7

In this chapter:
- *Menu Basics*
- *Working with Menus*
- *Multiple Menus*
- *Pop-up Menus*
- *Submenus*

Menus

Menus are the interface between the user and the program, and are the primary means by which a user carries out tasks. A Be program that makes use of menus usually places a menubar along the top of the content area of each application window—though it's easy enough to instead specify that a menubar appear elsewhere in a window.

A menu is composed of menu items, and resides in a menubar. You'll rely on the `BMenuBar`, `BMenu`, and `BMenuItem` classes to create menubar, menu, and menu item objects. Early in this chapter, you'll see how to create objects of these types and how to interrelate them to form a functioning menubar. After these menubar basics are described, the chapter moves to specific menu-related tasks such as changing a menu item's name during runtime and disabling a menu item or entire menu.

To offer the user a number of related options, create a single menu that allows only one item to be marked. Such a menu is said to be in radio mode, and places a checkmark beside the name of the most recently selected item. If these related items all form a subcategory of a topic that is itself a menu item, consider creating a submenu. A submenu is a menu item that, when selected, reveals still another menu. Another type of menu that typically holds numerous related options is a pop-up menu. A pop-up menu exists outside of a menubar, so it can be placed anywhere in a window. You'll find all the details of how to put a menu into radio mode, create a submenu, and create a pop-up menu in this chapter.

Menu Basics

A Be application can optionally include a menubar within any of its windows, as shown in Figure 7-1. In this figure, a document window belonging to the

StyledEdit program includes a menubar that holds four menus. As shown in the Font menu, a menu can include nested menus (submenus) within it.

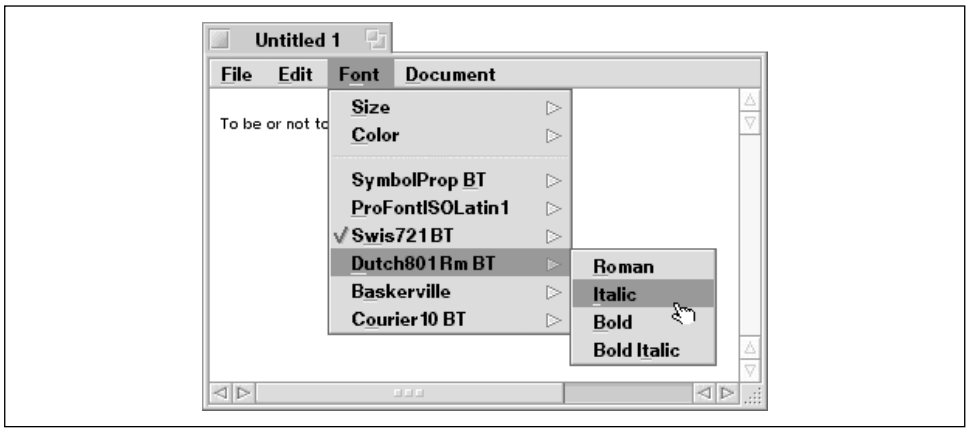

Figure 7-1. An application window can have its own menubar

Menus can be accessed via the keyboard rather than the mouse. To make the menubar the focus of keyboard keystrokes, the user presses both the Command and Escape keys. Once the menubar is the target of keystrokes, the left and right arrow keys can be used to drop, or display, a menu. Once displayed, items in a menu can be highlighted using the up and down arrow keys. The Enter key selects a highlighted item.

A second means of navigating menus and choosing menu items from the keyboard is through the use of *triggers*. One character in each menu name and in each menu item name is underlined. This trigger character is used to access a menu or menu item. After making the menubar the focus of the keyboard, pressing a menu's trigger character drops that menu. Pressing the trigger character of an item in that menu selects that item.

The topics of menubars, menus, and menu items are intertwined in such a way that moving immediately into a detailed examination of each in turn doesn't make sense. Instead, it makes more sense to conduct a general discussion of menu basics: creating menu item, menu, and menubar objects, adding menu item objects to a menu object, and adding a menu object to a menubar. That's what takes place on the next several pages. Included are a couple of example projects that include the code to add a simple menubar to a window. With knowledge of the interrelationship of the various menu elements, and a look at the code that implements a functional menubar with menus, it will be appropriate to move on to studies of the individual menu-related elements.

Adding a Menubar to a Window

The menubar, menu, and menu item are represented by objects of type `BMenuBar`, `BMenu`, and `BMenuItem`, respectively. To add these menu-related elements to your program, carry out the following steps:

1. Create a `BMenuBar` object to hold any number of menus.
2. Add the `BMenuBar` object to the window that is to display the menu.
3. For each menu that is to appear in the menubar:
    a. Create a `BMenu` object to hold any number of menu items.
    b. Add the `BMenu` object to the menubar that is to hold the menu.
    c. Create a `BMenuItem` object for each menu item that is to appear in the menu.

A menubar must be created before a menu can be added to it, and a menu must be created before a menu item can be added to it. However, the attaching of a menubar to a window and the attaching of a menu to a menubar needn't follow the order shown in the above list. For instance, a menubar, menu, and several menu items could all be created before the menu is added to a menubar.

When an example project in this book makes use of a menubar, its code follows the order given in the above list. It's worth noting that you will encounter programs that do things differently. Go ahead and rearrange the menu-related code in the `MyHelloWindow` constructor code in this chapter's first example project to prove that it doesn't matter when menu-related objects are added to parent objects.

Creating a menubar

The menubar is created through a call to the `BMenuBar` constructor. This routine accepts two arguments: a `BRect` that defines the size and location of the menubar, and a name for what will be the new `BMenuBar` object. Here's an example:

```
#define   MENU_BAR_HEIGHT   18.0

BRect     menuBarRect;
BMenuBar  *menuBar;

menuBarRect = Bounds();
menuBarRect.bottom = MENU_BAR_HEIGHT;

menuBar = new BMenuBar(menuBarRect, "MenuBar");
```

Convention dictates that a menubar appear along the top of a window's content area. Thus, the menubar's top left corner will be at point (0.0, 0.0) in window coordinates. The bottom of the rectangle defines the menu's height, which is typically 18 pixels. Because a window's menubar runs across the width of the window—regardless of the size of the window—the rectangle's right boundary can be set to the current width of the window the menubar is to reside in. The call to the `BWindow` member function `Bounds()` does that. After that, the bottom of the rectangle needs to be set to the height of the menu (by convention it's 18 pixels).

Creating a menubar object doesn't automatically associate that object with a particular window object. To do that, call the window's `BWindow` member function `AddChild()`. Typically a window's menubar will be created and added to the window from within the window's constructor. Carrying on with the above snippet, in such a case the menubar addition would look like this:

```
AddChild(menuBar);
```

Creating a menu

Creating a new menu involves nothing more than passing the menu's name to the `BMenu` constructor. For many types of objects, the object name is used strictly for "behind-the-scenes" purposes, such as in obtaining a reference to the object. A `BMenu` object's name is also used for that purpose, but it has a second use as well—it becomes the menu name that is displayed in the menubar to which the menu eventually gets attached. Here's an example:

```
BMenu  *menu;

menu = new BMenu("File");
```

Because one thinks of a menubar as being the organizer of its menus, it may be counterintuitive that the `BMenuBar` class is derived from the `BMenu` class—but indeed it is. A menubar object can thus make use of any `BMenu` member function, including the `AddItem()` function. Just ahead you will see how a menu object invokes `AddItem()` to add a menu item to itself. Here you see how a menubar object invokes `AddItem()` to add a menu to itself:

```
menuBar->AddItem(menu);
```

Creating a menu item

Each item in a menu is an object of type `BMenuItem`. The `BMenuItem` constructor requires two arguments: the menu item name as it is to appear listed in a menu,

and a BMessage object. Here's how a menu item to be used as an Open item in a File menu might be created:

```
#define    MENU_OPEN_MSG   'open'

BMenuItem  *menuItem;

menuItem = new BMenuItem("Open", new BMessage(MENU_OPEN_MSG));
```

Add the menu item to an existing menu by invoking the menu object's BMenu member function AddItem(). Here menu is the BMenu object created in the previous section:

```
menu->AddItem(menuItem);
```

While the above method of creating a menu item and adding it to a menu in two steps is perfectly acceptable, the steps are typically carried out in a single action:

```
menu->AddItem(new BMenuItem("Open", new BMessage(MENU_OPEN_MSG)));
```

Handling a Menu Item Selection

Handling a menu item selection is so similar to handling a control click that if you know one technique, you know the other. You're fresh from seeing the control (you either read Chapter 6, *Controls and Messages*, before this chapter, or you just jumped back and read it now, right?), so a comparison of menu item handling to control handling will serve well to cement in your mind the practice used in each case: message creation and message handling.

In Chapter 6, you read that to create a control, such as a button, you define a message constant and then use new along with the control's constructor to allocate both the object and the model message—as in this snippet that creates a standard push button labeled "OK":

```
#define    BUTTON_OK_MSG   'btmg'

BRect      buttonRect(20.0, 20.0, 120.0, 50.0);
BButton    *buttonOK;

buttonOK = new BButton(buttonRect, "OKButton",
                       "OK", new BMessage(BUTTON_OK_MSG));
```

Menu item creation is similar: define a message constant and then create a menu item object:

```
#define    MENU_OPEN_MSG   'open'

BMenuItem  *menuItem;

menuItem = new BMenuItem("Open", new BMessage(MENU_OPEN_MSG));
```

An application-defined message is sent from the Application Server to a window. The window receives the message in its `MessageReceived()` function. So the recipient window's `BWindow`-derived class must override `MessageReceived()`—as demonstrated in Chapter 6 and again here:

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage* message);
   ...
   ...
};
```

The implementation of `MessageReceived()` defines the action that occurs in response to each application-defined message. You saw several examples of this in Chapter 6, including a few projects that simply used `beep()` to respond to a message. Here's how `MessageReceived()` would be set up for a menu item message represented by a constant named `MENU_OPEN_MSG`:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{

   switch(message->what)
   {
      case MENU_OPEN_MSG:
         // open a file;
         break;

      default:
         BWindow::MessageReceived(message);
   }
}
```

Menubar Example Project

The SimpleMenuBar project generates a window that includes a menubar like the one in Figure 7-2. Here you see that the window's menubar extends across the width of the window, as expected, and holds a menu with a single menu item in it. Choosing the Beep Once item from the Audio menu sounds the system beep.

