C H A P T E R 5

Screen layout

GUI layout is an often-misunderstood area; a programmer could conceivably waste a lot of time on it. In this chapter, the three geometry managers, `Pack`, `Grid` and `Place` are covered in detail. Some advanced topics, including approaches to variable-size windows and the attendant problems of maintaining visually attractive and effective interfaces, will be presented.

5.1 Introduction to layout

Geometry managers are responsible for controlling the size and position of widgets on the screen. In Motif, widget placement is handled by one of several manager widgets. One example is the Constraint Widget class which includes the `XmForm` widget. Here, layout is controlled by attaching the widget by one, or more, of the top, bottom, left or right sides to adjacent widgets and containers. By choosing the appropriate combinations of attachments, the programmer can control a number of behaviors which determine how the widget will appear when the window is grown or shrunk.

Tk provides a flexible approach to laying out widgets on a screen. X defines several manager class widgets but in Tk, three geometry managers may be used. In fact, it is possible to use the managers with each other (although there are some rather important rules about how one goes about this). Tk achieves this flexibility by exploiting the X behavior that says widget geometry is determined by the geometry managers and *not* by the widgets themselves. Like X, if you do not manage the widget, it will not be drawn on the screen, although it will exist in memory.

Geometry managers available to Tkinter are these: the Packer, which is the most commonly used manager; the Grid, which is a fairly recent addition to Tk; the Placer, which has the least popularity, but provides the greatest level of control in placing widgets. You will see examples of all three geometry managers throughout the book. The geometry managers are available on all architectures supported by Tkinter, so it is not necessary to know anything about the implementation of the architecture-dependent toolkits.

5.1.1 Geometry management

Geometry management is a quite complex topic, because a lot of negotiation goes on between widgets, their containers, windows and the supporting window manager. The aim is to lay out one or more *slave* widgets as subordinates of a *master* widget (some programmers prefer to refer to *child* widgets and *parents*). *Master* widgets are usually containers such as a `Frame` or a `Canvas`, but most widgets can act as masters. For example, place a button at the bottom of a frame. As well as simply locating slaves within masters, we want to control the behavior of the widget as more widgets are added or when the window is shrunk or grown.

The negotiation process begins with each slave widget requesting width and height adequate to display its contents. This depends on a number of factors. A button, for example, calculates its required size from the length of text displayed as the label and the selected font size and weight.

Next, the master widget, along with its geometry manager, determines the space available to satisfy the requested dimensions of the slaves. The available space may be more or less than the requested space, resulting in squeezing, stretching or overlapping of the widgets, depending on which geometry manager is being employed.

Next, depending on the design of the window, space within a master's *master* must be apportioned between all *peer* containers. The results depend on the geometry manager of the peer widgets.

Finally, there is negotiation between the toplevel widget (normally the toplevel shell) and the window manager. At the end of negotiations the available dimensions are used to determine the final size and location in which to draw the widgets. In some cases there may not be enough space to display all of the widgets and they may not be realized at all. Even after this negotiation has completed when a window is initialized, it starts again if any of the widgets change configuration (for example, if the text on a button changes) or if the user resizes the window. Fortunately, it is a lot easier to use the geometry managers than it is to discuss them!

A number of common schemes may be applied when a screen is designed. One of the properties of the Packer and to a lesser extent the Grid, is that it is possible to allow the geometry manager to determine the final size of a window. This is useful when a window is created dynamically and it is difficult to predict the population of widgets. Using this approach, the window changes size as widgets are added or removed from the display. Alternatively, the designer might use the Placer on a fixed-size window. It really depends on the effect that is wanted.

Let's start by looking at the Packer, which is the most commonly used manager.

5.2 Packer

The Packer positions slave widgets in the master by adding them one at a time from the outside edges to the center of the window. The Packer is used to manage rows, columns and combinations of the two. However, some additional planning may have to be done to get the desired effect.

The Packer works by maintaining a list of slaves, or the *packing list*, which is kept in the order that the slaves were originally presented to the Packer. Take a look at figure 5.1 (this figure is modeled after John Ousterhout's description of the Packer).

Figure 5.1(1) shows the space available for placing widgets. This might be within a frame or the space remaining after placing other widgets. The Packer allocates a parcel for the next slave to be processed by slicing off a section of the available space. Which side is allocated is determined by the options supplied with the pack request; in this example, the `side=LEFT` and `fill=Y` options have been specified. The actual size allocated by the Packer is determined by a number of factors. Certainly the size of the slave is a starting point, but the available space and any optional padding requested by the slave must be taken into account. The allocated parcel is shown in figure 5.1(2).

