.NET Game Programming with DirectX 9.0

ALEXANDRE SANTOS LOBÃO
AND
ELLEN HATTON

Apress™

.NET Game Programming with DirectX 9.0

Copyright ©2003 by Alexandre Santos Lobão and Ellen Hatton

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-051-1

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: David Jung

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Project Manager: Sofia Marchant

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Compositor: Diana Van Winkle, Van Winkle Design Group

Artist and Cover Designer: Kurt Krames

Indexer: Lynn Armstrong

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 9th 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 information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

CHAPTER 4

River Pla.Net: Tiled Game Fields, Scrolling, and DirectAudio

In this chapter we'll apply the concepts learned in the previous chapter about Direct3D to implement DirectX gaming classes (such as `GameEngine` and `Sprite`), so we'll easily be able to create high-speed graphics games. We'll also introduce basic DirectAudio concepts that will allow us to include sound effects and background music in our games.

Figure 4-1. River Pla.Net, a River Raid clone, is this chapter's sample game

We'll also examine the concept of tiled game fields and scrolling in games, and start implementing a clone of Activision's River Raid game, a popular title for Atari 2600 and VCS. Our sample game, shown in Figure 4-1, will be finished in the next chapter, where we'll introduce DirectInput and the use of force-feedback joysticks.

Scrolling games and tile-based games have been around since earlier video game consoles and home computers hit the shelves, and we often see games that use both techniques. We'll discuss some interesting points about each in the next sections.

Scrolling Games

Although the basic concept of scrolling games is very simple, there are many interesting variations we must consider when we start creating a new game. We can define scrolling games as the games in which the background moves in a continuous way. It's a very loose definition, but it'll suffice for our goals here.

Some of the typical choices we must make when coding scrolling games are discussed next.

Scrolling Direction

All scrolling games are either *vertical scrollers, horizontal scrollers,* or *full scrollers,* meaning that the background on these games scroll in a vertical direction, in a horizontal direction, or in any direction. We'll discuss some variations of these movements in this section.

The most common choice is to implement vertical "up-down" scrollers (as does the sample game for this chapter), where the background moves from the top to the bottom of the screen, and horizontal "right-left" scrollers, where the background moves from right to left. We don't see many scrolling games using the opposite direction schemes because using these directions makes our games seem more natural to players.

Full scrollers are harder to implement and to play, but when made correctly, they can lead to very interesting gameplay. Just imagine a game in which players can move their character in any direction: This might be an interesting feature, but the player could become disorientated, and the game objective would be less clear.

Parallax Scrolling

Parallax scrolling is an ingenious trick that gives players the feeling of being in a 3-D environment, even with flat images.

The basic idea is to create different layers of background objects, each one moving at different speeds. For example, if we are controlling a monkey in a jungle, we can create some bushes and trees that scroll at the same speed as the terrain, trees a little farther off that move a little slower, distant mountains that move very slowly, and maybe a fixed moon in the sky.

This approach creates a more lifelike game, but must be used with care because it can lead to visual clutter and confusion for the player. A good tip is to make distant objects with less vivid colors. This adds to the ambience without distracting the player.

Player or Engine-Controlled Scrolling

When coding the scrolling for our game, we need to decide whether the background will always be moving (except, perhaps, when facing some end-of-level bosses), if it will move depending solely on the player's input, or if the movement will be a combination of both.

In some scrolling games, the player is always in the same position on the screen (usually the middle), and the background rolls according to the player's movement: When a player moves the joystick to the right, his or her character walks to the right (moving in a fixed position), while the background moves to the left. Many race games use this approach.

Some games use a similar solution: A player walks freely in a restricted area, and when he or she gets near any border, the background starts to move until the player starts walking back toward the center of the screen.

Some other games use a combination of automatic scrolling with player-controlled scrolling; the player controls scrolling right or left, but is always moving from the top to the bottom of the screen.

One last group of games comprises the auto-scrolling ones, such as the sample we'll code in this chapter: The background simply goes on scrolling without player intervention, creating a nonstop action game.

Choosing the Scrolling Type

Even a topic as simple as choosing the scroll type we should use in our game may lead to extensive discussion. Of course there's a lot more we can do when coding scrolling games; don't be reluctant to try new ideas. For example, we can split the screen and make two areas with different scrolling behaviors, such as in the old arcade game Olympics, where the computer controls a character running in the upper middle of the screen and the player runs in the lower middle; each half-screen scrolls with its own speed.

The most appropriate type of scrolling will vary from game to game, and it will be up to us to make the final choice between code complexity and game playability.

Technical Tips for Scrolling Implementation

Since there are many ways to implement scrolling—from a "camera" moving over a big image through to the opposite extreme, scrolling based on tiles—there's no universal solution. However, keep in mind the following rules of thumb as design goals:

- Avoid loading images from disk exactly when they are needed. Although it may not be practical to load all images at the start of the game, try to load the images before they're needed; never depend on disk response time, or the game will probably lack smoothness.

- On the other hand, loading every image and creating every vertex buffer for the game when it starts is only practical in small game fields. In bigger games memory can run out in a short time; so balance memory use against the loading speed of the images. A simple technique to avoid memory shortage is dividing the game into levels, and loading the images only for the current level. While the user is distracted with a screen with the current score or a short message, the next level can be loaded.

Tile-Based Games

A tile is just a small piece of a graphic with a certain property that reveals its status for the game (a background, an enemy, an obstacle, a ladder, etc.). Creating a tiled game field is simply a matter of putting the tiles together in a logical fashion. We can do this by creating a level-map file with a level designer or even with a text editor; our game, when running, translates the tile codes in the file to graphical tiles on screen.

When coding tile-based games, the first question to ask is, Will our tiles be clearly visible, or will we try to hide the repetitive patterns?

There's no correct answer—it just depends on the game.

If we're working with a game that deals with visible blocks or bricks, there's no special trick to use when creating the tiles: We can simply list the tiles we'll use and draw them. Drawing some extra tiles can help the game to look more interesting to the user.

However, using seamless tiles is another matter. The following sections offer some practical tips for when we need seamless tiles.

Draw the Basic Tile Sets

When creating a new tile set, we first draw the basic tiles for each type of terrain: for example, one tile for water, one tile for grass, one tile for sand, etc. An example of a basic set is show in Figure 4-2.

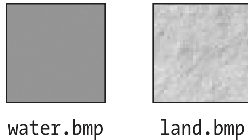

Figure 4-2. A basic set of tiles, comprising two terrain types

With the tiles presented in Figure 4-2 and in other figures in this chapter, we include suggested filenames. Using a logical filenaming scheme for your tiles can help you easily find specific tiles when you need them.

Keeping an eye on our "budget" of memory (how much memory we can use for textures), let's create some simple variations, such as adding different patterns to a sand tile, or some little bushes or small stones to a grass tile.

We should review our basic set, using the game project as a guide, to be sure that we create a tile for every terrain or object we need. Once we are satisfied with our basic set, we can go on to the next step: creating border tiles.

Create Border Tiles

To create border tiles, we must separate the tiles into groups that will have connections with each other, and then create the borders for the tiles in each group. We must do this because usually some tiles won't need to have borders with some of the others—for example, the tiles that will create internal parts of a building don't need to have any special border with the outside tiles.

Within every group, create the border tiles between each type of terrain. There are basically three types of borders we can create, as shown in Figure 4-3:

- **Border tiles:** With this kind of tile, one terrain type occupies almost all of the area of each tile, leaving just few pixels for the transition to the next terrain.

- **3/4-to-1/4 tiles:** One terrain occupies 3/4 of the tile and another terrain occupies the rest for this tile type. (Think about this texture as cutting a tile in four equal-sized squares and filling three of them with one type of terrain, and one with another.)

- **Half-to-half tiles:** With this kind of tile, each type of terrain occupies half of the tile; the transition between terrain types can be on the vertical, horizontal, or diagonal axis.

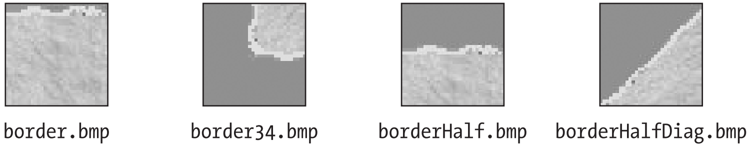

Figure 4-3. Example of border tiles

These basic border tiles will suffice to create a continuous-looking terrain, but if we have many of these transition tiles presented to the player on every screen, the set still won't suffice to create an illusion of a nontiled terrain. That's why we need to create extra borders between the most-used terrain types.

Include Extra Transition Tiles

For those transitions that will be presented most of the time to the player, include some different tiles for each transition and for the basic set, which will be used sparingly to break down the feeling of patterns of repetition. For example, when creating tiles between water and land, include some rocks, a bay, or a larger beach, so you can use them eventually to give more variation to the game visual. Examples of simple variations are shown in Figure 4-4.

