Learn VB .NET Through Game Programming

Matthew Tagliaferri

Apress™

Learn VB .NET Through Game Programming
Copyright ©2003 by Matthew Tagliaferri

ISBN (pbk): 1-59059-114-3

Printed and bound in the United States of America 12345678910

Technical Reviewer: Mike Burgess

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Managers: Laura Cheu and Nate McFadden

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Janet Vail

Proofreader and Indexer: Carol Burbo

Compositor: Kinetic Publishing Services, LLC

Artist: Dina Quan

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 8

Using DirectX

YOU MAY NOT HAVE NOTICED, but the drawing speed in some of the previous games was a bit slow—okay, maybe more than a bit slow. The slowdown was most noticeable in the NineTiles game from Chapter 3, "Understanding Object-Oriented Programming from the Start." In fact, I was originally going to add an opening animation sequence that showed all the tiles flipping over simultaneously, but this turned out to be too slow.

If you do some Google research on speed issues, you'll find that the bitmap rendering in the Graphics Device Interchange, Plus (GDI+) classes isn't quite ready for prime time. There are reports of unnecessary palette and color transformations going on behind the scenes when using the GDI+ classes for drawing. Trying to correct the problem by changing the color depth of the source bitmaps does nothing to speed up the drawing to any great degree.

Fortunately, for most of the games discussed to this point, blazing-fast bitmap rendering speed isn't necessary. The games should run at an acceptable speed. If you continue on the game-development track and create bigger and more complicated games, this speed will most likely become a barrier at some point, though. You have three ways to get around the graphics speed trap:

- Stop writing games until Microsoft addresses some of the GDI issues.
- Drop back down to the Win32 application programming interface (API) for graphics drawing (`bitblt`, `stretchblt`, and so on).
- Move over to DirectX drawing.

Obviously, the first option is no fun at all (plus, the book would have to end right here!). The second option is possible, but don't you get a slight feeling of failure when you have to stop using a cool new language and return to old habits? Plus, using the `bitblt` function could get tricky with all the device handles and such.

The third option sounds like the best choice by default. DirectX is a huge set of multimedia functionality built into the Windows operating system. It has gotten both bigger and better with each new release, and the latest release, DirectX 9, is no exception. DirectX 9 includes a managed class interface for the .NET developer. In other words, drawing using the DirectX classes is scarcely more difficult than drawing using the GDI+ classes. Thus, you'll break the speed barrier.

Installing DirectX 9

A version of DirectX is installed with every version of Windows, but the erstwhile developer needs the DirectX software development kit (SDK) to program to the DirectX libraries. You can find the SDK at `http://www.microsoft.com/windows/directx/default.aspx`. After downloading and installing it, you'll find a `DXSDK` folder on your C: drive packed full o' DirectX goodness. The huge help file might be the first thing you want to peruse, or you can dig right into the sample programs, which are available in Visual Basic .NET, C#, or C++.

NOTE *DirectX 9 is a large enough API that one could write an entire book about the library. In fact, someone has. Check out* .NET Game Programming with DirectX 9.0 *by Alexandre Santos Lobao and Ellen Hatton (Apress, 2003).*

This chapter focuses on one aspect of DirectX, known as *DirectDraw*. This functionality creates fast bitmap graphics, which is what needs to improve in the old games. In this chapter, you'll learn about DirectDraw through two example programs. The first is a "do-one-thing" program that simply introduces the concepts and renders a bunch of graphics to the screen to prove the speed of the DirectX library. The second program puts the concepts together to create the bulk of an arcade game.

Understanding DirectDraw Basics

The sample solution DirectXDemo demonstrates displaying bitmap images to the screen using the DirectX API. In a good display of conservation, it recycles one of the graphics from a prior project, the three-dimensional die. To demonstrate the speed of the DirectX API, this demo displays 250 spinning dice in random locations on the screen, as shown in Figure 8-1.

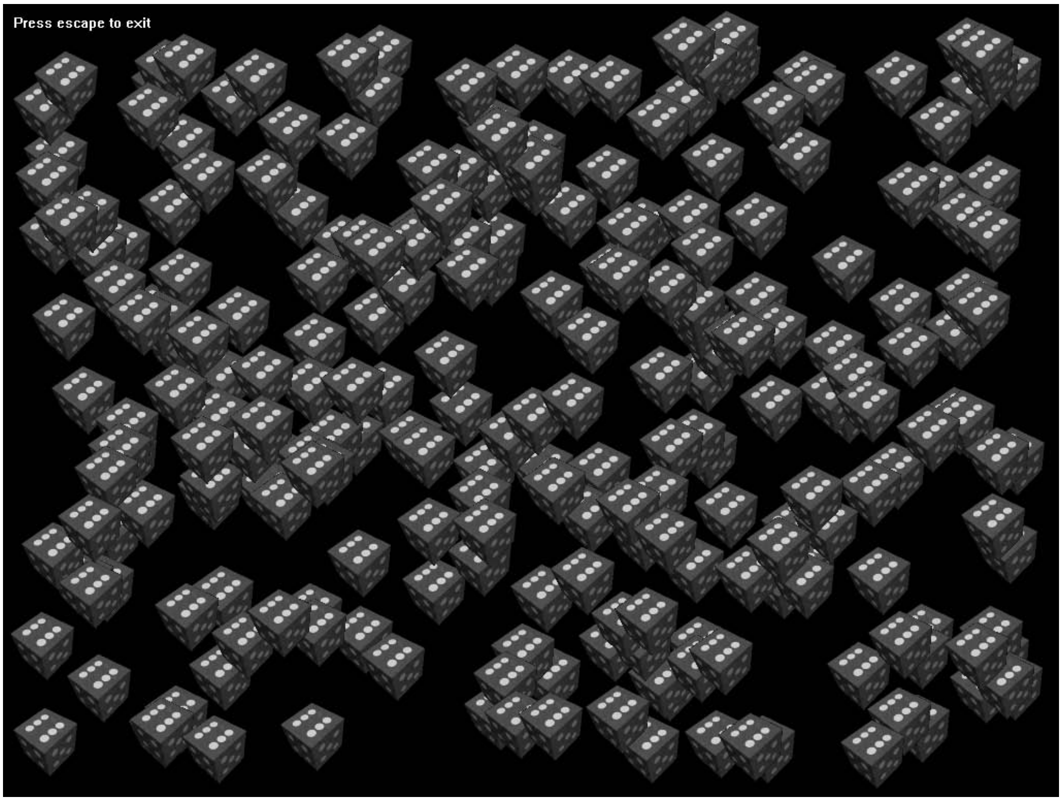

Figure 8-1. Spinning dice aplenty

DirectX drawing is based on the concept of *surfaces*. A surface is both a source for bitmap data and a destination. The DirectXDemo application utilizes three surfaces:

- The source bitmap
- The "screen" surface (also called the front surface)
- The back buffer surface, or back surface

DirectDraw achieves smooth animation by performing all drawing to a hidden surface, the back buffer, and then swapping the position of the front and back surfaces once rendering is complete. As a developer, you don't need to keep track of which surface is being displayed, however—all drawing always happens on the back buffer.

DirectX 9 encapsulates all of the functionality of a DirectDraw surface inside a .NET Framework managed class called (obviously enough) `Surface`. This class resides in the `Microsoft.DirectX.DirectDraw` namespace, which becomes available in Visual Studio .NET after installing the DirectX 9 SDK.

The other important class you'll use in a simple DirectDraw application is the `Device` class. The `Device` class encapsulates the capabilities of the system upon which the program is running.

Initializing a DirectDraw Application

Getting a DirectDraw application ready for rendering using the DirectX 9 managed classes requires setting up a `Device` instance and the front and back surfaces used for rendering. Listing 8-1 shows some private form variables and the initialization routine used in the demo application.