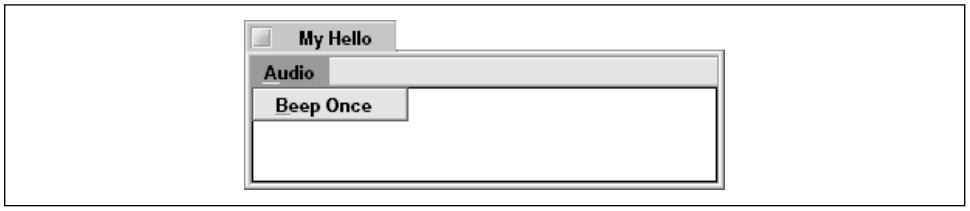

Figure 7-2. The SimpleMenuBar program's window

Preparing the window class for a menubar

If a window is to let a user choose items from a menubar, its `BWindow`-derived class must override `MessageReceived()`. Additionally, you may opt to keep track of the window's menubar by including a `BMenuBar` data member in the class. You can also include `BMenu` and `BMenuItem` data members in the class declaration, but keeping track of the menubar alone is generally sufficient. As demonstrated later in this chapter, it's a trivial task to find any menu or menu item and get a reference to it by way of a menubar reference. Here's how the window class header file (the *MyHelloWindow.h* file for this project) is set up for menu item handling:

```
#define   MENU_BEEP_1_MSG    'bep1'

class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage* message);

   private:
      MyDrawView     *fMyView;
      BMenuBar       *fMenuBar;
};
```

Creating the menubar, menu, and menu item

By default, the height of a menubar will be 18 pixels (though the system will automatically alter the menubar height to accommodate a large font that's used to display menu names). So we'll document the purpose of this number by defining a constant:

```
#define   MENU_BAR_HEIGHT   18.0
```

After the constant definition comes the `MyHelloWindow` constructor. In past examples, the first three lines of this routine created a view that occupies the entire content area of the new window. This latest version of the constructor uses the same three lines, but also inserts one new line after the call to `OffsetTo()`:

```
frame.OffsetTo(B_ORIGIN);
frame.top += MENU_BAR_HEIGHT + 1.0;

fMyView = new MyDrawView(frame, "MyDrawView");
AddChild(fMyView);
```

The `frame` is the `BRect` that defines the size and screen location of the new window. Calling the `BRect` function `OffsetTo()` with an argument of `B_ORIGIN` redefines the values of this rectangle's boundaries so that the rectangle remains the same overall size, but has a top left corner at window coordinate (0.0, 0.0). That's perfect for use when placing a new view in the window. Here, however, I want

the view to start not at the window's top left origin, but just below the menubar that will soon be created. Bumping the top of the `frame` rectangle down the height of the menu, plus one more pixel to avoid an overlap, properly sets up the rectangle for use in creating the view.

If you work on a project that adds a view and a menubar to a window, and mouse clicks on the menubar's menus are ignored, the problem most likely concerns the view. It's crucial to reposition a window's view so that it lies below the area that will eventually hold the menubar. If the view occupies the area that the menubar will appear in, the menubar's menus may not respond to mouse clicks. (Whether a menu does or doesn't respond will depend on the order in which the view and menubar are added to the window.) If the view overlaps the menubar, mouse clicks may end up directed at the view rather than the menubar.

The menubar is created by defining the bar's boundary and then creating a new `BMenuBar` object. A call to the `BWindow` function `AddChild()` attaches the menubar to the window:

```
BRect   menuBarRect;

menuBarRect = Bounds();
menuBarRect.bottom = MENU_BAR_HEIGHT;

fMenuBar = new BMenuBar(menuBarRect, "MenuBar");
AddChild(fMenuBar);
```

The menubar's one menu is created using the `BMenu` constructor. A call to the `BMenu` function `AddItem()` attaches the menu to the existing menubar:

```
BMenu   *menu;

menu = new BMenu("Audio");
fMenuBar->AddItem(menu);
```

A new menu is initially devoid of menu items. Calling the `BMenu` function `AddItem()` adds one item to the menu:

```
menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));
```

Subsequent calls to `AddItem()` append new items to the existing ones. Because the menu item won't be referenced later in the routine, and as a matter of convenience, the creation of the new menu item object is done within the call to `AddItem()`. We could expand the calls with no difference in the result. For instance, the above line of code could be written as follows:

```
BMenuItem  *theItem;

theItem = new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG));
menu->AddItem(theItem);
```

Here, in its entirety, is the `MyHelloWindow` constructor for the SimpleMenuBar project:

```
#define   MENU_BAR_HEIGHT   18.0

MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   frame.OffsetTo(B_ORIGIN);
   frame.top += MENU_BAR_HEIGHT + 1.0;

   fMyView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyView);

   BMenu  *menu;
   BRect  menuBarRect;

   menuBarRect.Set(0.0, 0.0, 10000.0, MENU_BAR_HEIGHT);
   fMenuBar = new BMenuBar(menuBarRect, "MenuBar");
   AddChild(fMenuBar);

   menu = new BMenu("Audio");
   fMenuBar->AddItem(menu);

   menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));

   Show();
}
```

Handling a menu item selection

To respond to the user's menu selection, all I did on this project was copy the `MessageReceived()` function that handled a click on a control in a Chapter 6 example project. The simplicity of this code sharing is further proof that a menu item selection is handled just like a control:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{

   switch(message->what)
   {
      case MENU_BEEP_1_MSG:
         beep();
         break;

      default:
         BWindow::MessageReceived(message);
   }
}
```

Window resizing and views

The SimpleMenuBar example introduces one topic that's only partially related to menus: how the resizing of a window affects a view attached to that window. A titled or document window (a window whose constructor contains a third parameter value of either `B_TITLED_WINDOW` or `B_DOCUMENT_WINDOW`) is by default resizable. (Recall that a `BWindow` constructor fourth parameter of `B_NOT_RESIZABLE` can alter this behavior.) The SimpleMenuBar window is resizable, so the behavior of the views within the window isn't static.

Like anything you draw, a menubar is a type of view. When a window that displays a menubar is resized, the length of the menubar is automatically altered to occupy the width of the window. This is a feature of the menubar, not your application-defined code.

Unlike a menubar, a `BView`-derived class needs to specify the resizing behavior of an instance of the class. This is done by supplying the appropriate Be-defined constant in the `resizingMode` parameter (the third parameter) to the `BView` constructor. In past examples, the `B_FOLLOW_ALL` constant was used for the `resizingMode`:

```
MyDrawView::MyDrawView(BRect rect, char *name)
   : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
}
```

The `B_FOLLOW_ALL` constant sets the view to be resized in conjunction with any resizing that takes place in the view's parent. If the view's parent is the window (technically, the window's top view) and the window is enlarged, the view will be enlarged proportionally. Likewise, if the window size is reduced, the view size is reduced. As a window is resized, it requires constant updating—so the `Draw()` function of each view in the window is repeatedly invoked. This may not always be desirable, as the SimpleMenuBar example demonstrates. This program's window is filled with a view of the class `MyDrawView`. In this project, the `Draw()` function for the `MyDrawView` class draws a rectangle around the frame of the view:

```
void MyDrawView::Draw(BRect)
{
   BRect  frame = Bounds();

   StrokeRect(frame);
}
```

If the `MyDrawView` view has a `resizingMode` of `B_FOLLOW_ALL`, the result of enlarging a window will be a number of framed rectangles in the window—one rectangle for each automatic call that's been made to `Draw()`. Figure 7-3 illustrates this.

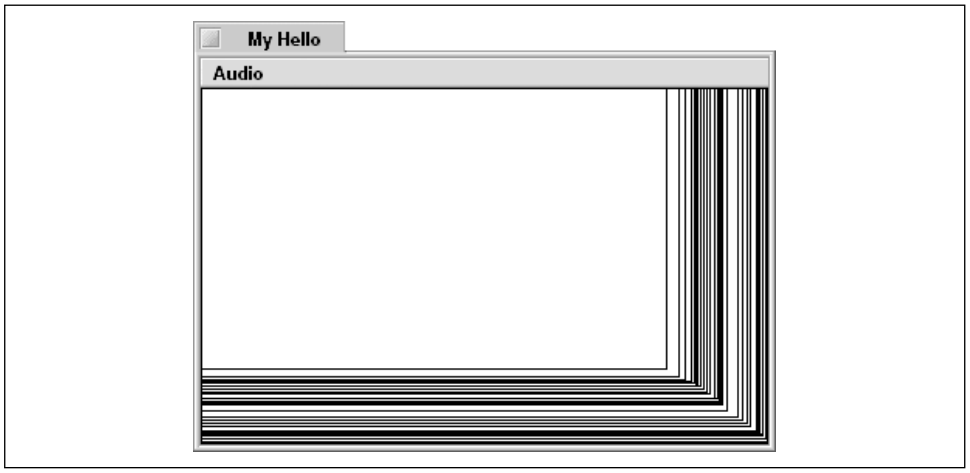

Figure 7-3. A view's resizing mode needs to be coordinated with window resizing

The SimpleMenuBar project avoids the above phenomenon by using the `B_FOLLOW_NONE` constant for the `resizingMode`:

```
MyDrawView::MyDrawView(BRect rect, char *name)
   : BView(rect, name, B_FOLLOW_NONE, B_WILL_DRAW)
{
}
```

This constant sets the view to remain a fixed size and at a fixed location in its parent—regardless of what changes take place in the parent's size. Figure 7-4 shows how the view in the SimpleMenuBar project's window looks when the program's window is enlarged.

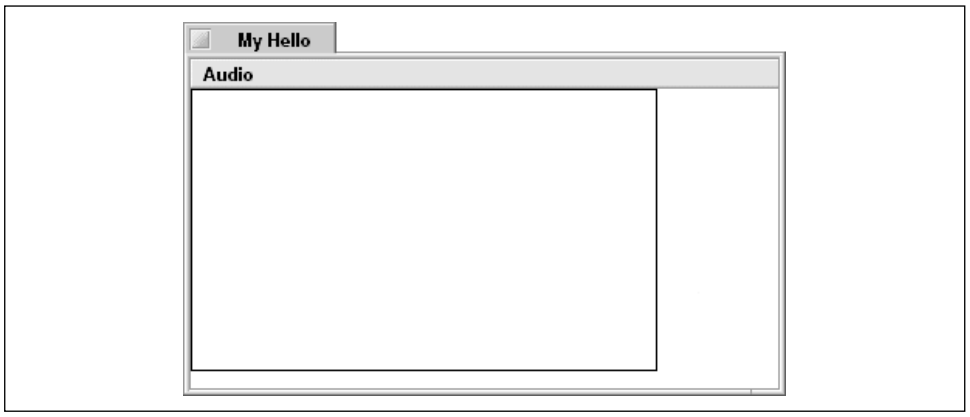

Figure 7-4. A fixed-size view is unaffected by window resizing

Menubar and Control Example Project

Now that you know how to add controls and menus to a window, there's a strong likelihood that you may want to include both within the same window. The MenuAndControl project demonstrates how to do this. As Figure 7-5 shows, the MenuAndControl program's window includes the same menubar that was introduced in the previous example (the SimpleMenuBar project). Sounding the system beep is accomplished by either choosing the one menu item or by clicking on the button. The view, which in this program doesn't occupy the entire window content area, remains empty throughout the running of the program. Here the view is used to elaborate on last section's discussion of window resizing and views. In this chapter's TwoMenus project, the view displays one of two drawings.

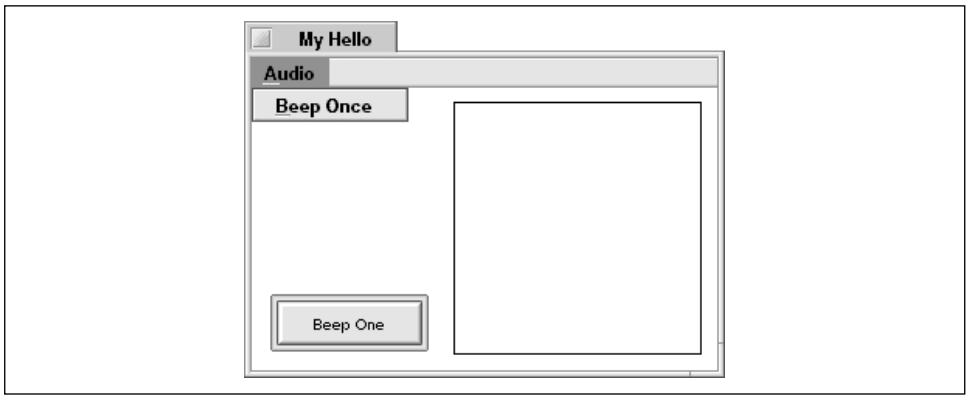

Figure 7-5. The MenuAndControl application window

Preparing the window class for a menubar and control

Both the push button control and the one menu item require the definition of a message constant:

