for the application using the `Tk_Name` macro or by invoking the Tcl command "`winfo name .`".

`Tk_CreateMainWindow` creates the application's main window, registers its name so that other applications can send commands to it, and adds all of Tk's commands to the interpreter. It returns the `Tk_Window` token for the main window. If an error occurs (e.g. `screenName` doesn't exist or the X server refused to accept a connection) then `Tk_CreateMainWindow` returns `NULL` and leaves an error message in `interp->result`.

`Tk_CreateWindowFromPath` adds a new window to an existing application. It's the procedure that's usually called when creating new widgets and it has the following prototype:

```
Tk_Window Tk_CreateWindowFromPath(Tcl_Interp *interp,
        Tk_Window tkwin, char *pathName, char *screenName);
```

The `tkwin` argument is a token for an existing window; its only purpose is to identify the application in which to create the new window. `PathName` gives the full name for the new window, such as "`.a.b.c`". There must not already exist a window by this name, but its parent (for example, "`.a.b`") must exist. If `screenName` is `NULL` then the new window is an internal window; otherwise the new window will be a top-level window on the indicated screen. `Tk_CreateWindowFromPath` returns a token for the new window unless an error occurs, in which case it returns `NULL` and leaves an error message in `interp->result`.

Tk also provides a third window-creation procedure called `Tk_CreateWindow`. This procedure is similar to `Tk_CreateWindowFromPath` except that the new window's name is specified a bit differently. See the reference documentation for details.

37.3 Setting a window's class

The procedure `Tk_SetClass` assigns a particular class name to a window. For example,

```
Tk_SetClass(tkwin, "Foo");
```

sets the class of window `tkwin` to "`Foo`". Class names are used by Tk for several purposes such as finding options in the option database and event bindings. You can use any string whatsoever as a class name when you invoke `Tk_SetClass`, but you should make sure the first letter is capitalized: Tk assumes in several places that uncapitalized names are window names and capitalized names are classes.

37.4 Deleting windows

The procedure Tk_DestroyWindow takes a Tk_Window as argument and deletes the window. It also deletes all of the window's children recursively. Deleting the main window of an application will delete all of the windows in the application and usually causes the application to exit.

37.5 Basic operations on Tk_Windows

Given a textual path name for a window, Tk_NameToWindow may be used to find the Tk_Window token for the window:

```
Tk_Window Tk_NameToWindow(Tcl_Interp *interp, char *pathName,
        Tk_Window tkwin);
```

PathName is the name of the desired window, such as ".a.b.c", and tkwin is a token for any window in the application of interest (it isn't used except to select a specific application). Normally Tk_NameToWindow returns a token for the given window, but if no such window exists it returns NULL and leaves an error message in interp->result.

Tk maintains several pieces of information about each Tk_Window and it provides a set of macros that you can use to access the information. See Table 37.2 for a summary of all the macros. Each macro takes a Tk_Window as an argument and returns the corresponding piece of information for the window. For example if tkwin is a Tk_Window then

```
Tk_Width(tkwin)
```

returns an integer value giving the current width of tkwin in pixels. Here are a few of the more commonly used macros:

- Tk_Width and Tk_Height return the window's dimensions; this information is used during redisplay for purposes such as centering text.
- Tk_WindowId returns the X identifier for the window; it is needed when invoking Xlib procedures during redisplay.
- Tk_Display returns a pointer to Xlib's Display structure corresponding to the window; it is also needed when invoking Xlib procedures.

Some of the macros, like Tk_InternalBorderWidth and Tk_ReqWidth, are only used by geometry managers (see Chapter 43) and others such as Tk_Visual are rarely used by anyone.

Macro Name	Result Type	Meaning
Tk_Attributes	XSetWindowAttributes *	Window attributes such as border pixel and cursor.
Tk_Changes	XWindowChanges *	Window position, size, stacking order.
Tk_Class	Tk_Uid	Name of window's class.
Tk_Colormap	Colormap	Colormap for window.
Tk_Depth	int	Bits per pixel.
Tk_Display	Display	X display for window.
Tk_Height	int	Current height of window in pixels.
Tk_InternalBorderWidth	int	Width of internal border in pixels.
Tk_IsMapped	int	1 if window mapped, 0 otherwise.
Tk_IsTopLevel	int	1 if top-level, 0 if internal.
Tk_Name	Tk_Uid	Name within parent. For main window, returns application name.
Tk_Parent	Tk_Window	Parent, or NULL for main window.
Tk_PathName	char *	Full path name of window.
Tk_ReqWidth	int	Requested width in pixels.
Tk_ReqHeight	int	Requested height in pixels.
Tk_Screen	Screen *	X Screen for window.
Tk_ScreenNumber	int	Index of window's screen.
Tk_Visual	Visual *	Information about window's visual characteristics.
Tk_Width	int	Current width of window in pixels.
Tk_WindowId	Window	X identifier for window.
Tk_X	int	X-coordinate within parent window.
Tk_Y	int	Y-coordinate within parent window.

Table 37.2. Macros defined by Tk for retrieving window state. Each macro takes a Tk_Window as argument and returns a result whose type is given in the second column. All of these macros are fast (they simply return fields from Tk's internal structures and don't require any interactions with the X server).

37.6 Create procedures

The create procedure for a widget must do five things: create a new Tk_Window; create and initialize a widget record; set up event handlers; create a widget command for the widget; and process configuration options for the widget. The create procedure should be the command procedure for a Tcl command named after the widget's class, and its client-

Data argument should be the Tk_Window token for the main window of the application (this is needed in order to create a new Tk_Window in the application).

Figure 37.1 shows the code for SquareCmd, which is the create procedure for square widgets. After checking its argument count, SquareCmd creates a new window for the widget and invokes Tk_SetClass to assign it a class of "Square". The middle part of SquareCmd allocates a widget record for the new widget and initializes it. The widget record for squares has the following definition:

```
typedef struct {
    Tk_Window tkwin;
    Display *display;
    Tcl_Interp *interp;
    int x, y;
    int size;
    int borderWidth;
    Tk_3DBorder bgBorder;
    Tk_3DBorder fgBorder;
    int relief;
    GC gc;
    int updatePending;
} Square;
```

The first field of the record is the Tk_Window for the widget. The next field, display, identifies the X display for the widget (it's needed during cleanup after the widget is deleted). Interp holds a pointer to the interpreter for the application. The x and y fields give the position of the upper-left corner of the square relative to the upper-left corner of the window, and the size field specifies the square's size in pixels. The last six fields are used for displaying the widget; they'll be discussed in Chapters 38 and 40.

After initializing the new widget record SquareCmd calls Tk_CreateEventHandler; this arranges for SquareEventProc to be called whenever the widget needs to be redrawn or when various other events occur, such as deleting its window or changing its size; events will be discussed in more detail in Chapter 39. Next SquareCmd calls Tcl_CreateCommand to create the widget command for the widget. The widget's name is the name of the command, SquareWidgetCmd is the command procedure, and a pointer to the widget record is the clientData for the command (using a pointer to the widget record as clientData allows a single C procedure to implement the widget commands for all square widgets; SquareWidgetCommand will receive a different clientData argument depending on which widget command was invoked). Then SquareCmd calls ConfigureSquare to process any configuration options specified as arguments to the command; Chapter 38 describes how the configuration options are handled. If an error occurs in processing the configuration options then SquareCmd destroys the window and returns an error. Otherwise it returns success with the widget's path name as result.

```
int SquareCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    Tk_Window main = (Tk_Window) clientData;
    Square *squarePtr;
    Tk_Window tkwin;

    if (argc < 2) {
        Tcl_AppendResult(interp, "wrong # args: should be \"",
                argv[0], " pathName ?options?\"", (char *) NULL);
        return TCL_ERROR;
    }

    tkwin = Tk_CreateWindowFromPath(interp, main, argv[1],
            (char *) NULL);
    if (tkwin == NULL) {
        return TCL_ERROR;
    }
    Tk_SetClass(tkwin, "Square");

    squarePtr = (Square *) malloc(sizeof(Square));
    squarePtr->tkwin = tkwin;
    squarePtr->display = Tk_Display(tkwin);
    squarePtr->interp = interp;
    squarePtr->x = 0;
    squarePtr->y = 0;
    squarePtr->size = 20;
    squarePtr->bgBorder = NULL;
    squarePtr->fgBorder = NULL;
    squarePtr->gc = None;
    squarePtr->updatePending = 0;

    Tk_CreateEventHandler(tkwin,
            ExposureMask|StructureNotifyMask, SquareEventProc,
            (ClientData) squarePtr);
    Tcl_CreateCommand(interp, Tk_PathName(tkwin),
        SquareWidgetCmd, (ClientData squarePtr),
        (Tcl_CmdDeleteProc *) NULL);
    if (ConfigureSquare(interp, squarePtr, argc-2, argv+2, 0)
            != TCL_OK) {
        Tk_DestroyWindow(squarePtr->tkwin);
        return TCL_ERROR;
    }
    interp->result = Tk_PathName(tkwin);
    return TCL_OK;
}
```

Figure 37.1. The create procedure for square widgets. This procedure is the command procedure for the `square` command.

37.7 Delayed window creation

Tk_CreateMainWindow and Tk_CreateWindowFromPath create the Tk data structures for a window, but they do not communicate with the X server to create an actual X window. If you create a Tk_Window and immediately fetch its X window identifier using Tk_WindowId, the result will be None. Tk doesn't normally create the X window for a Tk_Window until the window is mapped, which is normally done by a geometry manager (see Chapter 43). The reason for delaying window creation is performance. When a Tk_Window is initially created, all of its attributes are set to default values. Many of these attributes will be modified almost immediately when the widget configures itself. It's more efficient to delay the window's creation until all of its attributes have been set, rather than first creating the window and then asking the X server to modify the attributes later.

Delayed window creation is normally invisible to widgets, since the only time a widget needs to know the X identifier for a window is when it invokes Xlib procedures to display it. This doesn't happen until after the window has been mapped, so the X window will have been created by then. If for some reason you should need the X window identifier before a Tk_Window has been mapped, you can invoke Tk_MakeWindowExist:

```
void Tk_MakeWindowExist(tkwin);
```

This forces the X window for tkwin to be created immediately if it hasn't been created yet. Once Tk_MakeWindowExist returns, Tk_WindowId can be used to retrieve the Window token for it.

Chapter 38
Configuring Widgets

The phrase "configuring a widget" refers to all of the setup that must be done prior to actually drawing the widget's contents on the screen. A widget is configured initially as part of creating it, and it may be reconfigured by invoking its widget command. One of the largest components of configuring a widget is processing configuration options such as "`-borderwidth 1m`". For each option the textual value must be translated to an internal form suitable for use in the widget. For example, distances specified in floating-point millimeters must be translated to integer pixel values and font names must be mapped to corresponding `XFontStruct` structures. Configuring a widget also includes other tasks such as preparing X graphics contexts to use when drawing the widget and setting attributes of the widget's window, such as its background color.

This chapter describes the Tk library procedures for configuring widgets, and it presents the square widget's configure procedure and widget command procedure. Chapter 40 will show how to draw a widget once configuration is complete.

38.1 Tk_ConfigureWidget

Tk provides three library procedures, `Tk_ConfigureWidget`, `Tk_ConfigureInfo`, and `Tk_FreeOptions`, that do most of the work of processing configuration options (see Table 38.1). To use these procedures you first create a *configuration table* that describes all of the configuration options supported by your new widget class. When creating a new widget, you pass this table to `Tk_ConfigureWidget` along with `argc`/`argv` information describing the configuration options (i.e. all the arguments in the creation command after the widget name). You also pass in a pointer to the widget record for

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`int Tk_ConfigureWidget(Tcl_Interp *interp, Tk_Window tkwin, Tk_ConfigSpec *specs, int argc, char *argv[], char *widgRec, int flags)`	Processes a set of arguments from a Tcl command (`argc` and `argv`) using a table of allowable configuration options (`specs`) and sets the appropriate fiels of a widget record (`widgRec`). `Tkwin` is the widget's window. Normally returns `TCL_OK`; if an error occurs, returns `TCL_ERROR` and leaves an error message in `interp->result`. Flags is normally 0 or `TK_CONFIG_ARGV_ONLY` (see reference documentation for other possibilities).
`int Tk_ConfigureInfo(Tcl_Interp *interp, Tk_Window tkwin, Tk_ConfigSpec *specs, char *widgRec, char * argvName, flags)`	Finds the configuration option in `specs` whose command-line name is `argvName`, locates the value of that option in `widgRec`, and generates in `interp->result` a list describing that configuration option. If `argvName` is `NULL`, generates a list of lists describing all of the options in `specs`. Normally returns `TCL_OK`; if an error occurs, returns `TCL_ERROR` and leaves an error message in `interp->result`. `Flags` is normally 0 (see the reference documentation for other possibilities).
`Tk_FreeOptions(Tk_ConfigSpec *specs, char *widgRec, Display *display, int flags)`	Frees up any resources in `widgRec` that are used by `specs`. `Display` must be the widget's display. `Flags` is normally 0 but can be used to select particular entries in `specs` (see reference documentation for details).
`int Tk_Offset(type, field)`	This is a macro that returns the offset of a field named `field` within a structure whose type is `type`. Used when creating configuration tables.

Table 38.1. A summary of `Tk_ConfigureWidget` and related procedures and macros.

the widget. `Tk_ConfigureWidget` processes each option specified in `argv` according to the information in the configuration table, converting string values to appropriate internal forms, allocating resources such as fonts and colors if necessary, and storing the results into the widget record. For options that aren't explicitly specified in `argv`, `Tk_ConfigureWidget` checks the option database to see if a value is specified there. For options that still haven't been set, `Tk_ConfigureWidget` uses default values specified in the table.

When the `configure` widget command is invoked to change options, you call `Tk_ConfigureWidget` again with the `argc`/`argv` information describing the new option values. `Tk_ConfigureWidget` will process the arguments according to the table and modify the information in the widget record accordingly. When the `configure` widget command is invoked to read out the current settings of options, you call `Tk_ConfigureInfo`. It generates a Tcl result describing one or all of the widget's options in exactly the right form, so all you have to do is return this result from the widget command procedure.

Finally, when a widget is deleted you invoke `Tcl_FreeOptions`. `Tcl_FreeOptions` scans through the table to find options for which resources have been allocated, such as fonts and colors. For each such option it uses the information in the widget record to free up the resource.

