8

In this chapter:
- *Fonts*
- *Simple Text*
- *Editable Text*
- *Scrolling*

Text

The BeOS makes it simple to display text in a view—you've seen several examples of calling the `BView` functions `SetFont()` and `DrawString()` to specify which font a view should use and then draw a line of text. This approach works fine for small amounts of plain text; your application, however, is more likely to be rich in both graphics and text—so you'll want to take advantage of the `BFont`, `BStringView`, `BTextView`, `BScrollBar`, and `BScrollView` classes.

The `BFont` class creates objects that define the characteristics of fonts. You create a `BFont` object based on an existing font, then alter any of several characteristics. The BeOS is quite adept at manipulating fonts. You can alter basic font features such as size and spacing, but you can also easily change other more esoteric font characteristics such as shear and angle of rotation. You can use this new font in subsequent calls to `DrawString()`, or as the font in which text is displayed in `BStringView`, `BTextView`, or `BScrollView` objects.

A `BStringView` object displays a line of text, as a call to the `BView` function `DrawString()` does. Because the text of a `BStringView` exists as an object, this text knows how to update itself—something that the text produced by a call to `DrawString()` doesn't know how to do.

More powerful than the `BStringView` class is the `BTextView` class. A `BTextView` object is used to display small or large amounts of editable text. The user can perform standard editing techniques (such as cut, copy, and paste) on the text of a `BTextView` object. And the user (or the program itself) can alter the font or font color of some or all of the text in such an object.

If the text of a `BTextView` object extends beyond the content area of the object, a scrollbar simplifies the user's viewing. The `BScrollBar` class lets you add a scrollbar to a `BTextView`. Before adding that scrollbar, though, you should consider

creating a `BScrollView` object. As its name implies, such an object has built-in support for scrollbars. Create a `BTextView` object to hold the text, then create a `BScrollView` object that names the text view object as the scroll view's target. Or, if you'd like to scroll graphics rather than text, name a `BView` object as the target and then include a `BPicture` in that `BView`. While this chapter's focus is on text, it does close with an example adding scrollbars to a view that holds a picture.

Fonts

In the BeOS API, the `BFont` class defines the characteristics of a font—its style, size, spacing, and so forth. While the `BFont` class has not been emphasized in prior chapters, it has been used throughout this book. Every `BView` object (and thus every `BView`-derived object) has a current font that affects text displayed in that view. In previous examples, the `BView`-derived `MyDrawView` class used its `AttachedToWindow()` function to call a couple of `BView` functions to adjust the view's font: `SetFont()` to set the font, and `SetFontSize()` to set the font's size:

```
void MyDrawView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(24);
}
```

A view's current font is used in the display of characters drawn using the `BView` function `DrawString()`. Setting a view's font characteristics in the above fashion affects text produced by calls to `DrawString()` in each `MyDrawView` object.

The above snippet illustrates that the examples to this point have done little to alter the look of a font. Making more elaborate modifications is an easy task. Later in this chapter, you'll use some of the following techniques on text displayed in text view objects—editable text objects based on the `BTextView` class.

System Fonts

When designing the interface for your application, you'll encounter instances where you want a consistent look in displayed text. For example, your application may have a number of windows that include instructional text. In such a case, you'll want the text to have the same look from window to window. To ensure that your application can easily do this, the BeOS defines three fonts guaranteed to exist and remain constant for the running of your application.

The three global system fonts

The three constant fonts, or *global system fonts*, are `BFont` objects. When an application launches, these `BFont` objects are created, and three global pointers are

assigned to reference them. Table 8-1 shows these global `BFont` objects. Figure 8-1 shows a window running on my machine; the figure includes a line of text written in each of the three system fonts.

Table 8-1. Global Fonts and Their Usage

BFont Global Pointer	Common Font Usage
`be_plain_font`	Controls, such as checkboxes and buttons, have their labels displayed in this font. Menu items also appear in this font.
`be_bold_font`	Window titles appear in this font.
`be_fixed_font`	This font is used for proportional, fixed-width characters.

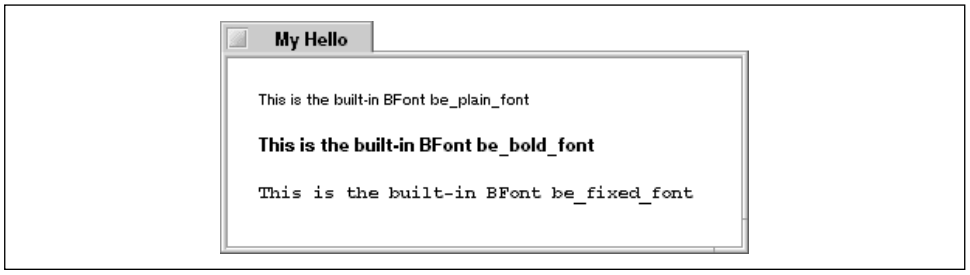

Figure 8-1. An example of text produced from the three global fonts

Contradictory as it sounds, the user can change the font that's used for any of the global system fonts. Figure 8-2 shows that the FontPanel preferences program lets the user pick a different plain, bold, or fixed font. This means that your application can't count on a global font pointer (such as `be_plain_font`) always representing the same font on all users' machines. You can, however, count on a global font pointer to always represent only a single font on any given user's machine—regardless of which font that is. So while you may not be able to anticipate what font the user will view when you make use of a global font pointer in your application, you are assured that the user will view the same font each time that global font pointer is used by your application.

Using a global system font

You've already seen how to specify one of the global fonts as the font to be used by a particular view: just call the `BView` function `SetFont()` within one of the view's member functions. The `AttachedToWindow()` snippet that appears above provides an example. That method initializes all of the objects of a particular class to use the same font. In the above example, all `MyDrawView` objects will initially display text in the font referenced by `be_bold_font`. For a particular view to have its current font set to a different system font, have that view call `SetFont()` after the view has been created:

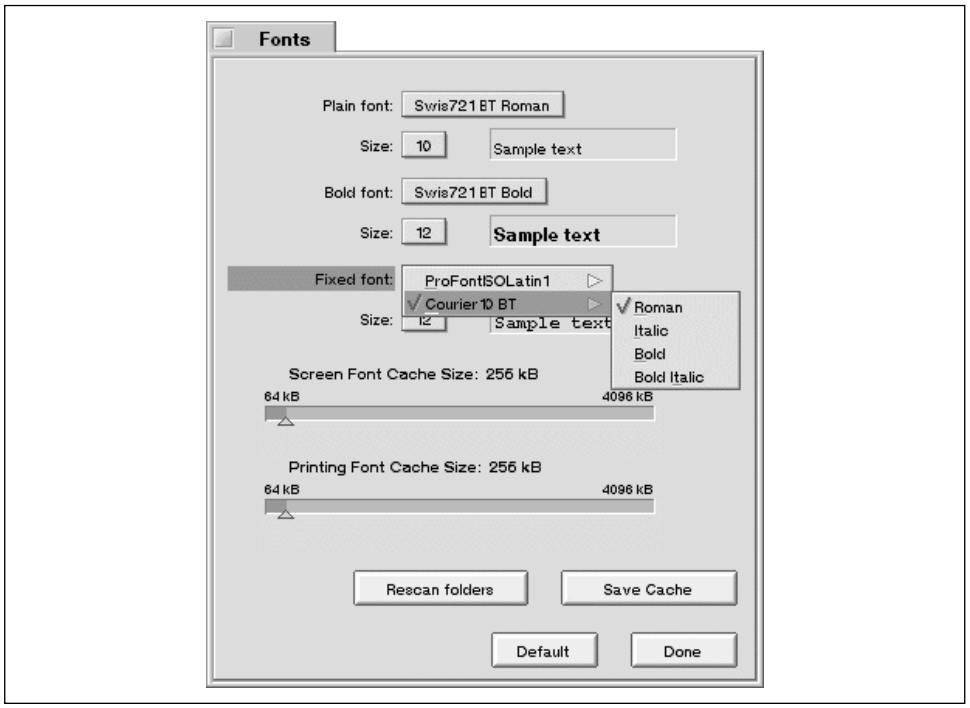

Figure 8-2. The FontPanel preferences application window

```
MyDrawView  *theDrawView;

theDrawView = new MyDrawView(frameRect, "MyDrawView");
theDrawView->SetFont(be_plain_font);
```

While a BeOS machine may have more than the three system fonts installed, your application shouldn't make any font-related assumptions. You can't be sure every user has a non-system font your application uses; some users may experience unpredictable results when running your application. If you want your program to display text that looks different from the global fonts (such as a very large font like 48 points), you can still use a global font to do so, as the next section illustrates.

Your program shouldn't force the user to have a particular non-system font on his or her machine, but it can give the user the option of displaying text in a non-system font. Consider a word processor you're developing. The default font should be `be_plain_font`. But your application could have a Font menu that allows for the display of text in any font on the user's computer. Querying the user's machine for available fonts is a topic covered in the `BFont` section of the Interface Kit chapter of the Be Book.

Global fonts are not modifiable

A global font is an object defined to be constant, so it can't be altered by an application. If a program could alter a global font, the look of text in other applications would be affected. Instead, programs work with copies of global fonts. While calling a `BView` function such as `SetFontSize()` may seem to be changing the size of a font, it's not. A call to `SetFontSize()` simply specifies the point size at which to display characters. The font itself isn't changed—the system simply calculates a new size for each character and displays text using these new sizes. Consider this snippet:

```
MyDrawView  *drawView1;
MyDrawView  *drawView2;

drawView1 = new MyDrawView(frameRect1, "MyDrawView1");
drawView1->SetFont(be_bold_font);
drawView1->SetFontSize(24);

drawView2 = new MyDrawView(frameRect2, "MyDrawView2");
drawView2->SetFont(be_bold_font);

drawView1->MoveTo(20.0, 20.0);
drawView1->DrawString("This will be bold, 24 point text");

drawView2->MoveTo(20.0, 20.0);
drawView2->DrawString("This will be bold, 12 point text");
```

This code specifies that the `MyDrawView` object `drawView1` use the `be_bold_font` in the display of characters. The code also sets this object to display these characters in a 24-point size. The second `MyDrawView` object, `drawView2`, also uses the `be_bold_font`. When drawing takes place in `drawView1`, it will be 24 points in size. A call to `DrawString()` from `drawView2` doesn't result in 24-point text, though. That's because the call to `SetFontSize()` didn't alter the font `be_bold_font` itself. Instead, it only marked the `drawView2` object to use 24 points as the size of text it draws.

Making global fonts unmodifiable is a good thing, of course. Having a global font remain static means that from the time your application launches until the time it terminates, you can always rely on the font having the same look. Of course, there will be times when your application will want to display text in a look that varies from that provided by any of the three global fonts. That's the topic of the next section.

Altering Font Characteristics

If you want to display text in a look that doesn't match one of the system fonts, and you want to be able to easily reuse this custom look, create your own `BFont`

object. Pass the `BFont` constructor one of the three global system fonts and the constructor will return a copy of it to your application:

```
BFont  theFont(be_bold_font);
```

The `BFont` object `theFont` is a copy of the font referenced by `be_bold_font`, so `theFont` can be modified. To do that, invoke the `BFont` member function appropriate for the characteristic to change. For instance, to set the font's size, call `SetSize()`:

```
theFont.SetSize(15.0);
```

A look at the `BFont` class declaration in the *Font.h* BeOS API header file hints at some of the other modifications you can make to a `BFont` object. Here's a partial listing of the `BFont` class:

```
class BFont {
   public:
               BFont();
               BFont(const BFont &font);
               BFont(const BFont *font);

      void     SetFamilyAndStyle(const font_family family,
                                 const font_style style);
      void     SetFamilyAndStyle(uint32 code);
      void     SetSize(float size);
      void     SetShear(float shear);
      void     SetRotation(float rotation);
      void     SetSpacing(uint8 spacing);
      ...
      void     GetFamilyAndStyle(font_family *family, font_style *style)
                  const;
      uint32   FamilyAndStyle() const;
      float    Size() const;
      float    Shear() const;
      float    Rotation() const;
      uint8    Spacing() const;
      uint8    Encoding() const;
      uint16   Face() const;
      ...
      float    StringWidth(const char *string) const;
      ...
}
```

For each member function that sets a font trait, there is a corresponding member function that returns the same trait. An examination of a few of these font characteristics provides a basis for understanding how fonts are manipulated.