```
#define   BUTTON_BEEP_MSG    'beep'
#define   MENU_BEEP_1_MSG    'bep1'
```

To handle messages from both the push button and the menu item, override `MessageReceived()`. Data members for the control and menubar appear in the `BWindow`-derived class as well:

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage* message);

   private:
      MyDrawView     *fMyView;
```

```
        BButton         *fButtonBeep;
        BMenuBar        *fMenuBar;
};
```

Creating the menu-related elements and the control

The `MyHelloWindow` constructor begins with the customary creation of a view for the window. Here, however, the view doesn't occupy the entire content area of the window. Recall that the SimpleMenuBar project set up the view's area like this:

```
frame.OffsetTo(B_ORIGIN);
frame.top += MENU_BAR_HEIGHT + 1.0;
```

The MenuAndControl project instead sets up the view's area as follows:

```
frame.Set(130.0, MENU_BAR_HEIGHT + 10.0, 290.0, 190.0);
```

Figure 7-5 shows that the resulting view occupies the right side of the program's window. Since a view in past examples occupied the entire content area of a window, items were added to the view in order to place them properly. For instance, if the `MyHelloWindow` defined a `BButton` data member named `fButtonBeep` and a `MyDrawView` data member named `fMyView`, the addition of the button to the window would look like this:

```
fMyView->AddChild(fButtonBeep);
```

The `MyHelloWindow` class declared in the MenuAndControl project does in fact include the two data members shown in the above line of code. This project's `MyHelloWindow` constructor, however, adds the button directly to the window rather than to the window's view. A call to the `BButton` function `MakeDefault()` serves to outline the button:

```
AddChild(fButtonBeep);
fButtonBeep->MakeDefault(true);
```

Looking back at Figure 7-5, you can see that in this project it wouldn't make sense to add the button to the window's view. If I did that, the button would end up being placed not on the left side of the window, but on the right side.

After adding the button to window, we create the menubar and add it to the window, create the menu and add it to the menubar, and create the menu item and add it to the menu. The menu-related code is identical to that used in the previous example (the SimpleMenuBar project). Note that there is no significance to my placing the control-related code before the menu-related code—the result is the same regardless of which component is added to the window first:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   frame.Set(130.0, MENU_BAR_HEIGHT + 10.0, 290.0, 190.0);
```

```
    fMyView = new MyDrawView(frame, "MyDrawView");
    AddChild(fMyView);

    fButtOnBeep = new BButton(buttonBeepRect, buttonBeepName,
                              buttonBeepLabEl, new BMessage(BUTTON_BEEP_MSG));
    AddChild(fButtonBeep);
    fButtonBeep->MakeDefault(true);

    BMenu  *menu;
    BRect  menuBarRect;

    menuBarRect.Set(0.0, 0.0, 10000.0, MENU_BAR_HEIGHT);
    fMenuBar = new BMenuBar(menuBarRect, "MenuBar");
    AddChild(fMenuBar);

    menu = new BMenu("Audio");
    fMenuBar->AddItem(menu);

    menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));

    Show();
}
```

Handling a menu item selection and a control click

It's important to keep in mind that "a message is a message"—a window won't distinguish between a message issued by a click on a control and a message generated by a menu item selection. So the same `MessageReceived()` function handles both message types:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_BEEP_MSG:
         beep();
         break;

      case MENU_BEEP_1_MSG:
         beep();
         break;

      default:
         BWindow::MessageReceived(message);
   }
}
```

Because a click on the Beep One button and a selection of the Beep Once menu item both result in the same action—a sounding of the system beep—I could have defined a single message constant. For instance, instead of defining both the `BUTTON_BEEP_MSG` and the `MENU_BEEP_1_MSG` constants, I could have simply defined, say, a `BEEP_MSG`:

```
#define   BEEP_MSG    'beep'
```

The One View Technique

Now you've seen numerous examples that establish one window-encompassing view, and one example that doesn't. Which method should you use? Sorry, but the answer is an ambiguous "It depends." It depends on whether your program will be making "universal" changes to a window, but it behooves you to get in the habit of always including a window-filling view in each of your `BWindow`-derived classes. If, much later in project development, you decide a window needs to be capable of handling some all-encompassing change or changes, you can just issue appropriate calls to the view and keep changes to your project's code to a minimum.

As an example of a universal change to a window, consider a window that displays several controls and a couple of drawing areas. If for some reason all of these items need to be shifted within the window, it would make sense to have all of the items attached to a view within the window rather than to the window itself. Then a call to the `BView` `MoveBy()` or `MoveTo()` member function easily shifts the window's one view, and its contents, within the window.

The second reason to include a window-filling view—programmer's preference—is related to the first reason. For each `BWindow`-derived class you define, you might prefer as a matter of habit to also define a `BView`-derived class:

```
class MyFillView : public BView {

   public:
                     MyDrawView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);
};
```

If you have no immediate plans for the view, simply implement the view class member functions as empty:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_NONE, B_WILL_DRAW)
{
}

void MyDrawView::AttachedToWindow()
{
}

void MyDrawView::Draw(BRect)
{
}
```

—Continued—

In the window's constructor, create and add a view. Then add all of the window's controls and other views to this main view:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   frame.OffsetTo(B_ORIGIN);

   fMyView = new MyFillView(frame, "MyFillWindowView");
   AddChild(fMyView);

        fButton = new BButton(buttonRect, buttonName,
                              buttonLabel, new BMessage(BUTTON_MSG));
   fMyView->AddChild(fButton);
   ...
   ...
}
```

The `BButton` constructor would then make use of this new message constant:

```
fButtonBeep = new BButton(buttonBeepRect, buttonBeepName,
                          buttonBeepLabel, new BMessage(BEEP_MSG));
```

The creation of the menu item makes use of this same message constant:

```
menu->AddItem(new BMenuItem("Beep Once", new BMessage(BEEP_MSG)));
```

A click on the button or a selection of the menu item would both result in the same type of message being sent to the window, so the `MessageReceived()` function would now need only one case label:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BEEP_MSG:
         beep();
         break;

      default:
         BWindow::MessageReceived(message);
   }
}
```

This scenario further demonstrates the notion that a window isn't interested in the source of a message—it cares only about the type of the message (as defined by the message constant). That's all well and good, but what's the likelihood of a real-world application having a window that includes both a control and a menu item that produce the same action? Perhaps higher than you might guess. It's a common practice in many programs to include a control window (usually referred

to as a palette) that as a matter of convenience holds a number of buttons that mimic the actions of commonly used menu items.

Window resizing and the view hierarchy

This chapter's first example, the SimpleMenuBar project, illustrated how window resizing affects a view. Including a control in the window of the MenuAndControl project provides an opportunity to illuminate resizing further.

A view created with a `resizingMode` of `B_FOLLOW_ALL` is one that is resized along with its resized parent. A `resizingMode` of `B_FOLLOW_NONE` fixes a view in its parent—even as the parent is resized. A view can also be kept fixed in size, but move within its parent. How it moves in relationship to the parent is dependent on which of the Be-defined constants `B_FOLLOW_RIGHT`, `B_FOLLOW_LEFT`, `B_FOLLOW_BOTTOM`, or `B_FOLLOW_TOP` are used for the `resizingMode`. Each constant forces the view to keep its present distance from one parent view edge. Constants can also be used in combination with one another by using the OR operator (|). In the MenuAndControl project, the `MyDrawView` constructor combines `B_FOLLOW_RIGHT` and `B_FOLLOW_BOTTOM`:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM, B_WILL_DRAW)
{
}
```

The result of this pairing of constants is a view that remains fixed to the parent view's (the window here) right and bottom. In Figure 7-6, the MenuAndControl window's size has been reduced horizontally and increased vertically, yet you see that the view has kept its original margin of about ten pixels from the window's right side and about ten pixels from the window's bottom edge.

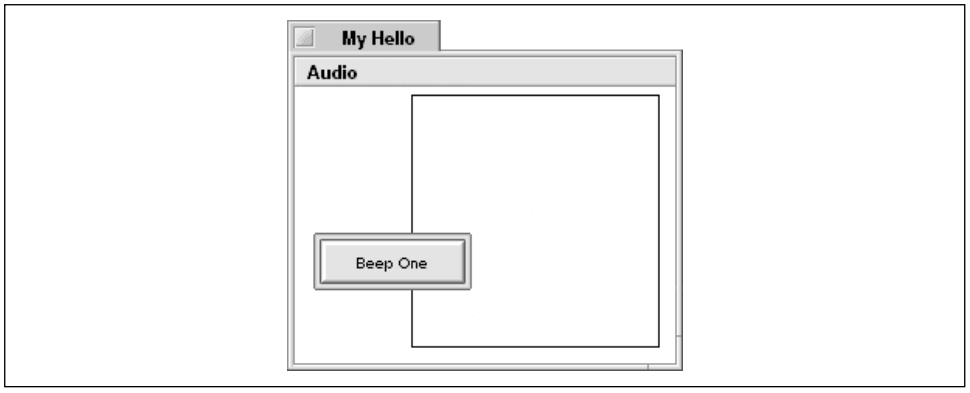

Figure 7-6. A view that keeps a constant-size right and bottom border in its parent

A complete description of all the `resizingMode` constants is found in the `BView` section of the Interface Kit chapter of the Be Book.

Figure 7-6 raises an interesting issue regarding the window's view hierarchy. In the figure, you see that the view appears to be drawn behind the button. As of this writing, the view hierarchy determines the drawing order of the views in a window. When a window requires updating, each view's `Draw()` function is automatically invoked. The order in which the `Draw()` functions are called is first dependent on the view hierarchy, starting from the window's top view down to its bottom views. For views on the same view hierarchy level, the order in which their `Draw()` functions are invoked depends on the order in which the views were added, or attached, to their parent view. The first view added becomes the first view redrawn. Such is the case with the `BButton` view and the `MyDrawView`. Each was added to the window, so these two views are at the same level of the view hierarchy, just under the window's top view. The `MyDrawView` was added first, so it is updated first. After its `Draw()` function is called, the `BButton` `Draw()` routine is called—thus giving the button the appearance of being in front of the `MyDrawView`:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   frame.Set(130.0, MENU_BAR_HEIGHT + 10.0, 290.0, 190.0);

   fMyView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyView);

   fButtonBeep = new BButton(buttonBeepRect, buttonBeepName,
                             buttonBeepLabel, new BMessage(BUTTON_BEEP_MSG));
   AddChild(fButtonBeep);
   ...
   ...
}
```

If the order of the two calls to `AddChild()` were switched, you would expect the button to be redrawn first, and the `MyDrawView` to be updated next. Give it a try by editing the *MyHelloWindow.cpp* file of the MenuAndControl project. When you do that, you'll see that running the program and shrinking the window results in the `MyDrawView` obscuring the button.

Notice that the discussion of view updating order starts of with "As of this writing…". There is no guarantee that this order based on view hierarchy will always be in effect. In short, don't make assumptions about view updating order. Instead, make an effort not to overlap views.

Working with Menus

Your program can get by with simple, static menus and menu items—but why stop there? The menubar, menus, and menu items of a program should reflect the current state of a program. You can make sure they do that by implementing menus so that they give the user visual cues as to what is being done, and what can and can't be done. For instance, a menu item can be marked with a checkmark to let the user know the item is currently in force. Or a menu item's name can be changed, if appropriate, to what is currently taking place in the program. A menu item—or an entire menu—can be disabled to prevent the user from attempting to perform some action that doesn't make sense at that point in the program is at. These and other menu-altering techniques are covered in this section.

Creating a Menu Item

Each menu item in a menu is an object based on the `BMenuItem` class. Menu item objects were introduced earlier in this chapter—here they're studied in much greater detail.

The BMenuItem class

A menu item is created using the `BMenuItem` constructor, the prototype of which is shown here:

```
BMenuItem(const char *label,
          BMessage   *message,
          char       shortcut = 0,
          uint32     modifiers = 0)
```

The first `BMenuItem` parameter, `label`, assigns the item its name, which is displayed as the item's label when the user pulls down the menu in which the item appears.

The `message` parameter assigns a message of a particular type to the menu item. When the user chooses the item, the message is delivered to the window that holds the menubar containing the menu item. That window's

`MessageReceived()` function becomes responsible for carrying out the action associated with the menu item.

The third `BMenuItem` constructor parameter, `shortcut`, is optional. The default value used by the constructor is `0`, but if a character is passed, that character becomes the menu item's keyboard shortcut. When the user presses the shortcut key in conjunction with a modifier key, the menu item is considered selected—just as if the user chose it from the menu. The fourth parameter, `modifiers`, specifies what key is considered the modifier key. A keyboard shortcut *must* include the Command key (which by default is the Alt key on a PC and the Command key on a Macintosh) as its modifier key, but it can also require that one or more other modifier keys be pressed in order to activate the keyboard shortcut. Any of the Be-defined modifier key constants, including `B_COMMAND_KEY`, `B_SHIFT_KEY`, `B_OPTION_KEY`, and `B_CONTROL`, can be used as the `modifiers` parameter. For instance, to designate that a key combination of Command-Q represent a means of activating a Quit menu item, pass `'Q'` as the shortcut parameter and `B_COMMAND_KEY` as the modifiers parameter. To designate that a key combination of Alt-Shift-W (on a PC) or Command-Shift-W (on a Mac) represents a means of closing all open windows, pass `'W'` as the `shortcut` parameter, and the ored constants `B_COMMAND_KEY | B_SHIFT_KEY` as the `modifiers` parameter.

Creating a BMenuItem object

A menu item is often created and added to a menu in one step by invoking the `BMenuItem` constructor from right within the parameter list of a call to the `BMenu` function `AddItem()`:

```
menu->AddItem(new BMenuItem("Start", new BMessage(START_MSG)));
```

Alternatively, a menu item can be created and then added to a menu in a separate step:

```
menuItem = new BMenuItem("Start", new BMessage(START_MSG));
menu->AddItem(menuItem);
```

Regardless of the method used, to this point the `BMenuItem` constructor has been passed only two arguments. To assign a keyboard shortcut to a menu item, include arguments for the optional third and fourth parameters. Here a menu item named "Start" is being given the keyboard shortcut Command-Shift-S (with the assumption that the slightly more intuitive keyboard shortcut Command-S is perhaps already being used for a "Save" menu item):

```
menu->AddItem(new BMenuItem("Start", new BMessage(START_MSG), 'S',
                            B_COMMAND_KEY | B_SHIFT_KEY));
```

If a menu item is associated with a keyboard shortcut, and if that shortcut uses a modifier key, a symbol for that modifier key appears to the right of the menu item.

The symbol provides the user with an indication of what key should be pressed in conjunction with the character key that follows the symbol. Figure 7-7 provides several examples. In that figure, I've set up a menu with four items. The name I've given each item reflects the modifier key or keys that need to be pressed in order to select the item. For instance, the first menu item is selected by pressing Command-A. This next snippet provides the code necessary to set up the menu shown in Figure 7-7:

```
menu->AddItem(new BMenuItem("Command", new BMessage(A_MSG), 'A',
                                                    B_COMMAND_KEY));
menu->AddItem(new BMenuItem("Command-Shift", new BMessage(B_MSG), 'B',
                            B_COMMAND_KEY | B_SHIFT_KEY));
menu->AddItem(new BMenuItem("Command-Shift-Option", new BMessage(C_MSG), 'C',
                            B_COMMAND_KEY | B_SHIFT_KEY | B_OPTION_KEY));
menu->AddItem(new BMenuItem("Command-Shift-Option-Control",
                            new BMessage(D_MSG), 'D',
                            B_COMMAND_KEY | B_SHIFT_KEY |
                            B_OPTION_KEY | B_CONTROL_KEY));
```

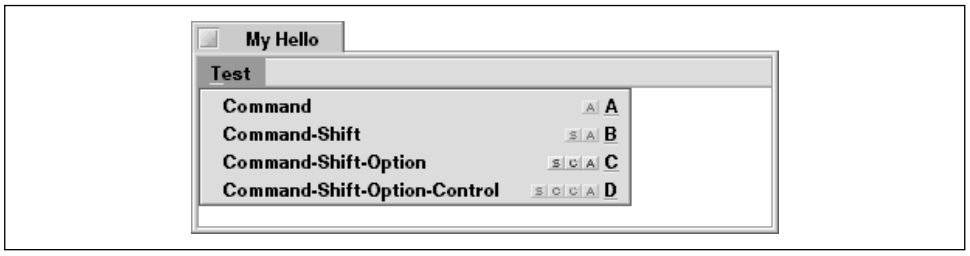

Figure 7-7. A menu that includes items that use several shortcut modifier keys

As illustrated in the preceding snippet and Figure 7-7, menu items are displayed in a menu in the order in which they were added to the `BMenu` object. To reposition items, simply rearrange the order of the calls to `AddItem()`.

A separator is a special menu item that is nothing more than an inactive gray line. It exists only to provide the user with a visual cue that some items in a menu are logically related, and are thus grouped together. To add a separator item, invoke the `BMenu` function `AddSeparatorItem()`:

```
menu->AddSeparatorItem();
```

Figure 7-15, later in this chapter, includes a menu that has a separator item.

Accessing a Menu Item

If a program is to make a change to a menu item, it of course needs access to the item. That can be accomplished by storing either the `BMenuItem` object or the `BMenuBar` object as a data member.

Storing a menu item in a data member

If you need access to a menu item after it is created, just store it in a local variable:

```
BMenu      *menu;
BMenuItem  *menuItem;
...
menu->AddItem(menuItem = new BMenuItem("Beep", new BMessage(BEEP_MSG)));
```

This snippet creates a menu item and adds it to a menu—as several previous examples have done. Here, though, the menu item object created by the `BMenuItem` constructor is assigned to the `BMenuItem` variable `menuItem`. Now the project has access to this one menu item in the form of the `menuItem` variable.

This menu item access technique is of limited use. Access to a menu item may need to take place outside of the function which created the menu item. If that's the case, a window class data member can be created to keep track of an item for the life of the window:

```
class MyHelloWindow : public BWindow {
   ...

   private:
      ...
      BMenuItem   *fMenuItem;
};
```

Now, when the menu item is created, store a reference to it in the `fMenuItem` data member:

```
menu->AddItem(fMenuItem = new BMenuItem("Beep", new BMessage(BEEP_MSG)));
```

Using this technique for creating the menu item, the menu item can be accessed from any member function of the window class.

Storing a menubar in a data member

If several menu items are to be manipulated during the running of a program, it may make more sense to keep track of just the menubar rather than each individual menu item:

```
class MyHelloWindow : public BWindow {
   ...

   private:
      ...
      BMenuBar   *fMenuBar;
};
```

If the menubar can be referenced, the `BMenu` member function `FindItem()` can be used to access any menu item in any of its menus. Pass `FindItem()` a menu's

label (the string that represents the menu item name that is displayed to the user), and the function returns that menu item's `BMenuItem` object:

```
BMenuItem  *theItem;

theItem = fMenuBar->FindItem("Beep");
```

Marking Menu Items

When selected, a menu item can be given a check mark to the left of the item name. When selected again, this same menu item can become unchecked. Figure 7-8 shows a menu with two marked items.

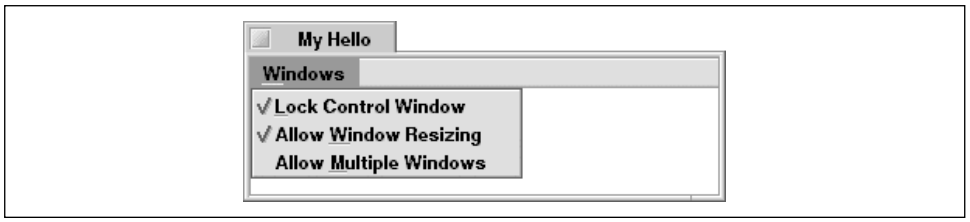

Figure 7-8. A menu with marked, or checked, menu items

Marking a menu item

To mark or unmark a menu item, invoke the item's `BMenuItem` function `SetMarked()`. Passing this function a value of `true` marks the item, while passing a value of `false` unmarks it. Attempting to mark an already marked item or unmark an already unmarked item has no effect. This next snippet sets up the Windows menu items shown in Figure 7-8. Assume that data members exist to keep track of each of the three Windows menu items, and that the menubar and menu have already been created:

```
BMenu      *menu;
BMenuItem  *menuItem;

menu->AddItem(fLockWindMenuItem = new BMenuItem("Lock Control Window",
              new BMessage(LOCK_WIND_MSG)));
fLockWindMenuItem->SetMarked(true);
menu->AddItem(fResizeWindMenuItem = new BMenuItem("Allow Window Resizing",
              new BMessage(RESIZE_WIND_MSG)));
fResizeWindMenuItem ->SetMarked(true)
menu->AddItem(fMultipleWindMenuItem = new BMenuItem("Allow Multiple Windows",
              new BMessage(MULTIPLE_WIND_MSG)));
```

If a menu item is to be initially marked (as the Lock Control Window and Allow Window Resizing items are in the above snippet), save a reference to the item when creating it. Then use that `BMenuItem` object to mark the item.

To find out whether a menu item is marked, call the `BMenuItem` function `IsMarked()`. For instance, after the user selects a menu item, call `IsMarked()` to determine whether to pass `SetMarked()` a value of `true` or `false` and thus toggle the menu's mark. The updating of an item's mark can take place in the `MessageReceived()` function, as shown here for the first menu item from the Windows menu of Figure 7-8:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case LOCK_WIND_MSG:
         BMenuItem  *theItem;

         theItem = fMenuBar->FindItem("Lock Control Window");
         if (theItem->IsMarked()) {
            theItem->SetMarked(false);
            // code to unlock a window (allow user to move it)
         }
         else {
            theItem->SetMarked(true);
            // code to lock a window (prevent user from moving it)
         }
      ...
      ...
      default:
         BWindow::MessageReceived(message);
   }
}
```

Marking a menu item in a menu of related items

A menu may treat all of its items as related options, with the intent of allowing only one item to be in force at any time. The Audio menu in Figure 7-9 provides an example of such a menu. Here the user is expected to choose one of the two beeping options. A subsequent click on the Beep button sounds the system beep either once or twice, depending on the currently selected menu item.

