```
    public static void SetSize(Window win)
    {
        RegistryKey key;
        key = Registry.CurrentUser.OpenSubKey(RegPath + win.Name);

        if (key != null)
        {
            Rect bounds = Rect.Parse(key.GetValue("Bounds").ToString());
            win.Top = bounds.Top;
            win.Left = bounds.Left;

            // Restore the size only for a manually sized
            // window.
            if (win.SizeToContent == SizeToContent.Manual)
            {
                win.Width = bounds.Width;
                win.Height = bounds.Height;
            }
        }
    }
}
```

To use this class in a window, you call the SaveSize() method when the window is closing and call the SetSize() method when the window is first opened. In each case, you pass a reference to the window you want the helper class to inspect. Note that in this example, each window must have a different value for its Name property.

Window Interaction

In Chapter 7, you considered the WPF application model, and you took your first look at how windows interact. As you saw there, the Application class provides you with two tools for getting access to other windows: the MainWindow and Windows properties. If you want to track windows in a more customized way—for example, by keeping track of instances of a certain window class, which might represent documents—you can add your own static properties to the Application class.

Of course, getting a reference to another window is only half the battle. You also need to decide how to communicate. As a general rule, you should minimize the need for window interactions, because they complicate code unnecessarily. If you do need to modify a control in one window based on an action in another window, create a dedicated method in the target window. That makes sure the dependency is well identified, and it adds another layer of indirection, making it easier to accommodate changes to the window's interface.

■ **Tip** If the two windows have a complex interaction, are developed or deployed separately, or are likely to change, you can consider going one step further and formalize their interaction by creating an interface with the public methods and implementing that interface in your window class.

Figures 23-3 and 23-4 show two examples for implementing this pattern. Figure 23-3 shows a window that triggers a second window to refresh its data in response to a button click. This window does not directly attempt to modify the second window's user interface; instead, it relies on a custom intermediate method called DoUpdate().

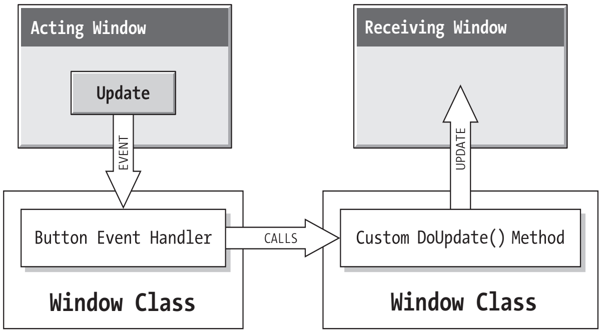


Figure 23-3. *A single window interaction*

The second example, Figure 23-4, shows a case where more than one window needs to be updated. In this case, the acting window relies on a higher-level application method, which calls the required window update methods (perhaps by iterating through a collection of windows). This approach is better because it works at a higher level. In the approach shown in Figure 23-3, the acting window doesn't need to know anything specific about the controls in the receiving window. The approach in Figure 23-4 goes one step further—the acting window doesn't need to know anything at all about the receiving window class.

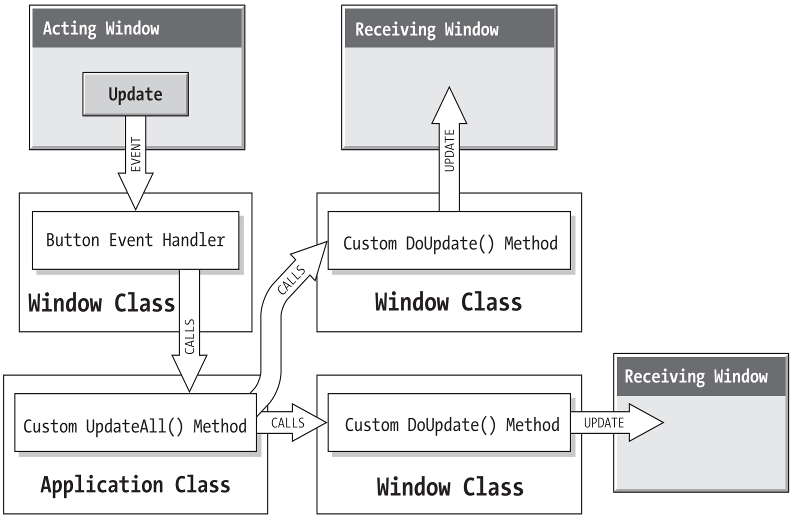


Figure 23-4. *A one-to-many window interaction*

■ **Tip** When interacting between windows, the Window.Activate() method often comes in handy. It transfers the activation to the window you want. You can also use the Window.IsActive property to test whether a window is currently the one and only active window.

You can go one step further in decoupling this example. Rather than having the Application class trigger a method in the various windows, it could simply fire an event and allow the windows to choose how to respond to that event.

■ **Note** WPF can help you abstract your application logic through its support for commands, which are application-specific tasks that can be triggered any way you like. Chapter 9 has the full story.

The examples in Figure 23-3 and Figure 23-4 show how separate windows (usually modeless) can trigger actions in one another. But certain other patterns for window interaction are simpler (such as the dialog model) and supplement this model (such as window ownership). You'll consider these features in the following sections.

Window Ownership

.NET allows a window to "own" other windows. Owned windows are useful for floating toolbox and command windows. One example of an owned window is the Find and Replace window in Microsoft Word. When an owner window is minimized, the owned windows are also minimized automatically. When an owned window overlaps its owner, it is always displayed on top.