Font size

An example of setting a `BFont` object's point size was shown above. An example of determining the current point size of that same `BFont` object follows.

```
float  theSize;

theSize = theFont.Size();
```

You've already seen that in order for a view to make use of a font, that font needs to become the view's current font. The `BView` function `SetFont()` performs that task. Numerous examples have demonstrated this routine's use in setting a view's font to one of the global system fonts, but you can use `SetFont()` with any `BFont` object. Here, one view is having its font set to the global system font `be_plain_font`, while another is having its font set to an application-defined `BFont` object:

```
BFont  theFont(be_bold_font);

theFont.SetSize(20.0);
drawView1->SetFont(&theFont);

drawView2->SetFont(be_plain_font);
```

This snippet demonstrates how to replace whatever font a view is currently using with another font—the `drawView1` view was making use of some font before the call to `SetFont()`. There will be times when you won't want to replace a view's font, but rather simply alter one or more of the traits of the view's current font. To do that, call the `BView` function `GetFont()` to first get a copy of the view's current font. Make the necessary changes to this copy, then call `SetFont()` to make it the view's new current font. Here, a view's current font has its size changed:

```
BFont  theFont;

theDrawView->GetFont(&theFont);
theFont.SetSize(32.0);
theDrawView->SetFont(&theFont);
```

Font shear

A font's shear is the slope, or angle, at which the font's characters are drawn. Pass the `BFont` function `SetShear()` a value in degrees and the routine will use it to adjust the amount of slope characters have. The range of values `SetShear()` accepts is 45.0 to 135.0. As Figure 8-3 shows, this angle is relative to the baseline on which characters are drawn. You'll also note that the degrees are measured clockwise. A value of 45.0 produces the maximum slant to the left, while a value of 135.0 produces the maximum slant to the right. The following code generates the three strings shown in Figure 8-3:

```
BFont theFont(be_plain_font);

theFont.SetSize(24.0);
theFont.SetShear(45.0);
theView->SetFont(&theFont);
theView->MovePenTo(110.0, 60.0);
```

```
theView->DrawString("Shear 45");

theFont.SetShear(90.0);
theView->SetFont(&theFont);
theView->MovePenTo(110.0, 140.0);
theView->DrawString("Shear 90");

theFont.SetShear(135.0);
theView->SetFont(&theFont);
theView->MovePenTo(110.0, 220.0);
theView->DrawString("Shear 135");
```

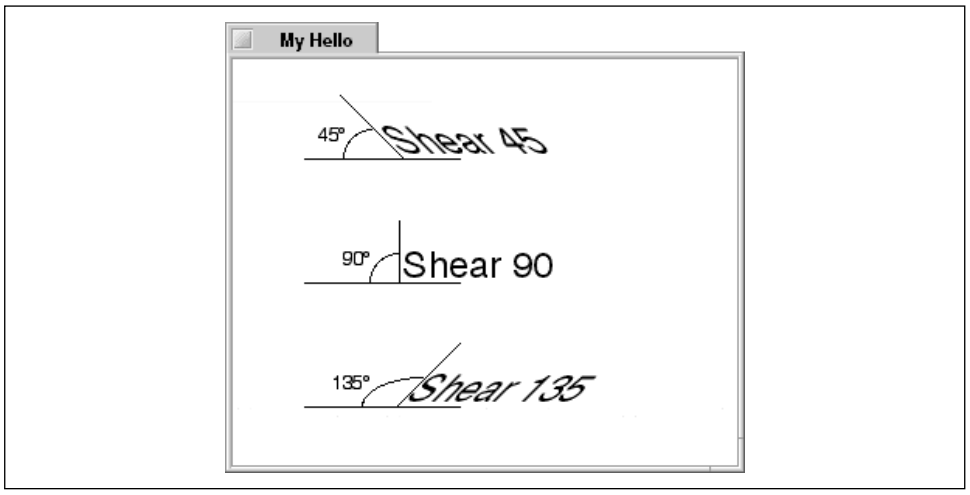

Figure 8-3. Output of text when the font's shear is varied

Font rotation

The `SetRotation()` function in the `BFont` class makes it easy to draw text that's rotated to any degree. Pass `SetRotation()` a value in degrees, and subsequent text drawn to the view will be rotated. The degrees indicate how much the baseline on which text is drawn should be rotated. Figure 8-4 shows that the angle is relative to the original, horizontal baseline. Degrees are measured clockwise: a positive rotation means that subsequent text will be drawn at an angle upward, while a negative rotation means that text will be drawn at an angle downward. This next snippet produces the text shown in the window in Figure 8-4:

```
BFont theFont(be_plain_font);

theFont.SetSize(24.0);
theFont.SetRotation(45.0);
theView->SetFont(&theFont);
theView->MovePenTo(70.0, 110.0);
theView->DrawString("Rotate 45");

theFont.SetRotation(-45.0);
```

```
theView->SetFont(&theFont);
theView->MovePenTo(190.0, 110.0);
theView->DrawString("Rotate -45");
```

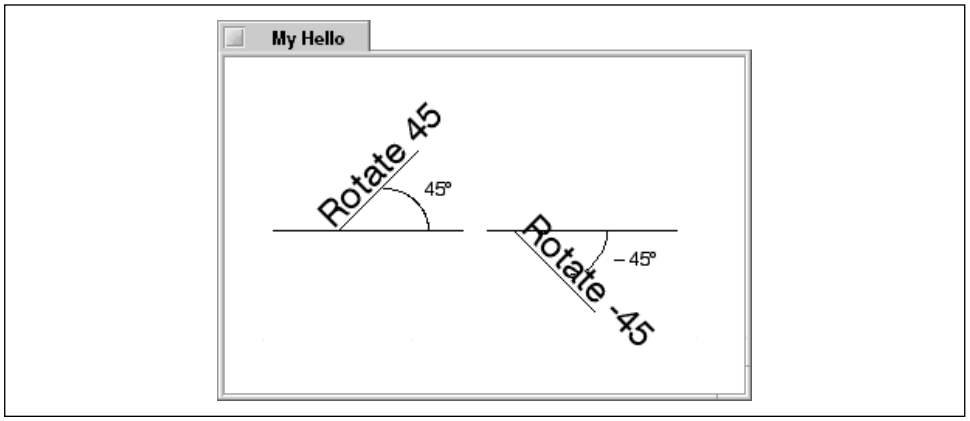

Figure 8-4. Output of text when the font's rotation is varied

Fonts Example Project

The FontSetting project demonstrates how to create `BFont` objects and use them as a view's current font. As Figure 8-5 shows, this example also demonstrates how to set the angle at which text is drawn, as well as how to rotate text.

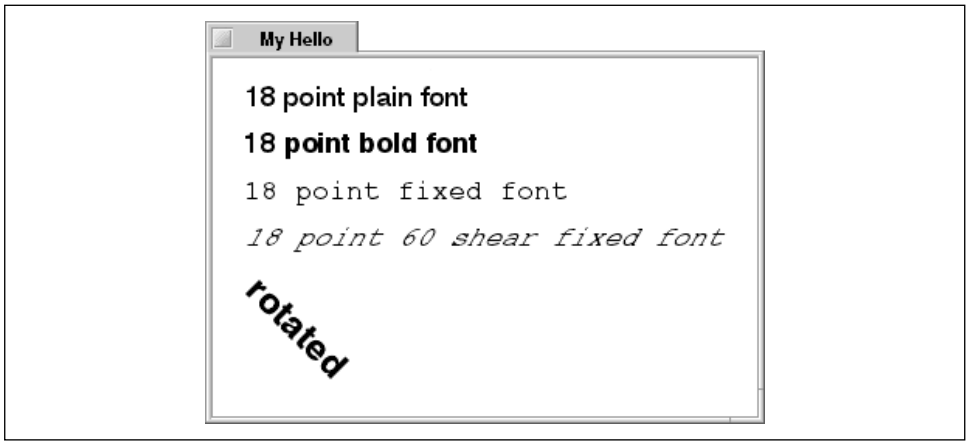

Figure 8-5. The FontSetting example program's window

I won't need a sophisticated program to show off a few of the things that can be done with fonts; a single menuless window will do. The FontSetting project's `MyHelloWindow` class has only one data member: the familiar drawing view `fMyView`. The `MyDrawView` class has no data members. Both the `MyDrawView`

constructor and the `MyDrawView` function `AttachedToWindow()` are empty. The only noteworthy function is the `MyDrawView` routine `Draw()`, shown here:

```
void MyDrawView::Draw(BRect)
{
   SetFont(be_plain_font);
   SetFontSize(18);
   MovePenTo(20, 30);
   DrawString("18 point plain font");

   SetFont(be_bold_font);
   SetFontSize(18);
   MovePenTo(20, 60);
   DrawString("18 point bold font");

   SetFont(be_fixed_font);
   SetFontSize(18);
   MovePenTo(20, 90);
   DrawString("18 point fixed font");

   BFont font;
   GetFont(&font);
   font.SetShear(120.0);
   SetFont(&font);
   MovePenTo(20, 120);
   DrawString("18 point 60 shear fixed font");

   SetFont(be_bold_font);
   GetFont(&font);
   font.SetSize(24.0);
   font.SetRotation(-45.0);
   SetFont(&font);
   MovePenTo(20, 150);
   DrawString("rotated");
}
```

The code in `Draw()` falls into five sections, each section ending with a call to `DrawString()`. Each of the first three sections:

- Sets the view's font to one of the three system fonts
- Sets the view to draw text in 18-point size
- Moves the pen to the starting location for drawing
- Draws a string

To draw each of the first three lines of text in 18-point size, note that after each call to `SetFont()`, `SetFontSize()` needs to be called. That's because a call to `SetFont()` uses all of the characteristics of the passed-in font. Thus, the second call to `SetFont()`—the call that sets the drawing view to draw in `be_bold_font`—sets the view to draw text in whatever point size the user defines for the `be_bold_font` (defined for the bold font in the FontPanel preferences window).

The fourth code section demonstrates how to change one aspect of a view's current font without affecting the font's other attributes. A call to `GetFont()` returns a copy of the view's current font. A call to the `BFont` function `SetShear()` alters the shear of the font. A call to `SetFont()` then establishes this font as the view's new current font.

The final section of code provides a second example of changing some characteristics of a view's current font without overwriting all of its traits. Here the view's font is set to `be_bold_font`, a copy is retrieved, and the size and rotation of the copied font are changed. This new font is then used as the view's current font before drawing the string "rotated."

Simple Text

Throughout this book you've seen that you can draw a string in any view by invoking the `BView`'s `DrawString()` function. `DrawString()` is a handy routine because it's easy to use—just call `MovePenTo()` or `MovePenBy()` to establish the starting point for a string, then pass `DrawString()` the text to draw. Drawing text with `DrawString()` has one distinct shortcoming, though. Unless the call is made from within the view's `Draw()` function, the text drawn by `DrawString()` won't automatically be updated properly whenever all or part of the text comes back into view after being obscured. A call to `DrawString()` simply draws text—it doesn't create a permanent association between the text and the view, and it doesn't create any kind of string object with the power to update itself. The `BStringView` class exists to overcome these deficiencies.

A `BStringView` object draws a single line of text, just as `DrawString()` does. Unlike the `DrawString()` text, however, the `BStringView` object's text automatically gets updated whenever necessary. While the text displayed by the `BStringView` object can be changed during runtime (see the "Setting the text in a string" section ahead), it isn't user-editable. It also doesn't word-wrap, and it can't be scrolled. That makes a `BStringView` object ideal for creating simple, static text such as that used for a label, but undesirable for displaying large amounts of text or user-editable text. For working with more sophisticated text objects, refer to the description of the `BTextView` class in this chapter's "Editable Text" section.

The BStringView Class

Create a `BStringView` object by invoking the `BStringView` constructor. The `BStringView` class is derived from the `BView` class. In creating a new string view object, the `BStringView` constructor passes all but its `text` parameter on to the `BView` constructor:

```
BStringView(BRect frame,
            const char *name,
```

```
        const char *text,
        uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
        uint32 flags = B_WILL_DRAW)
```

The `frame` parameter is a rectangle that defines the boundaries of the view. The text displayed by the `BStringView` object won't word wrap within this rectangle, so it must have a width sufficient to display the entire string. The `name` parameter defines a name by which the view can be identified at any time. The `resizingMode` parameter specifies the behavior of the view in response to a change in the size of the string view's parent view. The `flags` parameter is a mask consisting of one or more Be-defined constants that determine the kinds of notifications the view is to respond to.