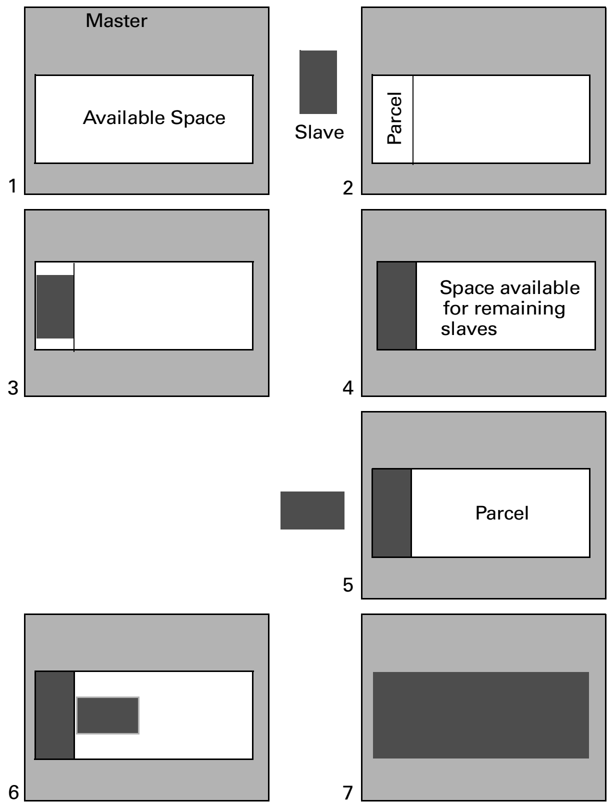


Figure 5.1 Packer operation

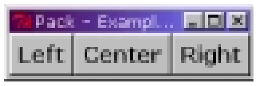


Figure 5.2 Pack geometry manager

Next, the slave is positioned within the parcel. If the available space results in a smaller parcel than the size of the slave, it may be squeezed or cropped, depending on the requested options. In this example, the slave is smaller than the available space and its height is increased to fill the available parcel. Figure 5.1(4) shows the available space for more slaves. In figure 5.1(5) we pack another slave with `side=LEFT` and `fill=BOTH` options. Again, the available parcel is larger than the size of the slave (figure 5.1(6)) so the widget is grown to fill the available space. The effect is shown in figure 5.1(7).

Here is a simple example of using the `pack` method, shown in figure 5.2:

Example_5_1.py

```
from Tkinter import *

class App:
   def __init__(self, master):
         Button(master, text='Left').pack(side=LEFT)
         Button(master, text='Center').pack(side=LEFT)      ❶
         Button(master, text='Right').pack(side=LEFT)

root = Tk()
root.option_add('*font', ('verdana', 12, 'bold'))
root.title("Pack - Example 1")
display = App(root)
root.mainloop()
```

Code comments

❶ The `side=LEFT` argument tells the Packer to start locating the widgets in the packing list from the left-hand side of the container. In this case the container is the default `Toplevel` shell created by the Tk initializer. The shell shrinks or expands to enclose the packed widgets.

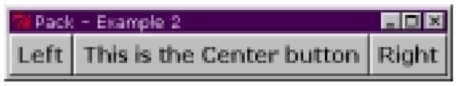
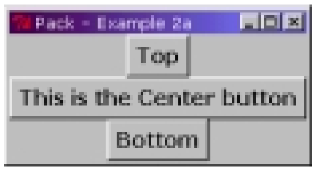


Figure 5.3 Packer accommodates requested widget sizes

Enclosing the widgets in a frame has no effect on the shrink-wrap effect of the Packer. In this example (shown in figure 5.3), we have increased the length of the text in the middle button and the frame is simply stretched to the requested size.

Example_5_2.py

```
         fm = Frame(master)
         Button(fm, text='Left').pack(side=LEFT)
         Button(fm, text='This is the Center button').pack(side=LEFT)
         Button(fm, text='Right').pack(side=LEFT)
         fm.pack()
```

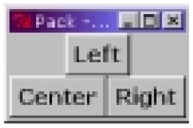


Packing from the top of the frame generates the result shown in figure 5.4. Note that the Packer centers the widgets in the available space since no further options are supplied and since the window is stretched to fit the widest widget.

Figure 5.4 Packing from the top side

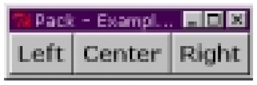
Example_5_2a.py

```
Button(fm, text='Top').pack(side=TOP)
Button(fm, text='This is the Center button').pack(side=TOP)
Button(fm, text='Bottom').pack(side=TOP)
```



Combining `side` options in the Packer list may achieve the desired effect (although more often than not you'll end up with an effect you did not plan on!). Figure 5.5 illustrates how unusual layouts may be induced.

Figure 5.5 Combining sides

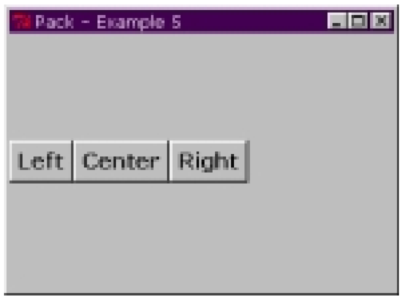


In all of these examples we have seen that the Packer negotiates the overall size of containers to fit the required space. If you want to control the size of the container, you will have to use *geometry* options, because attempting to change the `Frame` size (see example_5_4.py) has no effect as shown in figure 5.6.

Figure 5.6 Effect of changing frame size

Example_5_4.py

```
fm = Frame(master, width=300, height=200)
Button(fm, text='Left').pack(side=LEFT)
```



Sizing windows is often a problem when programmers start to work with Tkinter (and most other toolkits, for that matter) and it can be frustrating when there is no response as `width` and height options are added to widget specifications.

To set the size of the window, we have to make use of the `wm.geometry` option. Figure 5.7 shows the effect of changing the geometry for the root window.