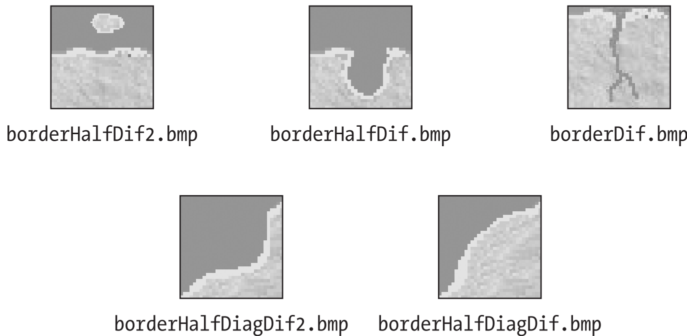

Figure 4-4. Simple variations of border tiles

To create a better set of tiles, test if the transitions for each tile are seamless in every direction (when we rotate the tiles). An improved game engine can use the same tiles with various rotations to achieve better results. An easy way to do this is to create some tiles with only borders (and a flat color at the middle), and use them as "masks" over other tiles, employing any graphical editor to hide the transitions between the base tiles and the masks. Ensuring that the border pixels are always the same will allow smooth transitions.

In Figure 4-5 we see part of a screen from Sid Meyer's Civilization. Although the terrain looks random at first glance, if we pay a little more attention we can see the same tiles used in different compositions, with great results.

© 2002 Infogrames Interactive, Inc.
All Rights Reserved.
Used With Permission.

Figure 4-5. Civilization: a successful example of a tile-based game

Creating New Game Classes

Looking at the similarities amongst the test programs we did in Chapter 3, we can choose some parts of the code to create DirectX versions for the two basic game classes we created in Chapter 2: a `GameEngine` class, which will be responsible for initializing, terminating, and managing the device operations, and a `Sprite` class, which will create some vertices and load the images as textures (transparent or otherwise) from image files.

NOTE *We'll try to maintain the class interfaces used in Chapter 2, but since using Direct3D is very different from using GDI+, don't be surprised if we find new ways to do the same things.*

We'll also extend our game class library by creating a `GameMusic` class according to the basic concepts we'll examine when studying the DirectAudio interface.

The GameEngine Class

To create the new `GameEngine` class, we'll use the lessons learned in Chapters 1 and 2 about game engines, plus the Direct3D concepts discussed in Chapter 3. The following sections present the concepts involved in the creation of this class.

The Class Interface

To include all we learned from the previous chapter, the `GameEngine` class must have some objects that will store references to Direct3D objects and a reference to the DirectAudio object (which controls the game music and sound effects, as we'll see). Another common theme we can see in the samples of the previous chapter is the use of flexible vertex formats to define figure vertices when creating a device, as well as the use of a background color when clearing the device.

Looking to the game engines from the samples of Chapters 1 and 2, we can again see some common properties, such as the window handle used for drawing, the width and height of the game field, and some flags to control whether the game is over or paused.

Looking again at the samples in Chapter 3, we can see a repetitive pattern in every Direct3D application. This gives us some clues about possible methods to include in our new `GameEngine` class:

1. Initialize the various Direct3D objects.

2. Enter a loop that will call the `Render` procedure between `BeginScene` and `EndScene` methods.

3. Dispose all Direct3D objects created.

With these ideas in mind, we can imagine three methods that can be called sequentially in a game, as shown in the pseudo-code here:

```
Dim MyGameEngine as clsGameEngine
    ...
    ' Initialize Direct3D and DirectAudio objects
    MyGameEngine.Initialize
    ' Start the game loop, the procedure will only return when the game is over
    MyGameEngine.Run
    ' Dispose the Direct3D and DirectAudio objects
MyGameEngine.Finalize
```

We'll need a fourth method: an empty `Render` method that will be called from within a loop on the `Run` method. Each game will create a new class, derived from the generic `GameEngine` class, that will implement the `Render` procedure and add any extra features to the `Initialize` and `Finalize` methods.

NOTE *For more information on flexible vertices and the objects and the methods mentioned in the preceding text, see Chapter 3.*

The suggested interface for the `GameEngine` class is shown in Figure 4-6; when creating new games, we can improve the class as needed.

GameEngine
-objDirect3D -objDirect3DDevice -objD3DX -objGameMusic -BackgroundColor -Width -Height -ScreenWinHandle -GameOver -Paused -D3DFVF_CustomVertex -IMAGE_PATH -SOUND_PATH
+Initialize() +Render() +Finalize() +Run()

Figure 4-6. The GameEngine class interface

The description of the interface members of the GameEngine class are shown in Table 4-1.

Table 4-1. Interface Members of the DirectX GameEngine Class

TYPE	NAME	DESCRIPTION
Property	ObjDirect3DDevice	The Device object, used by all graphical operations.
Property	BackgroundColor	The color used when clearing the device.
Property	Width	The width of the game field.
Property	Height	The height of the game field.
Property	ScreenWinHandle	The window handle used by all drawing functions.
Property	GameOver	If true, the game is over.
Property	Paused	If true, the game is paused. This flag and the preceding one store the current game status. Each game uses these flags to end or pause the game.
Constant	FVF_CustomVertex	The constant that will define which flexible vertex format we'll be using when creating the device and the vertices of the sprites.
Constants	IMAGE_PATH and SOUND_PATH	The relative paths where the images and the sound files are stored.
Method	Initialize	The procedure that will initialize Direct3D.
Method	Render	The rendering procedure. This procedure will be an empty overrideable function that must be implemented in the derived classes.
Method	Finalize	This method will dispose any objects created in the initialize procedure.
Method	Run	This method will simply have a BeginScene-EndScene block inside a loop, allowing the game programmer to start the game by calling the Run method.

The next code listing shows the definition of the `GameEngine` class, including the proposed properties, methods, and constants:

```
Imports Microsoft.DirectX.Direct3D
Imports Microsoft.DirectX

Public Class clsGameEngine
    Protected Shared objDirect3DDevice As Device = Nothing

    ' Simple textured vertices constant and structure
    Public Const FVF_CUSTOMVERTEX As VertexFormat = VertexFormat.Tex1 Or _
                                                    VertexFormat.Xyz

    ' defines the default background color as black
    Public BackgroundColor As Color = Color.FromArgb(255, 0, 0, 0)
    ' Images path, to be used by the child classes
    Protected Const IMAGE_PATH As String = "Images"

    Public Structure CUSTOMVERTEX
        Public X As Single
        Public Y As Single
        Public Z As Single
        Public tu As Single
        Public tv As Single
    End Structure

    Public Width As Integer = 25
    Public Height As Integer = 25

    Private ScreenWinHandle As System.IntPtr

    ' Controls the game end
    Public Shared GameOver As Boolean
    Public Shared Paused As Boolean

    Sub Run()
    Public Overrideable Sub Render()
    Public Function Initialize(Owner As windows.forms.control) As Boolean
    Protected Overrides Sub Finalize()
End Class
```

The `Imports` clause used in the beginning of the class is a new feature of Visual Basic .NET, and it allows us to use any of the objects of the imported namespace directly, without needing to inform the full object hierarchy. For example, instead of creating a Microsoft.DirectX. Direct3.D .Device object, we can simply use Device in our variable declarations.

Before writing the code for the class methods, let's ensure that we understand the scope modifiers used in the `GameEngine` class, as explained in the next section.

Understanding the Scope Modifiers

Now is a good time to look at the scope keywords used before variable and method declarations, and used extensively in the `GameEngine` class:

- `Private`: Visible only inside the class
- `Protected`: Visible only to the class and its derived classes
- `Public`: Visible to any code outside and inside the class

Other keywords used in this context are

- `Shared`: Any member declared with this keyword is shared with all the objects created for the class, and can be accessed directly by the class name (we don't need to create objects). Constants are shared by default, even when we don't use the shared keyword.
- `Overrideable`: This keyword indicates that a class member can be overridden by derived classes. In the preceding sample code, the `Render` procedure must be an overrideable function, since the code for it will be supplied by the derived classes, although it will be called by the `Run` method in the base class.
- `Overrides`: This keyword indicates that the class member is overriding a corresponding member of the base class. For example, to code a working `Finalize` event for any Visual Basic .NET class, we need to override the base class event `Finalize`.

- `Shadows`: When we want to redefine a function in a derived class, we can use this keyword. In this case, we aren't overriding a member from the base class, so when the method is called from the derived class, the method of this class will be called, and when a call is made from the base class, the method of the base class is called.

In the next section we'll examine the code for each method of the `GameEngine` class.