To support window ownership, the Window class adds two properties. Owner is a reference that points to the window that owns the current window (if there is one). OwnedWindows is a collection of all the windows that the current window owns (if any).

Setting up ownership is simply a matter of setting the Owner property, as shown here:

```
// Create a new window.
ToolWindow winTool = new ToolWindow();

// Designate the current window as the owner.
winTool.Owner = this;

// Show the owned window.
winTool.Show();
```

Owned windows are always shown modelessly. To remove an owned window, set the Owner property to null.

■ **Note** WPF does not include a system for building multiple document interface (MDI) applications. If you want more sophisticated window management, it's up to you to build it (or buy a third-party component).

An owned window can own another window, which can own another window, and so forth (although it's questionable whether this design has any practical use). The only limitations are that a window cannot own itself and two windows cannot own each other.

The Dialog Model

Often, when you show a window modally, you are offering the user some sort of choice. The code that displays the window waits for the result of that choice and then acts on it. This design is known as the *dialog model*. The window you show modally is the dialog box.

You can easily accommodate this design pattern by creating some sort of public property in your dialog window. When the user makes a selection in the dialog window, you would set this property and then close the window. The code that shows the dialog box can then check for this property and determine what to do next based on its value. (Remember that even when a window is closed, the window object, and all its control information, still exists until the variable referencing it goes out of scope.)

Fortunately, some of this infrastructure is already hardwired into the Window class. Every window includes a ready-made DialogResult property, which can take a true, false, or null value. Usually, true indicates the user chose to go forward (for example, clicked OK), while false indicates that the user canceled the operation.

Best of all, once you set the dialog result, it's returned to the calling code as the return value of the ShowDialog() method. That means you can create, show, and consider the result of a dialog box window with this lean code:

```
DialogWindow dialog = new DialogWindow();
if (dialog.ShowDialog() == true)
{
    // The user accepted the action. Full speed ahead.
}
else
{
    // The user canceled the action.
}
```

■ **Note** Using the DialogResult property doesn't prevent you from adding custom properties to your window. For example, it's perfectly reasonable to use the DialogResult property to inform the calling code whether an action was accepted or canceled and to provide other important details through custom properties. If the calling code finds a DialogResult of true, it can then check these other properties to get the information it needs.

You can take advantage of another shortcut. Rather than setting the DialogResult by hand after the user clicks a button, you can designate a button as the accept button (by setting IsDefault to true). Clicking that button automatically sets the DialogResult of the window to true. Similarly, you can designate a button as the cancel button (by setting IsCancel to true), in which case clicking it will set the DialogResult to Cancel. (You learned about IsDefault and IsCancel when you considered buttons in Chapter 6.)

■ **Note** The dialog model in WPF is different from that of Windows Forms. Buttons do not provide a DialogResult property, so you are limited to creating default and cancel buttons. The DialogResult is more limited—it can be only true, false, or null (which it is initially). Also, clicking a default or cancel button does not automatically close the window; you need to write the code to accomplish that.

Common Dialog Boxes

The Windows operating system includes many built-in dialog boxes that you can access through the Windows API. WPF provides wrappers for just a few of these.

■ **Note** There are good reasons that WPF doesn't include wrappers for all the Windows APIs. One of the goals of WPF is to decouple it from the Windows API so it's usable in other environments (like a browser) or portable to other platforms. Also, many of the built-in dialog boxes are showing their age and shouldn't be the first choice for modern applications. Windows 7 also discourages dialog boxes in favor of task-based panes and navigation.

The most obvious of these is the System.Windows.MessageBox class, which exposes a static Show() method. You can use this code to display a standard Windows message box. Here's the most common overload:

```
MessageBox.Show("You must enter a name.", "Name Entry Error",
  MessageBoxButton.OK, MessageBoxImage.Exclamation) ;
```

The MessageBoxButton enumeration allows you to choose the buttons that are shown in the message box. Your options include OK, OKCancel, YesNo, and YesNoCancel. (The less user-friendly AbortRetryIgnore isn't supported.) The MessageBoxImage enumeration allows you to choose the message box icon (Information, Exclamation, Error, Hand, Question, Stop, and so on).

Along with the MessageBox class, WPF includes specialized printing support that uses the PrintDialog (which is described in Chapter 29) and, in the Microsoft.Win32 namespace, the OpenFileDialog and SaveFileDialog classes.

The OpenFileDialog and SaveFileDialog classes acquire some additional features (some which are inherited from the FileDialog class). Both support a filter string, which sets the allowed file extensions. The OpenFileDialog class also provides properties that let you validate the user's selection (CheckFileExists) and allow multiple files to be selected (Multiselect). Here's an example that shows an OpenFileDialog and displays the selected files in a list box after the dialog box is closed:

```
OpenFileDialog myDialog = new OpenFileDialog();

myDialog.Filter = "Image Files(*.BMP;*.JPG;*.GIF)|*.BMP;*.JPG;*.GIF" +
  "|All files (*.*)|*.*";
myDialog.CheckFileExists = true;
myDialog.Multiselect = true;

if (myDialog.ShowDialog() == true)
{
    lstFiles.Items.Clear();
    foreach (string file in myDialog.FileNames)
    {
        lstFiles.Items.Add(file);
    }
}
```

You won't find any color pickers, font pickers, or folder browsers (although you can get these ingredients using the System.Windows.Forms classes from .NET 2.0).