Figure 5.6 Assigning the geometry of the Toplevel shell

Example_5_5.py

```
master.geometry("300x200")
```

5.2.1 Using the expand option

The `expand` option controls whether the Packer expands the widget when the window is resized. All the previous examples have accepted the default of `expand=NO`. Essentially, if `expand` is `true`, the widget *may* expand to fill the available space within its parcel; whether it does expand is controlled by the `fill` option (see “Using the fill option” on page 82).

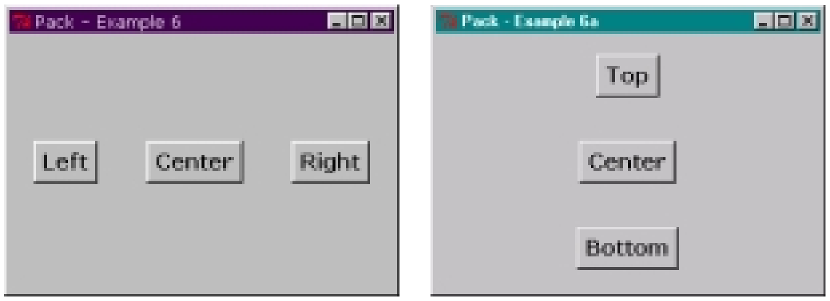


Figure 5.7 Expand without fill options

Example_5_6.py

```
Button(fm, text='Left').pack(side=LEFT, expand=YES)
Button(fm, text='Center').pack(side=LEFT, expand=YES)
Button(fm, text='Right').pack(side=LEFT, expand=YES)
```

Figure 5.7 shows the effect of setting expand to `true` (YES) without using the fill option (see Example_5_6.py). The vertical orientation in the second screen is similar to `side=TOP` (see Example_5_2a.py).

5.2.2 Using the fill option

Example_5_7.py illustrates the effect of combining `fill` and `expand` options; the output is shown in figure 5.9(1)

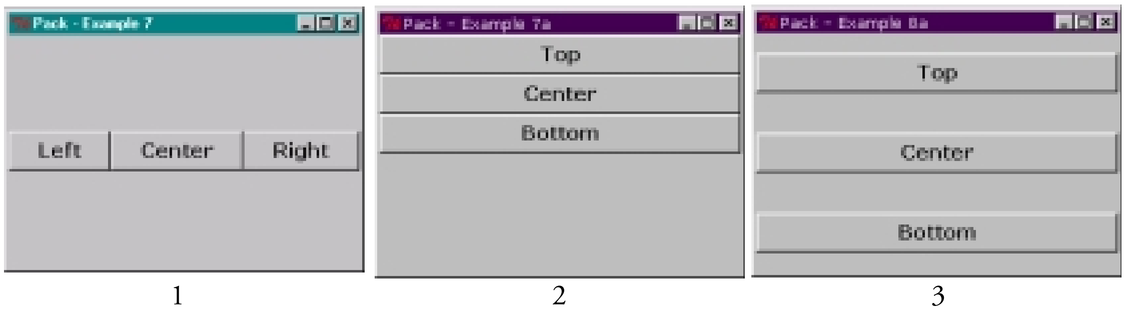


Figure 5.8 Using the fill option

Example_5_7.py

```
Button(fm, text='Left').pack(side=LEFT, fill=X, expand=YES)
Button(fm, text='Center').pack(side=LEFT, fill=X, expand=YES)
Button(fm, text='Right').pack(side=LEFT, fill=X, expand=YES)
```

If the `fill` option *alone* is used in Example_5_7.py, you will obtain a display similar to figure 5.9(2). By using `fill` and `expand` we see the effect shown in figure 5.9(3).

Varying the combination of `fill` and `expand` options may be used for different effects at different times. If you mix `expand` options, such as in example_5_8.py, you can allow some of the widgets to react to the resizing of the window while others remain a constant size. Figure 5.10 illustrates the effect of stretching and squeezing the screen.

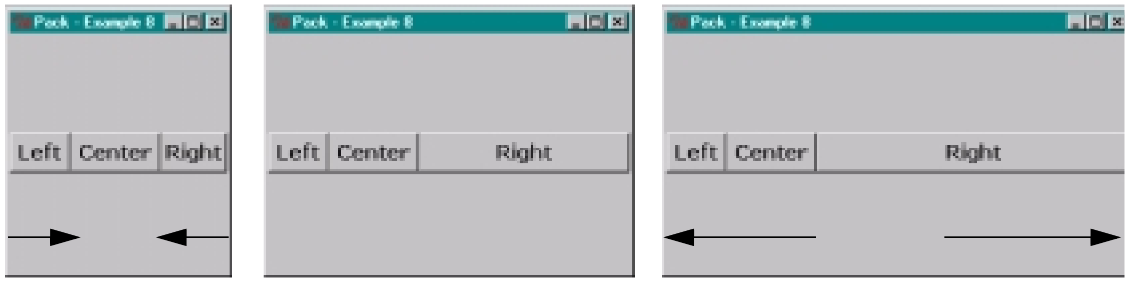


Figure 5.9 Allowing widgets to expand and fill independently

Example_5_8.py

```
Button(fm, text='Left').pack(side=LEFT, fill=X, expand=NO)
Button(fm, text='Center').pack(side=LEFT, fill=X, expand=NO)
Button(fm, text='Right').pack(side=LEFT, fill=X, expand=YES)
```

Using `fill=BOTH` allows the widget to use all of its parcel. However, it might create some rather ugly effects, as shown in figure 5.11. On the other hand, this behavior may be exactly what is needed for your GUI.