The `text` parameter establishes the text initially displayed by the `BStringView` object. The text can be passed between quotes or, as shown below, a variable of type `const char *` can be used as the `text` argument. After creating the string view object, call `AddChild()` to add the new object to a parent view:

```
BStringView  *theString;
BRect        stringFrame(10.0, 10.0, 250.0, 30.0);
const char   *theText = "This string will be automatically updated";

theString = new BStringView(stringFrame, "MyString", theText);
AddChild(theString);
```

For simplicity, this snippet hardcodes the string view's boundary. Alternatively, you could rely on the `StringWidth()` function to determine the pixel width of the string and then use that value in determining the coordinates of the view rectangle. In Chapter 7, *Menus*, this routine was introduced and discussed as a `BView` member function. Here you see that the `BFont` class also includes such a function. By default, a new `BStringView` object uses the `be_plain_font` (which is a global `BFont` object), so that's the object to use when invoking `StringWidth()`. Here, I've modified the preceding snippet to use this technique:

```
#define      FRAME_LEFT    10.0

BStringView  *theString;
BRect        stringFrame;
const char   *theText = "This string will be automatically updated";
float        textWidth;

textWidth = be_plain_font->StringWidth(theText);
stringFrame.Set(FRAME_LEFT, 10.0, FRAME_LEFT + textWidth, 30.0);

theString = new BStringView(stringFrame, "MyString", theText);
AddChild(theString);
```

Manipulating the Text in a String

Once a string view object is created, its text can be altered using a variety of BStringView member functions.

Setting the text in a string

The text of a BStringView object isn't directly editable by the user, but the program can change it. To do that, invoke the BStringView function SetText(), passing the new text as the only parameter. In the following snippet, the text of the string view object created in the previous snippet is changed from "This string will be automatically updated" to "Here's the new text":

```
theString->SetText("Here's the new text");
```

To obtain the current text of a string view object, call the BStringView member function Text():

```
const char  *stringViewText;

stringViewText = theString->Text();
```

Aligning text in a string

By default, the text of a BStringView object begins at the left border of the object's frame rectangle. You can alter this behavior by invoking the BStringView member function SetAlignment(). This routine accepts one of three Be-defined alignment constants: B_ALIGN_LEFT, B_ALIGN_RIGHT, or B_ALIGN_CENTER. Here the left-aligned default characteristic of the text of the BStringView object theString is altered such that it is now right-aligned:

```
theString->SetAlignment(B_ALIGN_RIGHT);
```

You can obtain the current alignment of a BStringView object's text by invoking the BStringView function Alignment(). This routine returns a value of type alignment. Unsurprisingly, the constants B_ALIGN_LEFT, B_ALIGN_RIGHT, and B_ALIGN_CENTER are of this type, so you can compare the returned value to one or more of these constants. Here, the alignment of the text in a BStringView object is checked to see if it is currently centered:

```
alignment  theAlignment;

theAlignment = theString->Alignment();

if (theAlignment == B_ALIGN_CENTER)
   // you're working with text that is centered
```

Changing the look of the text in the string

A new `BStringView` object's text is displayed in black and in the system plain font. A `BStringView` object is a `BView` object, so `BView` member functions such as `SetHighColor()`, `SetFont()`, and `SetFontSize()` can be invoked to change the characteristics of a string view object's text. Here, the color of the text of a `BStringView` object is changed from black to red by altering the string view's high color. The text's font and size are changed as well:

```
rgb_color  redColor = {255, 0, 0, 255};
theString->SetHighColor(redColor);
theString->SetFont(be_bold_font);
theString->SetFontSize(18);
```

You can make more sophisticated changes to the look of the text displayed in a `BStringView` object by creating a `BFont` object, modifying any of the font's characteristics (using the techniques shown in this chapter's "Fonts" section), and then using that font as the `BStringView` object's font. Here, the font currently used by a string view object is retrieved, its shear changed, and the altered font is again used as the string view object's font:

```
BFont theFont;

theString->GetFont(&theFont);
theFont.SetShear(100.0);
theString->SetFont(&theFont);
```

String View Example Project

The StringView project produces the window shown in Figure 8-6. The "Here's the new text" string is a `BStringView` object, so the text is automatically redrawn after the user obscures the window and then reveals it again. The Text menu holds a single item named `Test` that, when selected, does nothing more than generate a system beep. Subsequent examples in this chapter add to this menu.

Figure 8-6. The StringView example program's window

The `BStringView` object will be added to the window's main view—the window's one `MyDrawView` object. To make it easy for you to manipulate the string later in the program, I keep track of the string by making it a data member in the `MyDrawView` class.

```
class MyDrawView : public BView {

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

   private:
      BStringView       *fString;
};
```

The `BStringView` object's frame rectangle has a left boundary of 10 pixels. The `BStringView` object's parent view is the window's `fMyView` view. The width of the `MyDrawView` `fMyView` is the same as the window, so the default state for the `BStringView` text has the text starting 10 pixels from the left edge of the window. Figure 8-6 makes it clear that this isn't the starting point of the text. A call to `SetAlignment()` is responsible for this discrepancy—the string view object's text has been changed to right-aligned. The text's look has been changed from its default state by calling the `BView` functions `SetFont()` and `SetFontSize()`. You can't tell from Figure 8-6 that the text appears in red rather than black. It's a call to `SetHighColor()` that makes this color change happen. Here's the StringView project's `MyDrawView` constructor, which shows all the pertinent code:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect   stringFrame(10.0, 10.0, 250.0, 30.0);

   fString = new BStringView(stringFrame, "MyString",
                             "This string will be automatically updated");
   AddChild(fString);

   fString->SetText("Here's the new text");

   fString->SetAlignment(B_ALIGN_RIGHT);

   rgb_color  redColor = {255, 0, 0, 255};
   fString->SetHighColor(redColor);

   fString->SetFont(be_bold_font);
   fString->SetFontSize(18);
}
```

Editable Text

A `BStringView` object is ideal for displaying a small amount of uneditable text. When your application needs to display a larger amount of text that is user-editable, though, it's time to switch to a `BTextView` object. A `BTextView` object automatically implements keyboard editing, and makes it easy to add menu edit-

ing. And while a text view object initially displays all its text in a single font and a single color, you can easily alter the object to support multiple fonts and multiple colors—even within the same paragraph.

The BTextView Class

The `BTextView` class used to create an editable text object is derived from the `BView` class. So, as expected, several of the `BTextView` constructor parameters will be immediately familiar to you:

```
BTextView(BRect       frame,
          const char  *name,
          BRect       textRect,
          uint32      resizingMode,
          uint32      flags)
```

The `frame`, `name`, and `resizingMode` parameters serve the same purposes as they do for the `BView` class. The `flags` parameter is made up of one or more Be-defined constants that determine the kinds of notifications the view is to respond to. Regardless of which constant or constants you pass as the `flags` parameter, the `BTextView` constructor goes behind your back to add a couple more constants before forwarding `flags` to the `BView` constructor it invokes. These two `BTextView`-added constants are `B_FRAME_EVENTS`, to allow the `BTextView` object to reformat its text when it is resized, and `B_PULSE_NEEDED`, to allow the text insertion caret to blink properly.

The one `BTextView` constructor parameter unique to the `BTextView` class is `textRect`. This rectangle specifies the boundaries for the text that will eventually be placed in the `BTextView` object.

BTextView frame and text rectangles

At first glance, the purpose of the `BTextView` constructor's `textRect` rectangle may seem to be redundant—the `frame` parameter is also a boundary-defining rectangle. Here's the difference: the `frame` rectangle defines the size of the `BTextView` object, as well as where the `BTextView` object resides in its parent view. The `textRect` parameter defines the size of the text area within the `BTextView` object, and where within the `BTextView` object this text area is to be situated. By default, a `BTextView` object has a frame the size of the `frame` rectangle drawn around it. The `textRect` rectangle doesn't have a frame drawn around it. Thus, the `textRect` rectangle provides for a buffer, or empty space, surrounding typed-in text and the `BTextView` object's frame. Figure 8-7 illustrates this.

In Figure 8-7, the dark-framed rectangle represents the `frame` rectangle, the first parameter to the `BTextView` constructor. The light-framed rectangle represents the `textRect` rectangle. Neither of these rectangles would be visible to the user; I've shown them in the figure only to make it obvious where their boundaries are in

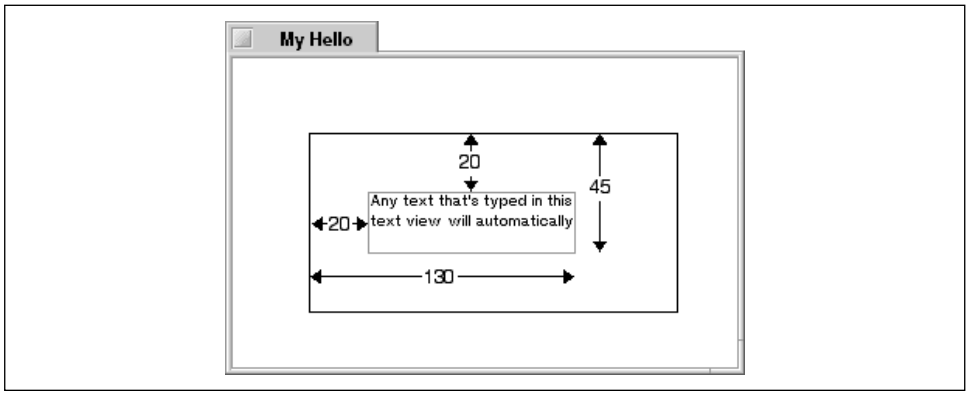

Figure 8-7. A BTextView object consists of two rectangles

this particular example. The arrows would not be in the window either—I've added them to make it clear that the coordinates of the `textBounds` rectangle are relative to the `viewFrame` rectangle. Here's the code that sets up a `BTextView` object like the one shown in Figure 8-7:

```
BTextView  *theTextView;
BRect      viewFrame(30.0, 30.0, 200.0, 110.0);
BRect      textBounds(20.0, 20.0, 130.0, 45.0);

theTextView = new BTextView(viewFrame, "TextView", textBounds,
                            B_FOLLOW_NONE, B_WILL_DRAW);
AddChild(theTextView);
```

In this snippet, the `viewFrame` rectangle defines the text view object frame to be 170 pixels wide by 80 pixels high. The `textBounds` rectangle specifies that the first character typed into the text view object will have 20 pixels of white space between the object's left edge and the character and 20 pixels of white space between the object's top edge and the top of the character. The `textBounds` rectangle's right boundary, 130, means there will be 40 pixels of white space between the end of a line of text and the text object's right boundary (see Figure 8-7).

While I've discussed at length the `BTextView` constructor parameters, I'm compelled to elaborate just a bit more on the two rectangles. Figure 8-7 and the accompanying code snippet exhibit a text object whose text area rectangle provides large and non-uniform borders between it and the text object itself. But it's much more typical to define a text area rectangle that has a small, uniform border. This example exaggerated the border size simply to make the relationship between the two rectangles clear.

Another point to be aware of is that the top and bottom coordinates of the text area rectangle become unimportant as the user enters text that exceeds the size of the text area rectangle. The bottom coordinate of the text area rectangle is always ignored—the text view object will accept up to 32K of text and will automatically

scroll the text as the user types, always displaying the currently typed characters. And as the text scrolls, the top coordinate of the text area rectangle becomes meaningless; the text view object will display the top line of scrolling text just a pixel or so away from the top of the text view object.

Text view example project

The TextView project displays a window like the one shown in Figure 8-8. To make the text view object's boundaries clear, the program outlines the object with a line one pixel in width. As it did for the StringView project, the Text menu holds a single item named Test. Choosing this item simply generates a system beep.

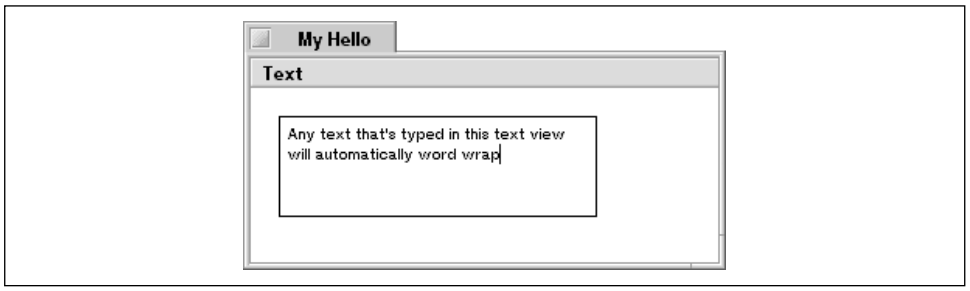

Figure 8-8. The TextView example program's window

The text view object will be added to the window-filling `MyDrawView`, so I've added a `BTextView` data member to the `MyDrawView` class:

```
class MyDrawView : public BView {

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

   private:
      BTextView      *fTextView;
};
```

The normally empty `MyDrawView` constructor now holds the code to create a `BTextView` object. The `viewFrame` rectangle defines the size and placement of the text view object. This rectangle is declared outside of the `MyDrawView` constructor because, as you see ahead, it sees additional use in other `MyDrawView` member functions. The `TEXT_INSET` constant is used in establishing the boundaries of the text view object's text area; that area will have a 3-pixel inset from each side of the text view object itself:

```
#define   TEXT_INSET   3.0

BRect  viewFrame(20.0, 20.0, 220.0, 80.0);
```

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

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new BTextView(viewFrame, "TextView", textBounds,
                             B_FOLLOW_NONE, B_WILL_DRAW);
   AddChild(fTextView);

   viewFrame.InsetBy(-2.0, -2.0);
}
```

After using `viewFrame` to establish the size and placement of the text view object, this rectangle's size is expanded by 2 pixels in each direction (recall from Chapter 5, *Drawing*, that a negative number as an argument to the `BView` member function `InsetBy()` moves the affected view's frame outward in one direction). This is done in preparation for drawing a border around the text view area.