Listing 8-1. Setting Up a DirectDraw Application

```
Private Const WID As Integer = 1024
Private Const HGT As Integer = 768

Private FDraw As Microsoft.DirectX.DirectDraw.Device
Private FFront As Microsoft.DirectX.DirectDraw.Surface
Private FBack As Microsoft.DirectX.DirectDraw.Surface

Private Sub InitializeDirectDraw()

    Dim oSurfaceDesc As New SurfaceDescription
    Dim oSurfaceCaps As New SurfaceCaps
    Dim i As Integer

    FDraw = New Microsoft.DirectX.DirectDraw.Device

    FDraw.SetCooperativeLevel(Me, _
      Microsoft.DirectX.DirectDraw._
      CooperativeLevelFlags.FullscreenExclusive)
    FDraw.SetDisplayMode(WID, HGT, 16, 0, False)

    With oSurfaceDesc
        .SurfaceCaps.PrimarySurface = True
        .SurfaceCaps.Flip = True
        .SurfaceCaps.Complex = True
        .BackBufferCount = 1
        FFront = New Surface(oSurfaceDesc, FDraw)
```

```
            oSurfaceCaps.BackBuffer = True
            FBack = FFront.GetAttachedSurface(oSurfaceCaps)
            FBack.ForeColor = Color.White
            .Clear()
        End With
    FNeedToRestore = True
End Sub
```

The `InitializeDirectDraw` procedure begins by creating an instance of a DirectDraw `Device` class and sets what's known as the *cooperative level*. The cooperative level specifies to the operating system the performance requirements of your application. Intensive games will want to use the `FullscreenExclusive` level used here, meaning that the application will create a full-screen window (unrelated to any form in the application) upon which the drawing will happen.

Next, the `SetDisplayMode` method sets the resolution of the window. The sample program creates a 1024×768 window using 16-bit color.

The remainder of the procedure defines the device as having one back buffer and then initializes the front and back surfaces. The front surface, `FFront`, is instantiated by calling the constructor and passing the `Device` variable to it. The back buffer, `FBack`, is retrieved by calling a method on the front surface (`GetAttachedSurface`). The last line within the `With` block clears the surface. Finally, a Boolean variable named `FNeedToRestore` is set to `True`, which tells the class that all of the DirectX surfaces require restoration before drawing can happen.

The program now has destination surfaces, but it still needs a source surface to store the die bitmap that contains the animated frames. You need to import the die bitmap into the solution as in Chapter 1, "Developing Your First Game." Listing 8-2 contains the code that loads this bitmap into a DirectDraw surface.

TIP *Don't forget to change the* `Build Action` *property on any bitmaps in your solution to* `Embedded Resource`.

Listing 8-2. Loading Bitmaps into `Surface` *Instances*

```
Private FDieSurf As Microsoft.DirectX.DirectDraw.Surface

Public Sub RestoreSurfaces()

    Dim oCK As New ColorKey
    Dim a As Reflection.Assembly = _
        System.Reflection.Assembly.GetExecutingAssembly()
    FDraw.RestoreAllSurfaces()

    If Not FDieSurf Is Nothing Then
        FDieSurf.Dispose()
        FDieSurf = Nothing
    End If

    FDieSurf = New Surface(a.GetManifestResourceStream( _
        "DirectXDemo.dicexrot.bmp"), New SurfaceDescription, FDraw)
    FDieSurf.SetColorKey(ColorKeyFlags.SourceDraw, oCK)

End Sub
```

The `Surface` class takes a resource stream as its first parameter. This is the same way that a GDI+ `Bitmap` class loads a resource that's embedded in the solution. The second parameter is a `SurfaceDescription` class instance. This class contains properties that can describe the surface (you can see another `SurfaceDescription` class being used in Listing 8-1 to describe some aspects of the front and back screen surfaces). For loading bitmaps, the surface description properties aren't needed because the attributes of the surface are retrieved from the attributes of the bitmap itself. Thus, the surface constructor in Listing 8-2 creates a blank, default `SurfaceDescription` with no specified properties.

The last line in Listing 8-2 sets the color key for the surface. A color key specifies one or more colors that are to be treated as transparent when rendering. Because no such color assignment happens in Listing 8-2, the `ColorKey` object named `oCK` declares pure black (RGB color `0, 0, 0`) as the transparent color. This program uses black because the background of the dice bitmap is also black.

Note that the surfaces of your application may be "lost" and require restoration. This is especially true in windowed DirectDraw applications (as opposed to full-screen applications) that can lose focus. Because of this possibility, the main drawing loop of the program needs to check that the device is ready before it can actually draw. Once the device comes back from a "not ready" state, the source bitmaps need to be re-created. This is why the method in Listing 8-2 is called `RestoreSurfaces` as opposed to a name that connotes a one-time load such as `LoadSurfaces`.

Creating the Drawing Loop

The drawing in the sample program happens in a method named `DrawFrame`. Listing 8-3 shows the majority of this routine, along with the `Form_Load` and `Form_KeyUp` events.

Listing 8-3. The `DrawFrame` *Method*

```
Private Sub Form1_Load(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles MyBase.Load

  Me.Cursor.Dispose()
  InitializeDirectDraw()
  SetupDice

   While Me.Created
      DrawFrame()
   End While
End Sub

Private Sub DrawFrame()

   If FFront Is Nothing Then Exit Sub

   'can't draw now, device not ready
   If Not FDraw.TestCooperativeLevel() Then
      FNeedToRestore = True
      Exit Sub
   End If

   If FNeedToRestore Then
      RestoreSurfaces()
      FNeedToRestore = False
   End If

   FBack.ColorFill(0)
   < drawing code removed>

   Try
      FBack.ForeColor = Color.White
      FBack.DrawText(10, 10, "Press escape to exit", False)
      FFront.Flip(FBack, FlipFlags.DoNotWait)
   Catch oEX As Exception
      Debug.WriteLine(oEX.Message)
   Finally
      Application.DoEvents()
   End Try
End Sub
```

```
Private Sub Form1_KeyUp(ByVal sender As Object, _
  ByVal e As System.Windows.Forms.KeyEventArgs) Handles MyBase.KeyUp

   If e.KeyCode = Keys.Escape Then
      Me.Close()
   End If
End Sub
```

The `Form_Load` event runs the initialization method that has already been discussed and then calls the `DrawFrame` method over and over in a loop. The loop continues to run as long as the `Created` property on the current form is set to `True`. One other interesting thing that happens in the `Form_Load` event is the disposal of the form's cursor so that it isn't visible on the game surface. Getting rid of the cursor is as easy as invoking its `Dispose` method.

The `DrawFrame` method does some checking before any drawing happens to make sure everything is in the correct state for drawing. The first check makes sure the front surface exists. If it doesn't exist, then there would be no destination surface to display to the user, so the draw loop exits immediately. The next test happens by calling `TestCooperativeLevel` on the `Device` object. If this method returns `False`, then the device isn't ready to draw, so again the draw loop exits. In addition, a form-level Boolean variable named `FNeedToRestore` is set, which indicates that the dice source surface object needs to be re-created.

Once the `TestCooperativeLevel` method returns `True`, drawing is almost ready to begin. If the `FNeedToRestore` variable is `True`, then the source bitmaps are loaded (or reloaded) by calling `RestoreSurfaces`. With this, everything is ready for the drawing to commence.

The first task performed is clearing the back buffer to black and using the `ColorFill` method on the `Surface` object. The code immediately after the `ColorFill` method is where the actual dice drawing takes place (but I've removed that code so that the focus is on the structure of the drawing loop itself).

The remainder of the `DrawFrame` method happens inside of an exception handler so that any errors are dealt with in a graceful manner. First, some text is drawn into the upper-left corner of the back buffer, indicating that the user can hit the Escape key to stop the application. Then, the back buffer is copied to the front buffer by calling the `Flip` method on the `Surface` class. `Flip` is actually an inaccurate description inherited from previous versions of DirectDraw, where two surfaces were in fact actually swapped, serving as back buffers and then front buffers in alternate frames. The method actually copies the contents of one `Surface` class to the other.

The `Catch` portion of the exception handler writes the error to the Debug window so that the developer can inspect it later, and the `Finally` portion calls an `Application.DoEvents` so that Windows messages can process normally. Without this `DoEvents`, the application wouldn't be able to intercept keystrokes, including the keystroke meant to shut down the application.

Finally, the KeyUp event handler for the form detects the pressing of the Escape key and closes the main form when detected. This stops all drawing and exits the application.

Setting Up the Dice Drawing

Taking a quick inventory, the program now has the capability to set up a full-screen DirectDraw surface and draws a black screen with the text *Press escape to exit* in the upper-left corner. A dice bitmap also loads into a Surface instance, but it isn't actually drawn anywhere yet. All that remains is the code to track a bunch of dice and to draw them onto the screen.