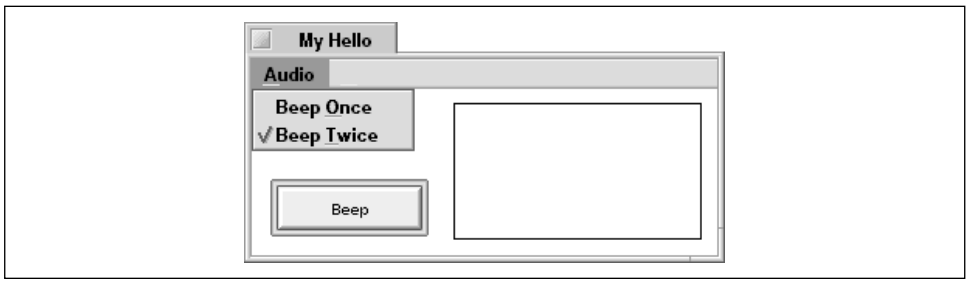

Figure 7-9. A menu with related options

If all of the items in a menu are to act as a single set of options, you can set up the menu to automatically handle the checking and unchecking of its items. The BMenu function SetRadioMode() instructs a menu to allow only one of its items to be marked at any time. Additionally, setting a menu to radio mode provides the menu with the power to automatically mark whatever item the user chooses, and to unmark the previously selected item. To set a menu to radio mode, pass SetRadioMode() a value of true. This next snippet sets up the Audio menu pictured in Figure 7-9, marks the Beep Twice item, and sets the Audio menu to radio mode:

```
BMenu      *menu;
BMenuItem  *menuItem;
...
menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));
menu->AddItem(menuItem = new BMenuItem("Beep Twice",
                                       new BMessage(MENU_BEEP_2_MSG)));
menuItem->SetMarked(true);
menu->SetRadioMode(true);
```

While a menu in radio mode will properly update its item mark in response to menu item selections, it is your responsibility to check which item is to be initially marked. As shown above, that's accomplished via a call to SetMarked(). If a menu item is checked, your code should also make sure that the feature or option to be set is in fact set.

The BMenu function FindMarked() returns the menu item object of the currently marked item in a menu. When a menu is in radio mode, only one item can be marked at any time. In the next snippet, the Audio menu shown in Figure 7-9 is kept track of by a BMenu data member named fAudioMenu. Calling FindMarked() on this item returns the BMenuItem object of the currently marked item:

```
BMenuItem  *theItem;

theItem = fAudioMenu->FindMarked();
```

FindMarked() can be used on a menu that isn't set to radio mode, too—but its usefulness is then diminished because there may be more than one item marked. If more than one item is marked, FindMarked() returns a reference to the first marked item encountered (it starts at the first item in a menu and moves down).

Changing a Menu Item's Label

A menu item's label can be changed at any time. To do that, gain access to the menu item and then invoke the BMenuItem function SetLabel(). In the next

snippet, a menu item named "Start Simulation" is being renamed to "Stop Simulation."

```
fSimMenuItem->SetLabel("Stop Simulation");
```

To find out the current label of a menu item, call the `BMenuItem` function `Label()`. A call to this routine can be made prior to a call to `SetLabel()` to determine the item's current label before changing it to a new string. The type of label-changing shown in the above snippet is a good candidate for the use of both `Label()` and `SetLabel()`. If the user starts some sort of action by choosing a menu item, that menu item's label might change in order to provide the user with a means of stopping the action. The `MessageReceived()` function holds the label-changing code:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case MENU_SIMULATION_RUN_MSG:
         const char  *menuItemLabel;

         menuItemLabel = fSimMenuItem->Label();
         if ((strcmp(menuItemLabel, "Start Simulation") == 0)) {
            fSimMenuItem->SetLabel("Stop Simulation");
            // invoke application-defined routine to start simulation
         }
         else {
            fSimMenuItem->SetLabel("Start Simulation");
            // invoke application-defined routine to stop simulation
         }
         break;

      default:
         BWindow::MessageReceived(message);
   }
}
```

This snippet is the first in this book to make use of a standard C library function. While the member functions of the classes of the BeOS take care of many of your programming needs, they of course can't account for every task your program is to perform. Before writing an application-defined routine, don't forget to fall back on your C and C++ background to select a standard library function where appropriate. Here I use the string comparison routine `strcmp()` to compare the characters in a menu item's label to the string "Start Simulation." If the strings are identical, `strcmp()` returns a value of `0`. In such a case, the menu item label needs to be changed to signal that the simulation is running and to allow the user to halt the action. As shown in the two comments in the above code, whatever it is that is to be simulated is left as an exercise for the reader!

Disabling and Enabling Menus and Menu Items

When a menu is created, the menu and each of its items are all initially enabled. The entire menu—including the menu's name in the menubar and all its items—or any individual menu item can be disabled.

Disabling and enabling a menu item

A disabled menu item appears dim and is inactive—the user can see the item and read its label, but can't select it (releasing the mouse button while the cursor is over the item has no effect). A menu item can be disabled or re-enabled by invoking the `BMenuItem` member function `SetEnabled()`. An argument of `true` enables the item, while an argument of `false` disables the item. By default, a menu item is enabled upon creation. To disable a newly created menu item, call `SetEnabled()` just after the menu item is created. Assuming that the menu's window keeps track of its menubar in a data member named `fMenuBar`, the following snippet could be used to disable a menu item named "Start":

```
BMenuItem  *theItem;

theItem = fMenuBar->FindItem("Start");
theItem->SetEnabled(false);
```

The current state of a menu item can be found by invoking the item's `IsEnabled()` function. This routine returns a value of `true` if the item is presently enabled, `false` if it's disabled.

If the enabling or disabling of a menu item is to take place in response to a message (whether initiated by a different menu item selection or a control click), include the menu item enabling/disabling code in `MessageReceived()`. Here a menu item named Advanced Options is enabled or disabled in response to an application-defined `TOGGLE_OPTIONS_MSG` message:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case TOGGLE_OPTIONS_MSG:
         BMenuItem  *theItem;

         theItem = fMenuBar->FindItem("Advanced Options");
         if (theItem->IsEnabled())
            theItem->SetEnabled(false);
         else
            theItem->SetEnabled(true);
      ...
      ...
      default:
         BWindow::MessageReceived(message);
   }
}
```

What happens when `SetEnabled()` or `IsEnabled()` is invoked depends on the state of the menu the item resides in. If the menu itself is disabled, attempting to enable an individual item in that menu will fail. Thus, even if your code hasn't explicitly disabled a specific menu item, `IsEnabled()` could return a value of `false` for that item.

Disabling and enabling a menu

When an entire menu is disabled, its name appears dim in the menubar. Clicking on the menu opens the menu and displays its items, but each of the items will be disabled. Now that you know of the `BMenuItem()` function `SetEnabled()`, it should come as no surprise that there is also a `BMenu` version of this routine. Passing a value of `false` to a menu object's `SetEnabled()` function disables the entire menu. If a program keeps track of an Audio menu in a `BMenu` data member named `fAudioMenu`, you could disable that menu with just this line of code:

```
fAudioMenu->SetEnabled(false);
```

The current state of a menu can be found by invoking the item's `IsEnabled()` function. Like the `BMenuItem` version, the `BMenu` version of this routine returns a value of `true` if the object is presently enabled.

A menu's state usually changes in response to a message. If that's the case, include the menu enabling or disabling code in `MessageReceived()`.

Menu and Menu Item Keyboard Access

Menu items can be accessed by the mouse, of course, but they can also be accessed from the keyboard.

Shortcut keys

This chapter's "Creating a Menu Item" section demonstrated how to assign a new `BMenuItem` a keyboard shortcut. Here the third and fourth arguments to the `BMenuItem` constructor set the new menu item to have a keyboard shortcut of Command-M:

```
menu->AddItem(new BMenuItem("Command", new BMessage(A_MSG), 'M',
                            B_COMMAND_KEY));
```

In most instances the above method works fine for establishing a keyboard shortcut. However, your program may want to assign a keyboard shortcut on the fly. This is particularly true if your program gives the user the power to modify menu

item keyboard shortcuts. If an existing menu item doesn't have a keyboard shortcut, or your program needs to change its currently defined keyboard shortcut, invoke the item's `SetShortcut()` function. The two arguments to this routine are identical to the third and fourth arguments that can be passed to the `BMenuItem` constructor. Here the keyboard shortcut for an Open menu item is being set to Command-Shift-O:

```
BMenuItem  *theItem;

theItem = fMenuBar->FindItem("Open");
theItem->SetShortcut('O', B_COMMAND_KEY | B_SHIFT_KEY)
```

To determine the current keyboard shortcut of a menu item, invoke the item's `Shortcut()` function. Pass this function a pointer to a 32-bit unsigned integer variable. `Shortcut()` will fill this variable with a mask that consists of all of the modifier keys that are a part of the shortcut for the menu item. `Shortcut()` will also return the shortcut character for the menu item:

```
uint32  *shortcutModifiers;
char    shortcutChar;

shortcutChar = theItem->Shortcut(shortcutModifiers);
```

To determine which modifier key or keys are a part of the shortcut, perform a bitwise AND on the `uint32` parameter. In the next snippet, a menu item named Calculate is examined to determine its shortcut key. If the character returned by `Shortcut()` is null, the item has no shortcut key. If the menu item does have a shortcut, the code goes on to determine which modifier keys are involved. Because all shortcut key combinations must include the Command key, no check is made to see if that key is a modifier. The code does, however, check to see if the Shift or Option keys are included in the shortcut key combination:

```
BMenuItem  *theItem;
uint32     *shortcutModifiers;
char       shortcutChar;
bool       hasShortcut = true;
bool       shiftKeyModifier = false;
bool       optionKeyModifier = false;

theItem = fMenuBar->FindItem("Calculate");
shortcutChar = theItem->Shortcut(shortcutModifiers);

if (shortcutChar == '\0')          // menu item doesn't have a shortcut key
   hasShortcut = false;
else {                             // menu item has a shortcut key
   if (*shortcutModifiers & B_SHIFT_KEY)
      shiftKeyModifier = true;
   if (*shortcutModifiers & B_OPTION_KEY)
      optionKeyModifier = true;
}
```

Keyboard triggers

A menu item can optionally be supplied with a shortcut key to benefit users who prefer the keyboard over the mouse. But every menu item is supplied with a *trigger* for the same reason. A trigger is a single character the user types in order to select a menu item. A trigger differs from a shortcut key in two ways. First, the trigger doesn't involve the use of a modifier key—simply pressing the trigger key is enough to choose the menu item. The second difference is that the trigger works only when the menu that holds the item in question is open, or dropped. Once a menu is open onscreen, the user can simply press a trigger key to select an item.

The trigger is one of the characters in the menu item name, and is indicated by being underlined. Typically, the trigger is the first character of one of the words that make up the menu item's name. Looking back at Figure 7-9 reveals two examples—there you see that the Audio menu's two items, Beep Once and Beep Twice, have triggers of "O" and "T," respectively.

Because a trigger can be used only on an open menu, different menus in the same menubar can have items with the same trigger. A menu item's trigger is assigned to the item by the system, so your program doesn't have to worry about which items end up with which triggers. If you do want responsibility for assigning a menu item a particular trigger, invoke that item's `SetTrigger()` function. Simply pass `SetTrigger()` the character that is to serve as the new trigger. Here a menu item named Jump is given a trigger of "u":

```
BMenuItem  *theItem;

theItem = fMenuBar->FindItem("Jump");
theItem->SetTrigger('u');
```

The character you pass to `SetTrigger()` must be either the menu item's shortcut key or a character in the menu item name. Failing both of these, the item will not be given a trigger—and whatever character had previously been assigned to the item won't be used either (so passing `SetTrigger()` an invalid character provides the exception to the rule that every menu item must have a trigger).

You can verify that a call to `SetTrigger()` worked according to plan by invoking the item's `Trigger()` function. Double-check to see if the above snippet works by following it with a call to `Trigger()`:

```
char  theTrigger;

theTrigger = theItem->Trigger();
```

With the exception of a menu item that's been given a trigger via a call to `SetTrigger()`, the system doesn't assign menu items triggers until runtime. That's done to avoid duplication of triggers within a menu. For this reason, calling `Trigger()` on a menu item that hasn't been manually assigned a trigger (by your project invoking the item's `SetTrigger()` function) serves little purpose—`Trigger()` will simply return `NULL` in such cases.