38.1.1 Tk_ConfigSpec tables

Most of the work in processing options is in creating the configuration table. The table is an array of records, each with the following structure:

```
typedef struct {
    int type;
    char *argvName;
    char *dbName;
    char *dbClass;
    char *defValue;
    int offset;
    int specFlags;
    Tk_CustomOption *customPtr;
} Tk_ConfigSpec;
```

The `type` field specifies the internal form into which the option's string value should be converted. For example, `TK_CONFIG_INT` means the option's value should be converted to an integer and `TK_CONFIG_COLOR` means that the option's value should be converted to a pointer to an `XColor` structure. For `TK_CONFIG_INT` the option's value must have the syntax of a decimal, hexadecimal, or octal integer and for `TK_CONFIG_COLOR` the option's value must have one of the forms for colors described in Section XXX. For `TK_CONFIG_COLOR` Tk will allocate an `XColor` structure, which must later be freed (e.g. by calling `Tk_FreeOptions`). More than 20 different option types are defined by Tk; see the reference documentation for details on each of the supported types.

`ArgvName` is the option's name as specified on command lines, e.g. "`-background`" or "`-font`". The `dbName` and `dbClass` fields give the option's name and class in the option database. The `defValue` field gives a default value to use for the option if it isn't specified on the command line and there isn't a value for it in the option database; `NULL` means there is no default for the option.

The `offset` field tells where in the widget record to store the converted value of the option. It is specified as a byte displacement from the beginning of the record. You should use the `Tk_Offset` macro to generate values for this field. For example,

```
Tk_Offset(Square, relief)
```

produces an appropriate offset for the `relief` field of a record whose type is `Square`.

The `specFlags` field contains an OR-ed combination of flag bits that provide additional control over the handling of the option. A few of the flags will be discussed below; see the reference documentation for a complete listing. Finally, the `customPtr` field provides additional information for application-defined options. It's only used when the type is TK_CONFIG_CUSTOM and should be NULL in other cases. See the reference documentation for details on defining custom option types.

Here is the option table for square widgets:

```
Tk_ConfigSpec configSpecs[] = {
     {TK_CONFIG_BORDER, "-background", "background",
"Background",
         "#cdb79e", Tk_Offset(Square, bgBorder),
         TK_CONFIG_COLOR_ONLY, (Tk_CustomOption *) NULL},
     {TK_CONFIG_BORDER, "-background", "background",
         "Background", "white", Tk_Offset(Square, bgBorder),
         TK_CONFIG_MONO_ONLY, (Tk_CustomOption *) NULL},
     {TK_CONFIG_SYNONYM, "-bd", "borderWidth", (char *) NULL,
         (char *) NULL, 0, 0, (Tk_CustomOption *) NULL},
     {TK_CONFIG_SYNONYM, "-bg", "background", (char *) NULL,
         (char *) NULL, 0, 0, (Tk_CustomOption *) NULL},
     {TK_CONFIG_PIXELS, "-borderwidth", "borderWidth",
         "BorderWidth", "1m", Tk_Offset(Square, borderWidth),
         0, (Tk_CustomOption *) NULL},
     TK_CONFIG_SYNONYM, "-fg", "foreground", (char *) NULL,
         (char *) NULL, 0, 0, (Tk_CustomOption *) NULL},
     {TK_CONFIG_BORDER, "-foreground", "foreground",
         "Foreground", "#b03060", Tk_Offset(Square, fgBorder),
         TK_CONFIG_COLOR_ONLY, (Tk_CustomOption *) NULL},
     {TK_CONFIG_BORDER, "-foreground", "foreground",
         "Foreground", "black", Tk_Offset(Square, fgBorder),
         TK_CONFIG_MONO_ONLY, (Tk_CustomOption *) NULL},
     {TK_CONFIG_RELIEF, "-relief", "relief", "Relief",
         "raised", Tk_Offset(Square, relief), 0,
         (Tk_CustomOption *) NULL},
     {TK_CONFIG_END, (char *) NULL, (char *) NULL, ,
         (char *) NULL, (char *) NULL, 0, 0,
         (Tk_CustomOption *) NULL}
};
```

This table illustrates three additional features of Tk_ConfigSpecs structures. First, there are two entries each for the -background and -foreground options. The first entry for each option has the TK_CONFIG_COLOR_ONLY flag set, which causes Tk to use that option if the display is a color display and to ignore it if the display is monochrome. The second entry specifies the TK_CONFIG_MONO_ONLY flag so it is only used for monochrome displays. This feature allows different default values to be specified for color and mono displays (the current color model for the window determines whether the it considered to be color or monochrome; see Section XXX). Second, the options -bd, -bg, and -fg have type TK_CONFIG_SYNONYM. This means that each of these options is a synonym for some other option; the dbName field identifies the other option and the other fields are ignored. For example, if the -bd option is specified with the above table, Tk will actually use the table entry for the -borderwidth option. Third, the last entry in the table must have type TK_CONFIG_END; Tk depends on this to locate the end of the table.

38.1.2 Invoking Tk_ConfigureWidget

Suppose that Tk_ConfigureWidget is invoked as follows:

```
Tcl_Interp *interp;
Tk_Window tkwin;
char *argv[] = {"-relief", "sunken", "-bg", "blue"};
Square *squarePtr;
int code;
...
code = Tk_ConfigureWidget(interp, tkwin, configSpecs,
        4, argv, (char *) squarePtr, 0);
```

A call much like this will occur if a square widget is created with the Tcl command

```
square .s -relief sunken -bg blue
```

The -relief option will be processed according to type TK_CONFIG_RELIEF, which dictates that the option's value must be a valid relief, such as "raised" or "sunken". In this case the value specified is sunken; Tk_ConfigureWidget converts this string value to the integer value TK_RELIEF_SUNKEN and stores that value in squarePtr->relief. The -bg option will be processed according to the configSpecs entry for -background, which has type TK_CONFIG_BORDER. This type requires that the option's value be a valid color name; Tk creates a data structure suitable for drawing graphics in that color in tkwin, and it computes additional colors for drawing light and dark shadows to produce 3-dimensional effects. All of this information is stored in the new structure and a token for that structure is stored in the bgBorder field of squarePtr. In Chapter 40 you'll see how this token is used to draw the widget.

Since the -borderwidth and -foreground options weren't specified in argv, Tk_ConfigureWidget looks them up in the option database using the information for those options in configSpecs. If it finds values in the option database then it will use them in the same way as if they had been supplied in argv.

If an option isn't specified in the option database then Tk_ConfigureWidget uses the default value specified in its table entry. For example, for -borderwidth it will use the default value "1m". Since the option has type TK_CONFIG_PIXELS, this string must specify a screen distance in one of the forms described in Section XXX. "1m" specifies a distance of one millimeter; Tk converts this to the corresponding number of pixels and stores the result as an integer in squarePtr->borderWidth. If the default value for an option is NULL then Tk_ConfigureWidget does nothing at all if there is no value in either argv or the option database; the value in the widget record will retain whatever value it had when Tk_ConfigureWidget is invoked.

Note: If an entry in the configuration table has no default value then you must initialize the corresponding field of the widget record before invoking Tk_ConfigureWidget. *If* *there is a default value then you need not initialize the field in the widget record since* `Tk_ConfigureWidget` *will always store a proper value there.*

38.1.3 Errors

`Tk_ConfigureWidget` normally returns `TCL_OK`. If an error occurs then it returns `TCL_ERROR` and leaves an error message in `interp->result`. The most common form of error is a value that doesn't make sense for the option type, such as "`abc`" for the `-bd` option. `Tk_ConfigureWidget` returns as soon as it encounters an error, which means that some of the fields of the widget record may not have been set yet; these fields will be left in an initialized state (such as `NULL` for pointers, `0` for integers, `None` for X resources, etc.).

38.1.4 Reconfiguring

`Tk_ConfigureWidget` gets invoked not only when a widget is created but also during the `configure` widget command. When reconfiguring you probably won't want to consider the option database or default values. You'll want to process only the options that are specified explicitly in `argv`, leaving all the unspecified options with their previous values. To accomplish this, specify `TK_CONFIG_ARGV_ONLY` as the last argument to `Tk_ConfigureWidget`:

```
code = Tk_ConfigureWidget(interp, tkwin, configSpecs,
        argc, argv, (char *) squarePtr,
        TK_CONFIG_ARGV_ONLY);
```

38.1.5 Tk_ConfigureInfo

If a `configure` widget command is invoked with a single argument, or with no arguments, then it returns configuration information. For example, if `.s` is a square widget then

```
.s configure -background
```

should return a list of information about the `-background` option and

```
.s configure
```

should return a list of lists describing all the options, as described in Section XXX. `Tk_ConfigureInfo` does all the work of generating this information in the proper format. For the square widget it might be invoked as follows:

```
code = Tk_ConfigureInfo(interp, tkwin, configSpecs,
        (char *) squarePtr, argv[2], 0);
```

`Argv[2]` specifies the name of a particular option (e.g. `-background` in the first example above). If information is to be returned about all options, as in the second example above, then `NULL` should be specified as the option name. `Tk_ConfigureInfo` sets `interp->result` to hold the proper value and returns `TCL_OK`. If an error occurs (because a bad option name was specified, for example) then Tk_ConfigureInfo stores an error message in interp->result and returns TCL_ERROR. In either case, the widget command procedure can leave interp->result as it is and return code as its completion code.

38.1.6 Tk_FreeOptions

The library procedure Tk_FreeOptions is usually invoked after a widget is deleted in order to clean up its widget record. For some option types, such as TK_CONFIG_BORDER, Tk_ConfigureWidget allocates resources which must eventually be freed. Tk_FreeOptions takes care of this:

```
void Tk_FreeOptions(Tk_ConfigSpec *specs, char *widgRec,
        Display *display, int flags);
```

Specs and widgRec should be the same as in calls to Tk_ConfigureWidget. Display identifies the X display containing the widget (it's needed for freeing certain options) and flags should normally be 0 (see the reference documentation for other possibilities). Tk_FreeOptions will scan specs looking for entries such as TK_CONFIG_BORDER whose resources must be freed. For each such entry it checks the widget record to be sure a resource is actually allocated (for example, if the value of a string resource is NULL it means that no memory is allocated). If there is a resource allocated then Tk_FreeOptions passes the value from the widget record to an appropriate procedure to free up the resource and resets the value in the widget record to a state such as NULL to indicate that it has been freed.

38.1.7 Other uses for configuration tables

Configuration tables can be used for other things besides widgets. They are suitable for any situation where textual information must be converted to an internal form and stored in fields of a structure, particularly if the information is specified in the same form as for widget options, e.g.

```
-background blue -width 1m
```

Tk uses configuration tables internally for configuring menu entries, for configuring canvas items, and for configuring display attributes of tags in text widgets.

38.2 Resource caches

The X window system provides a number of different resources for applications to use. Windows are one example of a resource; other examples are graphics contexts, fonts, pixmaps, colors, and cursors. An application must allocate resources before using them and free them when they're no longer needed. X was designed to make resource allocation and deallocation as cheap as possible, but it is still expensive in many situations because it requires communication with the X server (for example, font allocation requires communication with the server to make sure the font exists). If an application uses the same resource in several different places (e.g. the same font in many different windows) it is wasteful to allocate separate resources for each use: this wastes time communicating with the server and it wastes space in the X server to keep track of the copies of the resource.

Tk provides a collection of *resource caches* in order to reduce the costs of resource management. When your application needs a particular resource you shouldn't call Xlib to allocate it; call the corresponding Tk procedure instead. Tk keeps track of all the resources used by the application and allows them to be shared. If you use the same font in many different widgets, Tk will call X to allocate a font for the first widget, but it will re-use this font for all the other widgets. When the resource is no longer needed anywhere in the application (e.g. all the widgets using the font have been destroyed) then Tk will invoke the Xlib procedure to free up the resource. This approach saves time as well as memory in the X server.

If you allocate a resource through Tk you must treat it as read-only since it may be shared. For example, if you allocate a graphics context with `Tk_GetGC` you must not change the background color of the graphics context, since this would affect the other uses of the graphics context. If you need to modify a resource after creating it then you should not use Tk's resource caches; call Xlib directly to allocate the resource so that you can have a private copy.

Most of the resources for a widget are allocated automatically by `Tk_ConfigureWidget`, and `Tk_ConfigureWidget` uses the Tk resource caches. The following subsections describe how to use the Tk resource caches directly, without going through `Tk_ConfigureWidget`.

38.2.1 Graphics contexts

Graphics contexts are the resource that you are most likely to allocate directly. They are needed whenever you draw information on the screen and `Tk_ConfigureWidget` does not provide facilities for allocating them. Thus most widgets will need to allocate a few graphics contexts in their configure procedures. The procedure `Tk_GetGC` allocates a graphics context and is similar to the Xlib procedure `XCreateGC`:

```
GC Tk_GetGC(Tk_Window tkwin, unsigned long valueMask,
        XGCValues *valuePtr)
```

The `tkwin` argument specifies the window in which the graphics context will be used. `ValueMask` and `ValuePtr` specify the fields of the graphics context. `ValueMask` is an OR-ed combination of bits such as `GCForeground` or `GCFont` that indicate which fields of `valuePtr` are significant. `ValuePtr` specifies values of the selected fields. `Tk_GetGC` returns the X resource identifier for a graphics context that matches `valueMask` and `valuePtr`. The graphics context will have default values for all of the unspecified fields.

When you're finished with a graphics context you must free it by calling `Tk_FreeGC`:

```
Tk_FreeGC(Display *display, GC gc)
```

The `display` argument indicates the display for which the graphics context was allocated and the `gc` argument identifies the graphics context (`gc` must have been the return value from some previous call to `Tk_GetGC`). There must be exactly one call to `Tk_FreeGC` for each call to `Tk_GetGC`.

38.2.2 Other resources

Although resources other than graphics contexts are normally allocated and deallocated automatically by `Tk_ConfigureWidget` and `Tk_FreeOptions`, you can also allocate them explicitly using Tk library procedures. For each resource there are three procedures. The first procedure (such as `Tk_GetColor`) takes a textual description of the resource in the same way it might be specified as a configuration option and returns a suitable resource or an error. The second procedure (such as `Tk_FreeColor`) takes a resource allocated by the first procedure and frees it. The third procedure takes a resource and returns the textual description that was used to allocate it. The following resources are supported in this way:

Bitmaps: the procedures `Tk_GetBitmap`, `Tk_FreeBitmap`, and `Tk_NameOfBitmap` manage `Pixmap` resources with depth one. You can also invoke `Tk_DefineBitmap` to create new internally-defined bitmaps, and `Tk_SizeOfBitmap` returns the dimensions of a bitmap.