Coding the Class Methods

There are no new concepts in these methods, so we can simply copy the code from one of the samples in the previous chapter and organize it as methods of the `GameEngine` class. As previously explained, we have an `Initialize` method to do the initialization (as we saw in Chapter 3) for a full-screen application using an orthogonal view. The `Finalize` method disposes of the objects created, and the `Run` method has the rendering loop, used in all programs in Chapter 3, that calls the empty `Render` method for each loop interaction. The `Render` method will be coded in the derived class, which will include specific features for each game.

```
Sub Run()
    Do While Not GameOver
        If (objDirect3DDevice Is Nothing) Then
            GameOver = True
            Exit Sub
        End If
        objDirect3DDevice.Clear(ClearFlags.Target, BackgroundColor, 1.0F, 0)
        objDirect3DDevice.BeginScene()
        ' Calls the Render sub, which must be implemented on the derived classes
        Render()
        objDirect3DDevice.EndScene()
        Try
            objDirect3DDevice.Present()
        Catch
            ' Some error ocurred, possibly in the Render procedure
        End Try
        Application.DoEvents()
    Loop
End Sub
```

```
Public Overrideable Sub Render()
    ' This sub is specific for each game,
    '    and must be provided by the game engine derived class
End Sub

Public Function Initialize(Owner as Windows.Forms.Control) As Boolean
    Dim WinHandle As IntPtr = Owner.handle
    Dim objDirect3Dpp As PresentParameters
    Initialize = True

    Try
        DispMode = Manager.Adapters(_
                   Manager.Adapters.Default.Adapter).CurrentDisplayMode
        DispMode.Width = 640
        DispMode.Height = 480
        ' Define the presentation parameters
        objDirect3Dpp = New PresentParameters()
        objDirect3Dpp.BackBufferFormat = DispMode.Format
        objDirect3Dpp.BackBufferWidth = DispMode.Width
        objDirect3Dpp.BackBufferHeight = DispMode.Height
        objDirect3Dpp.SwapEffect = SwapEffect.Discard

        objDirect3Dpp.Windowed = True 'False
        ' Create the device
        objDirect3DDevice = New Device(_
           Manager.Adapters.Default.Adapter, _
           DeviceType.Reference, WinHandle, _
           CreateFlags.SoftwareVertexProcessing, objDirect3Dpp)

        ' Tells the device which is the format of our custom vertices
        objDirect3DDevice.VertexFormat = FVF_CUSTOMVERTEX
        ' Turn off culling => front and back of the triangles are visible
        objDirect3DDevice.RenderState.CullMode = Cull.None
        ' Turn off  lighting
        objDirect3DDevice.RenderState.Lighting = False
        ' Turn on alpha blending, for transparent colors in sprites
        objDirect3DDevice.RenderState.SourceBlend = Blend.SourceAlpha
        objDirect3DDevice.RenderState.DestinationBlend = Blend.InvSourceAlpha
        ' The sprite objects must turn on alpha blending only if needed,
        '    using the following line:
        '    objDirect3DDevice.RenderState.AlphaBlendEnable = True
```

```
        '  Set the Projection Matrix to use a orthogonal view
        objDirect3DDevice.Transform.Projection = Matrix.OrthoOffCenterLH(0,_
            DispMode.Width, 0, DispMode.Height, 0.0F, 0.0F)
    Catch de As DirectXException
        MessageBox.Show("Could not initialize Direct3D. Error: " & _
               de.ErrorString, "3D Initialization.", MessageBoxButtons.OK, _
               MessageBoxIcon.Error)
        Initialize = False
    End Try

    ' Dispose the used objects
    DispMode = Nothing
    objDirect3Dpp = Nothing
End Function

Protected Overrides Sub Finalize()
    On Error Resume Next  ' We are leaving, ignore any errors

    If Not (objDirect3DDevice Is Nothing) Then objDirect3DDevice.Dispose()
    objDirect3DDevice = Nothing

    GC.Collect()
    MyBase.Finalize()
End Sub
```

In the initialize procedure, we used a set of common parameters for the device creation; we can change it as needed for each application.

In the next section we'll see the upgraded code for the second game class of our library: the `Sprite` class.

The Sprite Class

Here we'll attempt to create a generic `Sprite` class, which can be improved upon as needed, and can be used to create derived classes that can hold specific properties and methods according to the game being created.

We can use the basic interface for sprites defined in Chapter 2, with the `New`, `Draw`, and `Load` methods, and some simple properties. Looking back at Chapter 3, we can list some suggestions for other interface elements: values for the translation, scaling, and rotation operations in the x and the y axis, and a speed value for both axes (speed is just the counter to be used for the translation in every new frame drawn).

Because a sprite is drawn over a polygon, we'll need a property to store the vertex buffer and a helper function to create the flexible vertices. Because a sprite is a 2-D image, there's no need to store z values for the transformations.

TIP *An important point of this new* `Sprite` *class is that we'll never need to change the vertex coordinates of the sprite to perform any translations or rotations; we can use the matrix transformations as seen in Chapter 3 to do it faster.*

NOTE *For more information about flexible vertices, vertex buffers, and matrices, refer to Chapter 3.*

The complete interface for a Direct3D sprite is shown in Figure 4-7.

Sprite
-X -Y -SizeX -SizeY -IsTransparent -Direction -IMAGE_SIZE -ScaleFactor -SpeedX -SpeedY -TranslationX -TranslationY -ScaleX -ScaleY -RotationX -RotationY -SpriteImage -VertBuffer
+New() +Load() +Draw() +Dispose() +CreateFlexVertex()

Figure 4-7. The Sprite class interface

The `Sprite` class members are described in Table 4-2.

Table 4-2. Interface Members for the DirectX Sprite Class

TYPE	NAME	DESCRIPTION
Properties	`X` and `Y`	The upper-left position of the sprite.
Properties	`SizeX` and `SizeY`	The size of the sprite, in the x and y axes.
Property	`IsTransparent`	If true, the `Draw` function will draw a transparent sprite, loaded in the `Load` function. We don't need to store a color key property to say which color will be transparent; such a color is used only when loading the textures.
Property	`Direction`	The current direction the sprite is moving in. This property can be used to choose which image must be drawn.
Constant	`IMAGE_SIZE`	The default size for a square sprite.
Property	`ScaleFactor`	Same as the GDI+ `Sprite` class, it holds a constant used when creating the sprite, indicating whether the x and y values are pixel values or based on `IMAGE_SIZE`. Useful for creating tiled game fields.
Properties	`SpeedX` and `SpeedY`	The speed (translation increment per frame) of the sprite on the x and y axes.
Properties	`TranslationX` and `TranslationY`	The current translation value in each axis, from the initial x,y position.
Properties	`ScaleX` and `ScaleY`	The scale to be applied to the sprite in each axis.
Properties	`RotationX` and `RotationY`	The rotation in each axis.
Property	`SpriteImage`	The sprite texture, loaded from an image file.
Property	`VertBuffer`	The vertex buffer with the vertices of the sprite.
Method	`New`	Method for creating a new sprite.
Method	`Load`	Method for loading the image file from disk; it creates the vertices used to draw the image on the screen.
Method	`Draw`	Method that draws the sprite.
Method	`Dispose`	Method that disposes of the texture and the vertex buffer used by the sprite.
Method	`CreateFlexVertex`	Helper method used when creating the sprite vertex buffer.

The interface code for the `Sprite` class is shown here:

```
Imports Microsoft.DirectX.Direct3D
Public Class clsSprite
    Inherits clsGameEngine

    Public IsTransparent As Boolean = False
    Public Direction As enDirection

    Public X As Single
    Public Y As Single
    Public SizeX As Single = IMAGE_SIZE
    Public SizeY As Single = IMAGE_SIZE

    Public Const IMAGE_SIZE As Integer = 32
    Public ScaleFactor As enScaleFactor = enScaleFactor.enScaleSprite
    ' speed used in translation
    Public SpeedX As Single = 0
    Public SpeedY As Single = 0

    ' Values used for the operations
    Public TranslationX As Single = 0
    Public TranslationY As Single = 0
    Public ScaleX As Single = 1
    Public ScaleY As Single = 1
    Public RotationX As Single = 0
    Public RotationY As Single = 0

    Protected SpriteImage As Texture
    Protected VertBuffer As VertexBuffer
    Public Enum enScaleFactor
        enScalePixel = 1
        enScaleSprite = IMAGE_SIZE
    End Enum
    Public Enum enDirection
        North = 1
        NorthEast = 2
        East = 3
        SouthEast = 4
        South = 5
        SouthWest = 6
        West = 7
        NorthWest = 8
    End Enum
```

```
    Sub New(...)
    Function Load(...) As Boolean
    Private Function CreateFlexVertex(...) As CUSTOMVERTEX
    Sub Draw()
    Public Sub Dispose()
End Class
```

We must highlight two points in the preceding code: the use of default values for the properties (always use the most common ones), and the use of the `inherits` clause. The `Sprite` class will be closely related to the game engine, and it'll need to use some of the game engine properties to work properly, so we must create it as a `GameEngine`-derived class.