■ **Note** In previous versions of WPF, the dialog box classes always displayed the old-style Windows XP dialog boxes. In WPF 4, they've been updated with Windows Vista and Windows 7 support, which means that when you run your application on a newer version of Windows, you'll automatically get the modern dialog box style.

Nonrectangular Windows

Irregularly shaped windows are often the trademark of cutting-edge consumer applications such as photo editors, movie makers, and MP3 players, and they're even more common with WPF applications.

Creating a basic shaped window in WPF is easy. However, creating a slick, professional-looking shaped window takes more work—and, most likely, a talented graphic designer to create the outlines and design the background art.

A Simple Shaped Window

The basic technique for creating a shaped window is to follow these steps:

1. Set the Window.AllowsTransparency property to true.
2. Set the Window.WindowStyle property to None to hide the nonclient region of the window (the blue border). If you don't, you'll get an InvalidOperationException when you attempt to show the window.
3. Set the Background to be transparent (using the color Transparent, which has an alpha value of 0). Or set the Background to use an image that has transparent areas (regions that are painted with an alpha value of 0).

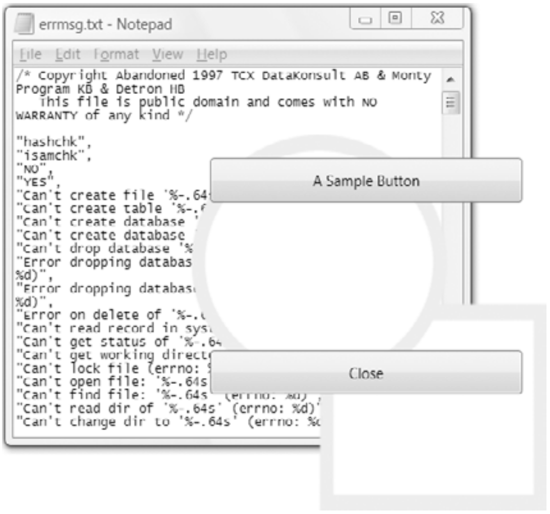
These three steps effectively remove the standard window appearance (known to WPF experts as the window *chrome*). To get the shaped window effect, you now need to supply some nontransparent content that has the shape you want. You have a number of options:

- Supply background art, using a file format that supports transparency. For example, you can use a PNG file to supply the background of a window. This is a simple, straightforward approach, and it's suitable if you're working with designers who have no knowledge of XAML. However, because the window will be rendered with more pixels at higher system DPIs, the background graphic may become blurry. This is also a problem if you choose to allow the user to resize the window.
- Use the shape-drawing features in WPF to create your background with vector content. This approach ensures that you won't lose quality regardless of the window size and system DPI setting. However, you'll probably want to use a XAML-capable design tool like Expression Blend.
- Use a simpler WPF element that has the shape you want. For example, you can create a nicely rounded window edge with the Border element. This gives you a modern Office-style window appearance with no design work.

Here's a bare-bones transparent window that uses the first approach and supplies a PNG file with transparent regions:

```
<Window x:Class="Windows.TransparentBackground" ...
    WindowStyle="None" AllowsTransparency="True"
    >
  <Window.Background>
    <ImageBrush ImageSource="squares.png"></ImageBrush>
  </Window.Background>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
      </Grid.RowDefinitions>
      <Button Margin="20">A Sample Button</Button>
      <Button Margin="20" Grid.Row="2">Another Button</Button>
    </Grid>
</Window>
```

Figure 23-5 shows this window with a Notepad window underneath. Not only does the shaped window (which consists of a circle and square) leave gaps through which you can see the content underneath, but also some buttons drift off the image and into the transparent region, which means they appear to be floating without a window.



```
/* Copyright Abandoned 1997 TCX DataKonsult AB & Monty
Program KB & Detron HB
   This file is public domain and comes with NO
WARRANTY of any kind */

"hashchk",
"isamchk",
"NO",
"YES",
"Can't create file '%-.64
"Can't create table '%-.6
"Can't create database '
"Can't create database
"Can't drop database '%
"Error dropping databas
%d)",
"Error dropping databas
%d)",
"Error on delete of '%-.
"Can't read record in sys
"Can't get status of '%-.64
"Can't get working direct
"Can't lock file (errno:
"Can't open file: '%-.64s
"Can't find file: '%-.64s' (errno: %d)
"Can't read dir of '%-.64s' (errno: %d)'
"Can't change dir to '%-.64s' (errno: %
```

Figure 23-5. A shaped window that uses a background image

If you've programmed with Windows Forms before, you'll probably notice that shaped windows in WPF have cleaner edges, especially around curves. That's because WPF is able to perform anti-aliasing between the background of your window and the content underneath to create the smoothened edge.

Figure 23-6 shows another, subtler shaped window. This window uses a rounded Border element to give a distinctive look. The layout is also simplified, because there's no way your content could accidentally leak outside the border, and the border can be easily resized with no Viewbox required.