Colors : the procedures `Tk_GetColor`, `Tk_FreeColor`, and `Tk_NameOfColor` manage `XColor` structures. You can also invoke `Tk_GetColorByValue` to specify a color with integer intensities rather than a string.

Cursors: the procedures `Tk_GetCursor`, `Tk_FreeCursor`, and `Tk_NameOfCursor` manage `Cursor` resources. You can also invoke `Tk_GetCursorFromData` to define a cursor based on binary data in the application.

Fonts: the procedures `Tk_GetFontStruct`, `Tk_NameOfFontStruct`, and `Tk_FreeFontStruct` manage `XFontStruct` structures.

3-D borders: the procedures `Tk_Get3DBorder`, `Tk_Free3DBorder`, and `Tk_NameOf3DBorder` manage `Tk_3DBorder` resources, which are used to draw objects with beveled edges that produce 3-D effects. Associated with these procedures are other procedures such as `Tk_Draw3DRectangle` that draw objects on the screen (see Section 40.3). In addition you can invoke `Tk_3DBorderColor` to retrieve the `XColor` structure for the border's base color.

38.3 Tk_Uids

When invoking procedures like Tk_GetColor you pass in a textual description of the resource to allocate, such as “red” for a color. However, this textual description is not a normal C string but rather a *unique identifier*, which is represented with the type Tk_Uid:

```
typedef char *Tk_Uid;
```

A Tk_Uid is like an atom in Lisp. It is actually a pointer to a character array, just like a normal C string, and a Tk_Uid can be used anywhere that a string can be used. However, Tk_Uid’s have the property that any two Tk_Uid’s with the same string value also have the same pointer value: if a and b are Tk_Uid’s and

```
(strcmp(a,b) == 0)
```

then

```
(a == b)
```

Tk uses Tk_Uid’s to specify resources because they permit fast comparisons for equality.

If you use Tk_ConfigureWidget to allocate resources then you won’t have to worry about Tk_Uid’s (Tk automatically translates strings from the configuration table into Tk_Uid’s). But if you call procedures like Tk_GetColor directly then you’ll need to use Tk_GetUid to turn strings into unique identifiers:

```
Tk_Uid Tk_GetUid(char *string)
```

Given a string argument, Tk_GetUid returns the corresponding Tk_Uid. It just keeps a hash table of all unique identifiers that have been used so far and returns a pointer to the key stored in the hash table.

Note: *If you pass strings directly to procedures like* Tk_GetColor *without converting them to unique identifiers then you will get unpredictable results. One common symptom is that the application uses the same resource over and over even though you think you’ve specified different values for each use. Typically what happens is that the same string buffer was used to store all of the different values. Tk just compares the string address rather than its contents, so the values appear to Tk to be the same.*

38.4 Other translators

Tk provides several other library procedures that translate from strings in various forms to internal representations. These procedures are similar to the resource managers in Section 38.2 except that the internal forms are not resources that require freeing, so typically there is just a “get” procedure and a “name of” procedure with no “free” procedure. Below is a quick summary of the availabile translators (see the reference documentation for details):

Anchors: Tk_GetAnchor and Tk_NameOfAnchor translate between strings containing an anchor positions such as “center” or “ne” and integers with values defined by symbols such as TK_ANCHOR_CENTER or TK_ANCHOR_NE.

Cap styles: Tk_GetCapStyle and Tk_NameOfCapStyle translate betwen strings containing X cap styles ("butt", "projecting", or "round") and integers with values defined by the X symbols CapButt, CapProjecting, and CapRound.

Join styles: Tk_JoinStyle and Tk_NameOfJoinStyle translate between strings containing X join styles ("bevel", "miter", or "round") and integers with values defined by the X symbols JoinBevel, JoinMiter, and JoinRound.

Justify styles: Tk_GetJustify and Tk_NameOfJustify translate between strings containing styles of justification ("left", "right", "center", or "fill") and integers with values defined by the symbols TK_JUSTIFY_LEFT, TK_JUSTIFY_RIGHT, TK_JUSTIFY_CENTER, and TK_JUSTIFY_FILL.

Reliefs: Tk_GetRelief and Tk_NameOfRelief translate between strings containing relief names ("raised", "sunken", "flat", "groove", or "ridge") and integers with values defined by the symbols TK_RELIEF_RAISED, TK_RELIEF_SUNKEN, etc.

Screen distances: Tk_GetPixels and Tk_GetScreenMM process strings that contain screen distances in any of the forms described in Section XXX, such as "1.5m" or "2". Tk_GetPixels returns an integer result in pixel units, and Tk_GetScreenMM returns a real result whose units are millimeters.

Window names: Tk_NameToWindow translates from a string containing a window path name such as ".dlg.quit" to the Tk_Window token for the corresponding window.

X atoms: Tk_InternAtom and Tk_GetAtomName translate between strings containing the names of X atoms (e.g. "RESOURCE_MANAGER") and X Atom tokens. Tk keeps a cache of atom names to avoid communication with the X server.

38.5 Changing window attributes

Tk provides a collection of procedures for modifying a window's attributes (e.g. background color or cursor) and configuration (e.g. position or size). These procedures are summarized in Table 38.2. The procedures have the same arguments as the Xlib procedures with corresponding names. They perform the same functions as the Xlib procedures except that they also retain a local copy of the new information so that it can be returned by the macros described in Section 37.5. For example, Tk_ResizeWindow is similar to the Xlib procedure XResizeWindow in that it modifies the dimensions of a window. However, it also remembers the new dimensions so they can be accessed with the Tk_Width and Tk_Height macros.

Only a few of the procedures in Table 38.2, such as Tk_SetWindowBackground, are normally invoked by widgets. Widgets should definitely *not* invoke procedures like

`Tk_ChangeWindowAttributes(Tk_Window tkwin, unsigned int value-`
`Mask,`
`XSetWindowAttributes *attsPtr)`

`Tk_ConfigureWindow(Tk_Window tkwin, unsigned int valueMask,` `XWindowChanges *valuePtr)`
`Tk_DefineCursor(Tk_Window tkwin, Cursor cursor)`
`Tk_MoveWindow(Tk_Window tkwin, int x, int y)`
`Tk_MoveResizeWindow(Tk_Window tkwin, int x, int y,` `unsigned int width, unsigned int height)`
`Tk_ResizeWindow(Tk_Window tkwin, unsgined int width,` `unsigned int height)`
`Tk_SetWindowBackground(Tk_Window tkwin, unsigned long pixel)`
`Tk_SetWindowBackgroundPixmap(Tk_Window tkwin, Pixmap pixmap)`
`Tk_SetWindowBorder(Tk_Window tkwin, unsigned long pixel)`
`Tk_SetWindowBorderPixmap(Tk_Window tkwin, Pixmap pixmap)`
`Tk_SetWindowBorderWidth(Tk_Window tkwin, int width)`
`Tk_SetWindowColormap(Tk_Window tkwin, Colormap colormap)`
`Tk_UndefineCursor(Tk_Window tkwin)`

Table 38.2. Tk procedures for modifying attributes and window configuration information. `Tk_ChangeWindowAttributes` and `Tk_ConfigureWindow` allow any or all of the attributes or configuration to be set at once (`valueMask` selects which values should be set); the other procedures set selected fields individually.

`Tk_MoveWindow` or `Tk_ResizeWindow`: only geometry managers should change the size or location of a window.

38.6 The square configure procedure

Figure 38.1 contains the code for the square widget's configure procedure. Its `argv` argument contains pairs of strings that specify configuration options.Most of the work is done by `Tk_ConfigureWidget`. Once `Tk_ConfigureWidget` returns, `Configur-`

```
int ConfigureSquare(Tcl_Interp *interp, Square *squarePtr,
        int argc, char *argv[], int flags) {
    if (Tk_ConfigureWidget(interp, squarePtr->tkwin, configSpecs,
            argc, argv, (char *) squarePtr, flags) != TCL_OK) {
        return TCL_ERROR;
    }
    Tk_SetWindowBackground(squarePtr->tkwin,
            Tk_3DBorderColor(squarePtr->bgBorder));
    if (squarePtr->gc == None) {
        XGCValues gcValues;
        gcValues.function = GXcopy;
        gcValues.graphics_exposures = False;
        squarePtr->gc = Tk_GetGC(squarePtr->tkwin,
                GCFunction|GCGraphicsExposures, &gcValues);
    }
    Tk_GeometryRequest(squarePtr->tkwin, 200, 150);
    Tk_SetInternalBorder(squarePtr->tkwin,
            squarePtr->borderWidth);
    if (!squarePtr->updatePending) {
        Tk_DoWhenIdle(DisplaySquare, (ClientData) squarePtr);
        squarePtr->updatePending = 1;
    }
    return TCL_OK;
}
```

Figure 38.1. The configure procedure for square widgets. It is invoked by the creation procedure and by the widget command procedure to set and modify configuration options.

`eSquare` extracts the color associated with the `-background` option and calls `Tk_SetWindowBackground` to use it as the background color for the widget's window. Then it allocates a graphics context that will be used during redisplay to copy bits from an off-screen pixmap into the window (unless some previous call to the procedure has already allocated the graphics context). Next `ConfigureSquare` calls `Tk_GeometryRequest` and `Tk_SetInternalBorderWidth` to provide information to its geometry manager (this will be discussed in Chapter 43). Finally, it arranges for the widget to be redisplayed; this will be discussed in Chapter 40.

38.7 The square widget command procedure

Figures 38.2 and 38.3 contain the C code for `SquareWidgetCommand`, which implements widget commands for square widgets. The main portion of the procedure consists of a series of `if` statements that compare `argv[1]` successively to "`configure`", "`position`", and "`size`", which are the three widget commands defined for squares. If

```
int SquareWidgetCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    Square *squarePtr = (Square *) clientData;
    int result = TCL_OK;

    if (argc < 2) {
        Tcl_AppendResult(interp, "wrong # args: should be \"",
                argv[0], " option ?arg arg ...?\"",
                (char *) NULL);
        return TCL_ERROR;
    }

    Tk_Preserve((ClientData) squarePtr);
    if (strcmp(argv[1], "configure") == 0) {
        if (argc == 2) {
            result = Tk_ConfigureInfo(interp, squarePtr->tkwin,
                    (char *) squarePtr, (char *) NULL, 0);
        } else if (argc == 3) {
            result = Tk_ConfigureInfo(interp, squarePtr->tkwin,
                    (char *) squarePtr, argv[2], 0);
        } else {
            result = ConfigureSquare(interp, squarePtr,
                    argc-2, argv+2, TK_CONFIG_ARGV_ONLY);
        }
    } else if (strcmp(argv[1], "position") == 0) {
        if ((argc != 2) && (argc != 4)) {
            Tcl_AppendResult(interp,"wrong # args: should be \"",
                    argv[0], " position ?x y?\"", (char *) NULL);
            goto error;
        }
        if (argc == 4) {
            if ((Tk_GetPixels(interp, squarePtr->tkwin,
                    argv[2], &squarePtr->x) != TCL_OK) ||
                    (Tk_GetPixels(interp, squarePtr->tkwin,
                    argv[3], &squarePtr->y) != TCL_OK)) {
                goto error;
            }
            KeepInWindow(squarePtr);
        }
        sprintf(interp->result, "%d %d", squarePtr->x,
                squarePtr->y);
    } else if (strcmp(argv[1], "size") == 0) {
```

Figure 38.2. The widget command procedure for square widgets. Continued in Figure 38.3.

```
    if ((argc != 2) && (argc != 3)) {
            Tcl_AppendResult(interp, "wrong # args: should be \"",
                    argv[0], " size ?amount?\"", (char *) NULL);
            goto error;
        }
        if (argc == 3) {
            int i;
            if (Tk_GetPixels(interp, squarePtr->tkwin, argv[2],
                    &i) != TCL_OK) {
                goto error;
            }
            if ((i <= 0) || (i > 100)) {
                Tcl_AppendResult(interp, "bad size \"", argv[2],
                        "\"", (char *) NULL);
                goto error;
            }
            squarePtr->size = i;
            KeepInWindow(squarePtr);
        }
        sprintf(interp->result, "%d", squarePtr->size);
    } else {
        Tcl_AppendResult(interp, "bad option \"", argv[1],
                "\": must be configure, position, or size",
                (char *) NULL);
        goto error;
    }
    if (!squarePtr->updatePending) {
        Tk_DoWhenIdle(DisplaySquare, (ClientData) squarePtr);
        squarePtr->updatePending = 1;
    }
    Tk_Release((ClientData) squarePtr);
    return result;

    error:
    Tk_Release((ClientData) squarePtr);
    return TCL_ERROR;
}
```

Figure 38.3. The widget command procedure for square widgets, continued from Figure 38.2.

argv[1] matches one of these strings then the corresponding code is executed; otherwise an error is generated.

The configure widget command is handled in one three ways, depending on how many additional arguments it receives. If at most one additional argument is provided then SquareWidgetCmd calls Tk_ConfigureInfo to create descriptive information for one or all of the widget's configuration options. If two or more additional arguments are

```
void KeepInWindow(Square *squarePtr) {
    int i, bd;
    bd = 0;
    if (squarePtr->relief != TK_RELIEF_FLAT) {
        bd = squarePtr->borderWidth;
    }
    i = (Tk_Width(squarePtr->tkwin) - bd)
            - (squarePtr->x + squarePtr->size);
    if (i < 0) {
        squarePtr->x += i;
    }
    i = (Tk_Height(squarePtr->tkwin) - bd)
            - (squarePtr->y + squarePtr->size);
    if (i < 0) {
        squarePtr->y += i;
    }
    if (squarePtr->x < bd) {
        squarePtr->x = bd;
    }
    if (squarePtr->y < bd) {
        squarePtr->y = bd;
    }
}
```

Figure 38.4. The `KeepInWindow` procedure adjusts the location of the square to make sure that it is visible in the widget's window.

provided then `SquareWidgetCmd` passes the additional arguments to `ConfigureSquare` for processing; `SquareWidgetCmd` specifies the `TK_CONFIG_ARGV_ONLY` flag, which `ConfigureSquare` passes on to `Tk_ConfigureWidget` so that options not specified explicitly by `argv` are left as-is.

The `position` and `size` widget commands change the geometry of the square displayed in the widget, and they have similar implementations. If new values for the geometry are specified then each command calls `Tk_GetPixels` to convert the argument(s) to pixel distances. The `size` widget command also checks to make sure that the new size is within a particular range of values. Then both commands invoke `KeepInWindow`, which adjusts the position of the square if necessary to ensure that it is fully visible in the widget's window (see Figure 38.4). Finally, the commands print the current values into `interp->result` to return them as result.