Let's see the code for the methods, starting with the `New` method. We'll create two overrides for the function: one for creating opaque sprites, and another for creating transparent sprites. The following code sample depicts the difference between these two overrides:

```
Sub New(strImageName As String,  startPoint As POINT, _
        Optional  Scale As enScaleFactor = enScaleFactor.enScaleSprite, _
        Optional  width As Integer = IMAGE_SIZE, _
        Optional  height As Integer = IMAGE_SIZE)
    X = startPoint.x
    Y = startPoint.y
    SizeX = width
    SizeY = height
    ScaleFactor = Scale
    If Not Load(strImageName) Then _
    Err.Raise(vbObjectError + 1, "clsSprite", _
           "Could not create the sprite textures")
End Sub

Sub New( strImageName As String,  colorKey As Integer,  startPoint As POINT, _
         Optional  Scale As enScaleFactor = enScaleFactor.enScaleSprite, _
         Optional  width As Integer = IMAGE_SIZE, _
         Optional  height As Integer = IMAGE_SIZE)
    ' When calling the New procedure with a colorKey,
    '   we want to create a transparent sprite
    IsTransparent = True
    X = startPoint.x
    Y = startPoint.y
    SizeX = width
    SizeY = height
    ScaleFactor = Scale
```

```
    If Not Load(strImageName, colorKey) Then _
       Err.Raise(vbObjectError + 1, "clsSprite", _
                  "Could not create the sprite textures")
End Sub
```

The `Load` procedure will receive an optional `colorKey` parameter that will be used to load a transparent texture if the `IsTransparent` property is set to true. Besides loading the texture from an image file, the `Load` procedure must create the vertex buffer used to show the sprite in the draw procedure, using the `CreateFlexVertex` helper procedure.

```
' Default colorKey is magenta
Function Load(strImageName As String, _
    Optional colorKey As Integer = &HFFFF00FF) As Boolean
    Dim vertices As CustomVertex()
    Dim i As Integer

    Try
        If IsTransparent Then
            'Load the transparent texture
            SpriteImage = TextureLoader.FromFile(objDirect3DDevice, _
               Application.StartupPath & "\" & IMAGE_PATH & "\" & strImageName, _
               64, 64, D3DX.Default, 0, Format.Unknown, Pool.Managed, _
               Filter.Point, Filter.Point, colorKey)
        Else
            SpriteImage = TextureLoader.FromFile(objDirect3DDevice, _
               Application.StartupPath & "\" & IMAGE_PATH & "\" & strImageName)
        End If

        VertBuffer = New VertexBuffer(GetType(CustomVertex), 4, _
            objDirect3DDevice, Usage.WriteOnly, FVF_CUSTOMVERTEX, Pool.Default)
        vertices = VertBuffer.Lock(0, 0)
        ' CreateFlags a square, composed of 2 triangles in a triangle strip
        vertices(0) = CreateFlexVertex(X * ScaleFactor, Y * ScaleFactor, 1, 0, 1)
        vertices(1) = CreateFlexVertex(X * ScaleFactor + SizeX, _
                                 Y * ScaleFactor, 1, 1, 1)
        vertices(2) = CreateFlexVertex(X * ScaleFactor, _
                                 Y * ScaleFactor + SizeY, 1, 0, 0)
        vertices(3) = CreateFlexVertex(X * ScaleFactor + SizeX, _
                                 Y * ScaleFactor + SizeY, 1, 1, 0)
        ' Release the vertex buffer and commits our vertex data
        VertBuffer.Unlock()
        Return True
```

```
    Catch de As DirectXException
        MessageBox.Show("Could not load image file " & strImageName & _
         ". Error: " & de.ErrorString, "3D Initialization.", _
         MessageBoxButtons.OK, MessageBoxIcon.Error)
        Return False
    End Try
End Function

Function CreateFlexVertex( X As Single,  Y As Single,  Z As Single,  _
         tu As Single,  tv As Single) As CUSTOMVERTEX
    CreateFlexVertex.X = X
    CreateFlexVertex.Y = Y
    CreateFlexVertex.Z = Z
    CreateFlexVertex.tu = tu
    CreateFlexVertex.tv = tv
End Function
```

The `Draw` method is very straightforward: It simply sets the texture and draws the rectangle defined by the vertex buffer created in the `Load` procedure, using the concepts shown in the previous chapter.

```
    Sub Draw()
        '  Turn on alpha blending only if the sprite has transparent colors
        If IsTransparent Then
            objDirect3DDevice.RenderState.AlphaBlendEnable = True
        End If
        Try
            objDirect3DDevice.SetTexture(0, SpriteImage)
            objDirect3DDevice.SetStreamSource(0, VertBuffer, 0)
            objDirect3DDevice.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2)
        Catch de As DirectXException
            MessageBox.Show("Could not draw sprite. Error: " & de.ErrorString, _
             "3D Initialization.", MessageBoxButtons.OK, MessageBoxIcon.Error)
        End Try
        '  Turn off alpha blending  if the sprite has transparent colors
        If IsTransparent Then
            objDirect3DDevice.RenderState.AlphaBlendEnable = False
        End If
    End Sub
```

The last method, `Dispose`, will only dispose of the texture and the vertex buffer created in the `Load` procedure. Calling the `Collect` method of the garbage collector will ensure a faster disposal of memory; and calling the `SupressFinalize` method of this class will prevent errors that can arise if the default finalizer is called before the objects are freed from memory. The `Dispose` method is shown in the next code sample:

```
Public Sub Dispose()
    On Error Resume Next  ' We are leaving, ignore any errors
    SpriteImage.Dispose()
    VertBuffer.Dispose()
    GC.Collect()
    GC.SuppressFinalize(Me)
End Sub
```

DirectAudio Classes

There are two different sets of components for audio input and output: Direct-Music, for background music playback, and DirectSound, for sound effects. These two sets together are sometimes called *DirectAudio*, although they are separate things. DirectMusic doesn't have a managed version, but we can access its features through COM interoperability.

.NET is kind of an evolution from COM architecture; but we still can use COM objects from .NET programs, and more: The .NET programs generate COM wrappers, so COM-based languages (such as the previous version of Visual Basic) can access .NET components too. To use non-managed DirectX features, we must include in our projects a reference to the VBDX8.DLL.

Besides the components for audio playback, DirectAudio includes Direct-Music Producer, which can be used to create new music based on chord maps, styles, and segments. We'll not enter into any details about DirectMusic Producer here, but if you want to exercise your composing skills, you'll find a lot of relevant material in DirectX SDK.

The main distinction between background music and sound effects is related to the file types used by each one. Sound effect files are MIDI or WAV files that store a single piece of sound that is usually played when a specific action occurs in a game (such as a player getting a bonus or dying). Background music can be produced using a MIDI file playing in a loop, but it's best done with segment (SGT) files. SGT files have a main piece of music and one or more motifs (or waves) that can be played any time, as the program commands, so the music can change subtly from time to time.

NOTE *A special music generation program is included with DirectX SDK, and it allows professional musicians to create segment files by connecting the computer to a music device (like a keyboard), or composing the music directly on the computer using instruments from the predefined libraries or even creating new ones. It's beyond the scope of this book to enter into details about the creation of segment files, but those who want to get a deeper knowledge of this subject will find many samples in the DirectMusic help feature on the DirectX SDK.*

A lot of theory and technical details are connected to DirectAudio, but we'll stick here to the simplest ways of generating sound for our application.

There aren't many steps we must follow to play sound, but we'll enclose these steps in two classes so every application doesn't need to include these initialization details. To play sounds using managed DirectSound, we need to perform the following four steps:

1. Create the DirectSound device object.

2. Create the DirectSound buffer object.

3. Load the WAV sound file into the buffer.

4. Play the sound using the buffer.

As we can see, only two objects are involved in playing sound through DirectSound:

- `Device`: Responsible for any generic operation regarding the sound device

- `Buffer`: Loads the sound files, sets specific properties, and plays the sounds

When playing files through DirectMusic, we'll have to implement some extra steps and different objects, due to the different nature of the sound files controlled. The steps to play any MIDI or SGT file using DirectMusic are as follows:

1. Create the `Performance` and `Loader` objects.
2. Initialize the `Performance` object.
3. Set the search directory from which the `Loader` object will load the sound files.
4. Load the file with the sound information of a `Segment` object.
5. If the file is an SGT file, download band and wave information for the file to the `Performance` object.
6. Play the sound from the segment object, choosing the audio path to play it in (primary or secondary).
7. If the file is an SGT file, play any included motifs as needed.

From these steps, we can see that there are three main objects we'll need to handle when playing sounds though DirectMusic: `Performance`, `Loader`, and `Segment`. These objects are described in more detail in the following list:

- `Performance`: Responsible for the management of all music playback. This object controls a set of instruments, with their special characteristics, and maps them to specific audio paths. It controls the music tempo, handles messages and events, and sets the music parameters. The performance mixes all sounds from primary and secondary audio paths seamlessly.
- `Loader`: Loads sound files, including instrument data, styles, bands, and collections. When we load a sound file, every related file is loaded too.
- `Segment`: Stores and plays each music piece as it is loaded from the sound file.

With these concepts in mind, we are ready to define the basic audio classes' interface, as shown in Figure 4-8.

GameSound
-Looping
+Load() +Play() +StopPlaying()

GameMusic
-Performance -Loader
+Initialize() +SetVolume() +Load() +Play() +PlayMotif() +StopPlaying() +Finalize()

Figure 4-8. The audio classes

We can add new properties and methods as needed when implementing audio management for our games.

In the next sections, we discuss the details for each game audio class.

The GameSound Class

As we can see in our class definition, to play a sound through managed DirectSound, we must load the buffer with the sound file and then play the sound. This way of working gives us two choices for playing multiple sounds in our games:

- Create one single `GameSound` object and call `Load` and `Play` methods for every sound to be played.
- Create a `GameSound` object for each sound to be played, load each sound once, and call the `Play` method for the specific object that holds the sound to be played.