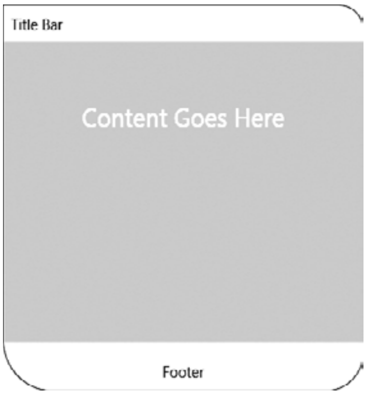


Figure 23-6. A shaped window that uses a Border

This window holds a Grid with three rows, which are used for the title bar, the footer bar, and all the content in between. The content row holds a second Grid, which sets a different background and holds any other elements you want (currently, it holds just a single TextBlock).

Here's the markup that creates the window:

```
<Window x:Class="Windows.ModernWindow" ...
    AllowsTransparency="True" WindowStyle="None"
    Background="Transparent"
    >
  <Border Width="Auto" Height="Auto" Name="windowFrame"
    BorderBrush="#395984" BorderThickness="1"
    CornerRadius="0,20,30,40" >
    <Border.Background>
      <LinearGradientBrush>
        <GradientBrush.GradientStops>
          <GradientStopCollection>
            <GradientStop Color="#E7EBF7" Offset="0.0"/>
            <GradientStop Color="#CEE3FF" Offset="0.5"/>
            </GradientStopCollection>
        </GradientBrush.GradientStops>
      </LinearGradientBrush>
    </Border.Background>

    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
      </Grid.RowDefinitions>

     <TextBlock Text="Title Bar" Margin="1" Padding="5"></TextBlock>

      <Grid Grid.Row="1" Background="#B5CBEF">
        <TextBlock VerticalAlignment="Center" HorizontalAlignment="Center"
         Foreground="White" FontSize="20">Content Goes Here</TextBlock>
      </Grid>

      <TextBlock Grid.Row="2" Text="Footer" Margin="1,10,1,1" Padding="5"
       HorizontalAlignment="Center"></TextBlock>
    </Grid>
  </Border>
</Window>
```

To complete this window, you would want to create buttons that mimic the standard maximize, minimize, and close buttons in the top-right corner.

A Transparent Window with Shaped Content

In most cases, WPF windows won't use fixed graphics to create shaped windows. Instead, they'll use a completely transparent background and then place shaped content on this background. (You can see how this works by looking at the button in Figure 23-5, which is hovering over a completely transparent region.)

The advantage of this approach is that it's more modular. You can assemble a window out of many separate components, all of which are first-class WPF elements. But more important, this allows you to take advantage of other WPF features to build truly dynamic user interfaces. For example, you might assemble shaped content that can be resized, or use animation to produce perpetually running effects right in your window. This isn't as easy if your graphics are provided in a single static file.

Figure 23-7 shows an example. Here, the window contains a Grid with one cell. Two elements share that cell. The first element is a Path that draws the shaped window border and gives it a gradient fill. The other element is a layout container that holds the content for the window, which overlays the Path. In this case, the layout container is a StackPanel, but you could also use something else (such as another Grid or a Canvas for coordinate-based absolute positioning). This StackPanel holds the close button (with the familiar X icon) and the text.

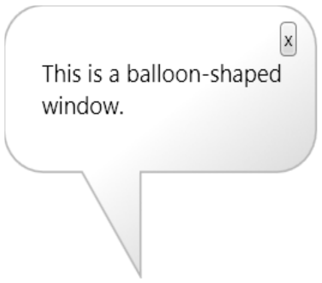


Figure 23-7. A shaped window that uses a Path

■ **Note** Even though Figure 23-5 and Figure 23-6 show different examples, they are interchangeable. In other words, you could create either one using the background-based approach or the shape-drawing approach. However, the shape-drawing approach gives you more abilities if you want to dynamically change the shape later and gives you the best quality if you need to resize the window.

The key piece of this example is the Path element that creates the backgrounds. It's a simple vector-based shape that's composed of a series of lines and arcs. Here's the complete markup for the Path:

```
<Path Stroke="DarkGray" StrokeThickness="2">
  <Path.Fill>
    <LinearGradientBrush StartPoint="0.2,0" EndPoint="0.8,1" >
      <LinearGradientBrush.GradientStops>
        <GradientStop Color="White" Offset="0"></GradientStop>
        <GradientStop Color="White" Offset="0.45"></GradientStop>
        <GradientStop Color="LightBlue" Offset="0.9"></GradientStop>
        <GradientStop Color="Gray" Offset="1"></GradientStop>
      </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
  </Path.Fill>
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="20,0" IsClosed="True">
          <LineSegment Point="140,0"/>
          <ArcSegment Point="160,20" Size="20,20" SweepDirection="Clockwise"/>
          <LineSegment Point="160,60"/>
          <ArcSegment Point="140,80" Size="20,20" SweepDirection="Clockwise"/>
          <LineSegment Point="70,80"/>
          <LineSegment Point="70,130"/>
          <LineSegment Point="40,80"/>
          <LineSegment Point="20,80"/>
          <ArcSegment Point="0,60" Size="20,20" SweepDirection="Clockwise"/>
          <LineSegment Point="0,20"/>
          <ArcSegment Point="20,0" Size="20,20" SweepDirection="Clockwise"/>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

Currently, the Path is fixed in size (as is the window), although you could make it resizable by hosting it in the Viewbox container that you learned about in Chapter 12. You could also improve this example by giving the close button a more authentic appearance—probably a vector X icon that's drawn on a red surface. Although you could use a separate Path element to represent a button and handle the button's mouse events, it's better to change the standard Button control using a control template (as described in Chapter 17). You can then make the Path that draws the X icon part of your customized button.

Moving Shaped Windows

One limitation of shaped forms is that they omit the nonclient title bar portion, which allows the user to easily drag the window around the desktop. In Windows Forms, this was a bit of a chore—you had to react to mouse events such as MouseDown, MouseUp, and MouseMove and move the window manually when the user clicked and dragged, or you had to override the WndProc() method and handle the low-level WM_NCHITTEST message. WPF makes the same task much easier. You can initiate window-dragging mode at any time by calling the Window.DragMove() method.

So, to allow the user to drag the shaped form you saw in the previous examples, you simply need to handle the MouseLeftButtonDown event for the window (or an element on the window, which will then play the same role as the title bar):

