

10.4 Inputs without Waiting

The previous programs for activating input devices and handling the events generated by them all used the same form: There is a single function that solicits one event at a time; the function waits for that event (e.g., with `waitforbuttonpress` or `menu`), and when the event occurs the event is processed. While waiting for the event, no computations are performed, and no other actions taken.

This form has the advantage of keeping events in a strict sequence. In `mandelbrotGUI`, for instance, the mouse could not be used for marking a region until the `zoom/pan/quit` menu selection was pending.

The single-function form of user interface is relatively simple to program. For simple kinds of input (selecting from a set of alternatives or selecting a region in a graphics display) it can provide an effective interface. But there are often situations where inputs take on more varied forms. For instance, in a word processor, you might use the mouse to select a region of text (click and drag), to place the insert cursor at a particular point in the text (simple click), or to activate any of numerous menu items. At the same time, you can be typing (providing input via the keyboard). Other actions might be going on in the background, such as saving text to a backup file or printing.

We can imagine a word processor that uses the one-event-at-a-time style: The user would first indicate whether she wants to insert text, mark a region, or so on. Then the mouse would be activated for that purpose. The first word processors actually worked this way (although without a mouse). It is not necessarily old-fashioned—the professional-level text editor `EMACS` does not need the mouse at all and simply solicits and handles text input. For instance, to move to the previous line in a body of text, one types `CNTRL-p`. But most users consider programs like `EMACS` unfriendly and too difficult to use.

Or, consider a computer game. Even when no user inputs are being given, the state of the system (the location of the enemy entities in a shoot-'em-up game, for instance) is changing. If computation needs to wait for the next event, how does the state change between events?

In a no-waiting interface such as used by a word processor or a game, many different types of events might occur at any time. To deal with complex situation, a different style of programming is called for. Rather than a single function handling all the possible events, a set of functions is created with one function for each type of event. When an event occurs, the corresponding function—termed a *callback function*—is invoked. The callback function can change the state, update the display, generate new events, and so on.

This form of system, where control is distributed across many functions, can greatly simplify the process of handling events. New types of events can

Key Term

be added by creating a new callback function and without necessarily altering any existing function. Nonetheless, there are significant challenges:

- Maintaining the state of the system in a way that is accessible by each of the many callback functions that are needed to read and update the state. In MATLAB, whose scoping rules don't allow functions to access data in another function's environment, this often means using global variables. In other languages, references would be a more appropriate way to allow access.
- Ensuring that each callback function updates the state in a valid way and that the display is consistent with the state.

A Matter of Style:

Meeting the challenges of implementing a callback system can lead to a system that is easy to use. But the programming difficulties are substantial and the programs can become complex. The single-function approach does create a clumsy and limited user interface, but it usually is much easier to write. The user interface examples in the rest of this book employ the single-function approach.

But the flexibility and power of the callback style have tremendous advantages when implementing a user interface. Commercial software almost always uses this style for user interfaces, and most computer programming systems support the callback style.

In deciding between the two for your own work, consider whether a simple but clumsy interface will do the job for you, whether the interface will be used by others, and whether it will be used frequently or just once in a while. In particular, for prototype software, it may be best to use the single-function style until you have determined exactly what functionality you want to implement. But there is one situation when the callback style is required: when computation must go on all of the time, even when there are no events. This is the case in computer games and other, perhaps more scientifically oriented, simulations.

As a simple example, consider a temperature converter as shown in Figure 10.2. The figure window is constructed by putting together several user interface control objects in the function `tempConvFig`:

```

[1] function [f,c,m] = tempConvFig
[2] % GUI for temperature converter
Create figure [3] fig = figure(1);
[4] set(fig,'units','centimeters',...
Set size and position [5] 'position',[5,5,5,3],...
[6] 'menubar','none');
Create editable display [7] f = uicontrol('style','edit',...
[8] 'units','centimeters','position',...
[9] [.5,2,2.5,.5],'string','');
```

```

Label for F box [10] flabel = uicontrol('style', 'text',...
[11]     'units', 'centimeters', 'position',...
[12]     [3.25, 2, 1, .5], 'string', 'Fahr');
Create editable display [13] c = uicontrol('style', 'edit',...
[14]     'units', 'centimeters', 'position',...
[15]     [.5, 1.25, 2.5, .5], 'string', '');
Label for C box [16] clabel = uicontrol('style', 'text',...
[17]     'units', 'centimeters', 'position',...
[18]     [3.25, 1.25, 1, .5], 'string', 'Celsius');
Display for messages [19] m = uicontrol('style', 'text',...
[20]     'units', 'centimeters',...
[21]     'position', [.5, .5, 4, .5],...
[22]     'string', '', 'ForegroundColor', [1, 0, 0]);
    
```

This function has been written to return three function handles: the parts of the display that will need to be updated in the temperature conversion process.

```
>> [f,c,m] = tempConvFig;
```

The returned arguments are handles to the fahrenheit and celsius text entry boxes and to a third display for messages.