`SquareWidgetCmd` invokes the procedures `Tk_Preserve` and `Tk_Release` as a way of preventing the widget record from being destroyed while the widget command is executing. Chapter 41 will discuss these procedures in more detail. The square widget is so simple that the calls aren't actually needed, but virtually all real widgets do need them so I put them in `SquareWidgetCmd` too.

Chapter 39
Events

This chapter describes Tk's library procedures for event handling. The code you'll write for event handling divides into three parts. The first part consists of code that creates event handlers: it informs Tk that certain callback procedures should be invoked when particular events occur. The second part consists of the callbacks themselves. The third part consists of top-level code that invokes the Tk event dispatcher to process events.

Tk supports three kinds of events: X events, file events (e.g. a particular file has just become readable), and timer events. Tk also allows you to create *idle callbacks*, which cause procedures to be invoked when Tk runs out of other things to do; idle callbacks are used to defer redisplays and other computations until all pending events have been processed. Tk's procedures for event handling are summarized in Table 39.1.

If you are not already familiar with X events, I recommend reading about them in your favorite Xlib documentation before reading this chapter.

39.1 X events

The X window server generates a number of different events to report interesting things that occur in the window system, such as mouse presses or changes in a window's size. Chapter XXX showed how you can use Tk's `bind` command to write event handlers as Tcl scripts. This section describes how to write event handlers in C. Typically you'll only use C handlers for four kinds of X events:

`Expose`: these events notify the widget that part or all of its window needs to be redisplayed.

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

```
void Tk_CreateEventHandler(Tk_Window tkwin, unsigned long mask,
    Tk_EventProc *proc, ClientData clientData)
```
Arranges for `proc` to be invoked whenever any of the events selected by `mask` occurs for `tkwin`.
```
void Tk_DeleteEventHandler(Tk_Window tkwin, unsigned long mask,
    Tk_EventProc *proc, ClientData clientData)
```
Deletes the event handler that matches `mask`, `proc`, and `clientData`, if such a handler exists.

```
void Tk_CreateFileHandler(int fd, int mask, Tk_FileProc *proc,
    ClientData clientData)
```
Arranges for `proc` to be invoked whenver one of the conditions indicated by `mask` occurs for the file whose descriptor number is `fd`.
```
void Tk_DeleteFileHandler(int fd)
```
Deletes the file handler for `fd`, if one exists.

```
Tk_TimerToken Tk_CreateTimerHandler(int milliseconds,
    Tk_TimerProc *proc, ClientData clientData)
```
Arranges for `proc` to be invoked after `milliseconds` have elapsed. Returns a token that can be used to cancel the callback.
```
void Tk_DeleteTimerHandler(Tk_TimerToken token)
```
Cancels the timer callback indicated by `token`, if it hasn't yet triggered.

```
void Tk_DoWhenIdle(Tk_IdleProc *proc, ClientData clientData)
```
Arranges for `proc` to be invoked when Tk has nothing else to do.
```
void Tk_CancelIdleCall(Tk_IdleProc *proc, ClientData clientData)
```
Deletes any existing idle callbacks for `idleProc` and `clientData`.

```
void Tk_CreateGenericHandler(Tk_GenericProc *proc,
    ClientData clientData)
```
Arranges for `proc` to be invoked whenever any X event is received by this process.
```
void Tk_DeleteGenericHandler(Tk_GenericProc *proc,
    ClientData clientData)
```
Deletes the generic handler given by `proc` and `clientData`, if such a handler exists.

```
void Tk_MainLoop(void)
```
Processes events until there are no more windows left in this process.
```
int Tk_DoOneEvent(int flags)
```
Processes a single event of any sort and then returns. `Flags` is normally 0 but may be used to restrict the events that will be processed or to return immediately if there are no pending events.

Table 39.1. A summary of the Tk library procedures for event handling.

ConfigureNotify: these events occur when the window's size or position changes so that it can adjust its layout accordingly (e.g. centered text may have to be repositioned).

FocusIn and FocusOut: these events notify the widget that it has gotten or lost the input focus, so it can turn on or off its insertion cursor.

DestroyNotify: these events notify the widget that its window has been destroyed, so it should free up the widget record and any associated resources.

The responses to these events are all relatively obvious and it is unlikely that a user or application developer would want to deal with the events so it makes sense to hard-code the responses in C. For most other events, such as key presses and mouse actions, it's better to define the handlers in Tcl with the bind command. As a widget writer you can create class bindings to give the widget its default behavior, then users can modify the class bindings or augment them with additional widget-specific bindings. By using Tcl as much as possible you'll make your widgets more flexible.

The procedure Tk_CreateEventHandler is used by widgets to register interest in X events:

```
void Tk_CreateEventHandler(Tk_Window tkwin, unsigned long
mask,
        Tk_EventProc *proc, ClientData clientData);
```

The tkwin argument identifies a particular window and mask is an OR'ed combination of bits like KeyPressMask and StructureNotifyMask that select the events of interest (refer to Xlib documentation for details on the mask values that are available). When one of the requested events occurs for tkwin Tk will invoke proc to handle the event. Proc must match the following prototype:

```
typedef void Tk_EventProc(ClientData clientData, XEvent
*eventPtr);
```

Its first argument will be the same as the clientData value that was passed to Tk_CreateEventHandler and the second argument will be a pointer to a structure containing information about the event (see your Xlib documentation for details on the contents of an XEvent structure). There can exist any number of event handlers for a given window and mask but there can be only one event handler with a particular tkwin, mask, proc, and clientData. If a particular event matches the tkwin and mask for more than one handler then all of the matching handlers are invoked, in the order in which they were created.

For example, the C code for the square widget deals with Expose, ConfigureNotify, and DestroyNotify events. To process these events, the following code is present in the create procedure for squares (see Figure 37.1 on page 335):

```
Tk_CreateEventHandler(squarePtr->tkwin,
        ExposureMask|StructureNotifyMask,
        SquareEventProc, (ClientData) squarePtr);
```

```
void SquareEventProc(ClientData clientData, XEvent *eventPtr) {
    Square *squarePtr = (Square *) clientData;
    if (eventPtr->type == Expose) {
        if ((eventPtr->xexpose.count == 0)
                && !squarePtr->updatePending) {
            Tk_DoWhenIdle(DisplaySquare, (ClientData) squarePtr);
            squarePtr->updatePending = 1;
        }
    } else if (eventPtr->type == ConfigureNotify) {
        KeepInWindow(squarePtr);
        if (!squarePtr->updatePending) {
            Tk_DoWhenIdle(DisplaySquare, (ClientData) squarePtr);
            squarePtr->updatePending = 1;
        }
    } else if (eventPtr->type == DestroyNotify) {
        Tcl_DeleteCommand(squarePtr->interp,
                Tk_PathName(squarePtr->tkwin));
        squarePtr->tkwin = NULL;
        if (squarePtr->flags & REDRAW_PENDING) {
            Tk_CancelIdleCall(DisplaySquare,
                    (ClientData) squarePtr);
        }
        Tk_EventuallyFree((ClientData) squarePtr, DestroySquare);
    }
}
```

Figure 39.1. The event procedure for square widgets.

The `ExposureMask` bit selects `Expose` events and `StructureNotifyMask` selects both `ConfigureNotify` and `DestroyNotify` events, plus several other types of events. The address of the widget's record is used as the `ClientData` for the callback, so it will be passed to `SquareEventProc` as its first argument.

Figure 39.1 contains the code for `SquareEventProc`, the event procedure for square widgets. Whenever an event occurs that matches `ExposureMask` or `StructureNotifyMask` Tk will invoke `SquareEventProc`. `SquareEventProc` casts its `clientData` argument back into a `Square *` pointer, then checks to see what kind of event occurred. For `Expose` events `SquareEventProc` arranges for the widget to be redisplayed. For `ConfigureNotify` events, `SquareEventProc` calls `KeepInWindow` to make sure that the square is still visible in the window (see Figure 38.4 on page 352), then `SquareEventProc` arranges for the widget to be redrawn. For `DestroyNotify` events `SquareEventProc` starts the process of destroying the widget and freeing its widget record; this process will be discussed in more detail in Chapter 41.

If you should need to cancel an existing X event handler you can invoke `Tk_DeleteEventHandler` with the same arguments that you passed to `Tk_CreateEventHandler` when you created the handler:

```
void Tk_DeleteEventHandler(Tk_Window tkwin, unsigned long
mask,
         Tk_EventProc *proc, ClientData clientData);
```

This deletes the handler corresponding to `tkwin`, `mask`, `proc`, and `clientData` so that its callback will not be invoked anymore. If no such handler exists then the procedure does nothing. Tk automatically deletes all of the event handlers for a window when the window is destroyed, so most widgets never need to call `Tk_DeleteEventHandler`.

39.2 File events

Event-driven programs like Tk applications should not block for long periods of time while executing any one operation, since this prevents other events from being serviced. For example, suppose that a Tk application attempts to read from its standard input at a time when no input is available. The application will block until input appears. During this time the process will be suspended by the operating system so it cannot service X events. This means, for example, that the application will not be able to respond to mouse actions nor will it be able to redraw itself. Such behavior is likely to be annoying to the user, since he or she expects to be able to interact with the application at any time.

File handlers provide an event-driven mechanism for reading and writing files that may have long I/O delays. The procedure `Tk_CreateFileHandler` creates a new file handler:

```
void Tk_CreateFileHandler(int fd, int mask, Tk_FileProc *proc,
         ClientData clientData);
```

The `fd` argument gives the number of a POSIX file descriptor (e.g. 0 for standard input, 1 for standard output, and so on). `Mask` indicates when `proc` should be invoked. It is an OR'ed combination of the following bits:

`TK_READABLE` means that Tk should invoke `proc` whenever there is data waiting to be read on `fd`;

`TK_WRITABLE` means that Tk should invoke `proc` whenever `fd` is capable of accepting more output data;

`TK_EXCEPTION` means that Tk should invoke `proc` whenever an exceptional condition is present for `fd`.

The callback procedure for file handlers must match the following prototype:

```
typedef void Tk_FileProc(ClientData clientData,
         int mask);
```

The `clientData` argument will be the same as the `clientData` argument to `Tk_CreateFileHandler` and `mask` will contain a combination of the bits `TK_READABLE`, `TK_WRITABLE`, and `TK_EXCEPTION` to indicate the state of the file at the time of the callback. There can exist only one file handler for a given file at a time; if you call `Tk_CreateFileHandler` at a time when there exists a handler for `fd` then the new handler replaces the old one.

Note: You can temporarily disable a file handler by setting its mask to 0. You can reset the mask later when you want to re-enable the handler.

To delete a file handler, call `Tk_DeleteFileHandler` with the same `fd` argument that was used to create the handler:

```
void Tk_DeleteFileHandler(int fd);
```

This removes the handler for `fd` so that its callback will not be invoked again.

With file handlers you can do event-driven file I/O. Rather than opening a file, reading it from start to finish, and then closing the file, you open the file, create a file handler for it, and then return. When the file is readable the callback will be invoked. It issues exactly one read request for the file, processes the data returned by the read, and then returns. When the file becomes readable again (perhaps immediately) then the callback will be invoked again. Eventually, when the entire file has been read, the file will become readable and the read call will return an end-of-file condition. At this point the file can be closed and the file handler deleted. With this approach, your application will still be able to respond to X events even if there are long delays in reading the file.

For example, `wish` uses a file handler to read commands from its standard input. The main program for `wish` creates a file handler for standard input (file descriptor 0) with the following statement:

```
...
Tk_CreateFileHandler(0, TK_READABLE, StdinProc, (ClientData)
NULL);
Tcl_DStringInit(&command);
...
```

In addition to creating the callback, this code initializes a dynamic string that will be used to buffer lines of input until a complete Tcl command is ready for evaluation. Then the main program enters the event loop as will be described in Section 39.6. When data becomes available on standard input `StdinProc` will be invoked. Its code is as follows:

```
void StdinProc(ClientData clientData, int mask) {
    int count, code;
    char input[1000];
    count = read(0, input, 1000);
    if (count <= 0) {
        ... handle errors and end of file ...
    }
    Tcl_DStringAppend(&command, input, count);
    if (Tcl_CmdComplete(Tcl_DStringValue(&command)) {
        code = Tcl_Eval(interp,
```

```
                    Tcl_DStringValue(&command));
            Tcl_DStringFree(&command);
            ...
        }
        ...
    }
```

After reading from standard input and checking for errors and end-of file, `StdinProc` adds the new data to the dynamic string's current contents. Then it checks to see if the dynamic string contains a complete Tcl command (it won't, for example, if a line such as "`foreach i $x {`" has been entered but the body of the `foreach` loop hasn't yet been typed). If the command is complete then `StdinProc` evaluates the command and clears the dynamic string for the next command.

Note: It is usually best to use non-blocking I/O with file handlers, just to be absolutely sure that I/O operations don't block. To request non-blocking I/O, specify the flag `O_NONBLOCK` *to the* `fcntl` *POSIX system call. If you use file handlers for writing to files with long output delays, such as pipes and network sockets, it's essential that you use use non-blocking I/O; otherwise if you supply too much data in a* `write` *system call the output buffers will fill and the process will be put to sleep.*

Note: For ordinary disk files it isn't necessary to use the event-driven approach described in this section, since reading and writing these files rarely incurs noticeable delays. File handlers are useful primarily for files like terminals, pipes, and network connections, which can block for indefinite periods of time.

39.3 Timer events

Timer events trigger callbacks after particular time intervals. For example, widgets use timer events to display blinking insertion cursors. When the cursor is first displayed in a widget (e.g. because it just got the input focus) the widget creates a timer callback that will trigger in a few tenths of a second. When the timer callback is invoked it turns the cursor off if it was on, or on if it was off, and then reschedules itself by creating a new timer callback that will trigger after a few tenths of a second more. This process repeats indefinitely so that the cursor blinks on and off. When the widget wishes to stop displaying the cursor altogether (e.g. because it has lost the input focus) it cancels the callback and turns the cursor off.