Clicking on a text view object causes a blinking insertion point caret to appear in the text area of that object. The programmer can "jump start," or force, this caret to appear in a text view object by making the object the focus view. The final setup work for a `MyDrawView` object takes place in the `AttachedToWindow()` member function, so that's an appropriate enough place to make a call to the `BView` function `MakeFocus()`:

```
void MyDrawView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(12);

   fTextView->MakeFocus();
}
```

The `AttachedToWindow()` calls to `SetFont()` and `SetFontSize()` don't affect the font used in the text view object. They're called by the `MyDrawView` object, so they affect text drawn directly in such an object (there just doesn't happen to be any text drawn in this example's `MyDrawView` object). To change the font of a text view object, invoke the `BTextView` function `SetFontAndColor()` from the text view object. Refer to "Text Characteristics," in this chapter.

The `MyDrawView` constructor ended with the coordinates of the rectangle `viewFrame` being enlarged a couple of pixels in each direction. This was done to define a rectangle with boundaries just outside the boundaries of the text view object. When used as an argument to `StrokeRect()`, this rectangle provides a frame for the text view object. I've placed the call to `StrokeRect()` in the `MyDrawView Draw()` function so that this frame always gets appropriately updated:

```
void MyDrawView::Draw(BRect)
{
   StrokeRect(viewFrame);
}
```

You might be tempted to try to surround a text view object with a frame by simply calling `StrokeRect()` from the text view object. This won't work, because the text view object holds text, not graphics. Instead, draw the frame in the text view object's parent view as I've done above. The `fTextView` object was added to the `MyDrawView` object, so I draw the text view object's border in the `MyDrawView` object.

Text Editing

By default, the user can select and edit some or all of the text that appears in a text view object. `BTextView` member functions, along with several Be-defined message constants, provide you with a great degree of control over the level of editing you want to allow in each text view object in a window.

Default text editing

Regardless of which editing menu items you choose to include or not include in the menubar of the text object object's parent window, the following text editing shortcut keys are automatically supported:

- Command-x: Cut
- Command-c: Copy
- Command-v: Paste
- Command-a: Select All

You can verify that this notion of automatic text editing is true in practice by running the previous example program, TextView. Then type a few characters, select some or all of it, and press the Command and "X" keys. Even though the TextView project includes no text editing menu items and no text editing code, the selected text will be cut.

You can deny the user the ability to edit text in the text view object by calling the `BTextView` function `MakeEditable()`, passing a value of `false`:

```
fTextView->MakeEditable(false);
```

After disabling text editing, you can again enable editing by calling `MakeEditable()` with an argument of `true`. You can check the current editing state of a text object by calling `IsEditable()`:

```
bool  canEdit;

canEdit = fTextView->IsEditable();
```

If you disable text editing for a text object, you may also want to disable text selection. Like text editing, by default, text in a text object can be selected by clicking and dragging the mouse. If you disable text editing, the user will be able to select any number of characters in the text object. Since the user will be able to select and copy text, but won't be able to paste copied text back into the view, this could lead to some confusion. To prevent the user from selecting text by invoking the `BTextView` member function `MakeSelectable()`, pass a value of `false` as the sole argument:

```
fTextView->MakeSelectable(false);
```

You can enable text selection by again calling `MakeSelectable()`, this time with an argument of `true`. You can check the current text selection state of a text view object by calling `IsSelectable()`:

```
bool  canSelect;

canSelect = fTextView->IsSelectable();
```

Menu items and text editing

Users may not intuitively know that a text object automatically handles keyboard shortcuts for copying, cutting, pasting, and selecting all of the object's text. When it comes time to perform text editing, the user will no doubt look in the menus of a window's menubar for the basic editing menu items: the Cut, Copy, Paste, and Select All items. If you include one or more `BTextView` objects in a window of your program, you'd be wise to include these four menu items in an Edit menu.

As you've seen, a `BTextView` object automatically provides shortcut key editing—you don't need to write any code to enable the shortcut key combinations to work. The system also automatically supports menu item editing—menu item editing is easy to enable on a text view object, but you do need to write a little of your own code. While you don't do any work to give a text view object shortcut key editing, you do need to do a little work to give that same object menu editing. All you need to do is build a menu with any or all of the four basic editing items. If you include the proper messages when creating the menu items, editing

will be appropriately handled without any other application-defined code being present.

You're most familiar with the *system message*: a message that has a corresponding hook function to which the system passes the message. A different type of message the system recognizes and reacts to is the *standard message*. A standard message is known to the system, and may be issued by the system, but it doesn't have a hook function. Among the many standard messages the BeOS provides are four for editing, represented by the Be-defined constants `B_CUT`, `B_COPY`, `B_PASTE`, and `B_SELECT_ALL`. This brief definition of the standard message should tide you over until Chapter 9, *Messages and Threads*, where this message type is described in greater detail. The following snippet demonstrates how an Edit menu that holds a Cut menu item could be created. Assume that this code was lifted from the constructor of a `BWindow`-derived class constructor, and that a menubar referenced by a `BMenuBar` object named `fMenuBar` already exists:

```
BMenu      *menu;
BMenuItem  *menuItem;

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

menu->AddItem(menuItem = new BMenuItem("Cut", new BMessage(B_CUT), 'X'));
menuItem->SetTarget(NULL, this);
```

Recall from Chapter 7 that the first parameter to the `BMenuItem` constructor, `label`, specifies the new menu item's label—the text the user sees in the menu. The second parameter, `message`, associates a message with the menu item. The third parameter, `shortcut`, assigns a shortcut key to the menu item. To let the menu item be responsible for cutting text, you must pass the Be-defined `B_CUT` standard message constant as shown above. The other required step is to set the currently selected text view object as the destination of the message.

The `BInvoker` class exists to allow objects to send a message to a `BHandler` object. The `BMenuItem` class is derived from the `BInvoker` class, so a menu item object can be invoked to send a message to a target. That's exactly what happens when the user selects a menu item. A window object is a type of `BHandler` (the `BWindow` class is derived from `BHandler`), so it can be the target of a menu item message. In fact, by default, the target of a menu item is the window that holds the menu item's menubar. Typically, a menu item message is handled by the target window object's `MessageReceived()` function, as has been demonstrated at length in Chapter 7. While having the window as the message recipient is often desirable, it isn't a requirement. The `BInvoker` function `SetTarget()` can be invoked by a `BInvoker` object (such as a `BMenuItem` object) to set the message target to any other `BHandler` object. The above snippet calls `SetTarget()` to set the active text view object to be the Cut menu item's target.

The first parameter to `SetTarget()` is a `BHandler` object, while the second is a `BLooper` object. Only one of these two parameters is ever used; the other is always passed a value of `NULL`. I'll examine both possibilities next.

If the target object is known at compile time, you can pass it as the first argument and pass `NULL` as the second argument. If the window involved in the previous snippet had a single text view object referenced by an `fMyText` data member, the call to `SetTarget()` could look like this:

```
menuItem->SetTarget(fMyText, NULL);
```

If the window has more than one text view object, however, setting an editing menu item message to target one specific text view object isn't desirable—selecting the menu item won't have any effect on text that is selected in a text view object other than the one referenced by `fMyText`. The remedy in such a case is to call `SetTarget()` as shown here:

```
menuItem->SetTarget(NULL, this);
```

When `SetTarget()` is called with a first argument of `NULL`, the second argument is a `BLooper` object. Passing a looper object doesn't set the looper object itself as the target—it sets the looper object's *preferred handler* to be the target. An object's preferred handler is dependent on the object's type, and can vary as the program runs. If the above line of code appears in a `BWindow`-derived class constructor, the `this` argument represents the `BWindow`-derived object being created. In the case of a `BWindow`-derived object, the preferred handler is whichever of the window's `BHandler` objects is the focus object when the window receives a message. For editing, this makes perfect sense—you'll want an editing operation such as the cutting of text to affect the text in the current text view object.

Earlier I mentioned that the default target of a menu item message is the menu's window. In the previous call to `SetTarget()`, the second argument is `this`, which is the menu's window. If that makes it seem like the call to `SetTarget()` is redundant, keep in mind that the object passed as the second argument to `SetTarget()` doesn't become the new target. Instead, that object's preferred handler becomes the target.

Other editing menu items are implemented in a manner similar to the Cut menu item. This next snippet adds Cut, Copy, Paste, and Select All menu items to an Edit menu and, at the same time, provides a fully functional Edit menu that supports editing operations in any number of text view objects:

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

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

menu->AddItem(menuItem = new BMenuItem("Cut", new BMessage(B_CUT), 'X'));
menuItem->SetTarget(NULL, this);
menu->AddItem(menuItem = new BMenuItem("Copy", new BMessage(B_COPY), 'C'));
menuItem->SetTarget(NULL, this);
menu->AddItem(menuItem = new BMenuItem("Paste", new BMessage(B_PASTE), 'V'));
menuItem->SetTarget(NULL, this);
menu->AddItem(menuItem = new BMenuItem("Select All",
                                       new BMessage(B_SELECT_ALL), 'A'));
menuItem->SetTarget(NULL, this);
```

The handling of an edit item comes from the standard message (such as `B_CUT`) that you use for the `message` parameter in the invocation of the `BMenuItem` constructor—not from the string (such as `"Cut"`) that you use for the `label` parameter. While the user will be expecting to see the familiar Cut, Copy, Paste, and Select All menu item names, you could just as well give these items the names Expunge, Mimic, Inject, and Elect Every. From a more practical standpoint, a program designed for non-English speaking people can include native text in the edit menu.

Text editing menu item example project

The TextViewEdit project is a modification of this chapter's TextView project. As shown in Figure 8-9, four menu items have been added to the already present Test item in the Text menu. For simplicity, I've added these four editing items to the existing Text menu, but your application should stick with convention and include these items in a menu titled Edit.

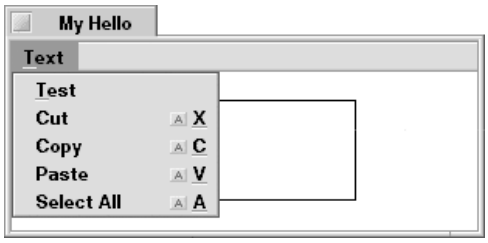

Figure 8-9. The TextViewEdit example program's window

For this TextViewEdit project, the `MyDrawView` class and the implementation of the `MyDrawView` member functions are all unchanged from the TextView project:

- `MyDrawView` class has a `BTextView` data member named `fTextView`.
- The `MyDrawView` constructor creates a `BTextView` object and assigns it to `fTextView`.
- `AttachedToWindow()` sets the focus view and sets up a rectangle to serve as a border for the text view object.
- `Draw()` draws the text view object's border.

The modifications to the project are all found in the `MyHelloWindow` constructor. Here, the four editing menu items are added to the already present Test menu item:

```
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;
   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("Text");
   fMenuBar->AddItem(menu);

   menu->AddItem(new BMenuItem("Test", new BMessage(TEST_MSG)));
   menu->AddItem(menuItem = new BMenuItem("Cut", new BMessage(B_CUT), 'X'));
   menuItem->SetTarget(NULL, this);
   menu->AddItem(menuItem = new BMenuItem("Copy", new BMessage(B_COPY),
                                          'C'));
   menuItem->SetTarget(NULL, this);
   menu->AddItem(menuItem = new BMenuItem("Paste", new BMessage(B_PASTE),
                                          'V'));
   menuItem->SetTarget(NULL, this);
   menu->AddItem(menuItem = new BMenuItem("Select All",
                                           new BMessage(B_SELECT_ALL), 'A'));
   menuItem->SetTarget(NULL, this);