Create a class named SimpleDie, shown in Listing 8-4, to keep track of each die object. It's referred to as "simple" because the code contains no capability to move around on the screen; each die simply spins in place.

Listing 8-4. Class to Keep Track of One Die on the Screen

```
Public Class SimpleDie
    Private FLocation As Point
    Private FFrame As Integer

    Public Sub New(ByVal p As Point)
        FLocation = p
    End Sub

    ReadOnly Property pLocation() As Point
        Get
            Return FLocation
        End Get
    End Property

    Public Sub Draw(ByVal FDest As Surface, ByVal FSource As Surface)

        Dim oRect As Rectangle

        oRect = New Rectangle((FFrame Mod 6) * 72, (FFrame \ 6) * 72, 72, 72)

        FDest.DrawFast(FLocation.X, FLocation.Y, FSource, oRect, _
            DrawFastFlags.DoNotWait Or DrawFastFlags.SourceColorKey)

        FFrame = (FFrame + 1) Mod 36
    End Sub
End Class
```

This class stores only two pieces of information—a screen coordinate that's passed into the class constructor and a private `Frame` variable that's incremented as each frame is drawn. The sole method on the class is the `Draw` method, which takes two DirectDraw `Surface` instances as parameters: the source image and the destination. This `Draw` method calculates a source rectangle based on the current frame and then uses a `DrawFast` method on the DirectDraw `Surface` class to transfer that part of the source surface to itself. The `DrawFast` method takes as parameters a coordinate pair (the place the bitmap should be drawn on the destination), the source surface, a rectangle that represents the portion of the source surface to copy, and some flags that can specify some additional functionality. In this case, the flags specify to use the color key of the source surface when drawing to determine transparency and to draw as quickly as possible by indicating the `DoNotWait` flag.

The demonstration program shows the speed of DirectDraw as compared to GDI+ drawing, so it should display lots of dice on the screen at the same time. The program is written in such a way that the number of dice displayed is a constant that you can easily change. You can store the information for the 250 die object instances using an `ArrayList` to store as many class instances as you want. Listing 8-5 shows the `SetupDice` method and the modified `DrawFrame` method with the code in place to draw the dice.

Listing 8-5. Initializing 250 `SimpleDie` *Object Instances*

```
Private FDice As ArrayList
Private Const NUMDICE As Integer = 250

Private Sub SetupDice()

   Dim d As SimpleDie
   Dim r As New Random

   FDice = New ArrayList
   Do While FDice.Count < NUMDICE
      d = New SimpleDie(New Point(r.Next(0, WID - 72), r.Next(0, HGT - 72)))
      FDice.Add(d)
   Loop

End Sub

Private Sub DrawFrame()

   Dim d As SimpleDie

   <code removed>
```

```
    FBack.ColorFill(0)

    For Each d In FDice
        d.Draw(FBack, FDieSurf)
    Next

    Try
        FBack.ForeColor = Color.White
        FBack.DrawText(10, 10, "Press escape to exit", False)
        FFront.Flip(FBack, FlipFlags.DoNotWait)
    Catch oEX As Exception
        Debug.WriteLine(oEX.Message)
    Finally
        Application.DoEvents()
    End Try
End Sub
```

As you can see, you can modify the number of dice shown by altering only the constant definition NUMDICE. For instance, I cranked it up to 1,000, and it was still much faster than my GDI+ experiments in the early days of designing the NineTiles game. The SetupDice method creates random locations in the horizontal range of 0 and the width of the screen, minus the width of the die frame, and the vertical range of 0 to the height of the screen, minus the height of a die frame.

CAUTION *DirectDraw doesn't effectively handle drawing "off the edges" of a surface, so you'll have do some math to keep from trying to draw at coordinates less than 0 or greater than the width of the destination surface.*

Finally, the bitmap data for the die *isn't* stored in the die class. When it's time to draw the die, the source surface data is passed into the class for drawing. You would obviously not create an identical surface instance for each die class—it would be random access memory (RAM) suicide to store the die frames bitmap in memory 250 times. The next example also uses this approach, where sprites with the same appearance look outside of themselves to get their sprite data.

Building an Arcade Game

With DirectDraw capabilities so easily within the grasp of the Visual Basic programmer, you'll now write an arcade game and get some sprites interacting on the screen. The arcade game is called *SpaceRocks*, and it involves a little spaceship floating around on the screen and shooting at some asteroids. (Sound familiar? Not to me, either.) Figure 8-2 shows a stirring game of SpaceRocks in action.

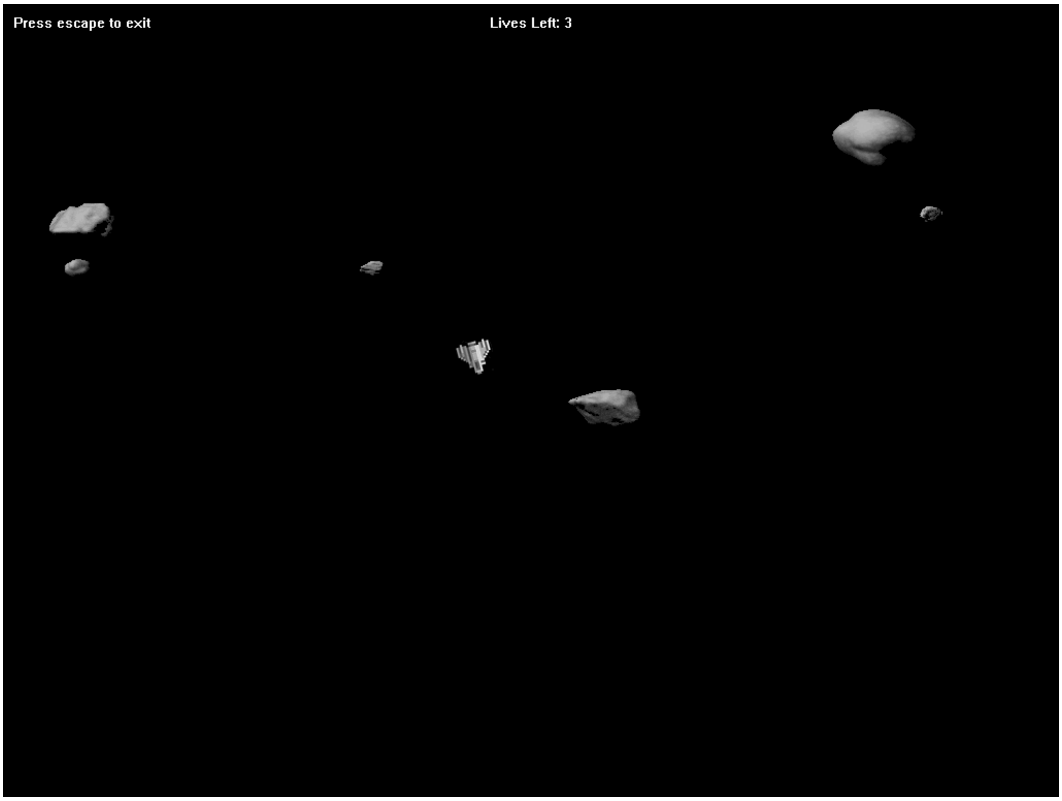

Figure 8-2. SpaceRocks, ahoy!

The structure of this game isn't unlike the structure of the DirectXDemo program previously described, but there's one further level of abstraction between this program and the last. In the previous program, much of the coding happened at the form level, such as the storage of the DirectDraw `Device` and `Surface` variables and the `Dice` object array. In SpaceRocks, a reusable class named `dxWorld` sets up the DirectDraw surface and device objects and handles the basic functions such as clearing the back buffer to black and flipping the back buffer to the front. Think of this as the generic game class; any future games you write will be subclasses of this class. Listing 8-6 shows portions of the `dxWorld` class (with some already-discussed elements removed).