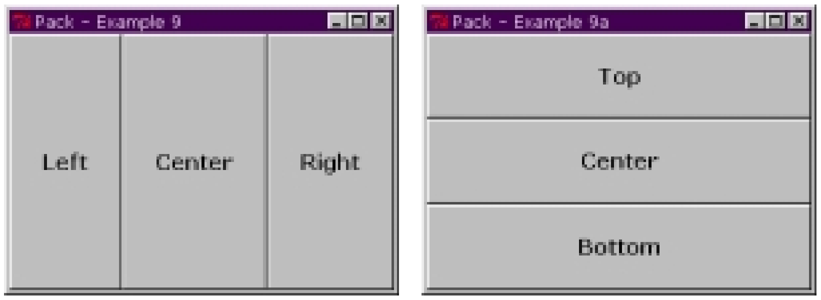


Figure 5.10 Using fill=BOTH

5.2.3 Using the padx and pady options

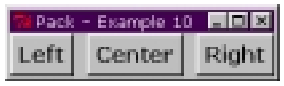


Figure 5.11 Using padx to create extra space

The `padx` and `pady` options allow the widget to be packed with additional space around it. Figure 5.12 shows the effect of adding `padx=10` to the pack request for the center button. Padding is applied to the specified left/right or top/bottom sides for `padx` and `pady` respectively. This may not achieve the effect you want, since if you place two widgets side by side, each with a `padx=10`, there will be 20 pixels between the two widgets and 10 pixels to the left and right of the pair. This can result in some unusual spacing.

5.2.4 Using the anchor option

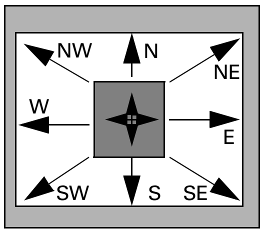


Figure 5.12 Anchoring a widget within the available space

The `anchor` option is used to determine where a widget will be placed within its parcel when the available space is larger than the size requested and none or one `fill` direction is specified. Figure 5.13 illustrates how a widget would be packed if an anchor is supplied. The option `anchor=CENTER` positions the widget at the center of the parcel. Figure 5.14 shows how this looks in practice.

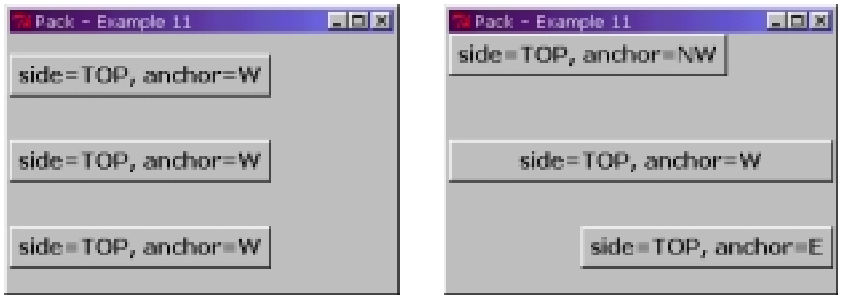


Figure 5.13 Using the anchor option to place widgets

5.2.5 Using hierarchical packing

While it is relatively easy to use the Packer to lay out simple screens, it is usually necessary to apply a hierarchical approach and employ a design which packs groups of widgets within frames and then packs these frames either alongside one other or inside other frames. This allows much more control over the layout, particularly if there is a need to fill and expand the widgets.

Figure 5.15 illustrates the result of attempting to lay out two columns of widgets. At first glance, the code appears to work, but it does not create the desired layout. Once you have packed a slave using `side=TOP`, the remaining space is below the slave, so you cannot pack alongside existing parcels.

Example_5_12.py

```
fm = Frame(master)
Button(fm, text='Top').pack(side=TOP, anchor=W, fill=X, expand=YES)
Button(fm, text='Center').pack(side=TOP, anchor=W, fill=X, expand=YES)
Button(fm, text='Bottom').pack(side=TOP, anchor=W, fill=X, expand=YES)
Button(fm, text='Left').pack(side=LEFT)
Button(fm, text='This is the Center button').pack(side=LEFT)
Button(fm, text='Right').pack(side=LEFT)
fm.pack()
```

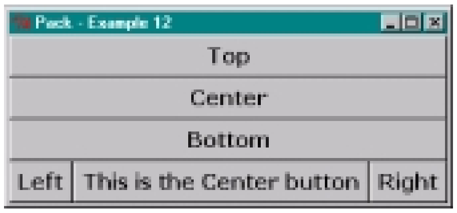


Figure 5.14 Abusing the Packer

All we have to do is to pack the two columns of widgets in separate frames and then pack the frames side by side. Here is the modified code:

Example_5_13.py

```
fm = Frame(master)
Button(fm, text='Top').pack(side=TOP, anchor=W, fill=X, expand=YES)
Button(fm, text='Center').pack(side=TOP, anchor=W, fill=X, expand=YES)
Button(fm, text='Bottom').pack(side=TOP, anchor=W, fill=X, expand=YES)
fm.pack(side=LEFT)
fm2 = Frame(master)
Button(fm2, text='Left').pack(side=LEFT)
Button(fm2, text='This is the Center button').pack(side=LEFT)
Button(fm2, text='Right').pack(side=LEFT)
fm2.pack(side=LEFT, padx=10)
```

Figure 5.16 shows the effect achieved by running Example_5_13.py.