Figure 7-9 shows that the menu itself has a trigger—the "A" key serves as the trigger for the Audio menu. Like a menu item trigger, the system automatically assigns a trigger to a menu. Again like a menu item, your project can override the system-supplied trigger character. To do that, invoke the `BMenu` version of `SetTrigger()` on a menu object. Here the Audio menu is created and its trigger set to "U":

```
BMenu       *menu;
...
menu = new BMenu("Audio");
fMenuBar->AddItem(menu);
menu->SetTrigger('U');
// now add menu items to menu
```

Menu Characteristics Example Projects

In this chapter's example projects folder you'll find three projects that alter the characteristics of menus: TwoItemMenu, DisableMenuItem, and FindItemByMark. Each contains only a few lines of new code, so I'll forego thorough code walkthroughs and describe each only briefly.

Adding and altering menu items example project

The menu in each of this chapter's previous example projects consisted of just a single item. The TwoItemMenu project adds a second item. This project also adds a shortcut key to each item—Command-1 for the Beep Once item and Command-2 for the Beep Twice item. Figure 7-10 shows that the system has assigned each item a trigger that is the same character as that used in the item's shortcut key. Finally, the project demonstrates a menu set to radio mode—selecting one menu item checks that item and unchecks the other item.

Most of the code included in the TwoMenuItem project will be quite familiar to you. The code that's pertinent to the menu item topics in this section comes from the `MyHelloWindow` constructor:

```
BMenu       *menu;
BMenuItem   *menuItem;
...
menu = new BMenu("Audio");
```

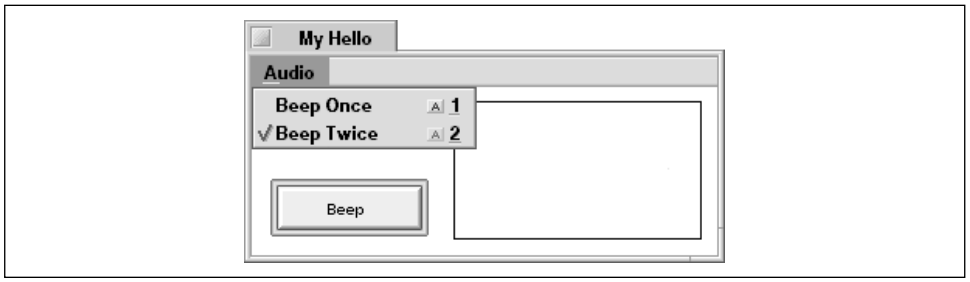

Figure 7-10. Menu items with shortcut keys

```
fMenuBar->AddItem(menu);
menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG),
                            '1', B_COMMAND_KEY));
menu->AddItem(menuItem = new BMenuItem("Beep Twice", new BMessage(MENU_BEEP_
2_MSG),
                                       '2', B_COMMAND_KEY));
menuItem->SetMarked(true);
menu->SetRadioMode(true);
```

Menu item disabling and enabling example project

The DisableMenuItem project adds a few lines of code to the TwoMenuItem project to demonstrate how your program can toggle a menu item's state based on a message sent by a control. Clicking on the Beep button disables the Beep Once menu item in the Audio menu. Clicking on the Beep button again enables the same item. This menu-related code is found in the `MessageReceived()` `case` section for the message issued by the button control:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_BEEP_MSG:
         // code to beep the appropriate number of times goes here

         BMenuItem  *theItem;

         theItem = fMenuBar->FindItem("Beep Once");
         if (theItem->IsEnabled())
            theItem->SetEnabled(false);
         else
            theItem->SetEnabled(true);
         break;

      case MENU_BEEP_1_MSG:
         fNumBeeps = 1;
         break;

      case MENU_BEEP_2_MSG:
         fNumBeeps = 2;
```

```
            break;

         default:
            BWindow::MessageReceived(message);
      }
   }
```

Accessing a menu item from a menu object

The preceding two projects use a `MyHelloWindow` class data member named `fNumBeeps` to keep track of how many times the system beep should sound in response to a click on the Beep button. The FindItemByMark project omits this data member, and doesn't keep track of which menu item is currently selected. Instead, it waits until the user clicks the Beep button before determining which menu item is currently marked. Clicking the Beep button results in the issuing of a `BUTTON_BEEP_MSG` that reaches the `MessageReceived()` function. Here the `BMenu` member function `FindMarked()` is used to find the currently checked menu item. Once the item object is obtained, its place in the menu is found by calling the `BMenu` function `IndexOf()`. A menu's items are indexed starting at 0, so adding 1 to the value returned by `IndexOf()` provides the number of beeps to play. The following snippet is from the `MessageReceived()` function of the project's `MyHelloWindow` class:

```
case BUTTON_BEEP_MSG:
   bigtime_t  microseconds = 1000000;  // one second
   int32      i;

   BMenuItem  *theItem;
   int32      itemIndex;
   int32      numBeeps;

   theItem = fAudioMenu->FindMarked();
   itemIndex = fAudioMenu->IndexOf(theItem);
   numBeeps = itemIndex + 1;

   for (i = 1; i <= numBeeps; i++)
   {
      beep();
      if (i != numBeeps)
         snooze(microseconds);
   }

   break;
```

Multiple Menus

Rather than jumping right into new topics, I'll provide a bit of a transition by presenting an example that includes two menus in its menubar. The example isn't entirely gratuitous, though—much of its code will reappear in upcoming discus-

sions. Figure 7-11 shows the window, and the new Visual menu that's been added to the existing Audio menu, for the TwoMenus program. Choosing Draw Circles from the Visual menu draws a number of concentric circles in the window, while Draw Squares draws, yes, a number of concentric squares!

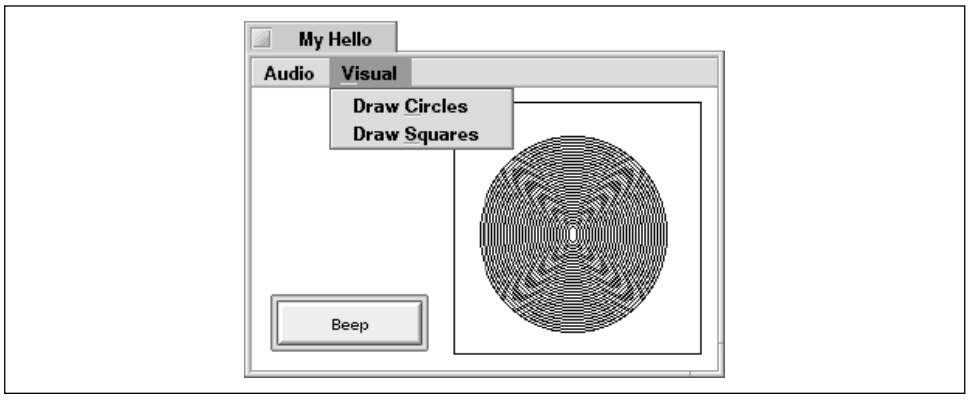

Figure 7-11. The TwoMenus application window

The *MyHelloWindow.h* header file in the TwoMenus project defines five message constants—one for the window's button and one for each of the four menu items.

```
#define   BUTTON_BEEP_MSG          'beep'
#define   MENU_BEEP_1_MSG          'bep1'
#define   MENU_BEEP_2_MSG          'bep2'
#define   MENU_DRAW_CIRCLES_MSG    'circ'
#define   MENU_DRAW_SQUARES_MSG    'squa'
```

The `MyHelloWindow` class holds four data members. `fMyView` is used for drawing the circles or squares, and `fNumBeeps` holds the number of times the system beep is to be played when the Beep button is clicked. Once the button and menubar are created, they aren't accessed outside of the `MyHelloWindow` constructor. Thus, I could have declared `BButton` and `BMenuBar` variables local to that routine rather than making each a data member. However, I've opted to set the project up from the start with the assumption that it will grow in complexity well beyond this trivial version. If I later need to add features that alter either the button or menu items (such as disabling and so forth), I'm all set.

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage* message);

   private:
      MyDrawView     *fMyView;
      BButton        *fButtonBeep;
```

```
        BMenuBar        *fMenuBar;
        int32           fNumBeeps;
};
```

Some of the `MyHelloWindow` constructor code is familiar to you, so I won't show the routine in its entirety. Here's the constructor without the view and button code:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   ...
   ...
   BMenu       *menu;
   BMenuItem   *menuItem;
   BRect       menuBarRect;

   menuBarRect.Set(0.0, 0.0, 10000.0, MENU_BAR_HEIGHT);
   fMenuBar = new BMenuBar(menuBarRect, "MenuBar");
   AddChild(fMenuBar);

   menu = new BMenu("Audio");
   fMenuBar->AddItem(menu);
   menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));
   menu->AddItem(menuItem = new BMenuItem("Beep Twice",
                                          new BMessage(MENU_BEEP_2_MSG)));
   menu->SetRadioMode(true);
   menuItem->SetMarked(true);
   fNumBeeps = 2;

   menu = new BMenu("Visual");
   fMenuBar->AddItem(menu);
   menu->AddItem(new BMenuItem("Draw Circles",
                               new BMessage(MENU_DRAW_CIRCLES_MSG)));
   menu->AddItem(new BMenuItem("Draw Squares",
                               new BMessage(MENU_DRAW_SQUARES_MSG)));
   Show();
}
```

The `MessageReceived()` routine handles each of the five types of messages the window might receive. A selection of either of the items from the Visual menu results in the application-defined function `SetViewPicture()` being called, followed by a call to the `BView` routine `Invalidate()` to force the view to update.

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_BEEP_MSG:
         // beep fNumBeeps times
         break;

      case MENU_BEEP_1_MSG:
         fNumBeeps = 1;
```

```
            break;

         case MENU_BEEP_2_MSG:
            fNumBeeps = 2;
            break;

         case MENU_DRAW_CIRCLES_MSG:
            fMyView->SetViewPicture(PICTURE_CIRCLES);
            fMyView->Invalidate();
            break;

         case MENU_DRAW_SQUARES_MSG:
            fMyView->SetViewPicture(PICTURE_SQUARES);
            fMyView->Invalidate();
            break;

         default:
            BWindow::MessageReceived(message);
      }
   }
```

The drawing code could have been kept in the `MessageReceived()` function, but I've decided to place it in a `MyDrawView` member function in order to keep `MessageReceived()` streamlined. The `SetViewPicture()` function defines a `BPicture` object based on the `int32` argument passed to the routine. The value of that argument is in turn based on the menu item selected by the user:

```
void MyDrawView::SetViewPicture(int32 pictureNum)
 {
   BRect  aRect;
   int32  i;

   switch (pictureNum)
   {
      case PICTURE_SQUARES:
         BeginPicture(fPicture);
            aRect.Set(15.0, 20.0, 140.0, 150.0);
            for (i = 0; i < 30; i++) {
               aRect.InsetBy(2.0, 2.0);
               StrokeRect(aRect);
         }
         fPicture = EndPicture();
         break;

      case PICTURE_CIRCLES:
         BeginPicture(fPicture);
            aRect.Set(15.0, 20.0, 140.0, 150.0);
            for (i = 0; i < 30; i++) {
               aRect.InsetBy(2.0, 2.0);
               StrokeEllipse(aRect);
            }
         fPicture = EndPicture();
         break;
   }
}
```

The picture defined in `SetViewPicture()` is a new data member that's been added to the `MyDrawView` class. Here you see the `BPicture` data member and the newly added declaration of the `SetViewPicture()` function:

```
#define   PICTURE_SQUARES   1
#define   PICTURE_CIRCLES   2


class MyDrawView : public BView {

   public:
                     MyDrawView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);
      void           SetViewPicture(int32 pictureNum);

   private:
      BPicture       *fPicture;
};
```

The whole purpose of storing the circles or squares `BPicture` object in a `MyDrawView` data member is so that the picture will be automatically updated whenever the view it's drawn in needs updating. That's accomplished by adding a call to the `BView` function `DrawPicture()` to the `MyDrawView` member function `Draw()`:

```
void MyDrawView::Draw(BRect)
{
   BRect  frame = Bounds();

   StrokeRect(frame);
   DrawPicture(fPicture);
}
```

Pop-up Menus

A pop-up menu is a menu that exists within the content area of a window rather than within a menubar. The pop-up menu can be positioned anywhere in a window (or anywhere in a view in a window). Like a menu in a menubar, a pop-up menu's content is displayed when the user clicks on the menu. Figure 7-12 shows a pop-up menu, both before and after being clicked, that holds two items.