Listing 8-6. The Ancestor Class for Future Games, `dxWorld`

```
Public MustInherit Class dxWorld

   Private FFrm As Form
   Private FNeedToRestore As Boolean = False

   Protected oRand As New Random
   Protected oDraw As Microsoft.DirectX.DirectDraw.Device
   Protected oFront As Microsoft.DirectX.DirectDraw.Surface
   Protected oBack As Microsoft.DirectX.DirectDraw.Surface
   Protected oJoystick As Microsoft.DirectX.DirectInput.Device

   Public Sub New(ByVal f As Form)
      MyBase.New()

      FFrm = f
      FFrm.Cursor.Dispose
      AddHandler FFrm.KeyDown, AddressOf FormKeyDown
      AddHandler FFrm.KeyUp, AddressOf FormKeyUp
      AddHandler FFrm.Disposed, AddressOf FormDispose

      InitializeDirectDraw()
      InitializeJoystick()
      InitializeWorld()

      Do While FFrm.Created
         DrawFrame()
      Loop
   End Sub

   Protected Overridable Sub FormDispose(ByVal sender As Object, _
     ByVal e As System.EventArgs)

      If Not (oJoystick Is Nothing) Then
         oJoystick.Unacquire()
      End If

   End Sub
```

```
    ReadOnly Property WorldRectangle() As Rectangle
        Get
            Return New Rectangle(0, 0, WID, HGT)
        End Get
    End Property

    'override for better keyboard handling
    Protected MustOverride Sub FormKeyDown(ByVal sender As Object, _
      ByVal e As System.Windows.Forms.KeyEventArgs)

    'override for better keyboard handling
    Protected Overridable Sub FormKeyUp(ByVal sender As Object, _
      <similar to prior discussion, removed>
    End Sub

    Private Sub InitializeDirectDraw()
      <similar to prior discussion, removed>
    End Sub

    'override to set up your world objects
    Protected MustOverride Sub InitializeWorld()

    'override when bitmaps have to be reloaded
    Protected Overridable Sub RestoreSurfaces()
        oDraw.RestoreAllSurfaces()
    End Sub

    Private Sub DrawFrame()
      <similar to prior discussion, removed>
    End Sub

    'override. put all your drawing in here.
    Protected Overridable Sub DrawWorldWithinFrame()
        Try
            oBack.ForeColor = Color.White
            oBack.DrawText(10, 10, "Press escape to exit", False)
        Catch oEX As Exception
            Debug.WriteLine(oEX.Message)
        End Try
    End Sub
End Class
```

The constructor for the `dxWorld` class takes a form as a parameter, and this form is dynamically assigned event handlers for its `KeyUp`, `KeyDown`, and `Dispose` events. The form used as the parameter for this class needs to have almost no code in it at all, except for the code that sets up an instance of this `dxWorld` class (actually, an instance of a descendant of the `dxWorld` class because `dxWorld` itself is declared `MustInherit`). As shown in Listing 8-7, creating an instance of this game on the form happens in four lines of code on an empty form.

Listing 8-7. Creating a New `dxWorld` *Instance*

```
Public Class fMain
    Inherits System.Windows.Forms.Form

    Dim FWorld As dxWorld.dxWorld

    Private Sub fMain_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load

        FWorld = New dxWorld.dxSpaceRocks(Me)
    End Sub
End Class
```

All of the important variables in the `dxWorld` class are declared as protected so that they'll be accessible in the descendant classes. This includes the `Surface` variables for the front and back surface and the DirectDraw `Device` object. There's also a `Random` object instance set up so that random numbers can be generated from anywhere inside the class or its descendants.

Setting Up a Joystick

You might also notice a variable named `oJoystick`, which is of type `Microsoft.DirectX.DirectInput.Device`. Yes, the new game class will be able to handle joystick input as well as keyboard input. Getting the joystick ready happens in the `InitializeJoystick` method on the `dxWorld` class, as shown in Listing 8-8.

Listing 8-8. The `InitializeJoystick` *Method*

```
Private Sub InitializeJoystick()

   Dim oInst As DeviceInstance
   Dim oDOInst As DeviceObjectInstance
```

```
    'get the first attached joystick
    For Each oInst In Manager.GetDevices( _
      DeviceClass.GameControl, EnumDevicesFlags.AttachedOnly)

        oJoystick = New Microsoft.DirectX._
          DirectInput.Device(oInst.InstanceGuid)
        Exit For
    Next

    If Not (oJoystick Is Nothing) Then

        oJoystick.SetDataFormat(DeviceDataFormat.Joystick)
        oJoystick.SetCooperativeLevel(FFrm, _
          Microsoft.DirectX.DirectInput. _
          CooperativeLevelFlags.Exclusive Or _
          Microsoft.DirectX.DirectInput.CooperativeLevelFlags.Foreground)

        ' Set the numeric range for each axis to +/- 256.
        For Each oDOInst In oJoystick.Objects
            If 0 <> (oDOInst.ObjectId And _
              CInt(DeviceObjectTypeFlags.Axis)) Then

                oJoystick.Properties.SetRange(ParameterHow.ById, _
                  oDOInst.ObjectId, New InputRange(-256, +256))
            End If
        Next
    End If
End Sub
```

InitializeJoystick retrieves the first game control device that it finds attached to the machine and then sets the range of all Axis objects within that joystick to have a range from –256 to +256. The standard game pad will have an x-axis and a y-axis; some three-dimensional controllers, such as SpaceBall, may have an x-axis, y-axis, and z-axis to be defined. Based on Listing 8-8, all axis objects associated with the joystick will be found and have their range set.

You'll see the code that shows how to poll the joystick for data and use it to update the game state later in the section "Setting Up the Ship Control Code." You first need to see how to set up the game elements themselves.

Creating the dxSprite Class

The base information to keep track of an object on the screen is stored in a class named dxSprite, as shown in Listing 8-9. This class is somewhat similar in structure to the SimpleDie class defined for the DirectXDemo project.

Listing 8-9. The `dxSprite` *Class Interface*

```
Public MustInherit Class dxSprite

    Public Event GetSurfaceData(ByVal sender As dxSprite, _
        ByRef oSource As Microsoft.DirectX.DirectDraw.Surface, _
        ByRef oRect As Rectangle)

    Property Location() As PointF
    Property Size() As Size
    Overridable Property Frame() As Integer
    ReadOnly Property BoundingBox() As Rectangle
    ReadOnly Property WorldBoundingBox() As Rectangle
    Property pShowBoundingBox() As Boolean
    ReadOnly Property Center() As PointF

    Public MustOverride Sub Move()
    Public Sub Draw(ByVal oSurf As Microsoft.DirectX.DirectDraw.Surface)
End Class
```

Much of the definition of this class is straightforward and doesn't require explanation. There are a few members, however, that do require a bit of clarification. The `GetSurfaceData` event is used as a callback so that the sprite class doesn't have to store surface (source bitmap) data directly. The reason you might not want to store surface data with the sprite was hinted at in the DirectXDemo application earlier. First, a game might contain dozens (hundreds?) of instances of the same sprite, and you certainly don't want to store multiple copies of the same bitmap data in each individual sprite instance.

Second, a single object may have several bitmaps to represent it depending on the state in which it might be. For the SpaceRocks game, for example, the ship object has three possible sprites: a ship with a fire trail, a ship without a trail, and an exploding ship for when it gets hit by a rock.

Using an event to retrieve the proper sprite based on the state of the object in question helps to decouple the sprite class from the game class. Note that there's nothing directly relevant to an arcade space-shooting-rock-type game in the class definition shown in Listing 8-9. The goal is to keep the `dxSprite` class generic enough to reuse in different projects (but you'll be creating SpaceRocks-specific subclasses for this game).

The `Draw` method of the sprite class is also nonstandard. As mentioned earlier in the chapter, DirectDraw doesn't take kindly to copying surfaces off the edge of the destination surface. The copy fails miserably, in fact, and crashes the program. Even if this crash is handled gracefully with a structure exception handler, the sprite “winks” out of existence as it reaches the edge of the destination surface instead of smoothly scrolling off the screen.

You must do some nasty rectangle manipulation to fix this problem. If you want to draw a ship partially off the left side of the screen, for example, then the program has to clip off the left side of the sprite and draw only the right portion of the rectangle on the left side. If the ship sprite is moving left, then each frame will clip more and more of the left side of the ship until it disappears. Figure 8-3 shows a sprite off the left side of the screen. The gray area is the area to be clipped.