This is an important technique which will be seen in several examples throughout the book. For an example which uses several embedded frames, take a look at Examples/chapter17/Example_16_9.py, which is available online.

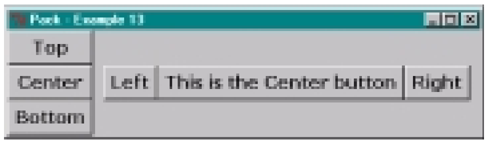


Figure 5.15 Hierarchical packing

5.3 Grid

Many programmers consider the `Grid` geometry manager the easiest manager to use. Personally, I don't completely agree, but you will be the final judge. Take a look at figure 5.17. This is a fairly complex layout task to support an image editor which uses a "by example" motif. Laying this out using the `Packer` requires a hierarchical approach with several nested Frames to enclose the target widgets. It also requires careful calculation of padding and other factors to achieve the final layout. It is much easier using the Grid.

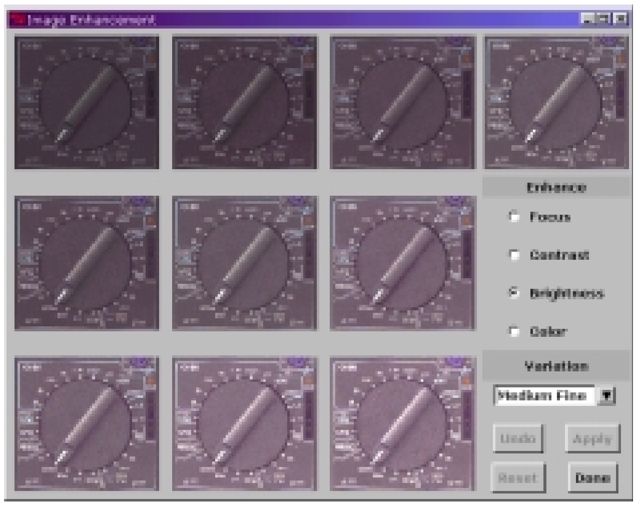


Figure 5.16 An image enhancer using Grid geometry management

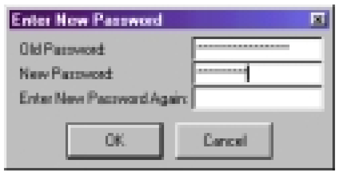


Figure 5.17 A dialog laid out using Grid

Before we tackle laying out the image editor, let's take a look at a simpler example. We'll create a dialog containing three labels with three entry fields, along with `OK` and `Cancel` buttons. The fields need to line up neatly (the example is a change-password dialog). Figure 5.18 shows what the Grid manager does for us. The code is quite simple, but I have removed some less-important lines for clarity:

Example_5_14.py

```
class GetPassword(Dialog):
    def body(self, master):
        self.title("Enter New Password")

        Label(master, text='Old Password:').grid(row=0, sticky=W)       1
        Label(master, text='New Password:').grid(row=1, sticky=W)
        Label(master, text='Enter New Password Again:').grid(row=2, sticky=W)
```

```
self.oldpw   = Entry(master, width = 16, show='*')
self.newpw1  = Entry(master, width = 16, show='*')     2
self.newpw2  = Entry(master, width = 16, show='*')

self.oldpw.grid(row=0, column=1, sticky=W)
self.newpw1.grid(row=1, column=1, sticky=W)            3
self.newpw2.grid(row=2, column=1, sticky=W)
```

Code comments

1 First, we create the labels. Since we do not need to preserve a reference to the label, we can apply the `grid` method directly. We specify the `row` number but allow the `column` to default (in this case to column 0). The `sticky` attribute determines where the widget will be attached within its cell in the grid. The `sticky` attribute is similar to a combination of the `anchor` and `expand` options of the Packer and it makes the widget look like a packed widget with an `anchor=W` option.

2 We *do* need a reference to the `entry` fields, so we create them separately.

3 Finally, we add the entry fields to the grid, specifying both `row` and `column`.

Let's go back to the image editor example. If you plan the layout for the fields in a grid it is easy to see what needs to be done to generate the screen. Look at figure 5.19 to see how the areas are to be gridded. The important feature to note is that we need to span both rows and columns to set aside the space for each of the components. You may find it convenient to sketch out designs for complex grids before committing them to code. Here is the code for the image editor. I have removed some of the code, since I really want to focus on the layout and not the operation of the application. The full source code for this example is available online.