The procedure `Tk_CreateTimerHandler` creates a timer callback:

```
Tk_TimerToken Tk_CreateTimerHandler(int milliseconds,
        Tk_TimerProc *proc, ClientData clientData);
```

The `milliseconds` argument specifies how many milliseconds should elapse before the callback is invoked. `Tk_CreateTimerHandler` returns immediately, and its return value is a token that can be used to cancel the callback. After the given interval has elapsed Tk will invoke `proc`. `Proc` must match the following prototype:

```
void Tk_TimerProc(ClientData clientData);
```

Its argument will be the same as the `clientData` argument passed to `Tk_CreateTimerHandler`. `Proc` is only called once, then Tk deletes the callback automatically. If you want `proc` to be called over and over at regular intervals then `proc` should reschedule itself by calling `Tk_CreateTimerHandler` each time it is invoked.

Note: There is no guarantee that `proc` *will be invoked at exactly the specified time. If the application is busy processing other events when the specified time occurs then* `proc` *won't be invoked until the next time the application invokes the event dispatcher, as described in Section 39.6.*

`Tk_DeleteTimerHandler` cancels a timer callback:

```
void Tk_DeleteTimerHandler(Tk_TimerToken token);
```

It takes a single argument, which is a token returned by a previous call to `Tk_CreateTimerHandler`, and deletes the callback so that it will never be invoked. It is safe to invoke `Tk_DeleteTimerHandler` even if the callback has already been invoked; in this case the procedure has no effect.

39.4 Idle callbacks

The procedure `Tk_DoWhenIdle` creates an *idle callback*:

```
void Tk_DoWhenIdle(Tk_IdleProc *proc, ClientData clientData);
```

This arranges for `proc` to be invoked the next time the application becomes idle. The application is idle when Tk's main event-processing procedure, `Tk_DoOneEvent`, is called and no X events, file events, or timer events are ready. Normally when this occurs `Tk_DoOneEvent` will suspend the process until an event occurs. However, if there exist idle callbacks then all of them are invoked. Idle callbacks are also invoked when the `update` Tcl command is invoked. The `proc` for an idle callback must match the following prototype:

```
typedef void Tk_IdleProc(ClientData clientData);
```

It returns no result and takes a single argument, which will be the same as the `clientData` argument passed to `Tk_DoWhenIdle`.

`Tk_CancelIdleCall` deletes an idle callback so that it won't be invoked after all:

```
void Tk_CancelIdleCall(Tk_IdleProc *proc, ClientData
clientData);
```

`Tk_CancelIdleCall` deletes all of the idle callbacks that match `idleProc` and `clientData` (there can be more than one). If there are no matching idle callbacks then the procedure has no effect.

Idle callbacks are used to implement the delayed operations described in Section XXX. The most common use of idle callbacks in widgets is for redisplay. It is generally a bad idea to redisplay a widget immediately when its state is modified, since this can result in multiple redisplays. For example, suppose the following set of Tcl commands is invoked to change the color, size, and location of a square widget `.s`:

```
.s configure -foreground purple
.s size 2c
.s position 1.2c 3.1c
```

Each of these commands modifies the widget in a way that requires it to be redisplayed, but it would be a bad idea for each command to redraw the widget. This would result in three redisplays, which are unnecessary and can cause the widget to flash as it steps through a series of changes. It is much better to wait until all of the commands have been executed and then redisplay the widget once. Idle callbacks provide a way of knowing when all of the changes have been made: they won't be invoked until all available events have been fully processed.

For example, the square widget uses idle callbacks for redisplaying itself. Whenever it notices that it needs to be redrawn it invokes the following code:

```
if (!squarePtr->updatePending) {
    Tk_DoWhenIdle(DisplaySquare, (ClientData) squarePtr);
    squarePtr->updatePending = 1;
}
```

This arranges for `DisplaySquare` to be invoked as an idle handler to redraw the widget. The `updatePending` field of the widget record keeps track of whether `DisplaySquare` has already been scheduled, so that it will only be scheduled once. When `DisplaySquare` is finally invoked it resets `updatePending` to zero.

39.5 Generic event handlers

The X event handlers described in Section 39.1 only trigger when particular events occur for a particular window managed by Tk. Generic event handlers provide access to events that aren't associated with a particular window, such as `MappingNotify` events, and to events for windows not managed by Tk (such as those in other applications). Generic event handlers are rarely needed and should be used sparingly.

To create a generic event handler, call `Tk_CreateGenericHandler`:

```
void Tk_CreateGenericHandler(Tk_GenericProc *proc,
        ClientData clientData);
```

This will arrange for `proc` to be invoked whenever any X event is received by the application. `Proc` must match the following prototype:

```
typedef int Tk_GenericProc(ClientData clientData,
        XEvent *eventPtr);
```

Its `clientData` argument will be the same as the `clientData` passed to `Tk_CreateGenericHandler` and `eventPtr` will be a pointer to the X event. Generic handlers are invoked before normal event handlers, and if there are multiple generic handlers then they are called in the order in which they were created. Each generic handler returns an integer result. If the result is non-zero it indicates that the handler has completely processed the event and no further handlers, either generic or normal, should be invoked for the event.

The procedure Tk_DeleteGenericHandler deletes generic handlers:

```
Tk_DeleteGenericHandler(Tk_GenericProc *proc,
        ClientData clientData);
```

Any generic handlers that match proc and clientData are removed, so that proc will not be invoked anymore.

Note: Tk_CreateGenericHandler *does nothing to ensure that the desired events are actually sent to the application. For example, if an application wishes to respond to events for a window in some other application then it must invoke* XSelectInput *to notify the X server that it wants to receive the events. Once the events arrive, Tk will dispatch them to the generic handler. However, an application should never invoke* XSelectInput *for a window managed by Tk, since this will interfere with Tk's event management.*

39.6 Invoking the event dispatcher

The preceding sections described the first two parts of event management: creating event handlers and writing callback procedures. The final part of event management is to invoke the Tk event dispatcher, which waits for events to occur and invokes the appropriate callbacks. If you don't invoke the dispatcher then no events will be processed and no callbacks will be invoked.

Tk provides two procedures for event dispatching: Tk_MainLoop and Tk_DoOneEvent. Most applications only use Tk_MainLoop. It takes no arguments and returns no result and it is typically invoked once, in the main program after initialization. Tk_MainLoop calls the Tk event dispatcher repeatedly to process events. When all available events have been processed it suspends the process until more events occur, and it repeats this over and over. It returns only when every Tk_Window created by the process has been deleted (e.g. after the "destroy ." command has been executed). A typical main program for a Tk application will create a Tcl interpreter, call Tk_CreateMainWindow to create a Tk application plus its main window, perform other application-specific initialization (such as evaluating a Tcl script to create the application's interface), and then call Tk_MainLoop. When Tk_MainLoop returns the main program exits. Thus Tk provides top-level control over the application's execution and all of the application's useful work is carried out by event handlers invoked via Tk_MainLoop.

The second procedure for event dispatching is Tk_DoOneEvent, which provides a lower level interface to the event dispatcher:

```
int Tk_DoOneEvent(int flags)
```

The flags argument is normally 0 (or, equivalently, TK_ALL_EVENTS). In this case Tk_DoOneEvent processes a single event and then returns 1. If no events are pending then `Tk_DoOneEvent` suspends the process until an event arrives, processes that event, and then returns 1.

For example, `Tk_MainLoop` is implemented using `Tk_DoOneEvent`:

```
void Tk_MainLoop(void) {
     while (tk_NumMainWindows > 0) {
          Tk_DoOneEvent(0);
     }
}
```

The variable `tk_NumMainWindows` is maintained by Tk to count the total number of main windows (i.e. applications) managed by this process. `Tk_MainLoop` just calls `Tk_DoOneEvent` over and over until all the main windows have been deleted.

`Tk_DoOneEvent` is also used by commands such as `tkwait` that want to process events while waiting for something to happen. For example, the "`tkwait window`" command processes events until a given window has been deleted, then it returns. Here is the C code that implements this command:

```
int done;
...
Tk_CreateEventHandler(tkwin, StructureNotifyMask,
WaitWindowProc,
          (ClientData) &done);
done = 0;
while (!done) {
     Tk_DoOneEvent(0);
}
...
```

The variable `tkwin` identifies the window whose deletion is awaited. The code creates an event handler that will be invoked when the window is deleted, then invokes `Tk_DoOneEvent` over and over until the `done` flag is set to indicate that `tkwin` has been deleted. The callback for the event handler is as follows:

```
void WaitWindowProc(ClientData clientData, XEvent *eventPtr) {
     int *donePtr = (int *) clientData;
     if (eventPtr->type == DestroyNotify) {
          *donePtr = 1;
     }
}
```

The `clientData` argument is a pointer to the flag variable. `WaitWindowProc` checks to make sure the event is a `DestroyNotify` event (`StructureNotifyMask` also selects several other kinds of events, such as `ConfigureNotify`) and if so it sets the flag variable to one.

The `flags` argument to `Tk_DoOneEvent` can be used to restrict the kinds of events it will consider. If it contains any of the bits `TK_X_EVENTS`, `TK_FILE_EVENTS`, `TK_TIMER_EVENTS`, or `TK_IDLE_EVENTS`, then only the events indicated by the specified bits will be considered. Furthermore, if `flags` includes the bit `TK_DONT_-WAIT`, or if no X, file, or timer events are requested, then `Tk_DoOneEvent` won't suspend the process; if no event is ready to be processed then it will return immediately with a 0 result to indicate that it had nothing to do. For example, the “`update idletasks`” command is implemented with the following code, which uses the `TK_IDLE_EVENTS` flag:

```
while (Tk_DoOneEvent(TK_IDLE_EVENTS) != 0) {
    /* empty loop body */
}
```

Chapter 40
Displaying Widgets

Tk provides relatively little support for actually drawing things on the screen. For the most part you just use Xlib functions like `XDrawLine` and `XDrawString`. The only procedures provided by Tk are those summarized in Table 40.1, which create three-dimensional effects by drawing light and dark shadows around objects (they will be discussed more in Section 40.3). This chapter consists mostly of a discussion of techniques for delaying redisplays and for using pixmaps to double-buffer redisplays. These techniques reduce redisplay overheads and help produce smooth visual effects with mimimum flashing.

40.1 Delayed redisplay

The idea of delayed redisplay was already introduced in Section 39.4. Rather than redrawing the widget every time its state is modified, you should use `Tk_DoWhenIdle` to schedule the widget's display procedure for execution later, when the application has finished processing all available events. This allows any other pending changes to the widget to be completed before it's redrawn.

Delayed redisplay requires you to keep track of what to redraw. For simple widgets such as the square widget or buttons or labels or entries, I recommend the simple approach of redrawing the entire widget whenever you redraw any part of it. This eliminates the need to remember which parts to redraw and it will have fine performance for widgets like the ones mentioned above.

For larger and more complex widgets like texts or canvases it isn't practical to redraw the whole widget after each change. This can take a substantial amount of time and cause annoying delays, particularly for operations like dragging where redisplays happen many

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`void Tk_Fill3DRectangle(Display *display, Drawable drawable, Tk_3DBorder border, int x, int y, int width, int height, int borderWidth, int relief)`
Fills the area of `drawable` given by `x`, `y`, `width`, and `height` with the background color from `border`, then draws a 3-D border `borderWidth` pixels wide around (but just inside) the rectangle. `Relief` specifies the 3-D appearance of the border.
`void Tk_Draw3DRectangle(Display *display, Drawable drawable, Tk_3DBorder border, int x, int y, int width, int height, int borderWidth, int relief)`
Same as `Tk_Fill3DRectangle` except only draws the border.	

`void Tk_Fill3DPolygon(Display *display, Drawable drawable, Tk_3DBorder border, XPoint *pointPtr, int numPoints, int borderWidth, int leftRelief)` Fills the area of a polygon in `drawable` with the background color from `border`. The polygon is specified by `pointPtr` and `numPoints` and need not be closed. Also draws a 3-D border around the polygon. `Border-Width` specifies the width of the border, measured in pixels to the left of the polygon's trajectory (if negative then the border is drawn on the right). `LeftRelief` specifies the 3-D appearance of the border (e.g. `TK_RELIE-F_RAISED` means the left side of the trajectory appears higher than the right). `void Tk_Fill3DPolygon(Display *display, Drawable drawable, Tk_3DBorder border, XPoint *pointPtr, int numPoints, int borderWidth, int leftRelief)` Same as `Tk_Fill3DPolygon`, except only draws the border without filling the interior of the polygon.	

Table 40.1. A summary of Tk's procedures for drawing 3-D effects.

times per second. For these widgets you should keep information in the widget record about which parts of the widget need to be redrawn. The display procedure can then use this information to redraw only the affected parts.

I recommend recording what to redraw in the simplest (coarsest) way that gives adequate performance. Keeping redisplay information on a very fine grain is likely to add complexity to your widgets and probably won't improve performance noticeably over a coarser mechanism. For example, the Tk text widget does not record what to redraw on a character-by-character basis; instead, it keeps track of which lines on the screen need to be redrawn. The minimum amount that is ever redrawn is one whole line. Most redisplays only involve one or two lines, and today's workstations are fast enough to redraw hundreds of lines per second, so the widget can keep up with the user even if redraws are occurring dozens of times a second (such as when the user is dragging one end of the selection). Tk's canvases optimize redisplay by keeping a rectangular bounding box that includes all of the modified objects. If two small objects at opposite corners of the window are modified simultaneously then the redisplay area will include the entire window, but this doesn't happen very often. In more common cases, such as dragging a single small object, the bounding box approach requires only a small fraction of the window's area to be redrawn.

40.2 Double-buffering with pixmaps

If you want to achieve smooth dragging and other visual effects then you should not draw graphics directly onto the screen, because this tends to cause annoying flashes. The reason for the flashes is that widgets usually redisplay themselves by first clearing an area to its background color and then drawing the foreground objects. While you're redrawing the widget the monitor is continuously refreshing itself from display memory. Sometimes the widget will be refreshed on the screen after it has been cleared but before the objects have been redrawn. For this one screen refresh the widget will appear to be empty; by the time of the next refresh you'll have redrawn all the objects so they'll appear again. The result is that the objects in the widget will appear to flash off, then on. This flashing is particularly noticeable during dynamic actions such as dragging or animation where redisplays happen frequently.

To avoid flashing it's best to use a technique called *double-buffering*, where you redisplay in two phases using an off-screen pixmap. The display procedure for the square widget, shown in Figure 40.1, uses this approach. It calls `XCreatePixmap` to allocate a pixmap the size of the window, then it calls `Tk_Fill3DRectangle` twice to redraw the widget in the pixmap. Once the widget has been drawn in the pixmap, the contents are copied to the screen by calling `XCopyArea`. With this approach the screen makes a smooth transition from the widget's previous state to its new state. It's still possible for the screen to refresh itself during the copy from pixmap to screen but each pixel will be drawn in either its correct old value or its correct new value.