```
<TextBlock Text="Title Bar" Margin="1" Padding="5"
 MouseLeftButtonDown="titleBar_MouseLeftButtonDown"></TextBlock>
```

In your event handler, you need only a single line of code:

```
private void titleBar_MouseLeftButtonDown(object sender,
  MouseButtonEventArgs e)
{
    this.DragMove();
}
```

Now the window follows the mouse around the screen, until the user releases the mouse button.

Resizing Shaped Windows

Resizing a shaped window isn't as easy. If your window is roughly rectangular in shape, the easiest approach is to add a sizing grip to the bottom-right corner by setting the Window.ResizeMode property to CanResizeWithGrip. However, the sizing grip placement assumes that your window is rectangular. For example, if you're creating a rounded window effect using a Border object, as shown earlier in Figure 23-6, this technique may work. The sizing grip will appear in the bottom-right corner, and depending how much you've rounded off that corner, it may appear over the window surface where it belongs. But if you've created a more exotic shape, such as the Path shown earlier in Figure 23-7, this technique definitely won't work; instead, it will create a sizing grip that floats in empty space next to the window.

If the sizing grip placement isn't right for your window, or you want to allow the user to size the window by dragging its edges, you'll need to go to a bit more work. You can use two basic approaches. You can use .NET's platform invoke feature (p/invoke) to send a Win32 message that resizes the window. Or you can simply track the mouse position as the user drags to one side, and resize the window manually, by setting its Width property. The following example uses the latter approach.

Before you can use either approach, you need a way to detect when the user moves the mouse over the edges of the window. At this point, the mouse pointer should change to a resize cursor. The easiest way to do this in WPF is to place an element along the edge of each window. This element doesn't need to have any visual appearance—in fact, it can be completely transparent and let the window show through. Its sole purpose is to intercept mouse events.

A 5-unit wide Rectangle is perfect for the task. Here's how you might place a Rectangle that allows right-side resizing in the rounded-edge window shown in Figure 23-6:

```
<Grid>
    ...
    <Rectangle Grid.RowSpan="3" Width="5"
     VerticalAlignment="Stretch" HorizontalAlignment="Right"
     Cursor="SizeWE" Fill="Transparent"
     MouseLeftButtonDown="window_initiateWiden"
     MouseLeftButtonUp="window_endWiden"
     MouseMove="window_Widen"></Rectangle>
</Grid>
```

The Rectangle is placed in the top row but is given a RowSpan value of 3. That way, it stretches along all three rows and occupies the entire right side of the window. The Cursor property is set to the mouse cursor you want to show when the mouse is over this element. In this case, the "west-east" resize cursor does the trick—it shows the familiar two-way arrow that points left and right.

The Rectangle event handlers toggle the window into resize mode when the user clicks the edge. The only trick is that you need to capture the mouse to ensure you continue receiving mouse events even if the mouse is dragged off the rectangle. The mouse capture is released when the user releases the left mouse button.

```
bool isWiden = false;

private void window_initiateWiden(object sender, MouseEventArgs e)
{
    isWiden = true;
}
```

```
private void window_Widen(object sender, MouseEventArgs e)
{
    Rectangle rect = (Rectangle)sender;
    if (isWiden)
    {
        rect.CaptureMouse();
        double newWidth = e.GetPosition(this).X + 5;
        if (newWidth > 0) this.Width = newWidth;
    }
}

private void window_endWiden(object sender, MouseEventArgs e)
{
    isWiden = false;

    // Make sure capture is released.
    Rectangle rect = (Rectangle)sender;
    rect.ReleaseMouseCapture();
}
```

Figure 23-8 shows the code in action.

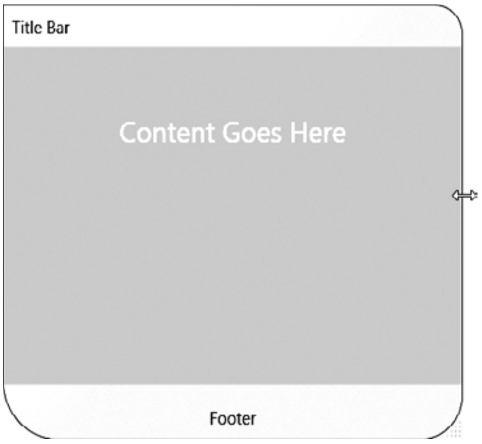


***Figure 23-8.** Resizing a shaped window*

Putting It All Together: A Custom Control Template for Windows

Using the code you've seen so far, you can build a custom-shaped window quite easily. However, if you want to use a new window standard for your entire application, you'll be forced to manually restyle every window with the same shaped border, header region, close buttons, and so on. In this situation, a better approach is to adapt your markup into a control template that you can use on any window. (If you're a bit sketchy on the inner workings of control templates, you might want to review the information in Chapter 17 before you continue with the rest of this section.)

The first step is to consult the default control template for the Window class. For the most part, this template is pretty straightforward, but it includes one detail you might not expect: an AdornerDecorator element. This element creates a special drawing area called the *adorner layer* over the rest of the window's client content. WPF controls can use the adorner layer to draw content that should appear superimposed over your elements. This includes small graphical indicators that show focus, flag validation errors, and guide drag-and-drop operations. When you build a custom window, you need to ensure that the adorner layer is present, so that controls that use it continue to function.

With that in mind, it's possible to identify the basic structure that the control template for a window should take. Here's a standardized example with markup that creates window like the one shown in Figure 23-8:

```
<ControlTemplate x:Key="CustomWindowTemplate" TargetType="{x:Type Window}">
  <Border Name="windowFrame" ... >
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto"></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition Height="Auto"></RowDefinition>
      </Grid.RowDefinitions>

      <!-- The title bar. -->
      <TextBlock Text="{TemplateBinding Title}"
       FontWeight="Bold"></TextBlock>
      <Button Style="{StaticResource CloseButton}"
       HorizontalAlignment="Right"></Button>

      <!-- The window content. -->
      <Border Grid.Row="1">
        <AdornerDecorator>
          <ContentPresenter></ContentPresenter>
        </AdornerDecorator>
      </Border>

      <!-- The footer. -->
      <ContentPresenter Grid.Row="2" Margin="10"
       HorizontalAlignment="Center"
       Content="{TemplateBinding Tag}"></ContentPresenter>

      <!-- The resize grip. -->
      <ResizeGrip Name="WindowResizeGrip" Grid.Row="2"
       HorizontalAlignment="Right" VerticalAlignment="Bottom"
       Visibility="Collapsed" IsTabStop="False" />

      <!-- The invisible rectangles that allow dragging to resize. -->
      <Rectangle Grid.Row="1" Grid.RowSpan="3" Cursor="SizeWE"
       VerticalAlignment="Stretch" HorizontalAlignment="Right"
       Fill="Transparent" Width="5"></Rectangle>
      <Rectangle Grid.Row="2" Cursor="SizeNS"
       HorizontalAlignment="Stretch" VerticalAlignment="Bottom"
       Fill="Transparent" Height="5"></Rectangle>
    </Grid>
  </Border>
  <ControlTemplate.Triggers>
    <Trigger Property="ResizeMode" Value="CanResizeWithGrip">
      <Setter TargetName="WindowResizeGrip"
       Property="Visibility" Value="Visible"></Setter>
    </Trigger>
  </ControlTemplate.Triggers>
</ControlTemplate>
```

The top-level element in this template is a Border object for the window frame. Inside that is a Grid with three rows. The contents of the Grid break down as follows:

- The top row holds the title bar, which consists of an ordinary TextBlock that displays the window title and a close button. A template binding pulls the window title from the Window.Title property.
- The middle row holds a nested Border with the rest of the window content. The content is inserted using a ContentPresenter. The ContentPresenter is wrapped in the AdornerDecorator, which ensures that the adorner layer is placed over your element content.
- The third row holds another ContentPresenter. However, this content presenter doesn't use the standard binding to get its content from the Window.Content property. Instead, it explicitly pulls its content from the Window.Tag property. Usually, this content is just ordinary text, but it could include any element content you want to use.

■ **Note** The Tag property is used because the Window class doesn't include any property that's designed to hold footer text. Another option is to create a custom class that derives from Window and adds a Footer property.

- Also in the third row is a resize grip. A trigger shows the resize grip when the Window.ResizeMode property is set to CanResizeWithGrip.
- Finally, two invisible rectangles run along the right and bottom edge of the Grid (and thus the window). They allow the user to click and drag to resize the window.

Two details that aren't shown here are the relatively uninteresting style for the resize grip (which simply creates a small pattern of dots to use as the resize grip) and the close button (which draws a small *X* on a red square). This markup also doesn't include the formatting details, such as the gradient brush that paints the background and the properties that create a nicely rounded border edge. To see the full markup, refer to the sample code provided for this chapter.

The window template is applied using a simple style. This style also sets three key properties of the Window class that make it transparent. This allows you to create the window border and background using WPF elements.

```
<Style x:Key="CustomWindowChrome" TargetType="{x:Type Window}">
  <Setter Property="AllowsTransparency" Value="True"></Setter>
  <Setter Property="WindowStyle" Value="None"></Setter>
  <Setter Property="Background" Value="Transparent"></Setter>
  <Setter Property="Template"
   Value="{StaticResource CustomWindowTemplate}"></Setter>
</Style>
```

At this point, you're ready to use your custom window. For example, you could create a window like this that sets the style and fills in some basic content:

```
<Window x:Class="ControlTemplates.CustomWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="CustomWindowTest" Height="300" Width="300"
  Tag="This is a custom footer"
  Style="{StaticResource CustomWindowChrome}">