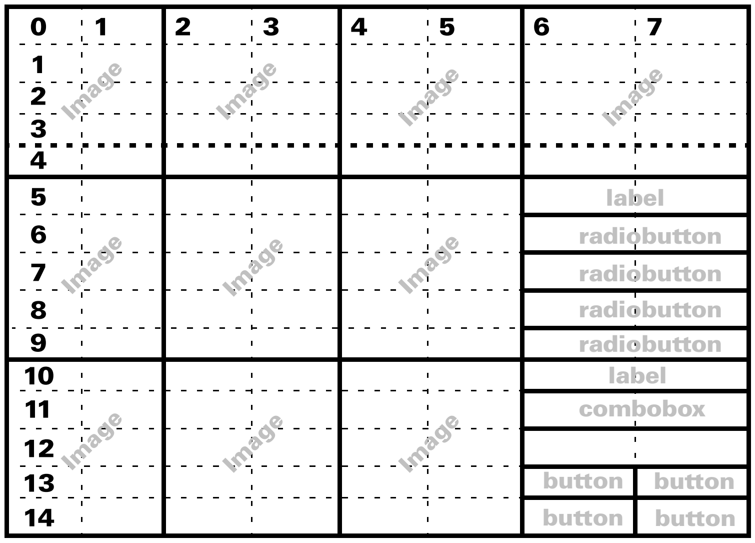


Figure 5.18 Designing the layout for a gridded display

imageEditor.py

```
from Tkinter import *
import sys, Pmw, Image, ImageTk, ImageEnhance                           ❶

class Enhancer:
  def __init__(self, master=None, imgfile=None):
       self.master = master
       self.masterImg = Image.open(imgfile)                             ❷
       self.masterImg.thumbnail((150, 150))

       self.images = [None]*9
       self.imgs   = [None]*9
       for i in range(9):                                              ❸
            image = self.masterImg.copy()
            self.images[i] = image
            self.imgs[i] = ImageTk.PhotoImage(self.images[i].mode,
                                               self.images[i].size)

       i = 0
       for r in range(3):
            for c in range(3):
                 lbl = Label(master, image=self.imgs[i])                  ❹
                 lbl.grid(row=r*5, column=c*2,
                      rowspan=5, columnspan=2,sticky=NSEW,
                      padx=5, pady=5)
                 i = i + 1

       self.original = ImageTk.PhotoImage(self.masterImg)
       Label(master, image=self.original).grid(row=0, column=6,
                                 rowspan=5, columnspan=2)
                                                                         ❺
       Label(master, text='Enhance', bg='gray70').grid(row=5, column=6,
                                      columnspan=2, sticky=NSEW)

       self.radio = Pmw.RadioSelect(master, labelpos = None,             ❻
                           buttontype = 'radiobutton', orient = 'vertical',
                           command = self.selectFunc)

       self.radio.grid(row=6, column=6, rowspan=4, columnspan=2)

# --- Code Removed ------------------------------------------------------

       Label(master, text='Variation',
             bg='gray70').grid(row=10, column=6,
                      columnspan=2, sticky=NSWE)

       self.variation=Pmw.ComboBox(master, history=0, entry_width=11,
                          selectioncommand = self.setVariation,
                          scrolledlist_items=('Fine','Medium Fine','Medium',
                                      'Medium Course','Course'))

       self.variation.selectitem('Medium')

       self.variation.grid(row=11, column=6, columnspan=2)
```

```
        Button(master, text='Undo',
               state='disabled').grid(row=13, column=6)            7

        Button(master, text='Apply',
               state='disabled').grid(row=13, column=7)
        Button(master, text='Reset',
               state='disabled').grid(row=14, column=6)
        Button(master, text='Done',
               command=self.exit).grid(row=14, column=7)

# --- Code Removed -------------------------------------------------

root = Tk()
root.option_add('*font', ('verdana', 10, 'bold'))
root.title('Image Enhancement')
imgEnh = Enhancer(root, sys.argv[1])
root.mainloop()
```

Code comments

1 This example uses the Python Imaging Library (PIL) to create, display, and enhance images. See “Python Imaging Library (PIL)” on page 626 for references to documentation supporting this useful library of image methods.

2 Although it’s not important in illustrating the grid manager, I left some of the PIL code in place to demonstrate how it facilitates handling images. Here, in the constructor, we open the master image and create a thumbnail within the bounds specified. PIL scales the image appropriately.

```
self.masterImg = Image.open(imgfile)
self.masterImg.thumbnail((150, 150))
```

3 Next we create a copy of the image and create a Tkinter PhotoImage placeholder for each of the images in the 3x3 grid.

4 Inside a double `for` loop we create a `Label` and place it in the appropriate cell in the grid, adding `rowspan` and `columnspan` options.

```
lbl = Label(master, image=self.imgs[i])
lbl.grid(row=r*5, column=c*2,
     rowspan=5, columnspan=2,sticky=NSEW, padx=5,pady=5)
```

Note that in this case the `sticky` option attaches the images to all sides of the grid so that the grid is sized to constrain the image. This means that the widget will stretch and shrink as the overall window size is modified.

5 Similarly, we grid a label with a different background, using the `sticky` option to fill all of the available cell.

```
Label(master, text='Enhance', bg='gray70').grid(row=5, column=6,
                              columnspan=2, sticky=NSEW)
```

6 The Pmw `RadioSelect` widget is placed in the appropriate cell with appropriate spans:

```
self.radio = Pmw.RadioSelect(master, labelpos = None,
                   buttontype = 'radiobutton', orient = 'vertical',
                   command = self.selectFunc)

self.radio.grid(row=6, column=6, rowspan=4, columnspan=2)
```

7 Finally, we place the `Button` widgets in their allocated cells.

You have already seen one example of the ImageEditor in use (figure 5.17). The real advantage of the grid geometry manager becomes apparent when you run the application with another image with a different aspect. Figure 5.20 shows this well; the grid adjusts perfectly to the image. Creating a similar effect using the Packer would require greater effort.