For sounds that are constantly playing throughout the game, the second approach is better, because it won't waste time reloading the sounds.

The interface for the class will be as follows:

```
Imports Microsoft.DirectX.DirectSound
Public Class ClsGameSound
    Protected Const SOUND_PATH As String = "Sounds"
    Dim DSoundBuffer As SecondaryBuffer = Nothing
```

```
    Public Looping As Boolean = False
    Private Shared SoundDevice As Device = Nothing
    Sub New(WinHandle As Windows.Forms.Control)
    Function Load(strFileName As String) As Boolean) As Boolean
    Sub StopPlaying()
    Sub Play()
End class
```

In the next sections, we'll look at the code and details for each class method.

The New Method

On the `New` method we must initialize the sound device. Since we only need one device initialization for all the sounds we want to play, we must define the device object as `shared` (as we already did in the class definition), and include code in the `New` method to initialize the device only if it's not already initialized. The code for this method is presented in the following listing:

```
Sub New(WinHandle As System.Windows.Forms.Control)
    If SoundDevice Is Nothing Then
        SoundDevice = New Device()
        SoundDevice.SetCooperativeLevel(WinHandle, CooperativeLevel.Normal)
    End If
End Sub
```

Besides creating the device, we can see a specific initialization to inform the device of the appropriate cooperative level—in other words, how the sound device object will interact with other applications that may be using the device. The possible values for the `SetCoopLevel` enumeration follow:

- `Normal`: Specifies the device be used in a way that allows the smoothest multithreading and multitasking (multiple application) operation. Using this setting will force us to only use the default buffer and output formats, but this will suffice for our samples.

- `Priority`: Sets a priority level on the device, so we can change buffer and output formats. This member doesn't behave well with other applications trying to share the device, so we must use it only if we don't expect concurrency from other applications.

- `Write Only`: Specifies that the application plays only on primary buffers. This enumeration member will work only on real hardware devices; if DirectSound is emulating the device, the call to `SetCooperativeLevel` will fail.

Once we've created the device, we can load the sounds into a sound buffer, as described in the next section.

The Load Method

The `Load` method will be mainly based on the `CreateSoundBufferFromFile` function, which requires the following parameters:

```
<Buffer object>.CreateSoundBufferFromFile(FilePath, BufferDescription)
```

The first parameter is the path from which we want to load the sound file. The second is a description of specific properties we'll need for the buffer, from the capabilities we'll have (control volume, frequency, etc.) to how the device must act when playing this buffer, when another application gets the focus. In our sample, we'll only set one flag, `GlobalFocus`, which tells the device to continue playing the buffer even if other DirectSound applications have the focus.

The full code of the `Load` function is shown in the following sample:

```
Function Load(strFileName As String) As Boolean
    Try
        Dim Desc As BufferDesc = New BufferDesc()
        Desc.Flags = New BufferCapsFlags() = BufferCapsFlags.GlobalFocus Or _
                             BufferCapsFlags.LocSoftware

        DSoundBuffer = _
         SoundDevice.CreateSoundBufferFromFile(Application.StartupPath & _
                             "\" & SOUND_PATH & "\" & strFileName, Desc)
    Catch de As Exception
        MessageBox.Show("Could not load the sound file. Error: " & de.Message, _
            "Music Initialization", MessageBoxButtons.OK, MessageBoxIcon.Error)
    End Try
    Load = True
End Function
```

In the next section we'll discuss the last two methods of the `GameSound` class, `Play` and `StopPlaying`.

The Play and StopPlaying Methods

The `Play` and `StopPlaying` methods will use the `Play` and `Stop` methods belonging to the `Buffer` object, as shown in the next code listing. Similarly to the `CreateSoundBufferFromFile` function, the `Play` method will receive a structure with the playing flags; in our sample we'll use the default settings, and include an extra setting for looping the sound if the `Looping` property of the class is set.

```
Sub Play()
    Dim PlayFlags As BufferPlayFlags = BufferPlayFlags.Default
    If Looping Then PlayFlags = BufferPlayFlags.Looping

    If Not (DSoundBuffer Is Nothing) Then
        Try
            DSoundBuffer.Play(0, PlayFlags)
        Catch de As Exception
            MessageBox.Show("Error playing sound file. Error: " & de.Message, _
                  MessageBoxButtons.OK, MessageBoxIcon.Error)
        End Try
    End If
End Sub

Sub StopPlaying()
    If Not (DSoundBuffer Is Nothing) Then
        DSoundBuffer.Stop()
        DSoundBuffer.SetCurrentPosition(0)
    End If
End Sub
```

In the next section we'll discuss the second DirectAudio class, `GameMusic`.

The GameMusic Class

The basic class interface to access DirectMusic features is shown in the next code listing. The first line imports the library created by Visual Basic as a wrapper to the VBDX8.DLL file, used for COM access to all DirectX features, including DirectMusic.

```
Imports DxVBLibA
Public Class clsGameMusic
    Private Shared DMusicPerf As DirectMusicPerformance8 = Nothing
    Private Shared DMusicLoad As DirectMusicLoader8 = Nothing
    Private DMusicSegment As DirectMusicSegment8 = Nothing

    ' Background music is looped by default
    Public looping As Boolean = True

    ' Default sound files path
    Private Const SOUND_PATH As String = "Sounds"

    Sub SetVolume(intVolume As Integer)
    Function Initialize(WinHandle As IntPtr) As Boolean

    Function Load(strFileName As String, bolLooping As Boolean = True) As Boolean
    Sub Play()
    Function PlayMotif(strMotifName As String) As Boolean
    Sub StopPlaying()
    Protected Overrides Sub Finalize()
End Class
```

In this code sample, we can see that DirectMusic defined the `Performance` and `Loader` objects as private so they'll only be accessible to the class. This will prevent us from having to know internal details of the class when playing music from our games.

The Initialize Method

In the `Initialize` function, we need to create and initialize the class objects with the most commonly used values. The next code sample shows a possible implementation for this method:

```
Function Initialize(WinHandle As IntPtr) As Boolean
   ' CreateFlags our default objects
   Dim AudioParams As DMUS_AUDIOPARAMS

   DMusicPerf = DX8.DirectMusicPerformanceCreate()
   DMusicLoad = DX8.DirectMusicLoaderCreate()
```

```
    Try
       ' Initialize our performance object to use reverb
       DMusicPerf.InitAudio(WinHandle.ToInt32, _
                CONST_DMUS_AUDIO.DMUS_AUDIOF_ALL, AudioParams, , _
                CONST_DMUSIC_STANDARD_AUDIO_PATH.
                DMUS_APATH_SHARED_STEREOPLUSREVERB, 128)
        ' Turn on all auto download
        DMusicPerf.SetMasterAutoDownload(True)
         ' Set our search folder
         DMusicLoad.SetSearchDirectory(Application.StartupPath & "\" & SOUND_PATH)
         Initialize = True
    Catch de As Exception
       MessageBox.Show("Could not initialize DirectMusic. Error: " & de.Message, _
           "Music Initialization.", MessageBoxButtons.OK, MessageBoxIcon.Error)
      Initialize = False
    End Try
End Function
```

Some of the key functions used in the `Initialize` method deserve a better explanation. Let's start by taking a closer look at the `InitAudio` method and its possible values:

```
<Performance object>.InitAudio(hWnd, Flags, AudioParams, DirectSound, _
                               DefaultPathType, ChannelCount)
```

The first parameter, `HWnd`, receives the window handle. This will usually be the same window used for Direct3D device object creation. If we specify a window handle, we don't need to specify the DirectSound object, so DirectMusic creates a private one for its personal use, making our code simpler.

The second parameter, `Flags`, specifies a member of the `CONST_DMUS_AUDIO` enumeration that will state the requested features for the performance. Although you can specify different values, such as `BUFFERS` to fully support audio path buffers, or `3D` for supporting 3-D sounds, using `ALL`, as in the sample code, will prepare `Performance` to handle any kind of loaded sounds.

The third parameter, `AudioParams`, allow us to specify the desired control parameters for the sound synthesizer, and to be notified of which requests were granted. We can specify details such as the frequency of the sample and the number of voices used; but since we are using only the simplest features from DirectMusic, we'll let all flags remain set to their default values.

The DirectSound object is used when we are employing DirectMusic features to support DirectSound with playing WAV files; since we are dealing with separate classes for each one, we can simply omit this parameter.

The next parameter, DefaultPathType, receives a member of the DMUS_APATH enumeration, which specifies the default audio path type, as described in the following list:

- DYNAMIC_3D: Indicates the audio path will play to a 3-D buffer (the sounds are distributed on the speakers in order to create the illusion of a 3-D environment). For more information on 3-D sounds, refer to the DirectX SDK.

- DYNAMIC_MONO: Used for creating an audio path with mono buffering (all music sounds are of equal volume in each speaker).