Note: *If you compile the square widget into* `wish` *you can use the dragging script from Section 36.4 to compare double-buffering with drawing directly on the screen. To make a version of the square widget that draws directly on the screen, just delete the calls to* `XCreatePixmap`*,* `XCopyArea`*, and* `XFreePixmap` *in* `DisplaySquare` *and replace the* `pm` *arguments to* `Tk_Fill3DRectangle` *with* `TkWindowId(tkwin)`*. Or, you can use the version of the square widget that comes with the Tk distribution; it has a* `-dbl` *option that you can use to turn double-buffering on and off dynamically.*

40.3 Drawing procedures

Tk provides only four procedures for actually drawing graphics on the screen, which are summarized in Table 40.1. These procedures make it easy to produce the three-dimensional effects required for Motif widgets, where light and dark shadows are drawn around objects to make them look raised or sunken.

```
void DisplaySquare(ClientData clientData) {
    Square *squarePtr = (Square *) clientData;
    Tk_Window tkwin = squarePtr->tkwin;
    Pixmap pm;
    squarePtr->updatePending = 0;
    if (!Tk_IsMapped(tkwin)) {
        return;
    }
    pm = XCreatePixmap(Tk_Display(tkwin), Tk_WindowId(tkwin),
            Tk_Width(tkwin), Tk_Height(tkwin), Tk_Depth(tkwin));
    Tk_Fill3DRectangle(Tk_Display(tkwin), pm, squarePtr->bgBorder
         0, 0, Tk_Width(tkwin), Tk_Height(tkwin),
        squarePtr->borderWidth, squarePtr->relief);
    Tk_Fill3DRectangle(Tk_Display(tkwin), pm, squarePtr->fgBorder,
        squarePtr->x, squarePtr->y, squarePtr->size, squarePtr-
>size,
        squarePtr->borderWidth, squarePtr->relief);
    XCopyArea(Tk_Display(tkwin), pm, Tk_WindowId(tkwin),
        squarePtr->copyGC, 0, 0, Tk_Width(tkwin), Tk_Height(tkwin),
        0, 0);
    XFreePixmap(Tk_Display(tkwin), pm);
}
```

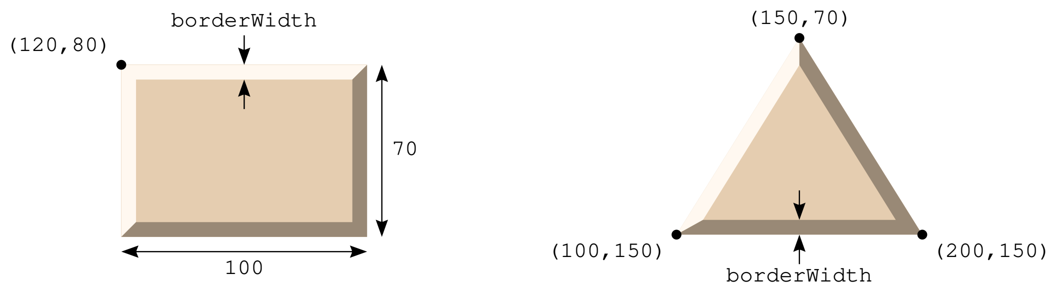
Figure 40.1. The display procedure for square widgets. It first clears `squarePtr->updatePending` to indicate that there is no longer an idle callback for `DisplaySquare` scheduled, then it makes sure that the window is mapped (if not then there's no need to redisplay). It then redraws the widget in an off-screen pixmap and copies the pixmap onto the screen when done.

Before using any of the procedures in Table 40.1 you must allocate a `Tk_3DBorder` object. A `Tk_3DBorder` records three colors (a base color for "flat" background surfaces and lighter and darker colors for shadows) plus X graphics contexts for displaying objects using those colors. Chapter 38 described how to allocate `Tk_3DBorders`, for example by using a configuration table entry of type `TK_CONFIG_BORDER` or by calling `Tk_Get3DBorder`.

Once you've created a `Tk_3DBorder` you can call `Tk_Fill3DRectangle` to draw rectangular shapes with any of the standard reliefs:

```
void Tk_Fill3DRectangle(Display *display, Drawable drawable,
        Tk_3DBorder border, int x, int y,int width, int
height,
        int borderWidth, int relief);
```

The `display` and `drawable` arguments specify the pixmap or window where the rectangle will be drawn. `Display` is usually specified as `Tk_Display(tkwin)` where `tkwin` is the window being redrawn. `Drawable` is usually the off-screen pixmap being used for display, but it can also be `Tk_WindowId(tkwin)`. `Border` specifies the col-




```
Tk_Fill3DRectangle(display,
drawable,
    border, 120, 80, 100, 70,
    borderWidth,
TK_RELIEF_RAISED);
```

(a)

```
static XPoint points[] =
{{200,150},
    {150,70}, {100,150}};
Tk_Fill3DPolygon(display,
drawable,
```

(b)

Figure 40.2. Figure (a) shows a call to `Tk_Fill3DRectangle` and the graphic that is produced; the border is drawn entirely inside the rectangular area. Figure (b) shows a call to `Tk_Fill3DPolygon` and the resulting graphic. The relief `TK_RELIEF_RAISED` specifies that the left side of the path should appear higher than the right, and that the border should be drawn entirely on the left side of the path if `borderWidth` is positive.

ors to be used for drawing the rectangle. `X`, `y`, `width`, `height`, and `borderWidth` specify the geometry of the rectangle and its border, all in pixel units (see Figure 40.2). Lastly, `relief` specifies the desired 3D effect, such as `TK_RELIEF_RAISED` or `TK_RELIEF_RIDGE`. `Tk_Fill3DRectangle` first fills the entire area of the rectangle with the “flat” color from `border` then it draws light and dark shadows `borderWidth` pixels wide around the edge of the rectangle to produce the effect specified by `relief`.

`Tk_Fill3DPolygon` is similar to `Tk_Fill3DRectangle` except that it draws a polygon instead of a rectangle:

```
void Tk_Fill3DPolygon(Display *display, Drawable drawable,
        Tk_3DBorder border, XPoint *pointPtr, int numPoints,
        int borderWidth, int leftRelief);
```

`Display`, `drawable`, and `border` all have the same meaning as for `Tk_Fill3-DRectangle`. `PointPtr` and `numPoints` define the polygon’s shape (see your Xlib documentation for information about `XPoint` structures) and `borderWidth` gives the width of the border, all in pixel units. `LeftRelief` defines the relief of the left side of the polygon’s trajectory relative to its right side. For example, if `leftRelief` is specified as `TK_RELIEF_RAISED` then the left side of the trajectory will appear higher than the right side. If leftRelief is TK_RELIEF_RIDGE or TK_RELIEF_GROOVE then the border will be centered on the polygon's trajectory; otherwise it will be drawn on the left side of the polygon's trajectory if borderWidth is positive and on the right side if borderWidth is negative. See Figure 40.2 for an example.

The procedures Tk_Draw3DRectangle and Tk_Draw3DPolygon are similar to Tk_Fill3DRectangle and Tk_Fill3DPolygon except that they only draw the border without filling the interior of the rectangle or polygon.

Chapter 41
Destroying Widgets

This chapter describes how widgets should clean themselves up when they are destroyed. For the most part widget destruction is fairly straightforward: it's just a matter of freeing all of the resources associated with the widget. However, there is one complicating factor, which is that a widget might be in use at the time it is destroyed. This leads to a two-phase approach to destruction where some of the cleanup may have to be delayed until the widget is no longer in use. Tk's procedures for window destruction, most of which have to do with delayed cleanup, are summarized in Table 41.1.

41.1 Basics

Widgets can be destroyed in three different ways. First, the `destroy` Tcl command can be invoked; it destroys one or more widgets and all of their descendants in the window hierarchy. Second, C code in the application can invoke `Tk_DestroyWindow`, which has the same effect as the `destroy` command:

```
void Tk_DestroyWindow(Tk_Window tkwin);
```

`Tk_DestroyWindow` is not invoked very often but it is used, for example, to destroy a new widget immediately if an error is encountered while configuring it (see Figure 37.1 on page 373). The last way for a widget to be destroyed is for someone to delete its X window directly. This does not occur very often, and is not generally a good idea, but in some cases it may make sense for a top-level window to be deleted externally (by the window manager, for example).

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`void Tk_DestroyWindow(Tk_Window tkwin)`	
	Destroys `tkwin` and all of its descendants in the widget hierarchy.
`void Tk_Preserve(ClientData clientData)`	
	Makes sure that `clientData` will not be freed until a matching call to `Tk_Release` has been made.
`void Tk_Release(ClientData clientData)`	
	Cancels a previous `Tk_Preserve` call for `clientData`. May cause `clientData` to be freed.
`void Tk_EventuallyFree(ClientData clientData Tk_FreeProc *freeProc)`	
	Invokes `freeProc` to free up `clientData` unless `Tk_Preserve` has been called for it; in this case `freeProc` won't be invoked until each `Tk_Preserve` call has been cancelled with a call to `Tk_Release`.

Table 41.1. A summary of the Tk library procedures for destroying widgets and delaying object cleanup.

A widget should handle all of these forms of window destruction in the same way using a handler for `DestroyNotify` events. Tk makes sure that a `DestroyNotify` event is generated for each window that is destroyed and doesn't free up its `Tk_Window` structure until after the handlers for the event have been invoked. When a widget receives a `DestroyNotify` event it typically does four things to clean itself up:

1. It deletes the widget command for the widget by calling `Tcl_DeleteCommand`.
2. It cancels any idle callbacks and timer handlers for the widget, such as the idle callback to redisplay the widget.
3. It frees any resources allocated for the widget. Most of this can be done by calling `Tk_FreeOptions`, but widgets usually have a few resources such as graphics contexts that are not directly associated with configuration options.
4. It frees the widget record.

For square widgets the first two of these actions are carried out in the event procedure, and the third and fourth actions are carried out in a separate procedure called `DestroySquare`. `DestroySquare` is the *destroy procedure* for square widgets; it is invoked indirectly from the event procedure using the mechanism discussed in Section 41.2 below. Its code is shown in Figure 41.1.

41.2 Delayed cleanup

The most delicate aspect of widget destruction is that the widget could be in use at the time it is destroyed; special precautions must be taken to delay most of the widget cleanup

```
void DestroySquare(ClientData clientData) {
    Square *squarePtr = (Square *) clientData;
    Tk_FreeOptions(configSpecs, (char *) squarePtr,
            squarePtr->display, 0);
    if (squarePtr->gc != None) {
        Tk_FreeGC(squarePtr->display, squarePtr->gc);
    }
    free((char *) squarePtr);
}
```

Figure 41.1. The destroy procedure for square widgets.

until the widget is no longer in use. For example, suppose that a dialog box `.dlg` contains a button that is created with the following command:

```
button .dlg.quit -text Quit -command "destroy .dlg"
```

The purpose of this button is to destroy the dialog box. Now suppose that the user clicks on the button with the mouse. The binding for `<ButtonRelease-1>` invokes the button's `invoke` widget command:

```
.dlg.quit invoke
```

The `invoke` widget command evaluates the button's `-command` option as a Tcl script, which destroys the dialog and all its descendants, including the button itself. When the button is destroyed a `DestroyNotify` event is generated, which causes the button's event procedure to be invoked to clean up the destroyed widget. Unfortunately it is not safe for the event procedure to free the button's widget record because the `invoke` widget command is still pending on the call stack: when the event procedure returns, control will eventually return back to the widget command procedure, which may need to reference the widget record. If the event procedure frees the widget record then the widget command procedure will make wild references into memory. Thus in this situation it is important to wait until the widget command procedure completes before freeing the widget record.

However, a button widget might also be deleted at a time when there is no `invoke` widget command pending (e.g. the user might click on some other button, which destroys the entire application). In this case the cleanup must be done by the event procedure since there won't be any other opportunity for the widget to clean itself up. In other cases there could be several nested procedures each of which is using the widget record, so it won't be safe to clean up the widget record until the last of these procedures finishes.

In order to handle all of these cases cleanly Tk provides a mechanism for keeping track of whether an object is in use and delaying its cleanup until it is no longer being used. `Tk_Preserve` is invoked to indicate that an object is in use and should not be freed:

```
void Tk_Preserve(ClientData clientData);
```

The clientData argument is a token for an object that might potentially be freed; typically it is the address of a widget record. For each call to Tk_Preserve there must eventually be a call to Tk_Release:

```
void Tk_Release(ClientData clientData);
```

The clientData argument should be the same as the corresponding argument to Tk_Preserve. Each call to Tk_Release cancels a call to Tk_Preserve for the object; once all calls to Tk_Preserve have been cancelled it is safe to free the object.

When Tk_Preserve and Tk_Release are being used to manage an object you should call Tk_EventuallyFree to free the object:

```
void Tk_EventuallyFree(ClientData clientData,
        Tk_FreeProc *freeProc);
```

ClientData must be the same as the clientData argument used in calls to Tk_Preserve and Tk_Release, and freeProc is a procedure that actually frees the object. FreeProc must match the following prototype:

```
typedef void Tk_FreeProc(ClientData clientData);
```

Its clientData argument will be the same as the clientData argument to Tk_EventuallyFree. If the object hasn't been protected with calls to Tk_Preserve then Tk_EventuallyFree will invoke freeProc immediately. If Tk_Preserve has been called for the object then freeProc won't be invoked immediately; instead it will be invoked later when Tk_Release is called. If Tk_Preserve has been called multiple times then freeProc won't be invoked until each of the calls to Tk_Preserve has been cancelled by a separate call to Tk_Release.

I recommend that you use these procedures in the same way as in the square widget. Place a call to Tk_Preserve at the beginning of the widget command procedure and a call to Tk_Release at the end of the widget command procedure, and be sure that you don't accidentally return from the widget command procedure without calling Tk_Release, since this would prevent the widget from ever being freed. Then divide the widget cleanup code into two parts. Put the code to delete the widget command, idle callbacks, and timer handlers directly into the event procedure; this code can be executed immediately without danger, and it prevents any new invocations of widget code. Put all the code to cleanup the widget record into a separate delete procedure like DestroySquare, and call Tk_EventuallyFree from the event procedure with the delete procedure as its freeProc argument.