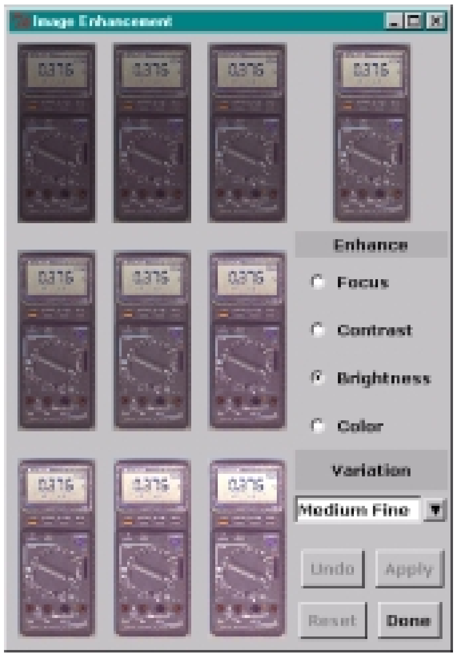


Figure 5.19 ImageEditor—scales for image size

5.4 Placer

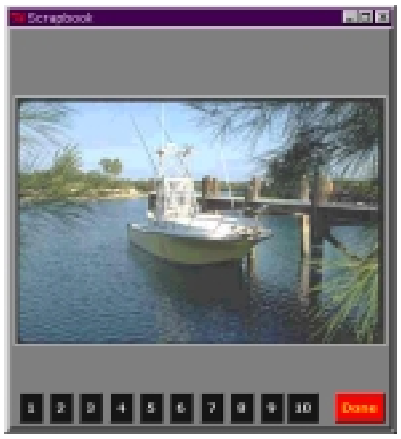


Figure 5.20 A simple scrapbook tool

The Placer geometry manager is the simplest of the available managers in Tkinter. It is considered difficult to use by some programmers, because it allows precise positioning of widgets within, or relative to, a window. You will find quite a few examples of its use in this book so I could take advantage of this precision. Look ahead to figure 9.5 on page 213 to see an example of a GUI that would be fairly difficult to implement using `pack` or `grid`. Because we will see so many examples, I am only going to present two simple examples here.

Let’s start by creating the simple scrapbook window shown in figure 5.21. Its function is to display some images, which are scaled to fit the window. The images are selected by clicking on the numbered buttons. It is quite easy to build a little application like this; again, we use PIL to provide support for images.

It would be possible to use `pack` to lay out the window (and, of course, `grid` would work if the image spanned most of the columns) but `place` provides some useful behavior when windows are *resized*. The Buttons in figure 5.21 are attached to *relative* positions, which means that they stay in the same relative position as the dimensions of the window change. You express relative positions as a real number with 0.0 representing *minimum* x or y and 1.0 representing *maximum* x or y. The minimum values for the axes are conventional for window coordinates with x0 on the left of the screen and y0 at the top of the screen. If you run scrapbook.py, test the effect of squeezing and stretching the window and you will notice how the buttons reposition. If you squeeze too much you will cause the buttons to collide, but somehow the effect using `place` is more acceptable than the clipping that occurs with `pack`. Here is the code for the scrapbook.

scrapbook.py

```
from Tkinter import *
import Image, ImageTk, os

class Scrapbook:
   def __init__(self, master=None):
        self.master = master
        self.frame = Frame(master, width=400, height=420, bg='gray50',
                     relief=RAISED, bd=4)

        self.lbl = Label(self.frame)                                      ❶
        self.lbl.place(relx=0.5, rely=0.48, anchor=CENTER)

        self.images = []                                                  ❷
        images = os.listdir("images")

        xpos = 0.05
        for i in range(10):
             Button(self.frame, text='%d'%(i+1), bg='gray10',
              fg='white', command=lambda s=self, img=i: \                 ❸
              s.getImg(img)).place(relx=xpos, rely=0.99, anchor=S)
             xpos = xpos + 0.08
             self.images.append(images[i])

        Button(self.frame, text='Done', command=self.exit,                ❹
         bg='red', fg='yellow').place(relx=0.99, rely=0.99, anchor=SE)
        self.frame.pack()
        self.getImg(0)

   def getImg(self, img):                                                 ❺
        self.masterImg = Image.open(os.path.join("images",
                                  self.images[img]))
        self.masterImg.thumbnail((400, 400))
        self.img = ImageTk.PhotoImage(self.masterImg)
        self.lbl['image'] = self.img

   def exit(self):
        self.master.destroy()

root = Tk()
```

```
root.title('Scrapbook')
scrapbook = Scrapbook(root)
root.mainloop()
```