  <StackPanel Margin="10">
    <TextBlock Margin="3">This is a test.</TextBlock>
    <Button Margin="3" Padding="3">OK</Button>
  </StackPanel>
</Window>
```

There's just one problem. Currently, the window lacks most of the basic behavior windows require. For example, you can't drag the window around the desktop, resize it, or use the close button. To perform these actions, you need code.

There are two possible ways to add the code you need: you could expand your example into a custom Window-derived class, or you could create a code-behind class for your resource dictionary. The custom control approach provides better encapsulation and allows you to extend the public interface of your window (for example, adding useful methods and properties that you can use in your application). The code-behind approach is a relatively lightweight alternative that allows you to extend the capabilities of a control template while letting your application continue to use the base control classes. It's the approach that you'll see in this example.

You've already learned how to create a code-behind class for your resource dictionary (see the "User-Selected Skins" section in Chapter 17). Once you've created the code file, it's easy to add the event handling code you need. The only challenge is that your code runs in the resource dictionary object, not inside your window object. That means you can't use the this keyword to access the current window. Fortunately, there's an easy alternative: the FrameworkElement.TemplatedParent property.

For example, to make the window draggable, you need to intercept a mouse event on the title bar and initiate dragging. Here's the revised TextBlock that wires up an event handler when the user clicks with the mouse:

```
<TextBlock Margin="1" Padding="5" Text="{TemplateBinding Title}"
 FontWeight="Bold" MouseLeftButtonDown="titleBar_MouseLeftButtonDown"></TextBlock>
```

Now you can add the following event handler to the code-behind class for the resource dictionary:

```
private void titleBar_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    Window win = (Window)
      ((FrameworkElement)sender).TemplatedParent;
    win.DragMove();
}
```

You can add the event handling code for the close button and resize rectangles in the same way. To see the finished resource dictionary markup and code, along with a template that you apply to any window, refer to the download examples for this chapter.

Of course, there's still a lot of polish needed before this window is attractive enough to suit a modern application. But it demonstrates the sequence of steps you need to follow to build a complex control template, with code, and it achieves a result that would have required custom control development in previous user interface frameworks.

The Aero Glass Effect

Surprisingly, WPF still doesn't have any built-in way for you to take advantage of the Aero glass effect first introduced in Windows Vista and supported in Windows 7. It gives windows their familiar blurred glass frames, through which you can see other windows and their content.

Before you go any further, it's worth noting that Aero glass won't work on Windows XP. To avoid problems, you should write code that checks the operating system and degrades gracefully when necessary. The easiest way to determine whether you're running on Windows Vista is to read the static OSVersion property from the System.Environment class. Here's how it works:

```
if (Environment.OSVersion.Version.Major >= 6)
{
    // Vista features are supported.
}
```

Applications running under Windows Vista and Windows 7 get the Aero glass effect for free in the nonclient region of the window. If you show a standard window with a standard window frame in WPF, and your application is running on an Aero-capable computer, you'll get the eye-catching translucent window frame.

■ **Note** To be Aero-capable, the computer must have a modern version of Windows other than Windows XP, Windows Vista Home Basic, and Windows 7 Starter. It must have the required video card support. Also, it must have the Aero glass feature switched on, which it is by default.

Some applications extend the Aero glass effect into the client area of the window. Two examples are Internet Explorer, which features the glass effect behind the address bar, and Media Player, which uses it behind the playback controls. You can perform the same magic in your own applications. You'll encounter only two limitations:

- The blurred glass area of your window always begins at the edges of your window. That means you can't create a glass "patch" somewhere in the middle. However, you can place completely opaque WPF elements on the glass frame to create a similar effect.
- The nonglass region inside your window is always defined as a rectangle.

WPF doesn't include classes for performing this effect. Instead, you need to call the DwmExtendFrameIntoClientArea() function from the Win32 API. (The *Dwm* prefix refers to the *Desktop Window Manager*, which controls this effect.) Calling this function allows you to extend the frame into your client area by making one or all of the edges thicker.

Here's how you can import the DwmExtendFrameIntoClientArea() function so it's callable from your application:

```
[DllImport("DwmApi.dll")]
public static extern int DwmExtendFrameIntoClientArea(
  IntPtr hwnd,
  ref Margins pMarInset);
```

You also need to define the fixed Margins structure, as shown here:

```
[StructLayout(LayoutKind.Sequential)]
public struct Margins
{
    public int cxLeftWidth;
    public int cxRightWidth;
    public int cyTopHeight;
    public int cyBottomHeight;
}
```

This has one potential stumbling block. As you already know, the WPF measurement system uses device-independent units that are sized based on the system DPI setting. However, the DwmExtendFrameIntoClientArea() uses physical pixels. To make sure your WPF elements line up with your extended glass frame, you need to take the system DPI into account in your calculations.

The easiest way to retrieve the system DPI is to use the System.Drawing.Graphics class, which exposes two properties that indicate the DPI of a window: DpiX and DpiY. The following code shows a helper method that takes a handle to a window and a set of WPF units, and returns a Margin object with the correspondingly adjusted measurement in physical pixels.