   Show();
}
```

The MyHelloWindow class includes a MessageReceived() member function that handles only the first menu item in the Test menu: the Text item. If MessageReceived() receives a TEST_MSG message, a call to beep() is made. The system routes the other message types (B_CUT, B_COPY, B_PASTE, and B_SELECT_ALL) to the focus view (which in this example is the one text view object) for automatic handling by that view.

Text Characteristics

While the characteristics of the text displayed by a BStringView object can be altered by invoking BView functions such as SetFont() and SetHighColor(), the characteristics of the text displayed by a BTextView object should be altered by invoking member functions of the BTextView class. For instance, to change either the color or font, or both, of a BTextView object's text, invoke the BTextView function SetFontAndColor().

Getting BTextView text characteristics

A new BTextView object's text is displayed in black and in the system's current plain font. A BTextView object is a BView object, so you might expect that changes to the object's text would be carried out by BView member functions such as SetHighColor() and SetFont(). While this is in fact true, it's important to note that the calls to these BView functions are made indirectly. That is, the BTextView class provides its own set of graphics member functions that a BTextView object should invoke in order to affect the object's text. Each of these BTextView functions in turn invokes whatever BView functions are needed in order to carry out its specific task. The BTextView functions GetFontAndColor() and SetFontAndColor() are of primary importance in changing the characteristics of text displayed in a text view object.

GetFontAndColor() is your means to accessing a text view object's current font so that you can alter its properties. Here's the prototype:

```
void GetFontAndColor(BFont     *font,
                     uint32    *sameProperties,
                     rgb_color *color = NULL,
                     bool      *sameColor = NULL)
```

GetFontAndColor() returns information about BTextView in its four parameters. After you call GetFontAndColor(), the font parameter holds a copy of the text view object's font. The sameProperties parameter is a mask that specifies which of a number of the text view object's font properties apply to all of the characters in the current selection. GetFontAndColor() combines a number of Be-defined constants to create the mask. For instance, if all of the characters in the

selected text are the same size, a test of the returned value of `sameProperties` will reveal that it includes the value of the Be-defined constant `B_FONT_SIZE`:

```
// invoke GetFontAndColor() here

if (sameProperties && B_FONT_SIZE)
   // all characters are the same size
```

If no text is currently selected at the time of the call to `GetFontAndColor()`, the `sameProperties` mask will be set such that all of the properties test `true`. This makes sense because all of the selected characters—all none of them—do indeed share the same properties!

The third parameter, `color`, specifies the RGB color in which the text view object's text is to be displayed. The color returned in this parameter is that of the first character in the selected text (or the character following the insertion point if no text is currently selected). The `sameColor` parameter indicates whether or not all of the selected text (or all characters if no text is currently selected) is of the color returned by the `color` parameter.

If your only interest is in the font of a text view object, the `color` and `sameColor` parameters can be safely ignored—these two parameters have a default value of `NULL`. Here's an example:

```
BFont      font;
uint32     sameProperties;

fTextView->GetFontAndColor(&font, &sameProperties);
```

If, instead, your only interest is in the color of a text view object's text, pass `NULL` as the font parameter:

```
uint32     sameProperties;
rgb_color  color;
bool       sameColor;

fTextView->GetFontAndColor(NULL, &sameProperties, &color, &sameColor);
```

Alternatively, you can pass a variable for each of the four arguments, then simply ignore the returned values of the variables that aren't of interest.

A call to `GetFontAndColor()` doesn't affect a text view object's text in any way—it simply returns to your program information about the text. Once you've obtained a font and color, you'll want to make changes to one or the other, or both.

Setting BTextView text characteristics

After obtaining a copy of a `BTextView` object's font, you can make any desired changes to the font and then pass these changes back to the text view object. The same applies to the text's color. The `BTextView` function `SetFontAndColor()` takes care of both of these tasks:

```
void SetFontAndColor(const BFont  *font,
                     uint32       properties = B_FONT_ALL,
                     rgb_color    *color = NULL)
```

The `font`, `properties`, and `color` parameters are the variables filled in by a previous call to `GetFontAndColor()`. In between the calls to `GetFontAndColor()` and `SetFontAndColor()`, invoke one or more `BFont` functions to change the desired font trait. For instance, to change the size of the font used to display the text of a `BTextView` object named `theTextView`, invoke the `BFont` function `SetSize()` as shown in the following snippet. Note that because this snippet isn't intended to change the color of the text in the `theTextView` text view object, the call to `GetFontAndColor()` omits the `color` and `sameColor` parameters:

```
BFont   font;
uint32  sameProperties;

theTextView->GetFontAndColor(&font, &sameProperties);
font.SetSize(24.0);
theTextView->SetFontAndColor(&font, B_FONT_ALL);
```

In this snippet, the `BFont` object `font` gets its value from `GetFontAndColor()`, is altered by the call to `SetSize()`, and then is passed back to the `theTextView` object by a call to `SetFontAndColor()`. The process is not, however, the same for the `sameProperties` variable.

Recall that the `GetFontAndColor()` `uint32` parameter `sameProperties` returns a mask that specifies which font properties apply to all of the characters in the current selection. The `SetFontAndColor()` `uint32` parameter `properties`, on the other hand, is a mask that specifies which properties of the `BFont` parameter passed to `SetFontAndColor()` should be used in the setting of the text view object's font.

Consider the following example: your program declares a `BFont` variable named `theBigFont` and sets a variety of its properties (such as size, rotation, and shear), but you'd only like `SetFontAndColor()` to apply this font's size property to a view object's font (perhaps your program has set other characteristics of `theBigFont` because it plans to use the font elsewhere as well). To do that, pass the Be-defined constant `B_FONT_SIZE` as the second argument to `SetFontAndColor()`:

```
BFont  theBigFont(be_plain_font);

theBigFont.SetSize(48.0);
```

```
theBigFont.SetRotation(-45.0);
theBigFont.SetShear(120.0);
theTextView->SetFontAndColor(&theBigFont, B_FONT_SIZE);
```

In this snippet, you'll note that there's no call to `GetFontAndColor()`. Unlike previous snippets, this code doesn't obtain a copy of the current font used by `theTextView`, alter it, and pass the font back to `theTextView`. Instead, it creates a new font based on the system font `be_plain_font`, changes some of that font's characteristics, then passes this font to `SetFontAndColor()`. The important point to note is that passing `B_FONT_SIZE` as the second properties mask tells `SetFontAndColor()` to leave all of the characteristics of the font currently used by `theTextView` unchanged except for its size. The new size will be that of the first parameter, `theBigFont`. The font control definitions such as `B_FONT_SIZE` are located in the *View.h* header file.

To change the color of a `BTextView` object's text, assign an `rgb_color` variable the desired color and then pass that variable as the third parameter to `SetFontAndColor()`. Here, the `BTextView theTextView` is set to display the characters in the current selection in red:

```
BFont      font;
uint32     sameProperties;
rgb_color  redColor = {255, 0, 0, 255};

theTextView->GetFontAndColor(&font, &sameProperties);
theTextView->SetFontAndColor(&font, B_FONT_ALL, &redColor);
```

Allowing multiple character formats in a BTextView

By default, all of the text that is typed or pasted into a `BTextView` object shares the same characteristics. That is, it all appears in the same font, the same size, the same color, and so forth. If the object is left in this default state, user attempts to change the characteristics of some of the text will fail—all of the text in the object will take on whatever change is made to any selected text. To allow for multiple character formats in a single text view object, pass a value of `true` to the `BTextView` function `SetStylable()`.

```
theTextView->SetStylable(true);
```

To reverse the setting and prevent multiple character formats in a text view object, pass `SetStylable()` a value of `false`. Note that in doing this any styles that were previously applied to characters in the text view will now be lost. To test a text view object's ability to support multiple character formats, call the `BTextView` function `IsStylable()`:

```
bool  supportsMultipleStyles;

supportsMultipleStyles = theTextView->IsStylable();
```

Changing the background color in a BTextView

Changes to the characteristics of text in a `BTextView` object are achieved by using a number of `BFont` functions in conjunction with the `BTextView` functions `GetFontAndColor()` and `SetFontAndColor()`. Changes to the background color of the text view object itself are accomplished by calling an inherited `BView` function rather than a `BTextView` routine. The `BView` function `SetViewColor()`, which was introduced in Chapter 5, is called to change a text view object's background. The call changes the background color of the entire text view object (that is, it changes the background color of the text view object's boundary or framing rectangle—a rectangle that includes the text area rectangle). Here, the background of a `BTextView` object is changed to pink:

```
rgb_color  pinkColor = {200, 150, 200, 255};
theTextView->SetViewColor(pinkColor);
```

Aligning text in a BTextView

A characteristic of the text in a `BTextView` that isn't dependent on the font is the text's placement within the text area of the text view object. By default, the text the user enters into a `BTextView` object is left-aligned. You can change the alignment by invoking the `BTextView` member function `SetAlignment()`. This routine works just like the `BStringView` version described earlier in this chapter: it accepts one of the three Be-defined alignment constants. Pass `B_ALIGN_LEFT`, `B_ALIGN_RIGHT`, or `B_ALIGN_CENTER` and the text that's currently in the text view object's text rectangle will be appropriately aligned: each line in the text view object will start at the alignment indicated by the constant. Here, a `BTextView` object named `theText` has its alignment set to centered:

```
theText->SetAlignment(B_ALIGN_CENTER);
```

Any text subsequently entered in the affected `BTextView` object continues to follow the new alignment.

The `BTextView` member function `Alignment()` is used to obtain the current alignment of a text view object's text. The `BTextView` version of `Alignment()` works just like the `BStringView` version. In this next snippet, the alignment of the text in a `BTextView` object is compared to the `B_ALIGN_RIGHT` constant to see if the text is currently right-aligned:

```
alignment  theAlignment;

theAlignment = theText->Alignment();

if (theAlignment == B_ALIGN_RIGHT)
   // the text in this object is right-aligned
```

Another `BTextView` function that affects text placement is `SetWordWrap()`. By default, the text in a text view object word wraps: typed or pasted text fills the width of the object's text rectangle and then continues on the following line. Text can instead be forced to remain on a single line until a newline character (a hard return established by a press of the Return key) designates that a new line be started. Passing `SetWordWrap()` a value of `false` turns word wrapping off for a particular text view object, while passing a value of `true` turns word wrapping on. To test the current word wrapping state of a text view object, call the `BTextView` function `DoesWordWrap()`.

While you can change the font characteristics (such as size and color) of individual characters within a `BTextView` object, you can't change the alignment of individual characters, words, or lines of text within a single `BTextView` object. The setting of a `BTextView` object's alignment or wrapping affects *all* characters in the view.

Text characteristics example project

The TextViewFont project demonstrates how to add support for multiple character formats in a `BTextView` object. Figure 8-10 shows a window with a Text menu and a `BTextView` object similar to the one appearing in this chapter's TextView and TextViewEdit example projects. Making a text selection and then choosing Alter Text from the Text menu increases the size of the font of the selected text. For good measure (and to demonstrate the use of color in a `BTextView`), the Alter Text menu also changes the color of the selected text from black to red. In Figure 8-10, the current selection consists of parts of the first and second words of text, and the Alter Text item has just been selected.

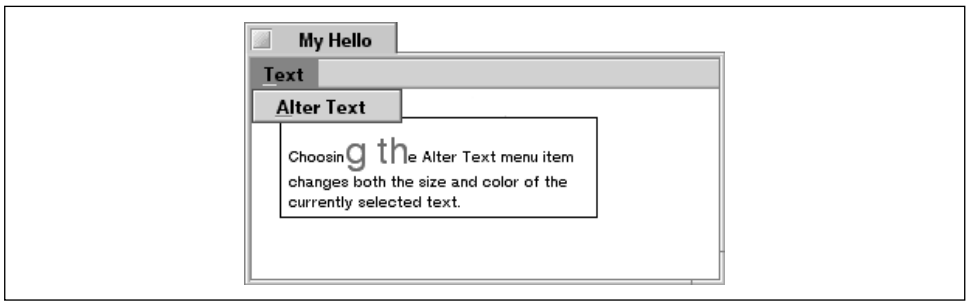

Figure 8-10. The TextViewFont example program's window

A menu item named Alter Text is fine for this simple example, but is a bit ambiguous for a real-world application. If your own program includes stylable text, then it should of course include separate menu items for each possible action. For instance, if your program allows the user to change the size of selected text, it should have Increase Size and Decrease Size menu items, or a menu item for each possible font point size. Similarly, if your program allows the color of text to be changed, it should have either a separate menu item for each possible color, or a menu item that brings up a color control (see Chapter 5 for a discussion of the `BColorControl` class).

Earlier in this chapter, you saw that a standard editing message such as `B_CUT` is assigned to the focus view by using menu-creation code like that shown here:

```
menu->AddItem(menuItem = new BMenuItem("Cut", new BMessage(B_CUT), 'X'));
menuItem->SetTarget(NULL, this);
```

You can set application-defined messages to go directly to the focus view in a similar manner. Here, `FONT_TEST_MSG` is an application-defined message type:

```
menu->AddItem(menuItem = new BMenuItem("Alter Text",
                                       new BMessage(FONT_TEST_MSG)));
menuItem->SetTarget(NULL, this);
```

For a standard editing message such as `B_CUT`, the work in handling the message is done; the system knows what to do with a Be-defined standard message. For my own application-defined message, however, a bit more code is needed because `BTextView` won't know how to handle an application-defined `FONT_TEST_MSG`. To create a `BTextView` object that can respond to an application-defined message, I defined a `MyTextView` class derived from `BTextView`. From *MyTextView.h*, here's the new class and the definition of the new message type:

```
#define   FONT_TEST_MSG     'fntt'

class MyTextView : public BTextView {

   public:
      MyTextView(BRect viewFrame, char *name, BRect textBounds);
      virtual void   MessageReceived(BMessage* message);
};
```

The MyTextView class consists of just two functions: a constructor to create a new object and a version of `MessageReceived()` that will be capable of handling the application-defined message. The implementation of these new functions can be found in *MyTextView.cp*. From that file, here's the `MyTextView` constructor:

```
MyTextView::MyTextView(BRect viewFrame, char *name, BRect textBounds)
   : BTextView(viewFrame, name, textBounds, B_FOLLOW_NONE, B_WILL_DRAW)
{
}
```

The MyTextView constructor is empty—it does nothing more than invoke the BTextView constructor. For this example, I'm satisfied with how a BTextView object looks and works, so I've implemented the MyTextView constructor such that it simply passes the arguments it receives on to the BTextView constructor. The real purpose of creating the BTextView-derived class is to create a class that overrides the BTextView version of MessageReceived(). The new version of this routine handles messages of the application-defined type FONT_TEST_MSG:

```
void MyTextView::MessageReceived(BMessage *message)
{
   switch (message->what) {

      case FONT_TEST_MSG:
         rgb_color  redColor = {255, 0, 0, 255};
         BFont      font;
         uint32     sameProperties;

         GetFontAndColor(&font, &sameProperties);
         font.SetSize(24.0);
         SetFontAndColor(&font, B_FONT_ALL, &redColor);
         break;

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

When an object of type MyTextView receives a FONT_TEST_MSG message, GetFontAndColor() is called to obtain a copy of the object's font. The size of the font is set to 24 points and SetFontAndColor() is called to pass the font back to the MyTextView object. When calling SetFontAndColor(), the rgb_color variable redColor is included in the parameter list in order to set the MyTextView object to use a shade of red in displaying text.

As is always the case, it's important that MessageReceived() include the default condition that invokes the inherited version of MessageReceived(). The MyTextView version of MessageReceived() handles only the application-defined FONT_TEST_MSG, yet a MyTextView object that is the focus view will also be receiving B_CUT, B_COPY, B_PASTE, and B_SELECT_ALL messages. When the MyTextView version of MessageReceived() encounters a message of one of these Be-defined types, it passes it on to the BTextView version of MessageReceived() for handling.

In this chapter's previous two examples, TextView and TextViewEdit, the MyDrawView class included a data member of type BTextView. Here I also

include a data member in the `MyDrawView` class, but I change it to be of type `MyTextView`:

```
class MyDrawView : public BView {

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

   private:
      MyTextView      *fTextView;
};
```

Like the previous example versions of the `MyDrawView` constructor, the TextView-Font project's version of the `MyDrawView` constructor creates a text view object. Here, however, the object is of the new `MyTextView` class type. Before exiting, the constructor calls the `BTextView` function `SetStylable()` to give the new `MyTextView` object support for multiple character formats. That ensures that changes made to the `MyTextView` object's text apply to only the current selection—not to all of the text in the object.

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

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new MyTextView(viewFrame, "TextView", textBounds);

   AddChild(fTextView);

   fTextView->SetStylable(true);
}
```

Scrolling

Text or graphics your application uses may not always fit within the confines of a view. Enter the scrollbar. The APIs for other operating systems that employ a graphical user interface include routines to work with scrollbars, but few implement this interface element as elegantly as the BeOS API does. The `BScrollBar` class makes it easy to add one or two scrollbars to any view. Better still, the `BScrollView` class creates a bordered object with one or two `BScrollBar` objects already properly sized to fit any view you specify. Best of all, scrollbar operation and updating is all automatic. Scrollbar objects are unique in that they

don't receive drawing or mouse down messages—the Application Server intercepts these messages and responds accordingly.

As part of the automatic handling of a scrollbar, the Application Server is responsible for changing the highlighting of a scrollbar. In Figure 8-11, you see the same scrollbar with two different looks. When the contents of the view a scrollbar is attached to exceed the size of the view, the scrollbar's knob appears and the scrollbar becomes enabled. As the content of the view increases, the scrollbar knob automatically decreases in size to reflect the lessening amount of the total text being displayed within the view.

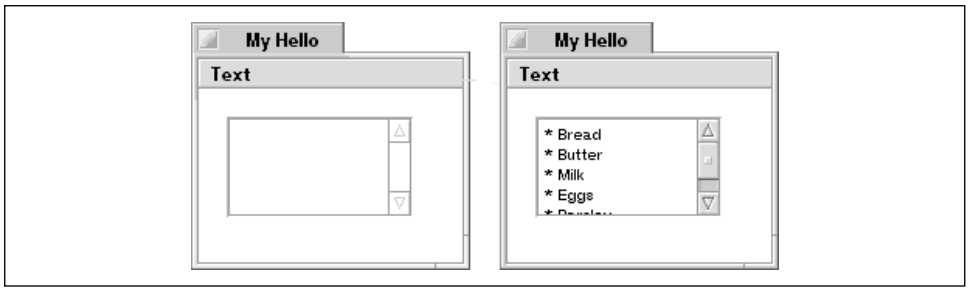

Figure 8-11. A scrollbar's look changes as the scrolled view's content increases

Keep in mind that from the ScrollBar preferences application the user can control the look and behavior of the scrollbars present in *all* applications that run on his or her machine. For instance, the ScrollBar preferences control whether programs display a scrollbar with a pair of scroll arrows on each end, or just one. With that in mind, you'll note that the look of the scrollbars in this chapter's figures differs (compare Figure 8-11 with Figure 8-13).

Scrollbars

To make use of a scrollbar, create a `BScrollBar` object, designate a different view to be the scrollbar's *target*—the thing to be scrolled—and add the `BScrollBar` object to the same parent view as the target view. Or, more likely, create a `BScrollView` object and let it do this work for you. Because the scrolling view is so handy, the "Scrolling" section's emphasis is on the `BScrollView` class. A `BScrollView` object includes one or two `BScrollBar` objects, though, so a study of the `BScrollBar` class won't prove to be time wasted.

The BScrollBar class

A scrollbar object is an instance of the `BScrollBar` class. Pass six arguments to its constructor, the prototype of which is shown here:

```
BScrollBar(BRect       frame,
           const char  *name,
           BView       *target,
           float        min,
           float        max,
           orientation  posture)
```

The first parameter, `frame`, is a rectangle that defines the boundaries of the scrollbar. The coordinates of the rectangle are relative to the scrollbar's parent view, not to the thing that is to be scrolled. User interface guidelines state that scrollbars should be of a uniform thickness. That is, all horizontal scrollbars should be of the same height, and all vertical scrollbars should be of the same width. Use the Be-defined constants `B_H_SCROLL_BAR_HEIGHT` and `B_V_SCROLL_BAR_WIDTH` to ensure that your program's scrollbars have the same look as those used in other Be applications. For instance, to set up the rectangle that defines a horizontal scrollbar to run along the bottom of a text object, you might use this code:

```
BRect  horizScrollFrame(20.0, 50.0, 220.0, 50.0 + B_H_SCROLL_BAR_HEIGHT);
```

The second parameter serves the same purpose as the name parameter in other `BView`-derived classes: it allows the view to be accessed by name using the `BView` function `FindView()`.

A scrollbar acts on a *target*—a view associated with the scrollbar. The target is what gets scrolled, and is typically an object that holds text or graphics. To bind a scrollbar to the object to be scrolled, follow these steps:

1. Create the target object.
2. Add the target object to a view (its parent view).
3. Create a scrollbar object, using the target object as the `target` parameter in the `BScrollBar` constructor.
4. Add the scrollbar object to the view that serves as the target object's parent.

For a target that displays graphics, the `min` and `max` parameters determine how much of the target view is displayable. The values you use for these two parameters will be dependent on the total size of the target. If the target view is a `BTextView` object, the values assigned to `min` and `max` are inconsequential—the scrollbar is always aware of how much text is currently in the target view and adjusts itself accordingly. The exception is if both `min` and `max` are set to 0. In such a case, the scrollbar is disabled and no knob is drawn, regardless of the target view's contents. Refer to the "Scrollbar range" section just ahead for more

details on choosing values for these two parameters when the target consists of graphics.

The last parameter, `posture`, designates the orientation of the scrollbar. To create a horizontal scrollbar, pass the Be-defined constant `B_HORIZONTAL`. For a vertical scrollbar, pass `B_VERTICAL`.

The following snippet provides an example of a call to the `BScrollBar` constructor. Here a horizontal scrollbar is being created for use with a previously created `BTextView` object named `theTextView`. Because the target is a text object, the values of `min` and `max` are arbitrarily selected:

```
BScrollBar  *horizScrollBar;
BRect       horizScrollFrame(20.0, 50.0, 220.0,
                             50.0 + B_H_SCROLL_BAR_HEIGHT);
float       min = 1.0;
float       max = 1.0;

horizScrollBar = new BScrollBar(scrollFrame, "ScrollBar", theTextView,
                                min, max, B_HORIZONTAL);
```

Scrollbar example project

This chapter's TextView example project demonstrated how a window can support editable text through the use of a `BTextView` object. Because the topic of scrollbars hadn't been presented when TextView was developed, the program's text view didn't include them. Now you're ready to add a vertical scrollbar, a horizontal scrollbar, or both to a text view. As shown in Figure 8-12, for the TextViewScrollBar project I've elected to add just a vertical scrollbar to the window's `BTextView` object.

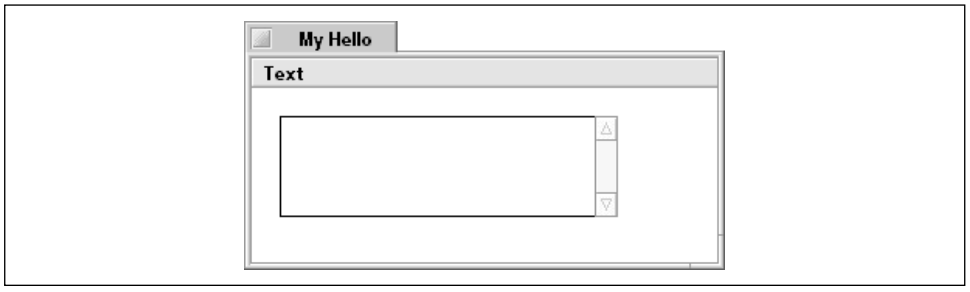

Figure 8-12. The TextViewScrollBar example program's window

The TextView example project shows how to set up `viewFrame` and `textBounds`, the rectangles that define the boundary of the window's `BTextView` object and the area that displays text in that object. Here I'll discuss only the code that's been added to the TextView project to turn it into the TextViewScrollBar project. All

those additions appear in the `MyDrawView` constructor. The new code involves the creation of a vertical scrollbar, and appears following the call to `InsetBy()`:

```
BRect  viewFrame(20.0, 20.0, 220.0, 80.0);


MyDrawView::MyDrawView(BRect rect, char *name)
   : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  textBounds;

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new BTextView(viewFrame, "MyTextView", textBounds,
                             B_FOLLOW_NONE, B_WILL_DRAW);
   AddChild(fTextView);

   viewFrame.InsetBy(-2.0, -2.0);