- DYNAMIC_STEREO: Specifies the sounds be played in a stereo environment (the sounds are distributed on the speakers according to how they were recorded—for example, if the percussion instruments were more to the left, the left speaker will have a louder percussion sound).

- SHARED_STEREOPLUSREVERB: Indicates the buffer created for the audio path has all the features of the stereo buffer, plus an environmental reverb (echo in music).

The ChannelCount parameter specifies the number of performance channels allocated to the audio path. In the code sample, we have 128 performance channels, which means that we can play up to 128 different sounds within the same Performance object.

A second function in the preceding sample that deserves a more detailed explanation is SetMasterAutoDownload:

```
<Performance object>. SetMasterAutoDownload(value)
```

This method is one of many used to set global parameters for the performance object, passing a single value as a parameter. The following lists a few more of the methods in this category:

- SetMasterVolume: Sets the volume, measured in hundreds of decibels, ranging from +20 (amplification) to –200 (attenuation). Values below –100 or above +10 will result in no audible difference, so the useful values are up to 10 times the default volume to 1/100 of it.

- SetMasterAutoDownload: Turns on and off automatic loading of instruments when loading the segment files that use them. We'll always want this parameter set to on.

- `SetMasterTempo`: Represents the "scale factor" for the tempo of the music. The default value is 1, so you can set this to 0.5 and have music playing at half the normal speed, or set this to 2 and double the speed (if you want to hear, say, Lou Reed singing like Madonna). Valid values range from 0.01 to 100.

There are other methods of this type, but these are the ones we'll most commonly want to set. We can create new methods for the `GameMusic` class to set these parameters, such as a `SetVolume` method to set the current volume for the performance object, as in the following code sample:

```
Sub SetVolume(intVolume As Integer)
    If Not (DMusicPerf Is Nothing) Then
        Try
              DMusicPerf.SetMasterVolume(intVolume)
        Catch de As Exception
              MessageBox.Show("Could not set the master volume. Error: " & _
                     de.Message, MessageBoxButtons.OK, MessageBoxIcon.Error)
        End Try
    End If
End Sub
```

NOTE *As you can imagine, there are complementary methods with the* `GetMaster` *prefix that are used to retrieve the value of a specific configuration parameter from the* `Performance` *object. It's beyond the scope of this book to explain every one of them; refer to DirectX SDK help for a comprehensive list.*

The last new function we saw in the sample, `SetSearchDirectory`, simply informs the `Loader` object of the directory from which the sound files will be loaded, so we won't need to give the path for every sound loaded.

The Load Method

The `Load` method will be mainly based on the `LoadSegment` function, which receives the file and path from which we want to load the sound file (MIDI or SGT). Since we already gave the search path for the `Loader` object, we can pass only the filename to the function.

The full code for a first version of the `Load` function is shown in the following sample:

```
Function Load(strFileName As String, Optional bolLooping As Boolean = True)_
               As Boolean
    ' Backgound music loops by default
    looping = bolLooping

    ' Load the music file
    Try
        DMusicSegment = DMusicLoad.LoadSegment(strFileName)
        ' If it's a segment file, we have some special treatment
        If strFileName.EndsWith(".sgt") Then
            If Not (DMusicSegment Is Nothing) Then '// Download the segment
                 DMusicSegment.Download(DMusicPerf.GetDefaultAudioPath)
            End If
        Else
            If strFileName.EndsWith(".mid") Or strFileName.EndsWith(".rmi") Then
                 DMusicSegment.SetStandardMidiFile()
            End If
        End If
    Catch de As Exception
        DMusicSegment = Nothing
        MessageBox.Show("Could not load the sound file. Error: " & de.Message, _
              "Music Initialization", MessageBoxButtons.OK, MessageBoxIcon.Error)
        Load = False
        Exit Function
    End Try
    Load = True
End Function
```

In VB .NET, every data type corresponds to a class definition, with its own set of properties and methods, so we can use these methods and properties as we would do with any kind of objects. In the preceding sample code, `EndsWith()` *is a method of the string data type, corresponding to the* `Right` *method in previous versions of Visual Basic, with the only difference being that we don't need to pass the number of characters to check.*

Another interesting point about the preceding code is the use of the `Try-Catch` *block with a specific type of exception. This structured error block, new in Visual Basic .NET, allows the programmer to catch generic errors or errors from a specific set (such as* `Exception`*).*

We can improve the music played from segment files using styles and motifs that are stored in those files by the music author. In the next section we'll examine what these are and how to apply them to our audio class.

Styles and Motifs

Styles and motifs are intrinsic parts of a segment file, and they allow us to choose from a set of previously created compositions to play at any time during music execution. Each segment file can have one or more styles recorded within it, and every style can have one or more motifs.

A *style* is a collection of instruments and music patterns or *motifs* (sequences of music values for each instrument present in the style). To read a specific style from a music segment, we use the `GetStyle` method. There are various `Get` methods in the `Segment` object, similar to the `GetMaster` methods present in the `Performance` object, as we saw before. To retrieve a style, we must pass the style number, from 0 to the number of styles present minus 1. Passing an invalid value will generate an error.

After choosing a specific style, we can retrieve its number of motifs using the `GetMotifCount` function.

Based on these two functions, we can extend our `Load` method to retrieve the number of styles and the number of motifs, setting new properties that will allow the game to retrieve the information from the `GameMusic` class. To do so, we must include the following new properties in the class:

```
Private DMusicStyle As DirectMusicStyle = Nothing
Public MotifCount As Integer = 0
Public StyleCount As Integer = 0
```

And we need to include the following lines in the `Load` method. Read the following code carefully so you understand the mechanism of reading styles and motifs.

```
    Dim strMotifName As String
    Dim bolLoopStyles As Boolean = True
    Dim bolLoopMotifs As Boolean = True

    Do While (bolLoopStyles)
        ' Count the styles
        Try
            DMusicStyle = DMusicSegment.GetStyle(StyleCount)
            StyleCount += 1
            ' Count the motifs of the style
            bolLoopMotifs = True
            MotifCount += DMusicStyle.GetMotifCount
        Catch
            ' The GetParam will throw an exception if there are no more styles.
            bolLoopStyles = False
        End Try
    Loop
    ' We start counting from zero, so add 1 to have the real Style count value
    StyleCount += 1
```

In the next section we'll discuss the methods for playing and stopping the audio.

The Play and StopPlaying Methods

Once the styles have been read, we can code the `Play` and `StopPlaying` methods, which will use the `PlaySegmentEx` and `StopEx` methods belonging to the `Performance` object, as shown in the following code sample. Both methods receive the segment object (created in the `Load` method) used to play or stop playing.

```
Sub Play()
    If Not (DMusicSegment Is Nothing) Then
        Try
            If Looping Then
                DMusicSegment.SetRepeats(CONST_DPNWAITTIME.INFINITE)
            Else
                DMusicSegment.SetRepeats(0)
            End If
            DMusicPerf.PlaySegmentEx(DMusicSegment, 0, 0)
        Catch de As Exception
            MessageBox.Show("Error playing music file. Error: " & de.Message, _
                MessageBoxButtons.OK, MessageBoxIcon.Error)
```

```
        End Try
    End If
End Sub

Sub StopPlaying()
    If Not (DMusicSegment Is Nothing) Then
       DMusicPerf.StopEx(DMusicSegment, 0, 0)
    End If
End Sub
```

In the next section, we look at how to play motifs, and create two new methods to add this feature to our `GameMusic` class.

Playing Motifs

To finish our `GameMusic` class, it'll be interesting to have one function to play a motif from the current music segment. We can use such a function to add some subtle variations to our background music, making it a little more exciting. We can simply do this by using the same `PlaySegmentEx` function. Instead of using the entire sound loaded as a segment object as the first parameter, we'll load the motif from the segment object (using the `GetMotif` function) and pass the motif as a parameter for that function.

We can create an analogous method that receives a number and plays the associated motif; such a method would be useful if we don't previously know the motif names, but do know how many there are (as calculated in the `Load` method).

The two overloaded methods are shown in the next listing, and it's up to each game to choose which one best suits its needs.

```
Function PlayMotif(strMotifName As String) As Boolean
        Dim Motif As DirectMusicSegment8
        Try
            ' Get the motif
            Motif = DMusicStyle.GetMotif(strMotifName)
            DMusicPerf.PlaySegmentEx(Motif, _
                CONST_DMUS_SEGF_FLAGS.DMUS_SEGF_DEFAULT Or _
                CONST_DMUS_SEGF_FLAGS.DMUS_SEGF_SECONDARY, 0)
            PlayMotif = True
        Catch
            PlayMotif = False
        End Try
End Function
```

```
Function PlayMotif(intMotifIndex As Integer) As Boolean
        Dim strMotifName As String
        Try
            ' Get the motif
            strMotifName = DMusicStyle.GetMotifName(intMotifIndex)
            ' Call the overloaded method which receives a string
            PlayMotif = PlayMotif(strMotifName)
        Catch
            PlayMotif = False
        End Try
End Function
```

The GetMotifName method, listed in the previous code sample, receives the motif number (ranging from one to GetMotifCount) and returns the motif name (as informed by the composer when creating the segment) as the second parameter. Calling this function with an invalid index will generate an error.