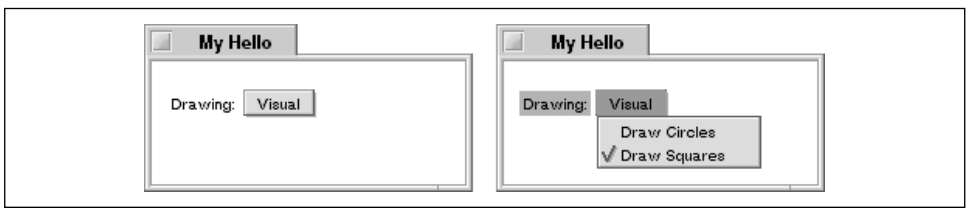

Figure 7-12. An example of a pop-up menu

A pop-up menu's default state is radio mode—the most recently selected item in the menu appears checked when the menu pops up. Figure 7-12 illustrates this for a menu with two items in it. A pop-up menu is most often in radio mode, so such a menu should be used to hold a related set of options. If the menu is to hold items that aren't directly related to one another, the menu should be housed within a menubar rather than existing as a pop-up menu. A context-sensitive pop-up menu is an exception—it behaves like a normal menu, albeit one that is not tied to a specific location in a window.

The BPopUpMenu Class

A pop-up menu is an object of the class `BPopUpMenu`. The `BPopUpMenu` class is derived from a class you've already studied—the `BMenu` class. Here's the `BPopUpMenu` constructor:

```
BPopUpMenu(const char  *name,
           bool        radioMode = true,
           bool        labelFromMarked = true,
           menu_layout layout = B_ITEMS_IN_COLUMN)
```

Like any menu, a pop-up menu has a name. The name is defined by the first `BPopUpMenu` constructor parameter and is present on the pop-up menu when the menu initially appears in a window. This pop-up menu name, however, can be changed to reflect the user's selection from the menu. The `labelFromMarked` parameter determines if that is to be the case.

The value of the `radioMode` parameter sets the pop-up menu's radio mode setting. By default, `radioMode` has a value of `true`. A value of `true` here means the same as it does for any other menu: choosing one item checks that item and unchecks whatever item was previously selected. The pop-up menu's radio mode value can be toggled by calling the `BMenu` function `SetRadioMode()`.

If the `labelFromMarked` parameter is set to `true`—as it is by default—the user's menu item choice from the pop-up menu determines the name the pop-up menu takes on. The original name won't reappear during the life of the window to which the pop up is attached. In Figure 7-12, for instance, the pop-up menu's name is Visual. If the user chooses, say, the Draw Squares item, the pop-up menu's name will change to Draw Squares. Setting `labelFromMarked` to `true` has the interesting side effect of automatically setting the pop-up menu to radio mode. That is, regardless of the value passed as the `radioMode` parameter of the `BPopUpMenu` constructor, the menu will be set to radio mode. If `labelFromMarked` is `false`, the pop-up menu's name will be fixed at its initial name (as defined by the value passed in as the first parameter) and its radio mode state will be determined by the value of the `radioMode` parameter.

The last BPopUpMenu parameter defines the layout of the pop-up menu. By default, a pop-up menu's items appear in a column—just like a menu held in a menubar. To instead have the items appear in a row, replace the B_ITEMS_IN_COLUMN value with another Be-defined constant: B_ITEMS_IN_ROW.

The BMenuField Class

A pop-up menu won't be placed in a menubar, so it doesn't have to be added to a BMenuBar object. A pop-up menu does, however, need to be added to an object capable of controlling the menu. The BMenuField class exists for this purpose. When a BMenuField object is created, a BPopUpMenu object is associated with it. Here's the BMenuField constructor:

```
BMenuField(BRect       frame,
           const char  *name,
           const char  *label,
           BMenu       *menu,
           uint32      resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
           uint32      flags = B_WILL_DRAW | B_NAVIGABLE)
```

The BMenuField is derived from the BView class. When a BMenuField object is created, four of the six BMenuField constructor parameters (frame, name, resizingMode, and flags) are passed on to the BView constructor.

The frame is a rectangle that defines the size of the BMenuField, which includes both a label and a pop-up menu. In Figure 7-12, you saw an example of a BMenuField object that has a label of "Drawing:" and a menu with the name "Visual." Recall that the source of the menu's name is the name parameter of the BPopUpMenu object. The BMenuField name parameter serves as a name for the BMenuField view, and isn't displayed onscreen. The resizingMode parameter specifies how the BMenuField is to be resized as its parent view is resized. The default value of B_FOLLOW_LEFT | B_FOLLOW_TOP means that the distance from the menu field's left side and its parent's parent's left side will be fixed, as will the distance from the menu field's top and its parent's top. The flags parameter specifies the notification the menu field is to receive. The default flags value of B_WILL_DRAW | B_NAVIGABLE means that the menu field view contains drawing, and should thus be subject to automatic updates, and that the menu field is capable of receiving and responding to keyboard input.

The BMenuField label parameter defines an optional label for the menu field. If you pass a string here, that string is displayed to the left of the pop-up menu that is a part of the menu field. To omit a label, pass NULL as the label parameter.

The menu parameter specifies the pop-up menu that is to be controlled by the menu field. While the class specified is BMenu, it is most likely that you'll pass a BPopUpMenu object here (BPopUpMenu is derived from BMenu, so it can be used).

Creating a Pop-up Menu

To create a pop-up menu, you first create a `BPopUpMenu` object, and then create a `BMenuField` object. Three of the four `BPopUpMenu` constructor parameters have default values, and those values generally suffice when creating a pop-up menu object—so creating the `BPopUpMenu` object often involves passing only a single argument to the `BPopUpMenu` constructor. Here a `BPopUpMenu` object for a pop-up menu named "Visual" is being created:

```
BPopUpMenu  *popUpMenu;

popUpMenu = new BPopUpMenu("Visual");
```

For a "regular" menu—one that resides in a menubar—the next step would typically be to add the new menu object to the existing menubar object with a call to `AddItem()`. A pop-up menu won't be placed in a menubar, so the above step is unnecessary. The pop-up menu does, however, need to be added to a menu field. The next steps are to create a `BMenuField` object that incorporates the `BPopUpMenu` object and then add this new menu field to a view (or window):

```
BMenuField  *menuField;
BRect       popUpMenuRect(10.0, 40.0, 105.0, 70.0);

menuField = new BMenuField(popUpMenuRect, "VisualPopUp", "Drawing",
                           popUpMenu);
AddChild(menuField);
```

The third argument to the `BMenuField` constructor specifies that the menu field have a label of "Drawing." In the previous snippet, the sole argument to the `BPopUpMenu` constructor specified that the pop-up menu have the name "Visual." The result of executing the above two snippets would be the menu field shown on the left side of Figure 7-12. The pop-up menu would be devoid of any items. To add a menu item to a pop up, have the pop-up menu invoke the `BMenu` function `AddItem()`. `BPopUpMenu` is derived from `BMenu`, and `BPopUpMenu` doesn't override the `BMenu` version of `AddItem()`—so adding menu items to a pop-up menu is handled in the exact same way as adding menu items to a "normal" menu that resides in a menubar. Here two items are added to the pop-up menu that was just created:

```
popUpMenu->AddItem(new BMenuItem("Draw Circles",
                                 new BMessage(MENU_DRAW_CIRCLES_MSG)));
popUpMenu->AddItem(new BMenuItem("Draw Squares",
                                 new BMessage(MENU_DRAW_SQUARES_MSG)));
```

At this point the menu field, and the pop-up menu that is a part of the menu field, match those shown on the right of Figure 7-12.

One reason the menu field label exists is to provide the user with information regarding the purpose of the menu field's pop-up menu. Once the user chooses

an item from the pop-up menu, the pop-up menu's name disappears, so the menu field label may then be of help. If the contents of the pop-up menu make the pop-up menu's purpose obvious, you may choose to forego the menu field label. To do that, simply pass `NULL` as the third argument to the `BMenuField` constructor. Compare this `BMenuField` object creation with the one created a couple of snippets back:

```
menuField = new BMenuField(popUpMenuRect, "VisualPopUp", NULL, popUpMenu);
```

The left side of Figure 7-13 shows how the menu field looks now. The middle part of the figure shows a menu item selection being made, while the right side of the figure shows how the menu field looks after choosing an item.

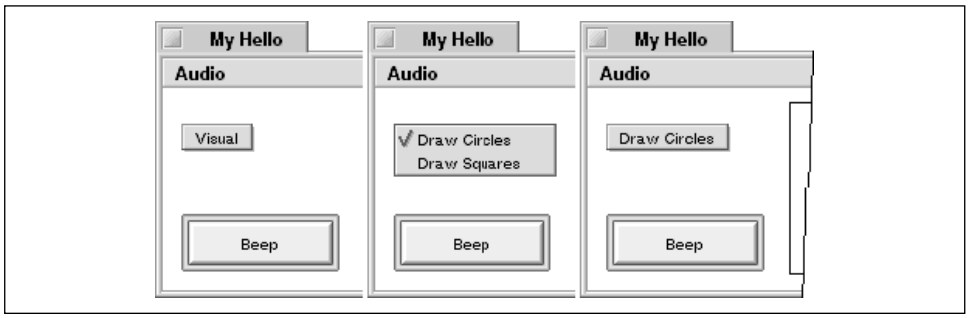

Figure 7-13. A menu field before, during, and after a menu item selection

You'll find the code that generates the window shown in Figure 7-13 in the MenuAndPopup project. The code varies little from that shown in the coverage of this chapter's TwoMenus project. But the `MyHelloWindow` constructors are different. The MenuAndPopup project uses the following code for adding a second menu to the menubar:

```
BMenuField  *menuField;
BPopUpMenu  *popUpMenu;
BRect       popUpMenuRect(10.0, 40.0, 120.0, 70.0);
const char  *popUpName = "VisualPopUp";
const char  *popUpLabel = NULL;

popUpMenu = new BPopUpMenu("Visual");
menuField = new BMenuField(popUpMenuRect, popUpName, popUpLabel, popUpMenu);
AddChild(menuField);
popUpMenu->AddItem(new BMenuItem("Draw Circles",
                                 new BMessage(MENU_DRAW_CIRCLES_MSG)));
popUpMenu->AddItem(new BMenuItem("Draw Squares",
                                 new BMessage(MENU_DRAW_SQUARES_MSG)));
```

Altering the Label/Pop-up Menu Divider

While no physical vertical line divides a menu field's label from its pop-up menu, there is indeed a defined boundary. By default, half of a menu field's width is devoted to the label, and half is assigned to the menu. Consider this snippet:

```
BPopUpMenu  *popUpMenu;

popUpMenu = new BPopUpMenu("Click Here");
popUpMenu->AddItem(new BMenuItem("Small", new BMessage(SML_MSG)));
popUpMenu->AddItem(new BMenuItem("Medium", new BMessage(MED_MSG)));
popUpMenu->AddItem(new BMenuItem("Large", new BMessage(LRG_MSG)));
popUpMenu->AddItem(new BMenuItem("Extra Large", new BMessage(XLG_MSG)));

BMenuField  *menuField;
BRect       popUpMenuRect(30.0, 25.0, 150.0, 50.0);

menuField = new BMenuField(popUpMenuRect, "PopUp", "Size", popUpMenu);
AddChild(menuField);
```

This code creates a pop-up menu and adds it to a menu field. The `BMenuField` object has a width of 120 pixels (150.0 – 30.0). Thus the menu field divider, which is given in coordinates local to the menu field's view, would be 60.0. As shown in the top two windows of Figure 7-14, this 1:1 ratio isn't always appropriate. Here the menu field label "Size" requires much less space than the pop-up menu name of "Click Here." Because the pop-up menu is constrained to half the menu field width, the pop-up name is automatically condensed—as is the "Extra Large" menu item after it is selected.