   BScrollBar  *verticalBar;
   BRect       scrollFrame;
   float       min = 1.0;
   float       max = 1.0;

   scrollFrame = viewFrame;
   scrollFrame.left = scrollFrame.right;
   scrollFrame.right = scrollFrame.left + B_V_SCROLL_BAR_WIDTH;

   verticalBar = new BScrollBar(scrollFrame, "VerticalBar", fTextView,
                                min, max, B_VERTICAL);
   AddChild(verticalBar);
   verticalBar->SetResizingMode(B_FOLLOW_NONE);
}
```

The scrollbar will be added to the target's parent view (`MyDrawView`), so the coordinates of the scrollbar's rectangle are relative to the parent view, not to the target view. The `scrollFrame` rectangle coordinates are first set to those of the target view. That gives the scrollbar the same top and bottom coordinates as the target, as desired. Then the scrollbar's left and right side coordinates are adjusted so that the scrollbar lies flush on the right side of the target view. The `scrollFrame` rectangle is then used as the first parameter to the `BScrollBar` constructor. The previously created `BTextView` object `fTextView` is specified as the scrollbar's target. The scrollbar is then added to `MyDrawView`, just as the target was previously added.

Graphical user interface conventions dictate that a vertical scrollbar be located flush against the right side of the area that is scrollable, and that a horizontal scrollbar be located flush against the bottom of the area that's scrollable. But nothing in the `BScrollBar` class *forces* you to follow that convention. A scrollbar is associated with a view to scroll by naming the view as the target in the `BScrollBar` constructor. Depending on the coordinates you choose for the `frame` rectangle parameter of the `BScrollBar` constructor, the scrollbar can be placed anywhere in the parent view and it will still scroll the contents of the target view.

Notice that in this example the resizing mode of the scrollbar is adjusted. The `BScrollBar` constructor doesn't include a sizing mode parameter. Instead, the specification of the scrolling bar's orientation (the last parameter to the `BScrollBar` constructor) defines a default behavior for the scrollbar. A vertically oriented scrollbar (like the one created here) resizes itself vertically. As the parent view changes in size vertically, the vertical scrollbar will grow or shrink vertically. As the parent view changes in size horizontally, the vertical scrollbar will follow the parent view.

In many cases these default characteristics are appropriate. In this project, they aren't. I've given the `BTextView` object a resizing mode of `B_FOLLOW_NONE`, indicating that the text view object will remain a fixed size as the parent view changes size. In such a case, I want the target view's scrollbar also to remain fixed in size and location. A call to the `BView` function `SetResizingMode()` takes care of that task.

Scrollbar range

If a scrollbar's target is a view that holds a graphical entity, such as a `BView` object that includes a `BPicture` object, the `BScrollBar` constructor `min` and `max` parameters take on significance. Together, `min` and `max` define a range that determines how much of the target view is displayable. Consider a view that is 250 pixels in width and 200 pixels in height, and is to be displayed in a view that is 100 pixels by 100 pixels in size. If this 100-by-100 pixel view has two scrollbars, and it's desired that the user be able to scroll the entire view, the range of the horizontal scrollbar should be 150 and the range of the vertical scrollbar should be 100. Figure 8-13 illustrates this.

In Figure 8-13, 100 pixels of the 250-pixel width of the view will always be displayed, so the horizontal range needs to be only 150 in order to allow the horizontal scrollbar to bring the remaining horizontal portions of the view through the display area. Similarly, 100 of the 200 vertical pixels will always be displayed in

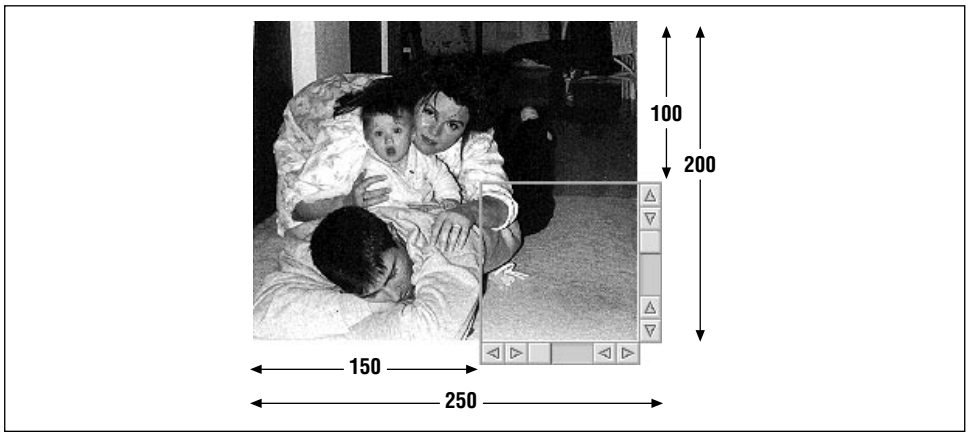

Figure 8-13. An example of determining the range of a pair of scrollbars

the view, so the vertical range needs to be 100 in order to pass the remaining 100 pixels through the display area. Assuming a graphics-holding view named `theGraphicView`, derived from a `BView` object, exists (as shown in Figure 8-13), the two scrollbars could be set up as shown in the following snippet. To see the relationship of the coordinates of the scrollbars to the view, refer to Figure 8-14.

```
BScrollBar  *horizScrollBar;
BRect       horizScrollFrame(170.0, 300.0, 270.0,
                             300.0 + B_H_SCROLL_BAR_HEIGHT);
float       horizMin =   0.0;
float       horizMax = 150.0;
BScrollBar  *vertScrollBar;
BRect       vertScrollFrame(270.0, 200.0, 270.0 + B_V_SCROLL_BAR_WIDTH,
                            300.0);
float       vertMin  =   0.0;
float       vertMax  = 100.0;

horizScrollBar = new BScrollBar(horizScrollFrame, "HScrollBar",
                                theGraphicView,
                                horizMin, horizMax, B_HORIZONTAL);
vertScrollBar  = new BScrollBar(vertScrollFrame, "VScrollBar",
                                theGraphicView,
                                vertMin, vertMax, B_VERTICAL);
```

If the target view changes size during program execution (for instance, your program may allow the user to replace the currently displayed contents of the view with different graphic), the range of any associated scrollbar should change, too. The `BScrollBar` function `SetRange()` exists for this purpose:

```
void SetRange(float  min,
              float  max)
```

The example just discussed has a horizontal scrollbar with a range of 0 to 150 pixels. If for some reason I wanted to allow the user to be able to scroll beyond the

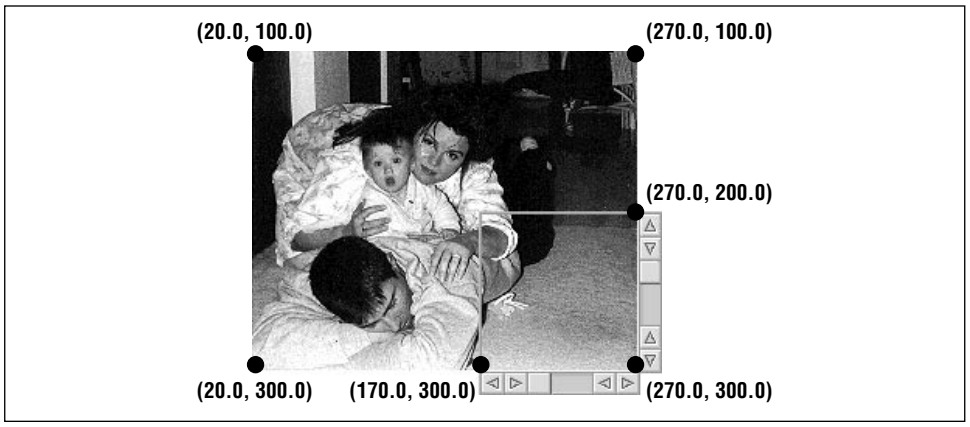

Figure 8-14. The coordinates of a pair of scrollbars and the picture to be scrolled

right edge of the view, I could increase the scrollbar's maximum value. Here I change the scrollbar's range to allow the user to view 50 pixels of white space past the view's right edge:

```
horizScrollBar->SetRange(0.0, 200.0);
```

The companion function to `SetRange()` is `GetRange()`. As expected, this function returns the current minimum and maximum scrolling values of a scrollbar:

```
void GetRange(float  *min,
              float  *max)
```

If this next snippet is executed after the preceding call to `SetRange()`, `min` should have a value of 0.0 and `max` should have a value of 200.0:

```
float  min;
float  max;

horizScrollBar->GetRange(&min, &max);
```

The ScrollViewPicture example near the end of this chapter provides an example of setting the scrollbar range for a `BView` object that's used as the target for a `BScrollView` object—a view that has built-in scrollbars.

Scrolling View

It's a relatively easy assignment to add scrollbars to a view, as just demonstrated in the TextViewScrollBar project. However, the BeOS API makes it easier still. The `BScrollView` class creates a scroll view object that serves as a container for another view. This contained view can hold either text (as a `BTextView` does) or graphics (as a `BPicture` does). Regardless of the content of its contained view,

the `BScrollView` object is responsible for adding scrollbars that allow for scrolling through the entire content and for making itself the parent of the contained view.

The BScrollView class

The seven parameters of the `BScrollView` constructor make it possible to create a scrolling view object that has one, two, or even no scrollbars:

```
BScrollView(const char    *name,
            BView         *target,
            uint32        resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
            uint32        flags = 0,
            bool          horizontal = false,
            bool          vertical = false,
            border_style  border = B_FANCY_BORDER)
```

The `name`, `resizingMode`, and `flags` parameters serve the purposes expected of a `BView`-derived class. The `name` parameter allows the scroll view object to be accessed by its name. The `resizingMode` specifies how the object is to be resized as the parent view changes size: the default value of `B_FOLLOW_LEFT | B_FOLLOW_TOP` indicates that the distance from the scroll view's left side and its parent's left side will be fixed, as will the distance from the scroll view's top and its parent's top. The `flags` parameter specifies the notification the scroll view object is to receive. The default value of 0 means that the object isn't to receive any notification.

The `target` parameter specifies the previously created view object to be surrounded by the scroll view object. The contents of the target view are what is to be scrolled. There's no need to specify any size for the new scroll view object. It will automatically be given a framing rectangle that accommodates the target view, any scrollbars that may be a part of the scroll view object, and a border (if present).

A scroll view object can have a horizontal scrollbar, a vertical scrollbar, both, or neither. The `horizontal` and `vertical` parameters specify which scrollbar or bars should be a part of the scroll view object. By default, the object includes no scrollbars (meaning the object serves as nothing more than a way to draw a border around a different view, as discussed in the `border` parameter description next). To include a scrollbar, simply set the appropriate `horizontal` or `vertical` parameter to `true`.

The `border` parameter specifies the type of border to surround the scroll view object. By default, a scroll view object has a fancy border; the appearance of a groove surrounds the object. To specify a plain line border, pass the Be-defined

constant B_PLAIN_BORDER as the last argument to the BScrollView constructor. To omit the border completely, pass B_NO_BORDER instead.

In this next snippet, a scroll view object is created with a plain border and a vertical scrollbar. Here it's assumed that theTextView is a BTextView object that isn't resizable. Because the target is fixed in the window in which it resides, the scroll view too can be fixed. As evidenced by the value of the resizingMode parameter (B_FOLLOW_NONE), theScrollView won't be resizeable:

```
BScrollView  *theScrollView;

theScrollView = new BScrollView("MyScrollView", theTextView, B_FOLLOW_NONE,
                                0, false, true, B_PLAIN_BORDER);
```

Scroll view example project

This chapter's TextViewScrollBar project modified the TextView example project to demonstrate how a BScrollBar object can be used to scroll the contents of a BTextView object. This latest project, ScrollViewText, achieves the same effect. Here, however, a BScrollView object is used to create the BTextView object's scrollbar. The resulting window looks similar to the one generated by the TextViewScrollBar project (refer back to Figure 8-12). Thanks to the BScrollView, however, here the border around the text view object has shading, as shown in Figure 8-15.

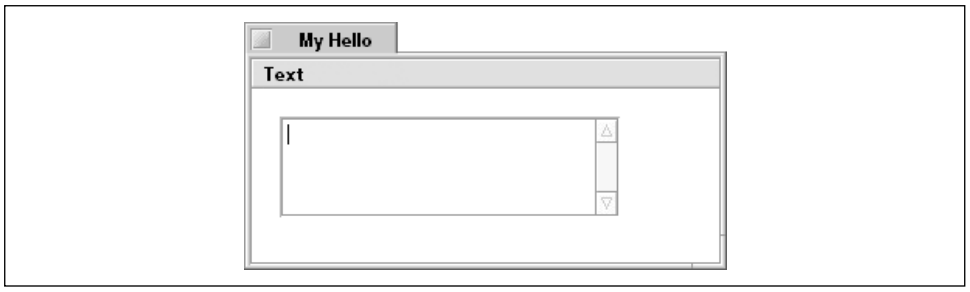

Figure 8-15. The ScrollViewText example program's window

While the results of the TextViewScrollBar and ScrollView Text projects are similar, the effort expended to obtain the results differs. Using a BScrollView object to supply the scrollbar (as done here) rather than using a BScrollBar object means there's no need to supply the scrollbar's coordinates. The BScrollView object takes care of the scrollbar's placement based on the location of the designated target. Additionally, there's no need to draw a border around the text view; the BScrollBar object takes care of that task too.

The MyDrawView class declaration is the same as it was for the original TextView project, with the addition of a BScrollView object:

```
class MyDrawView : public BView {

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

   private:
      BTextView         *fTextView;
      BScrollView       *fScrollView;
};
```

The `MyDrawView` constructor holds the scroll view code:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  viewFrame(20.0, 20.0, 220.0, 80.0);
   BRect  textBounds;

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new BTextView(viewFrame, "MyTextView", textBounds,
                             B_FOLLOW_NONE, B_WILL_DRAW);
   fScrollView = new BScrollView("MyScrollView", fTextView, B_FOLLOW_NONE,
                                 0, false, true);
   AddChild(fScrollView);
}
```

The `MyDrawView` constructor begins by setting up and creating a `BTextView` object. This code is the same as the code that appears in the TextView version of the `MyDrawView` constructor, with one exception. Here, a call to `AddChild()` doesn't immediately follow the creation of the text view object. The newly created `BTextView` object isn't added to the drawing view because it is instead added to the `BScrollView` object when it is passed to the scroll view's constructor. The `BScrollView` object is then added to the drawing view. When the program creates a window, that window's view hierarchy will look like the one pictured in Figure 8-16. In this figure, you see that the `BScrollView` object `fScrollView` is the parent to two views: the target view `fMyTextView` and a `BScrollBar` object created by the `BScrollView` constructor. Later, in the ScrollViewPicture project, you'll see how your code can easily access this implicitly created scrollbar.