This approach is a bit conservative but it's simple and safe. For example, most widgets have only one or two widget commands that could cause the widget to be destroyed, such as the invoke widget command for buttons. You could move the calls to Tk_Preserve and Tk_Release so that they only occur around code that might destroy the widget, such as a Tcl_GlobalEval call. This will save a bit of overhead by eliminating calls to Tk_Preserve and Tk_Release where they're not needed. However, Tk_Preserve and Tk_Release are fast enough that this optimization won't save much time and it means you'll constantly have to be on the lookout to add more calls to

Tk_Preserve and Tk_Release if you modify the widget command procedure. If you place the calls the beginning and end of the procedure you can make any modifications you wish to the procedure without having to worry about issues of widget cleanup. In fact, the square widget doesn't need calls to Tk_Preserve and Tk_Release at all, but I put them in anyway so that I won't have to remember to add them later if I modify the widget command procedure.

For most widgets the only place you'll need calls to Tk_Preserve and Tk_Release is in the widget command procedure. However, if you invoke procedures like Tcl_Eval anywhere else in the widget's code then you'll need additional Tk_Preserve and Tk_Release calls there too. For example, widgets like canvases and texts implement their own event binding mechanisms in C code; these widgets must invoke Tk_Preserve and Tk_Release around the calls to event handlers.

The problem of freeing objects while they're in use occurs in many contexts in Tk applications. For example, it's possible for the -command option for a button to change the button's -command option. This could cause the memory for the old value of the option to be freed while it's still being evaluated by the Tcl interpreter. To eliminate this problem the button widget evaluates a copy of the script rather than the original. In general whenever you make a call whose behavior isn't completely predictable, such as a call to Tcl_Eval and its cousins, you should think about all the objects that are in use at the time of the call and take steps to protect them. In some simple cases making local copies may be the simplest solution, as with the -command option; in more complex cases I'd suggest using Tk_Preserve and Tk_Release; they can be used for objects of any sort, not just widget records.

Note: Tk_Preserve *and* Tk_Release *implement a form of short-term reference counts. They are implemented under the assumption that objects are only in use for short periods of time such as the duration of a particular procedure call, so that there are only a few protected objects at any given time. You should not use them for long-term reference counts where there might be hundreds or thousands of objects that are protected at a given time, since they will be very slow in these cases.*

Chapter 42
Managing the Selection

This chapter describes how to manipulate the X selection from C code. The low-level protocols for claiming the selection and transmitting it between applications are defined by X's Inter-Client Communications Convention Manual (ICCCM) and are very complicated. Fortunately Tk takes care of all the low-level details for you and provides three simpler operations that you can perform on the selection:

- Create a *selection handler*, which is a callback procedure that can supply the selection when it is owned in a particular window and retrieved with a particular target.
- Claim ownership of the selection for a particular window.
- Retrieve the selection from its current owner in a particular target form.

Each of these three operations can be performed either using Tcl scripts or by writing C code. Chapter XXX described how to manipulate the selection with Tcl scripts and much of that information applies here as well, such as the use of targets to specify different ways to retrieve the selection. Tcl scripts usually just retrieve the selection; claiming ownership and supplying the selection are rarely done from Tcl. In contrast, it's common to create selection handlers and claim ownership of the selection from C code but rare to retrieve the selection. See Table 42.1 for a summary of the Tk library procedures related to the selection.

42.1 Selection handlers

Each widget that supports the selection, such as an entry or text, must provide one or more *selection handlers* to supply the selection on demand when the widget owns it. Each han-

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`Tk_CreateSelHandler(Tk_Window tkwin, Atom target, Tk_SelectionProc *proc, ClientData clientData, Atom format)`	Arranges for `proc` to be invoked whenever the selection is owned by `tkwin` and is retrieved in the form given by `target`. `Format` specifies the form in which Tk should transmit the selection to the requestor, and is usually `XA_STRING`.
`Tk_DeleteSelHandler(Tk_Window tkwin, Atom target)`	Removes the handler for `tkwin` and `target`, if one exists.
`Tk_OwnSelection(Tk_Window tkwin, Tk_LostSelProc *proc, ClientData clientData)`	Claims ownership of the selection for `tkwin` and notifies the previous owner, if any, that it has lost the selection. `Proc` will be invoked later when `tkwin` loses the selection.
`Tk_ClearSelection(Tk_Window tkwin)`	Cancels any existing selection for the display containing `tkwin`.
`int Tk_GetSelection(Tcl_Interp *interp, Tk_Window tkwin, Atom target, Tk_GetSelProc *proc, ClientData clientData)`	Retrieves the selection for `tkwin`'s display in the format specified by `target` and passes it to `proc` in one or more pieces. Returns `TCL_OK` or `TCL_ERROR` and leaves an error message in `interp->result` if an error occurs.

Table 42.1. A summary of Tk's procedures for managing the selection.

dler returns the selection in a particular target form. The procedure `Tk_CreateSelHandler` creates a new selection handler:

```
void Tk_CreateSelHandler(Tk_Window tkwin, Atom target,
        Tk_SelectionProc *proc, ClientData clientData,
        Atom format);
```

`Tkwin` is the window from which the selection will be provided; the handler will only be asked to supply the selection when the selection is owned by `tkwin`. `Target` specifies the target form in which the handler can supply the selection; the handler will only be invoked when the selection is retrieved with that target. `Proc` is the address of the handler callback, and `clientData` is a one-word value to pass to `proc`. `Format` tells Tk how to transmit the selection to the requestor and is usually `XA_STRING` (see the reference documentation for other possibilities).

The callback procedure for a selection handler must match the following prototype:

```
typedef int Tk_SelectionProc(ClientData clientData,
        int offset, char *buffer, int maxBytes);
```

The `clientData` argument will be the same as the `clientData` argument passed to `Tk_CreateSelHandler`; it is usually the address of a widget record. `Proc` should place a null-terminated string at `buffer` containing up to `maxBytes` of the selection starting at byte offset within the selection. The procedure should return a count of the number of non-null bytes copied, which must be maxBytes unless there are fewer than maxBytes left in the selection. If the widget no longer has a selection (because, for example, the user deleted the selected range of characters) the selection handler should return -1.

Usually the entire selection will be retrieved in a single request: offset will be 0 and maxBytes will be large enough to accommodate the entire selection. However, very large selections will be retrieved in transfers of a few thousand bytes each. Tk will invoke the callback several times using successively higher values of offset to retrieve successive portions of the selection. If the callback returns a value less than maxBytes it means that the entire remainder of the selection has been returned. If its return value is maxBytes it means that there may be additional information in the selection so Tk will call it again to retrieve the next portion. You can assume that maxBytes will always be at least a few thousand.

For example, Tk's entry widgets have a widget record of type Entry with three fields that are used to manage the selection:

string points to a null-terminated string containing the text in the entry;

selectFirst is the index in string of the first selected byte (or -1 if nothing is selected);

selectLast is the index of the last selected byte.

An entry will supply the selection in only one target form (STRING) so it only has a single selection handler. The create procedure for entries contains a statement like the following to create the selection handler, where entryPtr is a pointer to the widget record for the new widget:

```
Tk_CreateSelHandler(entryPtr->tkwin, XA_STRING,
        EntryFetchSelection, (ClientData) entryPtr,
        XA_STRING);
```

The callback for the selection handler is defined as follows:

```
int EntryFetchSelection(ClientData clientData, int offset,
        char *buffer, int maxBytes) {
    Entry *entryPtr = (Entry *) clientData;
    int count;
    if (entryPtr->selectFirst < 0) {
        return -1;
    }
    count = entryPtr->selectLast + 1 - entryPtr->selectFirst
        - offset;
    if (count > maxBytes) {
        count = maxBytes;
    }
    if (count <= 0) {
        count = 0;
    } else {
```

```
            strncpy(buffer, entryPtr->string
            + entryPtr->selectFirst + offset, count);
        }
        buffer[count] = 0;
        return count;
    }
```

If a widget wishes to supply the selection in several different target forms it should create a selection handler for each target. When the selection is retrieved, Tk will invoke the handler for the target specified by the retriever.

Tk automatically provides handlers for the following targets:

`APPLICATION`: returns the name of the application, which can be used to `send` commands to the application containing the selection.

`MULTIPLE`: used to retrieve the selection in multiple target forms simultaneously. Refer to ICCCM documenation for details.

`TARGETS`: returns a list of all the targets supported by the current selection owner (including all the targets supported by Tk).

`TIMESTAMP`: returns the time at which the selection was claimed by its current owner.

`WINDOW_NAME`: returns the path name of the window that owns the selection.

A widget can override any of these default handlers by creating a handler of its own.

42.2 Claiming the selection

The previous section showed how a widget can supply the selection to a retriever. However, before a widget will be asked to supply the selection it must first claim ownership of the selection. This usually happens during widget commands that select something in the widget, such as the `select` widget command for entries and listboxes. To claim ownership of the selection a widget should call `Tk_OwnSelection`:

```
void Tk_OwnSelection(Tk_Window tkwin, Tk_LostSelProc *proc,
        (ClientData) clientData);
```

`Tk_OwnSelection` will communicate with the X server to claim the selection for `tkwin`; as part of this process the previous owner of the selection will be notified so that it can deselect itself. `Tkwin` will remain the selection owner until either some other window claims ownership, `tkwin` is destroyed, or `Tk_ClearSelection` is called. When `tkwin` loses the selection Tk will invoke `proc` so that the widget can deselect itself and display itself accordingly. `Proc` must match the following prototype:

```
typedef void Tk_LostSelProc(ClientData clientData);
```

The `clientData` argument will be the same as the `clientData` argument to `Tk_OwnSelection`; it is usually a pointer to the widget's record.

Note: `Proc` *will only be called if some other window claims the selection or if* `Tk_ClearSelection` *is invoked. It will not be called if the owning widget is destroyed.*

If a widget claims the selection and then eliminates its selection (for example, the selected text is deleted) the widget has three options. First, it can continue to service the selection and return 0 from its selection handlers; anyone who retrieves the selection will receive an empty string. Second, the widget can continue to service the selection and return -1 from its selection handlers; this will return an error ("no selection") to anyone who attempts to retrieve it. Third, the widget can call `Tk_ClearSelection`:

```
void Tk_ClearSelection(Tk_Window tkwin);
```

The `tkwin` argument identifies a display. Tk will claim the selection away from whatever window owned it (either in this application or any other application on `tkwin`'s display) and leave the selection unclaimed, so that all attempts to retrieve it will result in errors. This approach will have the same effect returning -1 from the selection handlers except that the selection handlers will never be invoked at all.

42.3 Retrieving the selection

If an application wishes to retrieve the selection, for example to insert the selected text into an entry, it usually does so with the "`selection get`" Tcl command. This section describes how to retrieve the selection at C level, but this facility is rarely needed. The only situation where I recommend writing C code to retrieve the selection is in cases where the selection may be very large and a Tcl script may be noticeably slow. This might occur in a text widget, for example, where a user might select a whole file in one window and then copy it into another window. If the selection has hundreds of thousands of bytes then a C implementation of the retrieval will be noticeably faster than a Tcl implementation.

To retrieve the selection from C code, invoke the procedure `Tk_GetSelection`:

```
typedef int Tk_GetSelection(Tcl_Interp *interp,
        Tk_Window tkwin, Atom target, Tk_GetSelProc *proc,
        ClientData clientData);
```

The `interp` argument is used for error reporting. `Tkwin` specifies the window on whose behalf the selection is being retrieved (it selects a display to use for retrieval), and `target` specifies the target form for the retrieval. `Tk_GetSelection` doesn't return the selection directly to its caller. Instead, it invokes `proc` and passes it the selection. This makes retrieval a bit more complicated but it allows Tk to buffer data more efficiently. Large selections will be retrieved in several pieces, with one call to `proc` for each piece. `Tk_GetSelection` normally returns `TCL_OK` to indicate that the selection was successfully retrieved. If an error occurs then it returns `TCL_ERROR` and leaves an error message in `interp->result`.

`Proc` must match the following prototype:

```
typedef int Tk_GetSelProc(ClientData clientData,
        Tcl_Interp *interp, char *portion);
```

The `clientData` and `interp` arguments will be the same as the corresponding arguments to `Tk_GetSelection`. `Portion` points to a null-terminated ASCII string containing part or all of the selection. For small selections a single call will be made to `proc` with the entire contents of the selection. For large selections two or more calls will be made with successive portions of the selection. `Proc` should return `TCL_OK` if it successfully processes the current portion of the selection. If it encounters an error then it should return `TCL_ERROR` and leave an error message in `interp->result`; the selection retrieval will be aborted and this same error will be returned to `Tk_GetSelection`'s caller.

For example, here is code that retrieves the selection in target form `STRING` and prints it on standard output:

```
    ...
    if (Tk_GetSelection(interp, tkwin,
            Tk_InternAtom(tkwin, "STRING"), PrintSel,
            (ClientData) stdout) != TCL_OK) {
        ...
    }
    ...
int PrintSel(ClientData clientData, Tcl_Interp *interp,
        char *portion) {
    FILE *f = (FILE *) clientData;
    fputs(portion, f);
    return TCL_OK;
}
```

The call to `Tk_GetSelection` could be made, for example, in the widget command procedure for a widget, where `tkwin` is the `Tk_Window` for the widget and `interp` is the interpreter in which the widget command is being processed. The `clientData` argument is used to pass a `FILE` pointer to `PrintSel`. The output could be written to a different file by specifying a different `clientData` value.

Chapter 43
Geometry Management

Tk provides two groups of library procedures for geometry management. The first group of procedures implements a communication protocol between slave windows and their geometry managers. Each widget calls Tk to provide geometry information such as the widget's preferred size and whether or not it has an internal grid. Tk then notifies the relevant geometry manager, so that the widget does not have to know which geometry manager is responsible for it. Each geometry manager calls Tk to identify the slave windows it will manage, so that Tk will know who to notify when geometry information changes for the slaves. The second group of procedures is used by geometry managers to place slave windows. It includes facilities for mapping and unmapping windows and for setting their sizes and locations. All of these procedures are summarized in Table 43.1.

43.1 Requesting a size for a widget

Each widget is responsible for informing Tk of its geometry needs; Tk will make sure that this information is forwarded to any relevant geometry managers. There are three pieces of information that the slave can provide: requested size, internal border, and grid. The first piece of information is provided by calling `Tk_GeometryRequest`:

```
void Tk_GeometryRequest(Tk_Window tkwin, int width, height);
```

This indicates that the ideal dimensions for `tkwin` are `width` and `height`, both specified in pixels. Each widget should call `Tk_GeometryRequest` once when it is created and again whenever its preferred size changes (such as when its font changes); normally the calls to `Tk_GeometryRequest` are made by the widget's configure procedure. In

Copyright © 1993 Addison-Wesley Publishing Company, Inc.
All rights reserved. Duplication of this draft is permitted by individuals for personal use only. Any other form of duplication or reproduction requires prior written permission of the author or publisher. This statement must be easily visible on the first page of any reproduced copies. The publisher does not offer warranties in regard to this draft.