To devote more or less of a menu field to either the label or pop-up menu, use the `BMenuField` function `SetDivider()`. Pass this routine a floating-point value to be used as the new divider. This one argument should be expressed in coordinates local to the menu field. Consider our current example, which produces a menu field with a width of 120 pixels. To move the divider from its halfway point of 60 pixels to 30 pixels from the left edge of the menu field, pass a value of 30.0 to `SetDivider()`:

```
menuField->SetDivider(30.0);
```

The bottom two windows in Figure 7-14 show how the menu field looks after moving its divider. The pop-up menu now starts 30 pixels from the left of the menu field—just a few pixels to the left of the "Size" label. The pop-up menu now has room to expand horizontally; rather than being limited to 60 pixels in width, the menu can now occupy up to 90 of the menu field's 120 pixels.

You could use trial and error to find the amount of room appropriate for your pop up's label. But of course you'll instead rely on the `BView` function `StringWidth()`—the `BMenuField` class is derived from the `BView` class, so any

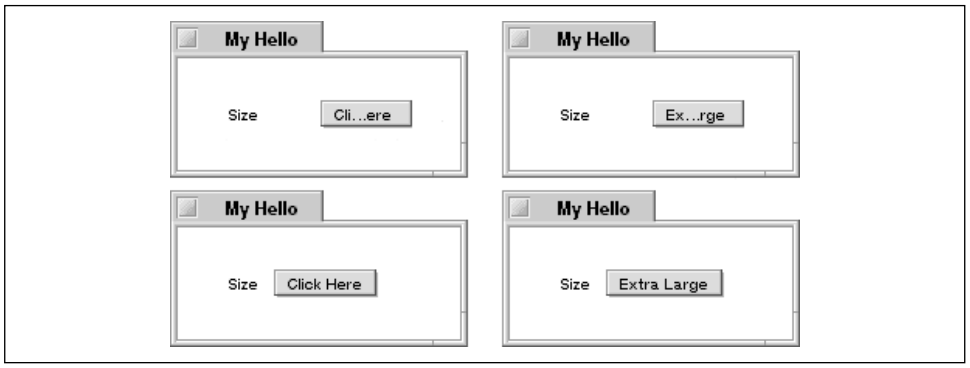

Figure 7-14. A menu field with its default divider (top) and an adjusted divider (bottom)

view member function can be invoked by a pop-up menu object. When passed a string, `StringWidth()` returns the number of pixels that string requires (based on the characteristics of the font currently used by the `BMenuField` object). For instance:

```
#define     LABEL_MARGIN5.0
float       labelWidth;

labelWidth = menuField->StringWidth("Size");
menuField->SetDivider(labelWidth + LABEL_MARGIN);
```

The above snippet determines the width of the string "Size" (the label used in the previous snippets), then uses that pixel width in setting the width of the space used to hold the label. Because the label starts a few pixels in from the left edge of the area reserved for the label, a few pixels are added as a margin, or buffer. If that wasn't done, the divider would be placed somewhere on the last character in the label, cutting a part of it off.

Submenus

A menu item can act as a *submenu*, or *hierarchical menu*—a menu within a menu. To operate a submenu, the user simply clicks on the submenu name, exposing a new menu of choices. To choose an item from the submenu, the user keeps the mouse button held down, slides the cursor onto the item, and releases the mouse button. Figure 7-15 provides an example of a submenu. Here a separator item and a submenu have been added to the Visual menu that was introduced in this chapter's TwoMenus project. The Number of Shapes submenu consists of three items: 10, 20, and 30.

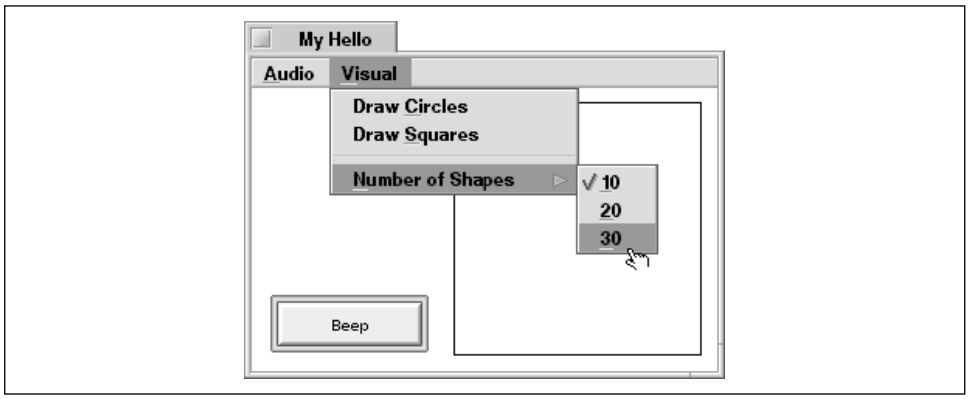

Figure 7-15. An example of a submenu

Creating a Submenu

A submenu is nothing more than a `BMenu` object that is added to another `BMenu` object in place of a menu item. Consider an Animals menu that has five types of animals for its menu items: armadillo, duck, labrador, poodle, and shepherd. Because three of the five animal types fall into the same category—dogs—this example would be well served by grouping the three dog items into a submenu. Figure 7-16 shows what the Animals menu would look like with a submenu, and this next snippet shows the code needed to produce this menu:

```
BMenu      *menu;
BMenu      *subMenu;

menu = new BMenu("Animals");
menuBar->AddItem(menu);
menu->AddItem(new BMenuItem("Armadillo", new BMessage(ARMADILLO_MSG)));
subMenu = new BMenu("Dogs");
menu->AddItem(subMenu);
subMenu->AddItem(new BMenuItem("Labrador", new BMessage(LAB_MSG)));
subMenu->AddItem(new BMenuItem("Poodle", new BMessage(POODLE_MSG)));
subMenu->AddItem(new BMenuItem("Shepherd", new BMessage(SHEPHERD_MSG)));
menu->AddItem(new BMenuItem("Duck", new BMessage(DUCK_MSG)));
```

Notice in this snippet that while I've given the variable used to represent the submenu the name `subMenu`, it really is nothing more than a `BMenu` object. The items in the Dogs submenu were added the same way as the items in the Animal menu—by invoking the `BMenu` member function `AddItem()`.

Submenu Example Project

The MenusAndSubmenus project builds an application that displays the window shown back in Figure 7-15. Most of the code in this project comes from the

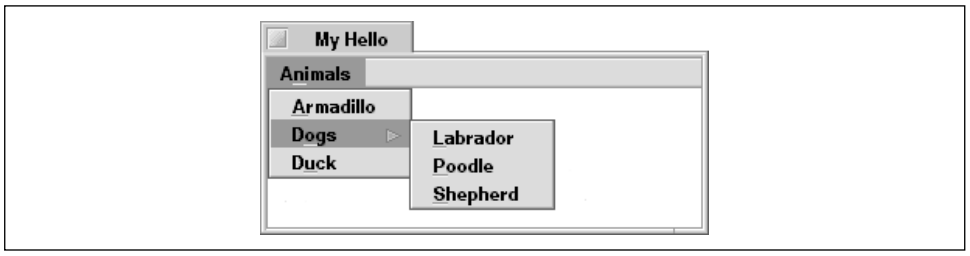

Figure 7-16. Categorizing things using a menu and submenu

TwoMenus project, along with new code supporting the new submenu. The `MyHelloWindow` class now holds a new `int32` data member named `fNumShapes` that keeps track of the number of circles or squares that should be used when Draw Circles or Draw Squares is selected:

```
class MyHelloWindow : public BWindow {
   ...
   ...
   private:
      MyDrawView     *fMyView;
      BButton        *fButtonBeep;
      BMenuBar       *fMenuBar;
      int32          fNumBeeps;
      int32          fNumShapes;
};
```

The `MyHelloWindow` constructor includes new code that adds a separator item to the Visual menu and creates and initializes the Number of Shapes submenu that's now housed as the last item in the Visual menu. Here's a part of the `MyHelloWindow` constructor:

```
BMenu      *menu;
BMenu      *subMenu;
BMenuItem  *menuItem;

// create menubar and add to window

// create Audio menu, add to menubar, add items to it, set
// to radio mode and mark one item

menu = new BMenu("Visual");
fMenuBar->AddItem(menu);
menu->AddItem(new BMenuItem("Draw Circles",
                            new BMessage(MENU_DRAW_CIRCLES_MSG)));
menu->AddItem(new BMenuItem("Draw Squares",
                            new BMessage(MENU_DRAW_SQUARES_MSG)));

menu->AddSeparatorItem();

subMenu = new BMenu("Number of Shapes");
menu->AddItem(subMenu);
```

```
subMenu->AddItem(menuItem = new BMenuItem("10",
                                 new BMessage(MENU_10_SHAPES_MSG)));
subMenu->AddItem(new BMenuItem("20", new BMessage(MENU_20_SHAPES_MSG)));
subMenu->AddItem(new BMenuItem("30", new BMessage(MENU_30_SHAPES_MSG)));

subMenu->SetRadioMode(true);
menuItem->SetMarked(true);
fNumShapes = 10;
```

As shown, a submenu can be set to radio mode, and an item in the submenu can be marked, just as is done for a menu that's added to a menubar.

The `MessageReceived()` function needs three new `case` sections—one to handle each of the three new messages that result from the submenu item selections:

```
case MENU_10_SHAPES_MSG:
   fNumShapes = 10;
   break;

case MENU_20_SHAPES_MSG:
   fNumShapes = 20;
   break;

case MENU_30_SHAPES_MSG:
   fNumShapes = 30;
   break;
```

Two of the existing `case` sections in `MessageReceived()` need modification. Now the number of shapes to use in the drawing of the concentric circles or squares gets passed to the `MyDrawView` member function `SetViewPicture()`:

```
case MENU_DRAW_CIRCLES_MSG:
fMyView->SetViewPicture(PICTURE_CIRCLES, fNumShapes);
   fMyView->Invalidate();
   break;

case MENU_DRAW_SQUARES_MSG:
fMyView->SetViewPicture(PICTURE_SQUARES, fNumShapes);
   fMyView->Invalidate();
   break;
```

The `MyDrawView` member function `SetViewPicture()` makes use of the new parameter as the index that determines how many times `InsetRect()` and `StrokeRect()` are called.