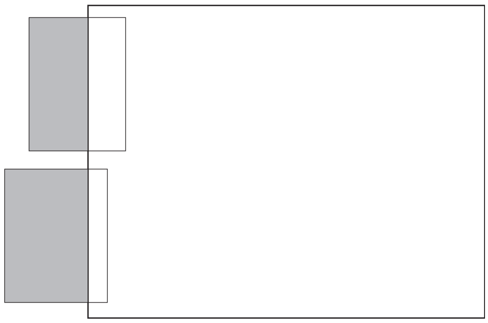

Figure 8-3. Two sprites partially off the left side of the screen. The gray area must be clipped, and only the white area should be drawn.

The nasty clipping math must also adjust the *bounding boxes* of each sprite. The bounding box represents a rectangle that surrounds the sprite and helps to test for collision between two sprites that might hit each other (the ship with a rock, for example). There are two representations of the bounding box stored for each sprite. One is declared in *sprite coordinates*, meaning that the upper-left corner in this bounding box is usually 0, 0. The second bounding box representation is stored in *world coordinates*, meaning that the upper-left corner is usually the same as the sprite’s `Location` property (the location on the screen). Listing 8-10 shows a portion of the `Draw` method.

Listing 8-10. A Portion of the `Draw` *Method*

```
Public Sub Draw(ByVal oSurf As Microsoft.DirectX.DirectDraw.Surface)

    Dim oSource As Microsoft.DirectX.DirectDraw.Surface
    Dim oRect As Rectangle
    Dim oPt As Point
    Dim iDiff As Integer
```

```
RaiseEvent GetSurfaceData(Me, oSource, oRect)

If oSource Is Nothing Then
   Exit Sub
Else
   Try
      FWBB = Me.BoundingBox         'start w/ normal bbox

      'start at the location
      oPt = New Point(System.Math.Floor(Location.X), _
         System.Math.Floor(Location.Y))

      If oPt.X < 0 Then
         'draw partial on left side
         oRect = New Rectangle(oRect.Left - oPt.X, oRect.Top, _
           oRect.Width + oPt.X, oRect.Height)

         If oPt.X + FWBB.Left < 0 Then

            FWBB = New Rectangle(0, FWBB.Top, _
            FWBB.Width + (oPt.X + FWBB.Left), FWBB.Height)

         Else

            FWBB = New Rectangle(FWBB.Left + oPt.X, FWBB.Top, _
              FWBB.Width, FWBB.Height)

         End If
         oPt.X = 0
      End If

      <lots of other rectangle-clipping code removed>

     'should never happen, just in case
      If oRect.Width <= 0 Or oRect.Height <= 0 Then Return

      'offset the bounding box by the world coordinates
      FWBB.Offset(oPt.X, oPt.Y)

      'draw the sprite
      oSurf.DrawFast(oPt.X, oPt.Y, oSource, oRect, _
       DrawFastFlags.DoNotWait Or DrawFastFlags.SourceColorKey)
```

```
            'draw the bounding box
            If Me.pShowBoundingBox Then
               oSurf.ForeColor = Color.Red
               oSurf.DrawBox(FWBB.Left, FWBB.Top, FWBB.Right, FWBB.Bottom)
            End If

         Catch oEx As Exception
            Debug.WriteLine("-------------------------------------")
            Debug.WriteLine(oEx.Message)
         End Try
      End If

   End Sub
```

Creating the dxSpaceRocks Class

The SpaceRocks game is implemented in the class named `dxSpaceRocks`, which is a descendant of the `dxWorld` class. This class contains the classes that store all of the game objects, including the ship, the rocks, and any bullets currently flying around. The rocks and bullets are stored in a different way because there can be multiple instances of these classes in the game at one time. The player's ship is always a lone instance, so the class that contains the ship information has a much different structure.

Setting Up the Game Class

Listing 8-11 shows the declaration of the game class and the instantiation of the private variables that track all the game objects.

Listing 8-11. The `dxSpaceRocks` *Class with the Game Object Class Definition and Initialization Code*

```
Public Class dxSpaceRocks
   Inherits dxWorld

   Private FShip As dxShipSprite
   Private FRocks As dxRockCollection
   Private FBullets As dxBulletCollection

   Protected Overrides Sub InitializeWorld()

      Dim oRand As New Random
```

```
        FShip = New dxShipSprite
        FShip.Location = New PointF(100, 100)
        FShip.Size = New Size(96, 96)
        FShip.pShowBoundingBox = False

        FRocks = New dxRockCollection
        FRocks.pShowBoundingBox = False

        FBullets = New dxBulletCollection
        FBullets.pShowBoundingBox = False

    End Sub

    Protected Overrides Sub RestoreSurfaces()
        MyBase.RestoreSurfaces()

        FShip.RestoreSurfaces(oDraw)
        FRocks.RestoreSurfaces(oDraw)
        FBullets.RestoreSurfaces(oDraw)
    End Sub

    <code removed>

End Class
```

The rock and bullet storage classes are collections, and their class names refer to them as such. The ship class, however, is a direct descendant of the `dxSprite` class, so its initialization is a bit different from the other two.

The procedure `RestoreSurfaces`, if you'll recall, is called when bitmap surface objects have to be re-created. Because the game class itself isn't storing any source surface objects, each game class has its own `RestoreSurfaces` method, and this method is called from the game's method of the same name. This procedure was originally declared as protected and `Overrideable` in the base `dxWorld` class, which gives you the ability to access it and override it in the subclass.

Setting Up the Game Class Drawing and Movement

Drawing for all descendants of the `dxWorld` class happens by overriding the protected method `DrawWorldWithinFrame`. Listing 8-12 shows that method.

Listing 8-12. The `DrawWorldWithinFrame` *Method*

```
Protected Overrides Sub DrawWorldWithinFrame()

   Dim p As New Point((WID / 2) - 40, 10)

   MyBase.DrawWorldWithinFrame()

   'joysticks don't generate events, so we update the ship
   'based on joystick state each turn
   UpdateShipState()

   FShip.Move()
   FRocks.Move()
   FBullets.Move()

   FBullets.Draw(oBack)
   FShip.Draw(oBack)
   FRocks.Draw(oBack)

   FBullets.BreakRocks(FRocks)

   oBack.ForeColor = Color.White
   Select Case FShip.Status
      Case ShipStatus.ssAlive
         oBack.DrawText(p.X, p.Y, "Lives Left: " & _
           FShip.LivesLeft, False)
           If FRocks.CollidingWith(FShip.WorldBoundingBox, _
             bBreakRock:=False) Then
              FShip.KillMe()
           End If

      Case ShipStatus.ssDying
         oBack.DrawText(p.X, p.Y, "Oops.", False)

      Case ShipStatus.ssDead
         If FShip.LivesLeft = 0 Then
            oBack.DrawText(p.X, p.Y, "Game Over", False)
         Else
            oBack.DrawText(p.X, p.Y, _
               "Hit SpaceBar to make ship appear " + _
                 "in middle of screen", False)
         End If
      End Select

  End Sub
```

The DrawWorldWithinFrame method runs once per every "clock tick" of the game engine. It controls both object movement and object drawing. At the start of the method is a call to a procedure named UpdateShipState. This procedure (described next) changes the state of the ship based on what joystick buttons are being pressed. Then, the program calls a Move method on the ship class and the rock and bullet collections. The Move method updates the position of every game object based on its current location, the direction it's traveling, and the speed at which it's traveling.

Once all the game objects have been moved, the Draw method of the three game class objects is called, passing in the variable that holds the back buffer DirectDraw surface. You've already seen the Draw method for the dxSprite class (with all the rectangle clipping logic), and the Draw method on the collection classes simply calls the Draw method for each dxSprite in their respective collections.

The remainder of the DrawWorldWithinFrame method handles the drawing of a text message at the top of the screen based on the current state of the player's ship. The game will report the number of lives the player has left, report a simple *Oops* as the ship explodes because of collision with a rock, give instructions on how to make the ship reappear if the user has lives left, or report *Game Over* if no lives remain. One other task is handled within this Case statement, and that's the collision check between the ship and the rocks (the CollidingWith method on the FRocks variable).

Setting Up the Ship Control Code

The remainder of the dxSpaceRocks class handles ship movement via keyboard or joystick. Listing 8-13 shows this code.