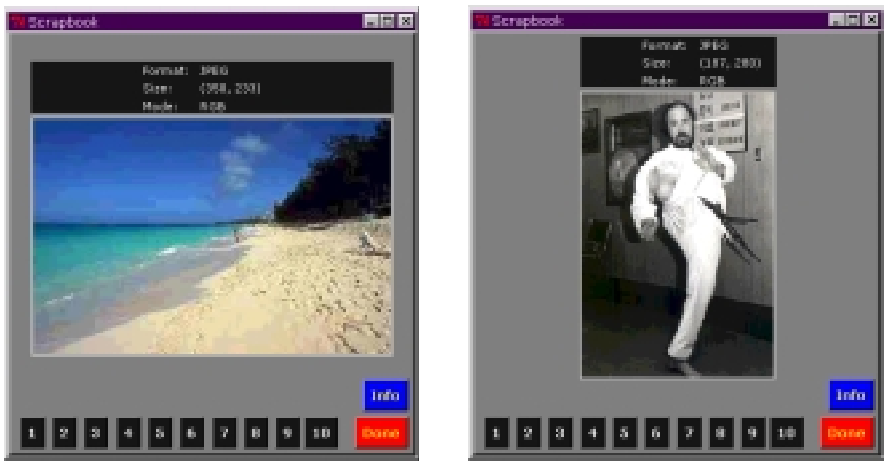
Code comments

1. We create the `Label` which will contain the image, placing it approximately in the center of the window and anchoring it at the center. Note that the relative placings are expressed as percentages of the width or height of the container.

    ```
    self.lbl.place(relx=0.5, rely=0.48, anchor=CENTER)
    ```

2. We get a list of files from the `images` directory
3. `place` really lends itself to be used for *calculated* positioning. In the loop we create a `Button`, binding the index of the button to the `activate` callback and placing the button at the next available position.
4. We put one button at the bottom right of the screen to allow us to **quit** the scrapbook. Note that we anchor it at the `SE` corner. Also note that we `pack` the outer frame. It is quite common to pack a group of widgets placed within a container. The Packer does all the work of negotiating the space with the outer containers and the window manager.
5. `getImg` is the PIL code to load the image, create a thumbnail, and load it into the `Label`.

In addition to providing precise window placement, `place` also provides *rubber sheet* placement, which allows the programmer to specify the size and location of the slave window in terms of the dimensions of the master window. It is even possible to use a master window which is *not* the parent of the slave. This can be very useful if you want to track the dimensions of an arbitrary window. Unlike `pack` and `grid`, `place` allows you to position a window outside the master (or sibling) window. Figure 5.22 illustrates the use of a window to display some of an image's properties in a window above each of the images. As the size of the image changes, the information window scales to fit the width of the image.



```
Format:  JPEG
Size:    (350, 233)
Mode:    RGB
```

```
Format:  JPEG
Size:    (187, 280)
Mode:    RGB
```

Figure 5.21 Adding a sibling window which tracks changes in attached window

The Placer has another important property: unlike the other Tkinter managers, it does not attempt to set the geometry of the master window. If you want to control the dimensions of container widgets, you must use widgets such as `Frames` or `Canvases` that have a `configure` option to allow you to control their sizes. Let's take a look at the code needed to implement the information window.

scrapbook2.py

```
from

Tkinter import *
import Image, ImageTk, os, string

class Scrapbook:
   def __init__(self, master=None):

# --- Code Removed ------------------------------------------------------

        Button(self.frame, text='Info', command=self.info,
         bg='blue', fg='yellow').place(relx=0.99, rely=0.90, anchor=SE)   1
        self.infoDisplayed = FALSE

   def getImg(self, img):

# --- Code Removed ------------------------------------------------------

        if self.infoDisplayed:                                             2
             self.info();self.info()

   def info(self):
        if self.infoDisplayed:
             self.fm.destroy()                                             4
             self.infoDisplayed = FALSE
        else:
             self.fm = Frame(self.master, bg='gray10')                     3
             self.fm.place(in_=self.lbl, relx=0.5,
                  relwidth=1.0, height=50, anchor=S,                       5
                  rely=0.0, y=-4, bordermode='outside')
             ypos = 0.15
             for lattr in ['Format', 'Size', 'Mode']:
                  Label(self.fm, text='%s:\t%s' % (lattr,
                       getattr(self.masterImg,
                            '%s' % string.lower(lattr))),
                       bg='gray10', fg='white',
                       font=('verdana', 8)).place(relx=0.3,                6
                       rely= ypos, anchor=W)
                  ypos = ypos + 0.35
             self.infoDisplayed = TRUE

# --- Code Removed ------------------------------------------------------
```

Code comments

1 We add a button to display the image information.

2 To force a refresh of the image info, we toggle the info display.

```
            self.info();self.info()
```

3 The `info` method toggles the information display.

4 If the window is currently displayed, we destroy it.

5 Otherwise, we create a new window, placing it above the image and setting its width to match that of the image. We also add a negative increment to the *y* position to provide a little whitespace.

```
            self.fm.place(in_=self.lbl, relx=0.5,
                 relwidth=1.0, height=50, anchor=S,
                 rely=0.0, y=-4, bordermode='outside')
```

6 The entries in the information window are placed programmatically.

5.5 Summary

Mastering the geometry managers is an important step in developing the ability to produce attractive and effective GUIs. When starting out with Tkinter, most readers will find `grid` and `pack` to be easy to use and capable of producing the best results when a window is resized. For very precise placement of widgets, `place` is a better choice. However, this does take quite a bit more effort.

You will see many examples of using the three managers throughout the book. Remember that it is often appropriate to combine geometry managers within a single window. If you do, you must be careful to follow some rules; if things are just not working out, then you have probably broken one of those rules!