These text entry boxes already have functionality; you can type and edit inside them in the familiar way. But doing so does not affect the other elements of the display. To implement this, we have to provide callback functions. We need a callback function for each type of event that might be generated. In our case, there are two such events: one for text entry in the fahrenheit box and one for text entry in the celsius box. These events will be triggered when the ENTER key is pressed in the text box, or when the screen cursor moves from inside the text box to outside it.

The first step in writing a callback function is to define the state of the system that the callback function will operate on. The callbacks will need to read the information in their own text entry box, convert this to the format



Figure 10.2. The temperature conversion graphical-user interface described in the text. (a) A successful conversion. (b) An error.

for the other box, and perhaps set the message text. To do this, the callback needs to be able to access the `f`, `c`, and `m` handles—no other information is needed. So the state will be the `f`, `c`, and `m` handles.

In order for the callback functions to be able to access the state, we will use a global variable, accessible from any function. This is a simple approach but incurs all of the problems described in Section 9.3.

```
>> global tempConvState;
>> tempConvState.f = f;
>> tempConvState.c = c;
>> tempConvState.m = m;
```

Here is the callback function to process text entry in the fahrenheit box. It reads the text in the fahrenheit box, converts the text to a number and the number from fahrenheit to celsius, converts the celsius number to a text string, and writes this in the celsius box. If the fahrenheit string cannot be successfully converted, a suitable message is displayed.

```
[1] function fahrenheitEntry
[2] global tempConvState;
[3]
[4] s = get(tempConvState.f, 'String');
Convert to a number [5] ftemp = str2num(s);
Conversion unsuccessful [6] if isempty(ftemp)
[7]     cstr = '';
[8]     mstr = 'Invalid Fahr entry';
[9] else
[10]     cstr = num2str( 5*(ftemp-32)/9 );
[11]     mstr = '';
[12] end
Update the display [13] set(tempConvState.c, 'String', cstr);
[14] set(tempConvState.m, 'String', mstr);
```

A similar callback, `celsiusEntry`, is written to process events arising from the celsius text box. No callback is needed for the message text since no events are generated there.

At this point, the system can be operated manually by invoking the callbacks on the command line. The callbacks are like any other function. In order to invoke them automatically when text is entered, we need to set the `Callback` property of the text boxes.

```
>> set(f, 'Callback', 'fahrenheitEntry');
>> set(c, 'Callback', 'celsiusEntry');
```

Ordinarily, all of these steps for creating the figure window, setting up the state of the system, and setting the callbacks would be packaged up into a function. Here is one:

```

[1] function tempConvGUI
[2] % a simple temperature conversion window
Set up the state [3] global tempConvState;
[4] [f,c,m] = tempConvFig;
[5] tempConvState.f = f;
[6] tempConvState.c = c;
[7] tempConvState.m = m;
Set callbacks [8] set(f,'Callback', 'fahrenheitEntry');
[9] set(c,'Callback', 'celsiusEntry');
    
```

Invoking this function displays the temperature-conversion window. Note that the command prompt returns immediately after the function is invoked. The temperature-conversion window is still active, and you can also use the command line for any other purpose. Editing text in one of the temperature windows invokes the appropriate callback just as if you had typed the callback at the command prompt. The callback style of interface allows multiple events to be processed, even in different windows.

There are many software tools available to simplify the process of writing a graphical-user interface. MATLAB includes a tool called `guide` that makes it easy to lay out the user interface controls, set their properties, and set up callbacks. `Guide` has a particularly clever system for organizing callback functions and passing state information to them. This is well described in the MATLAB documentation and other sources [22].

10.5 Exercises

Exercise 10.1:

The resolution of an ordinary wall clock is one second: The clock ticks once a second, and it's not possible to read off fractional seconds. The system clock on a computer ticks somewhat more often. Write a program to determine the resolution of the system clock. You can do this by calling `now` repeatedly in a loop and finding the smallest nonzero change in time between successive calls.

Exercise 10.2:

Modify `trafficTimer` to add

1. A command for saving the collected data to a file.
2. A means to add annotations to the data by selecting from a menu list of standard annotations (e.g., 'sunset', 'rainstorm', 'accident').

Exercise 10.3:

Make a table of the values given by `get(gcf, 'CurrentChar')` for the noncharacter keys (e.g., delete, insert, f5, and so on).