Listing 8-13. Ship Movement Code for the dxSpaceRocks *Class*

```
Public Class dxSpaceRocks
   Inherits dxWorld

   Private FLeftPressed As Boolean = False
   Private FRightPressed As Boolean = False
   Private FUpPressed As Boolean = False
   Private FSpacePressed As Boolean = False

   <some code removed>

   Protected Overrides Sub FormKeyDown(ByVal sender As Object, _
     ByVal e As System.Windows.Forms.KeyEventArgs)
```

```
        Select Case e.KeyCode
           Case Keys.Left
              FLeftPressed = True
           Case Keys.Right
              FRightPressed = True
           Case Keys.Up
              FUpPressed = True
           Case Keys.Space
              FSpacePressed = True
           Case Keys.B
              FShip.pShowBoundingBox = Not FShip.pShowBoundingBox
              FRocks.pShowBoundingBox = Not FRocks.pShowBoundingBox
              FBullets.pShowBoundingBox = Not FBullets.pShowBoundingBox
        End Select
    End Sub

    Protected Overrides Sub FormKeyUp(ByVal sender As Object, _
      ByVal e As System.Windows.Forms.KeyEventArgs)

        MyBase.FormKeyUp(sender, e)

        Select Case e.KeyCode
           Case Keys.Left
              FLeftPressed = False
           Case Keys.Right
              FRightPressed = False
           Case Keys.Up
              FUpPressed = False
           End Select
    End Sub

    Private Sub UpdateShipState()

       Dim oState As New JoystickState
       Dim bButtons As Byte()
       Dim b As Byte

       Dim p As PointF

       If Not oJoystick Is Nothing Then

          Try
             oJoystick.Poll()
```

```
  Catch oEX As InputException
     If TypeOf oEX Is NotAcquiredException Or _
       TypeOf oEX Is InputLostException Then

        Try
           ' Acquire the device.
           oJoystick.Acquire()
        Catch
           Exit Sub
        End Try

     End If
   End Try

   Try
      oState = oJoystick.CurrentJoystickState
   Catch
      Exit Sub
   End Try

   'ship is turning if x axis movement
   FShip.IsTurningRight = (oState.X > 100) Or FRightPressed
   FShip.IsTurningLeft = (oState.X < -100) Or FLeftPressed
   FShip.ThrustersOn = (oState.Y < -100) Or FUpPressed

   'any button pushed on the joystick will work
   bButtons = oState.GetButtons()
   For Each b In bButtons
      If (b And &H80) <> 0 Then
         FSpacePressed = True
         Exit For
      End If
   Next

Else
   FShip.IsTurningRight = FRightPressed
   FShip.IsTurningLeft = FLeftPressed
   FShip.ThrustersOn = FUpPressed
End If
```

```
            If FSpacePressed Then
                Select Case FShip.Status
                    Case ShipStatus.ssDead
                        'center screen
                        FShip.BringMeToLife(WID \ 2 - FShip.Size.Width \ 2, _
                                            HGT \ 2 - FShip.Size.Height \ 2)
                    Case ShipStatus.ssAlive
                        p = FShip.Center
                        p.X = p.X - 16
                        p.Y = p.Y - 16

                        FBullets.Shoot(p, FShip.Angle)
                    End Select
                    FSpacePressed = False
            End If
        End Sub
    End Class
```

Keyboard state is stored in Boolean variables named FLeftPressed, FRightPressed, FUpPressed, and FSpacePressed. These variables are set to True in the KeyDown event and to False in the KeyUp event (if the appropriate key is indeed being pressed, that is). By storing the variables in this way, the game allows for object movement as long as the correct key is down. For example, once a user presses the up arrow, the ship should have its thrusters on until the key is released. The Boolean FUpPressed will stay True as long as the arrow is down.

The B key is the last key that affects the game—it turns the bounding boxes on and off for debugging purposes.

NOTE *This was especially useful to me as I slowly coded the "sprite-half-off-the-screen" code in the* dxSprite*'s* Draw *method (see Listing 8-10 to relive the pain).*

The function UpdateShipState, called once per drawing frame, polls the joystick and keyboard Boolean variables for their states and updates the state of the ship accordingly. For example, if the joystick's x-axis has a value that's greater than 100, then the ship is turning clockwise. A move in the negative y direction on the joystick is the cue to turn on the thrusters. Pressing Button 1 on the joystick (or pressing the spacebar) either fires a bullet or brings a dead ship back to life.

Setting Up the Ship Class

The `dxShipSprite` class is a descendant of the `dxSprite` class discussed earlier. This class controls the player's ship as it cruises around on the screen. There are three graphics required for the ship—one for the ship with thrusters off, one with thrusters on, and one for an explosion sequence for when the ship is biting the dust. Figure 8-4 shows one frame of each of the bitmaps.

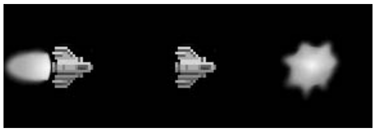

Figure 8-4. The first frame of the each of the three ship graphics

Drawing the Ship

The two ship graphics consist of 24 frames. Each frame represents a different rotation of the ship in a circle. There are 15 degrees of rotation between each frame (360 degrees / 24 frames = 15 degrees per frame). The explosion sequence is only six frames and was designed by hand (and not very well; bear in mind that I don't consider computer graphics design among my talents).

Drawing the correct graphic at the correct time is a function of what state the ship is in at the moment. There's an enumerated type declared called `ShipStatus` that defines whether the ship is currently okay, in the middle of exploding, or dead and gone. If the ship is gone, then the program obviously doesn't have to draw it at all. If the ship is in the middle of exploding, then the explosion graphic is chosen for display. If the ship is okay, then one of the two ship graphics are displayed, either with or without the thruster fire. The ship control code in Listing 8-13 hinted at the fact that the ship sprite has a property named `ThrustersOn`, and this property determines which of the two ship bitmaps to draw. Listing 8-14 shows the portion of the `dxShipSprite` class that loads the three bitmaps into DirectDraw `Surface` variables and the code that selects the correct surface to draw in a given frame.

Listing 8-14. Ship Sprite State and Graphics-Related Code

```
Public Enum ShipStatus
   ssAlive = 0
   ssDying = 1
   ssDead = 2
End Enum

Public Class dxShipSprite
   Inherits dxSprite

   Private FShipSurfaceOff As Microsoft.DirectX.DirectDraw.Surface
   Private FShipSurfaceOn As Microsoft.DirectX.DirectDraw.Surface
   Private FShipSurfaceBoom As Microsoft.DirectX.DirectDraw.Surface

   Public Sub New()
      MyBase.new()
      AddHandler Me.GetSurfaceData, AddressOf GetShipSurfaceData
   End Sub

   Private FStatus As ShipStatus
   ReadOnly Property Status() As ShipStatus
      Get
         Return FStatus
      End Get
   End Property

   'we can keep surfaces in the ship
   'sprite class because there's only one of them
   Public Sub RestoreSurfaces(ByVal oDraw As _
     Microsoft.DirectX.DirectDraw.Device)

      Dim oCK As New ColorKey

      Dim a As Reflection.Assembly = _
        System.Reflection.Assembly.GetExecutingAssembly()

      If Not FShipSurfaceOff Is Nothing Then
        FShipSurfaceOff.Dispose()
        FShipSurfaceOff = Nothing
      End If

      FShipSurfaceOff = New Surface(a.GetManifestResourceStream( _
        "SpaceRocks.ShipNoFire.bmp"), New SurfaceDescription, oDraw)
      FShipSurfaceOff.SetColorKey(ColorKeyFlags.SourceDraw, oCK)
```

```
    If Not FShipSurfaceOn Is Nothing Then
        FShipSurfaceOn.Dispose()
        FShipSurfaceOn = Nothing
    End If

    FShipSurfaceOn = New Surface(a.GetManifestResourceStream( _
      "SpaceRocks.ShipFire.bmp"), New SurfaceDescription, oDraw)
    FShipSurfaceOn.SetColorKey(ColorKeyFlags.SourceDraw, oCK)

    If Not FShipSurfaceBoom Is Nothing Then
        FShipSurfaceBoom.Dispose()
        FShipSurfaceBoom = Nothing
    End If

    FShipSurfaceBoom = New Surface(a.GetManifestResourceStream( _
      "SpaceRocks.Boom.bmp"), New SurfaceDescription, oDraw)
    FShipSurfaceBoom.SetColorKey(ColorKeyFlags.SourceDraw, oCK)

End Sub

Private Sub GetShipSurfaceData(ByVal aSprite As dxSprite, _
  ByRef oSurf As Surface, ByRef oRect As Rectangle)

    Dim aShip As dxShipSprite = CType(aSprite, dxShipSprite)

    Select Case aShip.Status
        Case ShipStatus.ssDead
            oSurf = Nothing

        Case ShipStatus.ssDying
            oSurf = FShipSurfaceBoom

        Case ShipStatus.ssAlive

            If aShip.ThrustersOn AndAlso _
              oRand.Next(0, Integer.MaxValue) Mod 10 <> 0 Then

                oSurf = FShipSurfaceOn
            Else
                oSurf = FShipSurfaceOff
            End If

    End Select
```

```
        oRect = New Rectangle((aShip.Frame Mod 6) * 96, _
          (aShip.Frame \ 6) * 96, 96, 96)

    End Sub
End Class
```