In the next section we'll discuss the final method of our class.

The Finalize Method

The last class method, Finalize, simply destroys the audio objects, making sure that the Performance object is closed before destroying it, using the CloseDown method.

```
Protected Overrides Sub Finalize()
    ' The object is being destroyed, so ignore any errors
    On Error Resume Next
    MyBase.Finalize()
    'Clean up DMusicSegment
    If Not (DMusicSegment Is Nothing) Then
        DMusicPerf.StopEx(DMusicSegment, 0, 0)
    End If
    DMusicSegment = Nothing
    DMusicStyle = Nothing
    DMusicLoad = Nothing
    If Not (DMusicPerf Is Nothing) Then
        DMusicPerf.CloseDown()
    End If
    DMusicPerf = Nothing
End Sub
```

Now that this class is finished, we have a complete set of classes to help us to include audio capabilities in our games. The most important point to note when using this kind of approach is that when coding a game, we need to be concerned with the game goals, not the less important details such as how to load music or initialize a device.

In the next section we'll discuss the proposal for the sample game used in this chapter and the next, which will allow us to test, in a practical way, our gaming class library.

The Game Proposal

We are going to do an Activision's River Raid (an old Atari console game) clone, but in this chapter we'll create only half of the game features. Since not everyone will remember the original River Raid game features (or will have ever played it), let's introduce the points we'll want to cover in our first version of the game.

In River Pla.Net the player will control a plane that is flying over a top-to-bottom scrolling river. Even when the player isn't moving the plane, the ground beneath it will be moving. As far as we know, the river goes on forever, so the main goal of the game is to live for as long as possible.

Here are some more details about the game:

- The plane will be controlled by keyboard arrows.

- There will be some obstacles along the river: ships, planes, and bridges. The ships and planes won't move in the first version of the game.

- The plane must always be flying over water; if it flies over land or over an obstacle (bridges, planes, or ships), it will be destroyed.

- To make the level design easier, the game map will be a text file, in which each character will represent a different tile when the game field is created.

- There'll be some gas barrels on the river, which will be collected by the player's plane when it flies over them. In the first version of the game, we won't create a fuel counter.

- After being destroyed, the player's plane will be invincible for a few seconds.

- The game must have background music and different sound effects for each player action: upon being destroyed, when in invincible mode, and while filling the gas tank.

When a team of developers creates a "real" game, the game proposal is normally followed by some drafts showing details about the game (like screen layout and some artwork samples), and must be refined until everyone in the team has a clear understanding of what the game will be. The game proposal goal is to answer the question: *What are we doing?*

Once everyone agrees about the game proposal, it's time to answer the next question we need to ask: *How will we do it?* The game project presents the technical details to answer this question, but both documents aren't static; they can (in fact, they must) be revised every time a new point of view arises and is agreed upon by the game development team. Care must be taken not to include every suggestion, or the planning stage will simply never end.

The last two important questions in a "real" game development are mainly targeted to commercial games (*How long will it take to finish the project? How much will it cost?*), and won't be discussed here.

The Game Project

Looking back at the project proposal, before starting anything else, we need to decide some higher level details for the game. It's good practice to think about how things will work before writing down any class diagrams or pseudo-code.

In our specific case, maybe the two most important points on the proposal are the following:

- The player will control a plane that is flying over a top-to-bottom scrolling river.

- To make the level design easier, the game map will be a text file, where each character will represent a different tile when creating the game field.

How do we *really* make scrolling games? How can we design a level with Notepad?

In our game, these questions are very much related. First, let's figure out the creation of the game field, and then think about how to do the translation and implementation of the other features described in the game proposal.

Defining the Game Tiles

River Pla.Net is one of those games that allows us to design the whole game field based on *tiles*. We can create the game field map file with a text editor, using the game program to translate the set of characters in the file to a set of tiles on screen.

Maybe the first thing we must think about when creating a tile-based game is what size our tiles will be. They aren't required to be square ones, but using squares is the best approach, since we can put the tiles together in any direction, without any problems.

To define the number of tiles, we must first decide the resolution for our game. A higher resolution will allow us to use more tiles, if the tiles are a fixed size, or force us to have larger images. Either approach can lead to a reduction in performance because they'll both use more memory than lower resolutions, so let's keep our sample to a 640×480 resolution to make it as fast as possible. With this resolution, if we have square tiles that are 32-pixels wide, we can have 15 tiles for the height and 20 tiles for the width.

Looking at the game proposal, we can list a basic set of tiles to fill the game goals:

- Land
- Water
- Ship
- Enemy plane
- Gas barrel
- Bridge

This reduced set of tiles is probably very close to the one used by the original River Raid, but using just these tiles will result in a very "blocky" game. In Figure 4-9 we see a basic set of tiles.

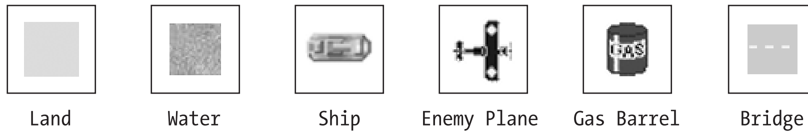

Figure 4-9. A basic set of tiles

Let's create a game field using any graphical tool (Microsoft's Paint will suffice) to cut and paste the tiles shown in Figure 4-9 so we can see a first "visual prototype" of our game, giving us a better idea about how it'll look. Figure 4-10 shows a screen drawn with these tiles.

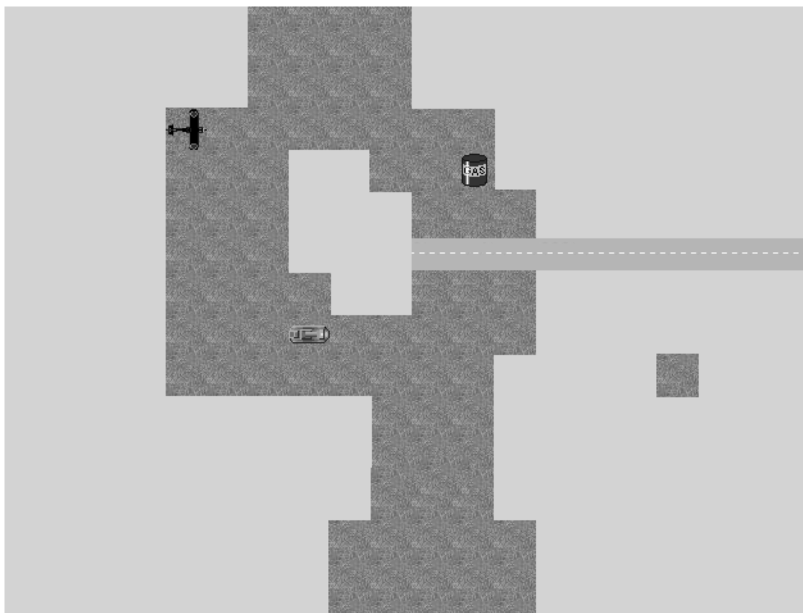

Figure 4-10. A first screen based on tiles

As we can see in this first screen, using only this set of tiles will result in a flat block game: All river "curves" are straight, and we can barely see the border between land and water. Creating borders is just a matter of drawing a new set of tiles that can be used to generate an island and a lake, and all river curves can be derived from this set. Figure 4-11 shows such a set of tiles.

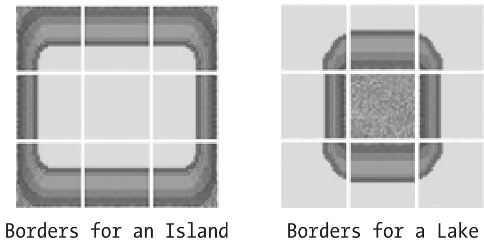

Figure 4-11. The border tiles

It's important to adopt tile names that will help us to find them easily. For the border tiles, we suggest giving all tiles the "border" prefix and a direction indicator. For example, for the border where the water meets the land to the north, the name would be "borderN", and for a Southwest border the name would be "borderSW". Take a closer look at Figure 4-12 to understand this naming convention and all the border tile names.

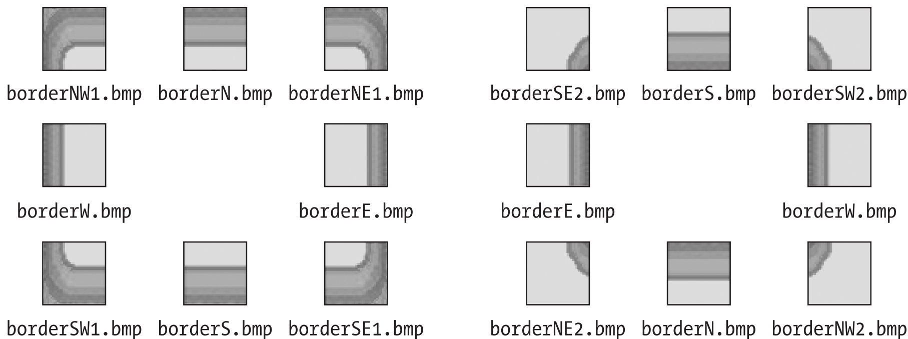

Figure 4-12. The names of the border tiles

The N, S, W, and E borders are used as island borders and as lake borders by just exchanging positions.