`Tk_GeometryRequest(Tk_Window tkwin, int width, int height)`	Informs the geometry manager for tkwin that the preferred dimensions for `tkwin` are `width` and `height`.
`Tk_SetInternalBorder(Tk_Window tkwin, int width)`	Informs any relevant geometry managers that `tkwin` has an internal border `width` pixels wide and that slave windows should not be placed in this border region.
`Tk_SetGrid(Tk_Window tkwin, int reqWidth, int reqHeight, int widthInc, int heightInc)`	Turns on gridded geometry management for `tkwin`'s top-level window and specifies the grid geometry. The dimensions requested by `Tk_GeometryRequest` correspond to grid dimensions of `reqWidth` and `reqHeight`, and `widthInc` and `heightInc` specify the dimensions of a single grid cell.
`Tk_ManageGeometry(Tk_Window tkwin, Tk_GeometryProc *proc, ClientData clientData)`	Arranges for `proc` to be invoked whenever `Tk_GeometryRequest` is invoked for `tkwin`. Used by geometry managers to claim ownership of a slave window.
`int Tk_ReqHeight(Tk_Window tkwin)`	Returns the height specified in the most recent call to `Tk_GeometryRequest` for `tkwin` (this is a macro, not a procedure).
`int Tk_ReqWidth(Tk_Window tkwin)`	Returns the width specified in the most recent call to `Tk_GeometryRequest` for `tkwin` (this is a macro, not a procedure).
`int Tk_InternalBorderWidth(Tk_Window tkwin)`	Returns the border width specified in the most recent call to `Tk_InternalBorderWidth` for `tkwin` (this is a macro, not a procedure).
`Tk_MapWindow(Tk_Window tkwin)`	Arranges for `tkwin` to be displayed on the screen whenever its ancestors are mapped.
`Tk_UnmapWindow(Tk_Window tkwin)`	Prevents `tkwin` and its descendants from appearing on the screen.
`Tk_MoveWindow(Tk_Window tkwin, int x, int y)`	Positions `tkwin` so that its upper-left pixel (including any borders) appears at coordinates `x` and `y` in its parent.
`Tk_MoveResizeWindow(Tk_Window tkwin, int x, int y, unsigned int width, unsigned int height)`	Changes `tkwin`'s position within its parent and also its size.
`Tk_ResizeWindow(Tk_Window tkwin, unsigned int width, unsigned int height)`	Sets the inside dimensions of `tkwin` (not including its external border, if any) to `width` and `height`.

Table 43.1. A summary of Tk's procedures for geometry management.

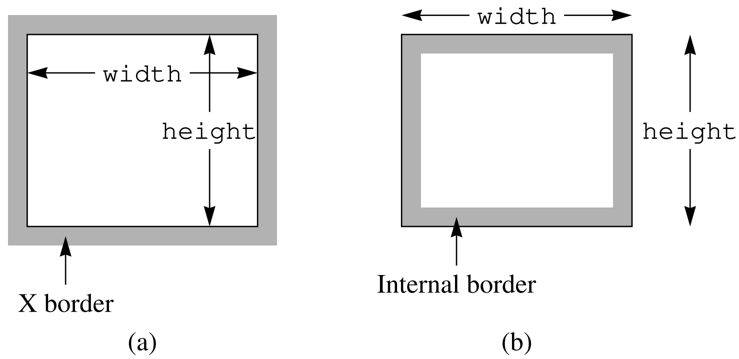


Figure 43.1. X borders and internal borders. (a) shows an official X border, which is drawn by X outside the area of the window. (b) shows an internal border drawn by a widget, where the area occupied by the border is part of the window's official area. In both figures `width` and `height` are the official X dimensions of the window.

addition, geometry managers will sometimes call `Tk_GeometryRequest` on a window's behalf. For example, the packer resets the requested size for each master window that it manages to match the needs of all of its slaves. This overrides the requested size set by the widget and results in the shrink-wrap effects shown in Chapter XXX.

43.2 Internal borders

The X window system allows each window to have a border that appears just outside the window. The official height and width of a window are the inside dimensions, which describe the usable area of the window and don't include the border. Unfortunately, though, X requires the entire border of a window to be drawn with a single solid color or stipple. To achieve the Motif three-dimensional effects, the upper and left parts of the border have to be drawn differently than the lower and right parts. This means that X borders can't be used for Motif widgets. Instead, Motif widgets draw their own borders, typically using Tk procedures such as `Tk_Draw3DRectangle`. The border for a Motif widget is drawn around the perimeter of the widget but inside the official X area of the widget. This kind of border is called an *internal border*. Figure 43.1 shows the difference between external and internal borders.

If a widget has an internal border then its usable area (the part that's inside the border) is smaller than its official X area. This complicates geometry management in two ways. First, each widget has to include the border width (actually, twice the border width) in the width and height that it requests via `Tk_GeometryRequest`. Second, if a master window has an internal border then geometry managers should not place slave windows on top of the border; the usable area for arranging slaves should be the area inside the border. In order for this to happen the geometry managers must know about the presence of the internal border. The procedure `Tk_SetInternalBorder` is provided for this purpose:

```
void Tk_SetInternalBorder(Tk_Window tkwin, int width);
```

This tells geometry managers that `tkwin` has an internal border that is `width` pixels wide and that slave widgets should not overlap the internal border. Widgets with internal borders normally call `Tk_SetInternalBorder` in their configure procedures at the same time that they call `Tk_GeometryRequest`. If a widget uses a normal X border, or if it has an internal border but doesn't mind slaves being placed on top of the border, then it need not call `Tk_SetInternalBorder`, or it can call it with a `width` of 0.

43.3 Grids

Gridded geometry management was introduced in Section XXX. The goal is to allow the user to resize a top-level window interactively, but to constrain the resizing so that the window's dimensions always lie on a grid. Typically this means that a particular subwindow displaying fixed-width text always has a width and height that are an integral number of characters. The window manager implements constrained resizes, but the application must supply it with the geometry of the grid. In order for this to happen, the widget that determines the grid geometry must call `Tk_SetGrid`:

```
void Tk_SetGrid(Tk_Window tkwin, int gridWidth, int
gridHeight,
          int widthInc, int heightInc);
```

The `gridWidth` and `gridHeight` arguments specify the number of grid units corresponding to the pixel dimensions requested in the most recent call to `Tk_GeometryRequest`. They allow the window manager to display the window's current size in grid units rather than pixels. The `widthInc` and `heightInc` arguments specify the number of pixels in a grid unit. Tk passes all of this information on to the window manager, and it will then constrain interactive resizes so that `tkwin`'s top-level window always has dimensions that lie on a grid defined by its requested geometry, `gridWidth`, and `gridHeight`.

Widgets that support gridding, such as texts, normally have a `-setgrid` option . If `-setgrid` is 0 then the widget doesn't call `Tk_SetGrid`; this is done if gridded resizing isn't wanted (e.g. the widget uses a variable-width font) or if some other widget in the top-level window is to be the one that determines the grid. If `-setgrid` is 1 then the widget calls `Tk_SetGrid`; typically this happens in the configure procedure at the same time that other geometry-related calls are made. If the widget's grid geometry changes (for example, its font might change) then the widget calls `Tk_SetGrid` again.

43.4 Geometry managers

The remainder of this chapter describes the Tk library procedures that are used by geometry managers. It is intended to provide the basic information that you need to write a new geometry manager. This section provides an overview of the structure of a geometry manager and the following sections describe the Tk library procedures.

A typical geometry manager contains four main procedures. The first procedure is a command procedure that implements the geometry manager's Tcl command. Typically each geometry manager provides a single command that is used by the application designer to provide information to the geometry manager: `pack` for the packer, `place` for the placer, and so on. The command procedure collects information about each slave and master window managed by the geometry manager and allocates a C structure for each window to hold the information. For example, the packer uses a structure with two parts. The first part is used if the window is a master; it includes information such as a list of slaves for that master. The second part is used if the window is a slave; it includes information such as the side against which the slave is to be packed and padding and filling information. If a window is both a master and a slave then both parts are used. Each geometry manager maintains a hash table (using Tcl's hash table facilities) that maps from widget names to the C structure for geometry management.

The second procedure for a geometry manager is its *layout procedure*. This procedure contains all of the actual geometry calculations. It uses the information in the structures created by the command procedure, plus geometry information provided by all of the slaves, plus information about the current dimensions of the master. The layout procedure typically has two phases. In the first phase it scans all of the slaves for a master, computes the ideal size for the master based on the needs of its slaves, and calls `Tk_GeometryRequest` to set the requested size of the master to the ideal size. This phase only exists for geometry managers like the packer that reflect geometry information upwards through the widget hierarchy. For geometry managers like the placer, the first phase is skipped. In the second phase the layout procedure recomputes the geometries for all of the slaves of the master.

The third procedure is a *request callback* that Tk invokes whenever a slave managed by the geometry manager calls `Tk_GeometryRequest`. The callback arranges for the layout procedure to be executed, as will be described below.

The final procedure is an event procedure that is invoked when a master window is resized or when a master or slave window is destroyed. If a master window is resized then the event procedure arranges for the layout procedure to be executed to recompute the geometries of all of its slaves. If a master or slave window is destroyed then the event procedure deletes all the information maintained by the geometry manager for that window. The command procedure creates event handlers that cause the event procedure to be invoked.

The layout procedure must be invoked after each call to the command procedure, the request callback, or the event procedure. Usually this is done with an idle callback, so that the layout procedure doesn't actually execute until all pending work is completed. Using an idle callback can save a lot of time in situations such as the initial creation of a complex panel. In this case the command procedure will be invoked once for each of many slave windows, but there won't be enough information to compute the final layout until all of the invocations have been made for all of the slaves. If the layout procedure were invoked immediately it would just waste time computing layouts that will be discarded almost immediately. With the idle callback, layout is deferred until complete information is available for all of the slaves.

43.5 Claiming ownership

A geometry manager uses the procedure `Tk_ManageGeometry` to indicate that it wishes to manage the geometry for a given slave window:

```
void Tk_ManageGeometry(Tk_Window tkwin, Tk_GeometryProc *proc,
        ClientData clientData);
```

From this point on, whenever `Tk_GeometryRequest` is invoked for `tkwin`, Tk will invoke `proc`. There can be only one geometry manager for a slave at a given time, so any previous geometry manager is cancelled. A geometry manager can also disown a slave by calling `Tk_ManageGeometry` with a null value for `proc`. `Proc` must match the following prototype:

```
typedef void Tk_GeometryProc(ClientData clientData,
        Tk_Window tkwin);
```

The `clientData` and `tkwin` arguments will be the same as those passed to `Tk_ManageGeometry`. Usually `Tk_ManageGeometry` is invoked by the command procedure for a geometry manager, and usually `clientData` is a pointer to the structure holding the geometry manager's information about `tkwin`.

43.6 Retrieving geometry information

When a widget calls `Tk_GeometryRequest` or `Tk_SetInternalBorder` Tk saves the geometry information in its data structure for the widget. The geometry manager's layout procedure can retrieve the requested dimensions of a slave with the macros `Tk_ReqWidth` and `Tk_ReqHeight`, and it can retrieve the width of a master's internal border with the macro `Tk_InternalBorderWidth`. It can also retrieve the master's actual dimensions with the `Tk_Width` and `Tk_Height` macros, which were originally described in Section 37.5.

Note: *Geometry managers need not worry about the gridding information provided with the* `Tk_SetGrid` *procedure. This information doesn't affect geometry managers at all. It is simply passed on to the window manager for use in controlling interactive resizes.*

43.7 Mapping and setting geometry

A geometry manager does two things to control the placement of a slave window. First, it determines whether the slave window is mapped or unmapped, and second, it sets the size and location of the window.

X allows a window to exist without appearing on the screen. Such a window is called *unmapped*: neither it nor any of its descendants will appear on the screen. In order for a window to appear, it and all of its ancestors (up through the nearest top-level window) must be *mapped*. All windows are initially unmapped. When a geometry manager takes responsibility for a window it must map it by calling `Tk_MapWindow`:

```
void Tk_MapWindow(Tk_Window tkwin);
```

Usually the geometry manager will call `Tk_MapWindow` in its layout procedure once it has decided where the window will appear. If a geometry manager decides not to manage a window anymore (e.g. in the "`pack forget`" command) then it must unmap the window to remove it from the screen:

```
void Tk_UnmapWindow(Tk_Window tkwin);
```

Some geometry managers may temporarily unmap windows during normal operation. For example, the packer unmaps a slave if there isn't enough space in its master to display it; if the master is enlarged later then the slave will be mapped again.

Tk provides three procedures that a geometry manager's layout procedure can use to position slave windows:

```
void Tk_MoveWindow(Tk_Window tkwin, int x, int y);
void Tk_ResizeWindow(Tk_Window tkwin, unsigned int width,
        unsigned int height);
void Tk_MoveResizeWindow(Tk_Window tkwin, int x, int y,
        unsigned int width, unsigned int height);
```

`Tk_MoveWindow` moves a window so that its upper left corner appears at the given location in its parent; `Tk_ResizeWindow` sets the dimensions of a window without moving it; and `Tk_MoveResize` both moves a window and changes its dimensions.

The position specified to `Tk_MoveWindow` or `Tk_MoveResizeWindow` is a position in the slave's parent. However, most geometry managers allow the master for a slave to be not only its parent but any descendant of the parent. Typically the layout procedure will compute the slave's location relative to its master; before calling `Tk_MoveWindow` or `Tk_MoveResizeWindow` it must translate these coordinates to the coordinate system of the slave's parent. The following code shows how to transform coordinates `x` and `y` from the master to the parent, assuming that `slave` is the slave window and `master` is its master:

```
int x, y;
Tk_Window slave, master, parent, ancestor;
...
for (ancestor = master; ancestor != Tk_Parent(slave);
        ancestor = Tk_Parent(ancestor)) {
```

```
        x += Tk_X(ancestor) + Tk_Changes(ancestor)->border_width;
        y += Tk_Y(ancestor) + Tk_Changes(ancestor)->border_width;
}
```