The RestoreSurfaces code is similar to what you saw in the DirectXDemo application, except that there are three surfaces to load instead of one. The routine GetShipSurfaceData is special because it serves as the event handler for the GetSurfaceData event for this object. If you'll recall, the GetSurfaceData event is raised from within the Draw method of the dxSprite class (see Listing 8-10 if you need a reminder). When the Draw method is ready to draw, it raises this event and expects the event handler to pass back the correct source Surface object that needs to be drawn, as well as a Rectangle object that indicates which portion of the source bitmap to draw. The routine GetShipSurfaceData does all of that work for the ship class. Based on the state of the ship and whether its thrusters are on or off, the appropriate bitmap is returned. The last line constructs a source rectangle based on the value of the Frame property, based on the knowledge that all of the ship graphics are 96-pixels wide and high.

NOTE *The game uses one additional trick when selecting a bitmap to display. Ten percent of the time, the* GetShipSurfaceData *routine returns the ship graphic without the thruster fire, even when thrusters are on. This gives the fire a little "flicker" effect.*

Moving the Ship

The ship's current location is stored in the Location property defined on the ancestor dxSprite class. The trick is figuring out how to move the location based on the current angle of the ship, whether the thrusters are currently on, and how long they've been on.

Properties control the velocity of the ship, which is how many pixels it moves per frame in both the x and y directions, and its acceleration, which controls how fast the velocity is increasing.

Listing 8-15 lists the Move method of the ship class, which is called once during every frame by the dxSpaceRocks game class.

Listing 8-15. The `Move` *Method of* `dxShipSprite`

```
Public Overrides Sub Move()

   Dim dx, dy As Single

   'we're only moving every x frames
   FSkipFrame = (FSkipFrame + 1) Mod 1000
   If FSkipFrame Mod 3 = 0 Then

      Select Case Me.Status
         Case ShipStatus.ssAlive
            Turn()

            If ThrustersOn Then
               Acceleration += 1

               dy = -Math.Sin(FAngle * Math.PI / 180) * Acceleration
               dx = Math.Cos(FAngle * Math.PI / 180) * Acceleration

               Velocity = New PointF(Velocity.X + dx, Velocity.Y + dy)
            Else
               Acceleration = 0
            End If

         Case ShipStatus.ssDying
            Frame += 1

            Velocity = New PointF(0, 0)
            Acceleration = 0

            'we're done drawing the boom
            If Frame >= 6 Then
               FStatus = ShipStatus.ssDead
            End If

         Case ShipStatus.ssDead
            'nothing
      End Select

   End If

   Location = New PointF(Location.X + Velocity.X, Location.Y + Velocity.Y)

End Sub
```

Note that there's a "governor" of sorts on the `Move` class in the form of an integer variable named `FSkipFrame`. This variable updates in every execution of the `Move` method, but it allows actual velocity and acceleration to change in every third execution. Without this governor, the ship's controls are far too touchy and hard to control.

The `Acceleration` property is an integer that keeps increasing as long as the ship's thrusters are turned on. (Actually, there's maximum acceleration defined in the property, so it does max out eventually.) The `Acceleration` variable, along with the current angle the ship is facing and some basic trigonometry, help determine the speed of the ship during this turn in both the x and y directions. This speed is stored in the `Velocity` property.

At the bottom of the `Move` method, the calculated velocity is added to the current location, which yields the new location of the ship.

Setting Up Rocks and Rock Collections

The rocks are simpler structures than the ship because they move at a constant speed and in a constant direction, and they aren't (directly) affected by the game player's control. This simplicity is counteracted by the fact that the game has to keep track of an undetermined number of them, however. Thus, a "manager" class keeps track of each rock.

The (rather cool) rock graphics themselves were created courtesy of POV-RAY models from Scott Hudson. The models represent digital representations of actual "potential earth-crossing" asteroids. Please visit the Web site `http://www.eecs.wsu.edu/~hudson/Research/Asteroids` for further information.

NOTE *You can find information on POV-RAY and raytracing in Appendix B, "Using POV-RAY and Moray."*

Creating the Rock Class

The rock class itself keeps track of the size of the rock (there are three possible sizes), the direction it's moving, which of the two graphics to use, which direction it's spinning, and how fast it's spinning. Listing 8-16 shows the public interface for this class.

Listing 8-16. The `dxRockSprite` *Class and Enumerated Type for Determining Rock Size*

```
Public Enum dxRockSize
   rsLarge = 0
   rsMed = 1
   rsSmall = 2
End Enum

Public Class dxRockSprite
   Inherits dxSprite

   Public Event RockBroken(ByVal aRock As dxRockSprite)
   Property pAlternateModel() As Boolean
   Property pSpinReverse() As Boolean
   Property pRockSize() As dxRockSize
   Property pRotSpeed() As Integer
   Property Velocity() As PointF
   Public Overrides Sub Move()
   Public Sub Break()

End Class
```

Details of this class are mostly trivial and unworthy of you (who by this time is a nearly expert game programmer). The `pRockSize` property is mildly interesting in that the bounding box of the rock is different depending on the size of the rock.

Creating the Rock Collection Class

The `dxRockCollection` class is much more interesting than the rock class. This class keeps track of the six different DirectDraw `Surface` objects that store the rock graphics (two rock shapes in three sizes each). It also keeps the pointers to each individual rock class and handles all of the interaction between the game and the rocks (you can think of this class as a sort of "rock broker"). To that end, several methods on the collection class simply perform functionality upon each rock in the collection. The `Draw` method is one such method, shown in Listing 8-17, which merely calls the like-named method on each object in the collection.

Listing 8-17. The `Draw` *Method (and Some Others)*

```
Public Sub Draw(ByVal oSurf As Microsoft.DirectX.DirectDraw.Surface)

   Dim aRock As dxRockSprite

   For Each aRock In FRocks
      aRock.Draw(oSurf)
   Next

End Sub
```

Another interesting piece of functionality in the rock collection is the pair of overloaded `AddRock` methods, shown in Listing 8-18. These methods add a new rock to the collection. It also includes the code that runs when a rock is shot and split in two.

Listing 8-18. Adding a New Rock to the Game in One of Two Ways

```
Private Overloads Function AddRock()

   Dim oPt As PointF
   'start location along the edges

   Select Case FRand.Next(0, Integer.MaxValue) Mod 4
      Case 0
         oPt = New PointF(0, FRand.Next(0, Integer.MaxValue) Mod HGT)
      Case 1
         oPt = New PointF(WID, FRand.Next(0, Integer.MaxValue) Mod HGT)
      Case 2
         oPt = New PointF(FRand.Next(0, Integer.MaxValue) Mod WID, 0)
      Case 3
         oPt = New PointF(FRand.Next(0, Integer.MaxValue) Mod WID, HGT)
   End Select

   Return AddRock(dxRockSize.rsLarge, oPt)
End Function

Private Overloads Function AddRock(ByVal pSize As dxRockSize, _
   ByVal p As PointF) As dxRockSprite

   Dim aRock As dxRockSprite

 aRock = New dxRockSprite
   With aRock
```

```
    .pShowBoundingBox = Me.pShowBoundingBox
    .pAlternateModel = FRand.Next(0, Integer.MaxValue) Mod 2 = 0
    .pSpinReverse = FRand.Next(0, Integer.MaxValue) Mod 2 = 0
    .pRotSpeed = FRand.Next(0, Integer.MaxValue) Mod 3
    .pRockSize = pSize
    Select Case pSize
       Case dxRockSize.rsLarge
          .Size = New Size(96, 96)
        Case dxRockSize.rsMed
           .Size = New Size(64, 64)
        Case dxRockSize.rsSmall
           .Size = New Size(32, 32)
    End Select

    .Location = p

    Do  'no straight up/down or left/right
       .Velocity = New PointF(FRand.Next(-3, 3), FRand.Next(-3, 3))
    Loop Until .Velocity.X <> 0 And .Velocity.Y <> 0

    .Move() 'the first move makes sure they're off the edge

    AddHandler .GetSurfaceData, AddressOf GetRockSurfaceData
      AddHandler .RockBroken, AddressOf RockBroken
 End With
    FRocks.Add(aRock)
End Function

Private Sub RockBroken(ByVal aRock As dxRockSprite)

    Select Case aRock.pRockSize
     Case dxRockSize.rsLarge
        AddRock(dxRockSize.rsMed, aRock.Location)
        AddRock(dxRockSize.rsMed, aRock.Location)

     Case dxRockSize.rsMed
        AddRock(dxRockSize.rsSmall, aRock.Location)
        AddRock(dxRockSize.rsSmall, aRock.Location)

     Case dxRockSize.rsSmall
        'nothing
 End Select
 FRocks.Remove(aRock)
End Sub
```