```
public static Margins GetDpiAdjustedMargins(IntPtr windowHandle,
  int left, int right, int top, int bottom)
{
    // Get the system DPI.
    System.Drawing.Graphics g = System.Drawing.Graphics.FromHwnd(windowHandle);
    float desktopDpiX = g.DpiX;
    float desktopDpiY = g.DpiY;

    // Set the margins.
    VistaGlassHelper.Margins margins = new VistaGlassHelper.Margins();
    margins.cxLeftWidth = Convert.ToInt32(left * (desktopDpiX / 96));
    margins.cxRightWidth = Convert.ToInt32(right * (desktopDpiX / 96));
    margins.cyTopHeight = Convert.ToInt32(top * (desktopDpiX / 96));
    margins.cyBottomHeight = Convert.ToInt32(right * (desktopDpiX / 96));

    return margins;
}
```

■ **Note** Unfortunately, the System.Drawing.Graphics is a part of Windows Forms. To gain access to it, you need to add a reference to the System.Drawing.dll assembly.

The final step is to apply the margins to the window using the DwmExtendFrameIntoClientArea() function. The following code shows an all-in-one helper method that takes the WPF margin measurements and a reference to a WPF window. It then gets the Win32 handle for the window, adjusts the margins, and attempts to extend the glass frame.

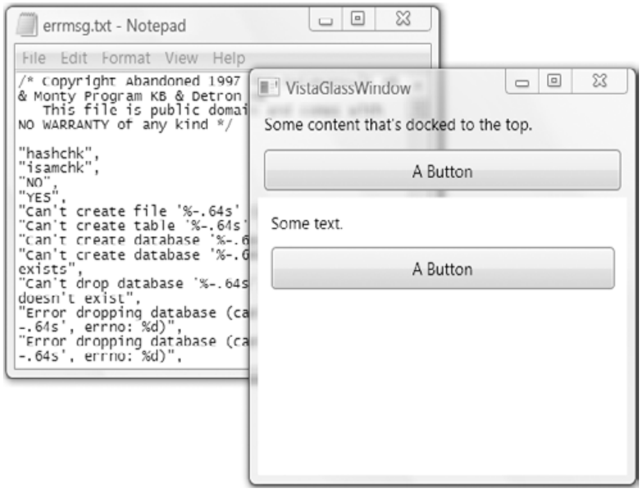
```
public static void ExtendGlass(Window win, int left, int right,
  int top, int bottom)
{
    // Obtain the Win32 window handle for the WPF window.
    WindowInteropHelper windowInterop = new WindowInteropHelper(win);
    IntPtr windowHandle = windowInterop.Handle;

    // Adjust the margins to take the system DPI into account.
    Margins margins = GetDpiAdjustedMargins(
      windowHandle, left, right, top, bottom);

    // Extend the glass frame.
    int returnVal = DwmExtendFrameIntoClientArea(windowHandle, ref margins);
    if (returnVal < 0)
    {
        throw new NotSupportedException("Operation failed.");
    }
}
```

The sample code for this chapter wraps all these ingredients into a single class, called VistaGlassHelper, which you can call from any window. For the code to work, you must call it before the window is shown. The Window.Loaded event provides the perfect opportunity. Additionally, you must remember to set the Background of your window to Transparent so the glass frame shows through the WPF drawing surface.

Figure 23-9 shows an example that thickens the top edge of the glass frame.


errmsg.txt - Notepad
File Edit Format View Help
VistaGlassWindow
Some content that's docked to the top.
A Button
Some text.
A Button

```
/* Copyright Abandoned 1997
& Monty Program KB & Detron
   This file is public domai
NO WARRANTY of any kind */

"hashchk",
"isamchk",
"NO",
"YES",
"Can't create file '%-.64s'
"Can't create table '%-.64s'
"Can't create database '%-.6
"Can't create database '%-.6
exists",
"Can't drop database '%-.64s
doesn't exist",
"Error dropping database (ca
-.64s', errno: %d)",
"Error dropping database (ca
-.64s', errno: %d)",
```

Figure 23-9. Extending the glass frame

When creating this window, the content at the top is grouped into a single Border element. That way, you can measure the height of the border and use that measurement to extend the glass frame. (Of course, the glass frame is set only once, when the window is first created. If you change content or resize the window and the Border grows or shrinks, it won't line up with the glass frame any longer.)

Here's the complete markup for the window:

```
<Window x:Class="Windows.VistaGlassWindow2"
    ...
    Loaded="window_Loaded" Background="Transparent"
    >
  <Grid >
    <DockPanel Name="mainDock" LastChildFill="True">
      <!-- The border is used to compute the rendered height with margins.
           topBar contents will be displayed on the extended glass frame.-->
      <Border Name="topBar" DockPanel.Dock="Top">
        <StackPanel>
          <TextBlock Padding="5">Some content that's docked to the top.</TextBlock>
          <Button Margin="5" Padding="5">A Button</Button>
        </StackPanel>
      </Border>
      <Border Background="White">
        <StackPanel Margin="5">
          <TextBlock Margin="5" >Some text.</TextBlock>
          <Button Margin="5" Padding="5">A Button</Button>
        </StackPanel>
      </Border>
    </DockPanel>
  </Grid>
</Window>
```

Notice that the second Border in this window, which contains the rest of the content, must explicitly set its background to white. Otherwise, this part of the window will be completely transparent. For the same reason, the second Border shouldn't have any margin space, or you'll see a transparent edge around it.

When the window is loaded, it calls the ExtendGlass() method and passes in the new coordinates. Ordinarily, the glass frame is 5 units thick, but this code adds to the top edge.

```
private void window_Loaded(object sender, RoutedEventArgs e)
{
    try
    {
        VistaGlassHelper.ExtendGlass(this, 5, 5,
          (int)topBar.ActualHeight + 5, 5);
    }
    catch
    {
        // A DllNotFoundException occurs if you run this on Windows XP.
        // A NotSupportedException is thrown if the
        // DwmExtendFrameIntoClientArea() call fails.
        this.Background = Brushes.White;
    }
}
```