The implementation of this project's `AttachedToWindow()` function is identical to the TextView projects: call `SetFont()` and `SetFontSize()` to specify font information for the drawing view, then call `fTextView->MakeFocus()` to start the cursor blinking in the `BTextView` object. The implementation of `Draw()` is simple—here it's an empty function. In the TextView project, `StrokeRect()` was invoked

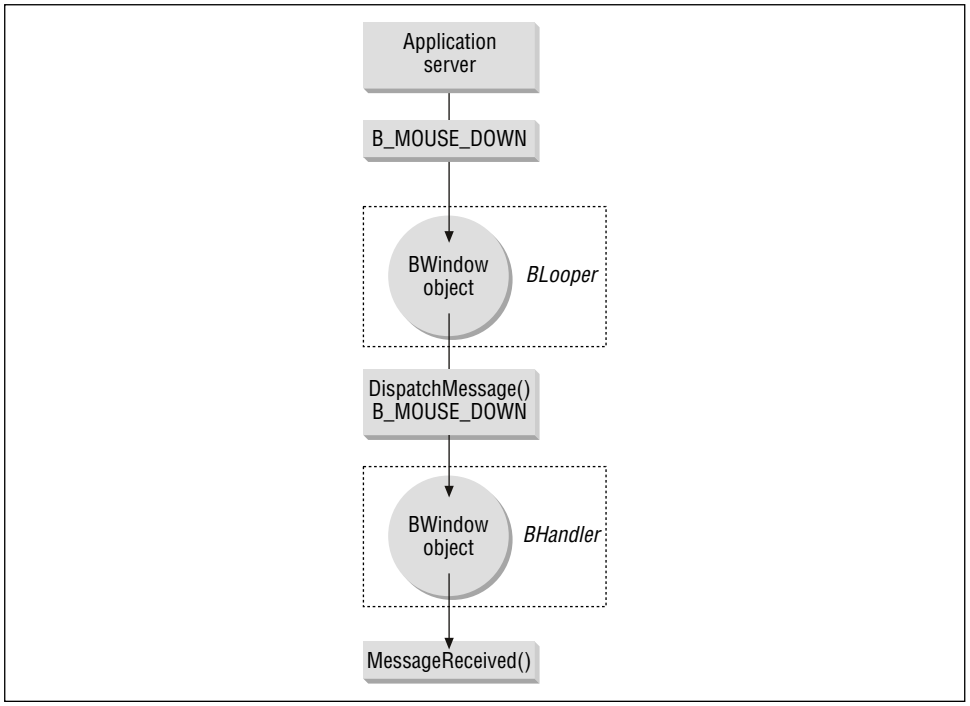

Figure 8-16. View hierarchy of the window of the ScrollViewText program

to draw a border around the `BTextView` object. Here, I rely on the `BScrollView` object's ability to automatically draw its own border.

```
void MyDrawView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(12);

   fTextView->MakeFocus();
}


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

Scrolling window example project

If your program offers the user text editing capabilities, it may make sense to provide a window that exists for just that purpose. Typically, a simple text editor displays a resizable window bordered by a vertical scrollbar and possibly a horizontal scrollbar. Figure 8-17 shows such a window—the window displayed by the ScrollViewWindow project.

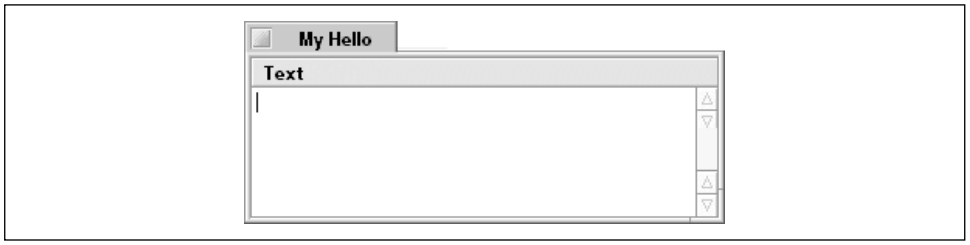

Figure 8-17. The ScrollViewWindow example program's window

The ScrollViewWindow project includes only slight modifications to the TextViewScrollBar project. All the changes are found in the `MyDrawView` constructor:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  viewFrame;
   BRect  textBounds;

   viewFrame = Bounds();
   viewFrame.right -= B_V_SCROLL_BAR_WIDTH;

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new BTextView(viewFrame, "MyTextView", textBounds,
                             B_FOLLOW_ALL, B_WILL_DRAW);

   fScrollView = new BScrollView("MyScrollView", fTextView,
                                 B_FOLLOW_ALL, 0, false, true);
   AddChild(fScrollView);
}
```

A call to the `BView` function `Bounds()` sets the coordinates of what is to be the `BScrollView` target, the `BTextView` view object, to the coordinates of the drawing view. The drawing view itself is the same size as the content area of the window it resides in, so this brings me close to my goal of making the entire content area of the window capable of holding user-entered text. The exception is that room needs to be allowed for the vertical scrollbar. Subtracting the width of this scrollbar results in a `viewFrame` rectangle with the desired size.

The other changes to the `MyDrawView` constructor involve the `resizingMode` parameter to both the `BTextView` constructor and the `BScrollView` constructor. In the previous example, the scroll view was fixed in size, so neither the scroll view nor its target needed to be concerned with resizing. Here I want the text area of the window to always occupy the entire content area of the window, less the window's scrollbar area. A `resizingMode` of `B_FOLLOW_ALL` (rather than

B_FOLLOW_NONE) tells both the BScrollView object and its BTextView target that they should automatically resize themselves as the user changes the parent window's size.

Accessing a BScrollView scrollbar

The BScrollView constructor lets you specify whether a BScrollView object should include a horizontal scrollbar, a vertical scrollbar, or both. The constructor is responsible for creating the appropriate number of BScrollBar objects and placing them within the BScrollView object. The BScrollView constructor gives each of its BScrollBar objects a default range of 0.0 to 1000.0. If the BScrollView object's target is a BTextView, these default minimum and maximum values always suffice—the BTextView object makes sure that the ranges of the scrollbars that target it are adjusted accordingly. If the BScrollView object's target is instead a view that holds graphics, you'll need to adjust the range of each scrollbar so that the user is guaranteed visual access to the entire target view.

This chapter's "Scrollbar range" section describes how to determine the range for a scrollbar to scroll graphics, as well as how to use the BScrollBar function SetRange() to set a scrollbar's minimum and maximum values. After determining the range a scroll view object's scrollbar should have, invoke the BScrollView function ScrollBar() to gain access to the scrollbar in question. Pass ScrollBar() the type of scrollbar (B_HORIZONTAL or B_VERTICAL) to access, and the routine returns the BScrollBar object. Then invoke that object's SetRange() function to reset its range. Here, access to the vertical scrollbar of a BScrollView object named theScrollView is gained, and the scrollbar's range is then set to a minimum of 0.0 and a maximum of 340.0:

```
BScrollBar  *scrollBar;

scrollBar = theScrollView->ScrollBar(B_VERTICAL);
scrollBar->SetRange(0.0, 340.0);
```

Scrollbar access example project

While topics such as the BStringView class and the BTextView class have made the display of text the focus of this chapter, I'll close with an example that demonstrates how to scroll a picture. The BScrollView class (covered later in this section) places no restrictions on what type of view is to be the designated target, so the differences between scrolling text and scrolling graphics are minimal. As shown in Figure 8-18, the ScrollViewPicture project demonstrates how to use a BPicture object as the target of a BScrollView object.

This example would work just fine with a very simple picture, such as one created by drawing a single large rectangle. I've opted to instead create a slightly more complex picture, and at the same time set up the project such that it's an

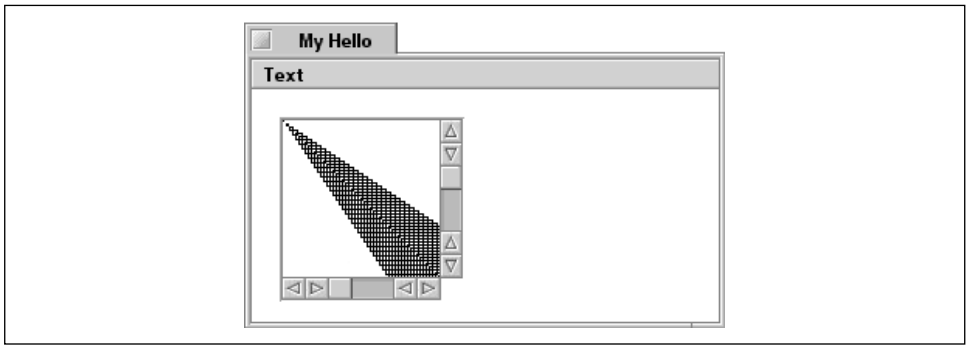

Figure 8-18. The ScrollViewPicture example program's window

easy task to make the picture far more complex. I've done that by creating a `BView`-derived class named `MyPictureView`. The declaration of this new class appears in the *MyPictureView.h* header file, while the implementations of the class member functions can be found in *MyPictureView.cpp*. Here's the `MyPictureView` declaration:

```
class MyPictureView : public BView {

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

   private:
      BPicture       *fPicture;
};
```

As you'll see in this example, a `MyPictureView` object will become the target of a `BScrollView` object. That means the `MyPictureView` class could include any number of graphics, and the scroll view object will view them all as a single entity. So while I've included a single `BPicture` data member in the `MyPictureView`, your `BView`-derived class could hold two, three, or three hundred pictures, along with other graphic data members such as polygons or bitmaps (see Chapter 5 for shape-drawing information and Chapter 10, *Files*, for bitmap details).

The `AttachedToWindow()` routine sets up the picture. Recall that such code can't appear in the constructor of the picture's parent view. That's because the `BPicture` definition relies on the current state of the picture's parent view, and the parent view's state isn't completely set up until the execution of `AttachedToWindow()`.

```
void MyPictureView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(12);
```

```
   BeginPicture(new BPicture);
      BRect  aRect;
      int32  i;

      for (i=0; i<100; i++) {
         aRect.Set(i*2, i*2, i*3, i*3);
         StrokeRect(aRect);
      }
   fPicture = EndPicture();
}
```

I mentioned earlier that your own `BView` class could contain more than one `BPicture` data member. Looking at the above version of `AttachedToWindow()`, you might think that's unnecessary because you can define a single `BPicture` object to include all the graphics a view needs. That may be the case—or it may not be. Your program might construct a `BView`-derived class by piecing together multiple `BPicture` objects. For instance, your application may give the user the opportunity to define a single large graphic by selecting numerous small pictures.

The `BScrollView` object will exist in a `MyDrawView` object, and will have a `BScrollView` object as its target. I've included `MyPictureView` and `BScrollView` data members in the `MyDrawView` class to keep track of these objects.

```
class MyDrawView : public BView {

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

   private:
      MyPictureView  *fPictureView;
      BScrollView    *fScrollView;
};
```

The `MyDrawView` constructor begins by creating what will be the target view: `fPictureView`. Next, it creates a `BScrollView` object with `fPictureView` as the target. After adding the scroll view to the drawing view, the default ranges of the scroll view's two scrollbars are altered. Looking back at the `MyPictureView` version of `AttachedToWindow()`, you'll note that the `BPicture` object will be 300 pixels wide and 300 pixels high. Yet the width and height of the `viewFrame` rectangle that is to display this picture are each only 100 pixels. The difference in the actual size of the picture and the size of the rectangle the picture's displayed in is used in resetting the scrollbar ranges:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  viewFrame(20.0, 20.0, 120.0, 120.0);

   fPictureView = new MyPictureView(viewFrame, "MyPictureView");

   fScrollView = new BScrollView("MyScrollView", fPictureView,
                                 B_FOLLOW_NONE, 0, true, true);
   AddChild(fScrollView);

   BScrollBar  *scrollBar;

   scrollBar = fScrollView->ScrollBar(B_VERTICAL);
   scrollBar->SetRange(0.0, 200.0);
   scrollBar = fScrollView->ScrollBar(B_HORIZONTAL);
   scrollBar->SetRange(0.0, 200.0);
}
```