The first AddRock function is the one that's used when a new, large size rock is to be added to the game. It takes no parameters. Its job is to select a random point along one of the four edges of the screen, and then it calls the second AddRock method, passing along the size of the new rock (always large) and the location it has selected.

The second AddRock method actually creates the new instance of the dxRockSprite class, sets up all of its properties, and then adds it to the ArrayList that holds all of the rock objects. This second AddRock method is used when a rock is shot and splits into two smaller pieces. You can see this code in the RockBroken routine, which serves as the event handler for the rock class event of the same name. When a large rock is broken, two medium-sized rocks are spawned at the same location of the large rock, and then the large rock is removed from the ArrayList named FRocks (and thus from the game). When a medium rock is broken, two smaller rocks are spawned in the same location, and the medium rock is removed from the ArrayList.

The last interesting function in the rock collection class is the CollidingWith function, which determines if an outside agent has crashed into a rock and whether that rock should break as a result (see Listing 8-19).

Listing 8-19. The CollidingWith *Function*

```
Public Function CollidingWith(ByVal aRect As Rectangle, _
   ByVal bBreakRock As Boolean) As Boolean

   Dim aRock As dxRockSprite

   For Each aRock In FRocks
      If aRock.WorldBoundingBox.IntersectsWith(aRect) Then
         If bBreakRock Then
            aRock.Break()
         End If
         Return True
      End If
   Next

   Return False
End Function
```

The collision code in the game relies on the bounding boxes of all of the game objects (ship, rocks, and bullets). The bounding boxes are all represented by .NET Framework Rectangle objects. One of the most useful methods built into the Rectangle class is the IntersectsWith class, which returns True if the current rectangle overlaps another passed-in rectangle parameter. The function shown in Listing 8-19 checks to see if the bounding box for each rock in the collection intersects with the rectangle that's passed into the function. If it finds an intersection, the function returns `True` and the rock involved in the collision either breaks or doesn't break, depending on the value of the `bBreakRock` parameter (collisions with bullets break the rock, and a collision with the ship leaves the rock intact).

Setting Up Bullets and Bullet Collections

Keeping with the pattern of discussing things in decreasing order of complexity, the bullet class is the simplest of the three major game elements. The bullet has only one graphic (with only a single frame) and can move in a single direction at a fixed speed. Like the rocks class, a "manager" class keeps track of multiple bullets on the screen.

Creating the Bullet Class

The bullet class is simple and short enough to list here in its entirety, as shown in Listing 8-20.

Listing 8-20. The Bullet Sprite Class

```
Public Class dxBulletSprite
   Inherits dxSprite

   Private FFrameAliveCount As Integer

   Sub New()
      MyBase.New()
      FBoundingBox = New Rectangle(10, 10, 12, 12)
   End Sub

   Private FVelocity As PointF
   Property Velocity() As PointF
      Get
         Return FVelocity
      End Get
      Set(ByVal Value As PointF)
         FVelocity = Value
      End Set
   End Property
```

```
    Public Overrides Sub Move()
        Location = New PointF(Location.X + Velocity.X, _
          Location.Y + Velocity.Y)
        FFrameAliveCount += 1
    End Sub

    ReadOnly Property pFrameAliveCount() As Integer
        Get
            Return FFrameAliveCount
        End Get
    End Property
End Class
```

The bullet class keeps track of velocity and a property known as `FrameAliveCount`. This property determines when a bullet has traveled far enough and should be removed from the screen. The `Move` method is extremely simple. It changes the location of the sprite by the value of the `Velocity` property in both the x and y directions.

Creating the Bullet Collection Class

The collection class that keeps track of multiple bullets shares many features with the rock collection class already discussed. It uses an `ArrayList` to store multiple instances of the `dxBulltetSprite` class. Listing 8-21 shows the `Shoot` method, which brings a new instance of the bullet class into the world.

Listing 8-21. The `Shoot` *Method*

```
Public Sub Shoot(ByVal p As PointF, ByVal iAngle As Integer)

    If FBullets.Count >= 4 Then Exit Sub

    Dim dx, dy As Single
    Dim aBullet As dxBulletSprite

    aBullet = New dxBulletSprite
    With aBullet
        .pShowBoundingBox = Me.pShowBoundingBox
        .Location = p
```

```
      dy = -Math.Sin(iAngle * Math.PI / 180) * 6
      dx = Math.Cos(iAngle * Math.PI / 180) * 6

      .Velocity = New PointF(dx, dy)
      .Move()

      AddHandler .GetSurfaceData, AddressOf GetBulletSurfaceData
   End With
   FBullets.Add(aBullet)
End Sub
```

The `Shoot` method first checks that there are fewer than four bullets already floating around in space. If four bullets are already on the screen, then the method returns without firing. If this check succeeds, though, then a new `dxBulletSprite` object is instantiated, properties are set (including the `Velocity` property, calculated from the angle parameter pass into the function), and the bullet is added to the `ArrayList`.

The method `BreakRocks`, shown in Listing 8-22, is called once in each drawing loop to see if the bullet has found its target.

Listing 8-22. The Method `BreakRocks`

```
Public Sub BreakRocks(ByVal FRocks As dxRockCollection)

   Dim aBullet As dxBulletSprite
   Dim i As Integer

   'check each bullet to see if it hits a rock
   'have to use a loop so you don't skip over when deleting
   i = 0
   Do While i < FBullets.Count
      aBullet = FBullets.Item(i)
      If FRocks.CollidingWith(aBullet.WorldBoundingBox, _
         bBreakRock:=True) Then

         FBullets.Remove(aBullet)
      Else
         i = i + 1
      End If
   Loop
End Sub
```

The method `BreakRocks` uses the `CollidingWith` function discussed in Listing 8-19 to determine if any of the bullets in this collection have collided with any rock in the game. A slightly tricky loop is employed in this method that requires some explanation. Whenever a collection is iterated and the possibility exists that elements in the collection will be removed during that iteration, then the program should never use the standard `For..Each` method to iterate, or the result is that items in the collection will be skipped. Instead, you should use a loop such as the one shown in Listing 8-22. This loop uses an integer counter to keep track of the place in the iteration. The trick is that if an element in the collection is deleted (in this case, a bullet), then the loop counter *isn't incremented*. Say the loop is an element 5 in a collection of 10, and this element is removed from the collection. After the removal, all of the elements after element 5 have "slid down" one place in the order, meaning the former element 6 is now element 5. By not incrementing the counter after a delete, the next iteration of the loop makes sure to check that next element.

Summary

Whew! What you may have thought was a reasonably simple game ended up being a complicated set of classes with some complex relationships. The result of this code, however, is a decent set of reusable classes for creating DirectDraw games. A "world" class encapsulates much of the DirectDraw setup code and surfaces for page flipping, a "sprite" class abstracts an on-screen object (which is generic enough for many uses because it doesn't attempt to store surface data within itself, instead employing an event to retrieve surface data from an outside source), and several examples of "manager" classes control several instances of similar game elements. You should be able to use this example and become the next Atari.