We can remake the screen from Figure 4-10 to add the borders, resulting in the screen shown in Figure 4-13.

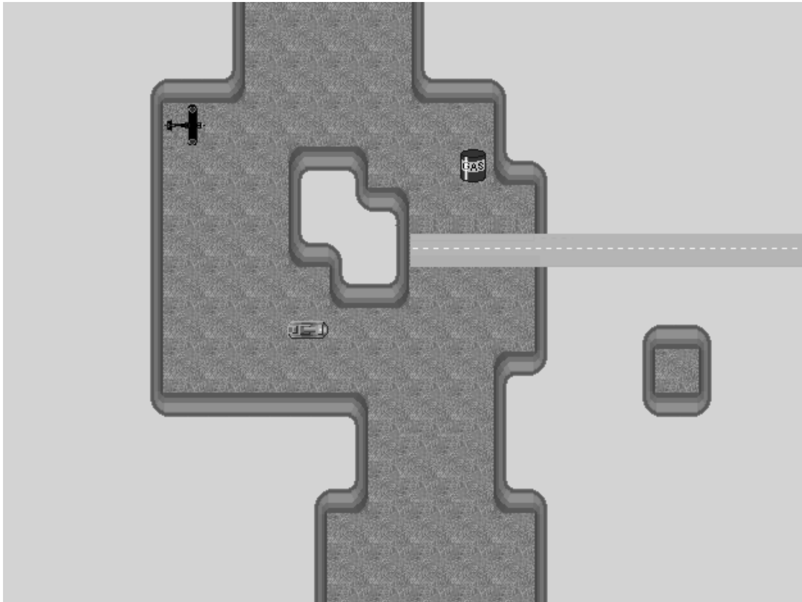

Figure 4-13. A second screen, based on a larger set of tiles

Initially it may seem as though these are the only tiles we'll ever need, but just imagine if the plane flies over a straight section of the river, with no bridges or opponents: The player would barely see the scrolling movement—the only tip would be the water movement. So we can add some "final touch" tiles, such as trees and mountains (at least two of each, to give some visual variation), and maybe create a bridge tile different from the road one, to give the game a nicer look and feel. Figure 4-14 shows the "final touch" tiles.

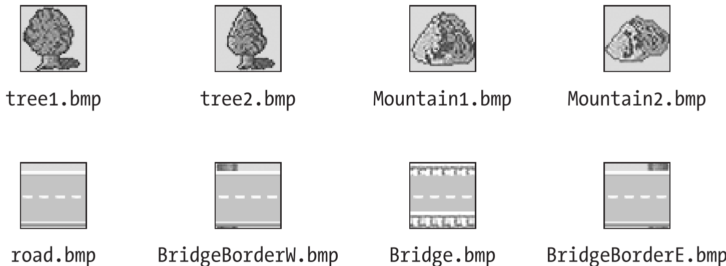

Figure 4-14. The "final touch" set of tiles

Creating a bridge with borders forces us to change the road tile and make it thin, and we have to add two new tiles to use when the road is over the river border. Our final screen, using all tiles, is shown in Figure 4-15.

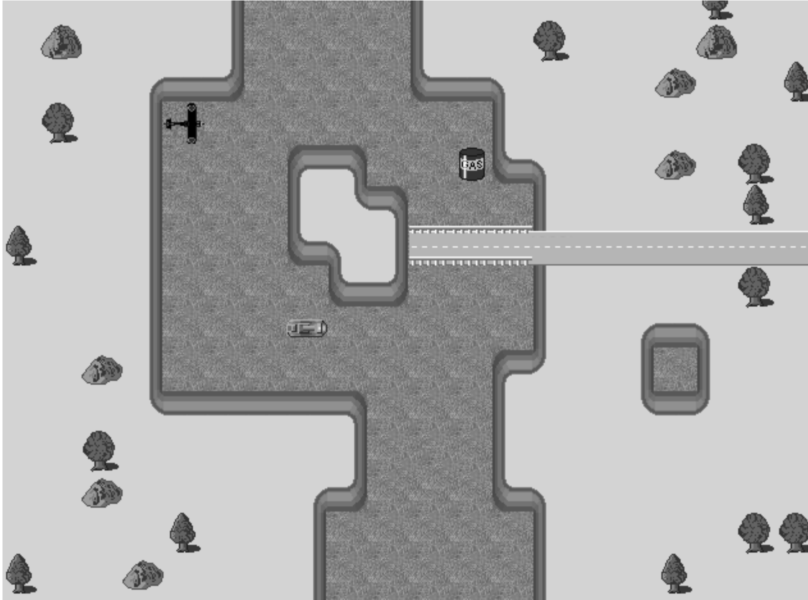

Figure 4-15. The final screen, using all sets of tiles

Of course, we can go on creating new tiles. For example, a diagonal border for the river would be interesting, so we can break the "blocky" visual that still persists. We can add different borderlines, with beaches or little bays; and we can add more "final touch" objects, such as houses, buildings, or even animals or people, to arrange around the screen.

The more tiles we include, the more flexibility the level designer will have to create our groundbreaking levels. But for our purpose here, the tiles we've already created will suffice.

Before going to the next topic, we shall define the char codes corresponding to each tile. Doing this at game project phase will allow the level designer to start writing the levels at the same time the programmers start coding. Then again, there's no rule for choosing the chars. A good approach is to choose chars that will give a visual clue about how the level will look; and as for the borders, we can simply choose North as 1 and go on sequentially in a clockwise direction. The characters chosen are shown in Table 4-3.

Table 4-3. The Tile Codes

CODE	TILE	CODE	TILE	CODE	TILE
1	borderN	9	borderSW2	G	Gas
2	borderNE1	A	borderW	(	BridgeBorderW
3	borderNE2	B	borderNW1	)	BridgeBorderE
4	borderE	C	borderNW2	-	Road
5	borderSE1	T	Tree	=	Bridge
6	borderSE2	M	Mountain	.	Land
7	borderS	S	Ship	_	Water
8	borderSW1	P	EnemyPlane	--	--

We have decided to create one single char for both types of trees and another one for both mountain types, so the game can randomly choose the image to use and have some subtle visual variations every time it's played.

With these codes, we can look again at our first test screen and create a corresponding map of it, as show in the following code listing:

```
.....4____A......T..
.M...4____A..T...M..
...975____876...M..T
.T.4P_______A.......
...4___B2__G86..M.T.
...4___AC2___A....T.
T..4___864===(------
...4____85___A....T.
...4___S_____A.976..
..M4________B3.4_A..
...C11112___A..C13..
..T.....4___A.......
..M....95___86......
....T..4_____A....TT
T..M...4_____A..T...
.......4_____A......
```

Scrolling Games

When talking about translation, remember that we have already made a scrolling object—our walking-man cube in Chapter 3, when performing translations on it. Since we already have a set of features in Direct3D that allow us to scroll an object

without having to move every vertex of it, we can use the same idea here: Simply change the `Transform.World` matrix of the device object to do the translation.

So all we need to do is create the game field and perform small translations on it, for each time frame, to make it scroll. But will we scroll the entire game field? Will moving all the tiles of the game field result in prolonged calculations?

Looking again at the samples in the last chapter, we see that setting the world matrix will only define the transformations to be applied to the vertex buffer when we call the `DrawPrimitivesUP` function; so there's no performance difference when setting the matrix for a scene with a few dozen vertices or for a scene with several thousand.

So what we want to do is to draw the minimum number of vertices possible. Remember, our game field will be defined by a text file. For performance reasons we must load it only once, when starting the game, converting each char in the text file to a tile that will be in a fixed (x,y) position, depending on the position of the char in the map text file. The tiles will be fixed throughout the game, and we'll scroll over them, changing the translation value of the world matrix, to "move the camera" over the game field. Just like a plane moving over a real river, the river is fixed on the ground, and we move over it.

So to minimize the vertices drawn, we can store the current line from the bottom of the screen, and draw the 15 next lines—as mentioned in the previous section, our visual game field will be 20 tiles wide and 15 tiles tall. Since we won't be performing translations on the objects the size of a tile, we'll need to draw an extra line to avoid the top of the screen not being drawn appropriately. It's like being in *The Matrix* (the movie): The world will exist only when we are looking at it.

Another good idea to minimize the number of vertices being drawn is simply remove the Land tile from our set, and draw the other sets over a green background. To make the background green, all we need to do is to use the green color as a parameter of the `Clear` method of the device object.

The Class Diagram

Since we already have all base classes for the game engine, sprite, game music, and sound, all we need to do is to create derived classes that will supply specific characteristics, according to the game's special needs.

Looking at the game proposal, we can only see two candidates for new classes: `Player` and `Tile`, which will be derived from the `Sprite` class. Of course, we'll need a class derived from the game engine, too (let's call it the `RiverEngine` class), to implement the game code.

After a little brainstorming over the game proposal, we have come up with the class diagram shown in Figure 4-16.

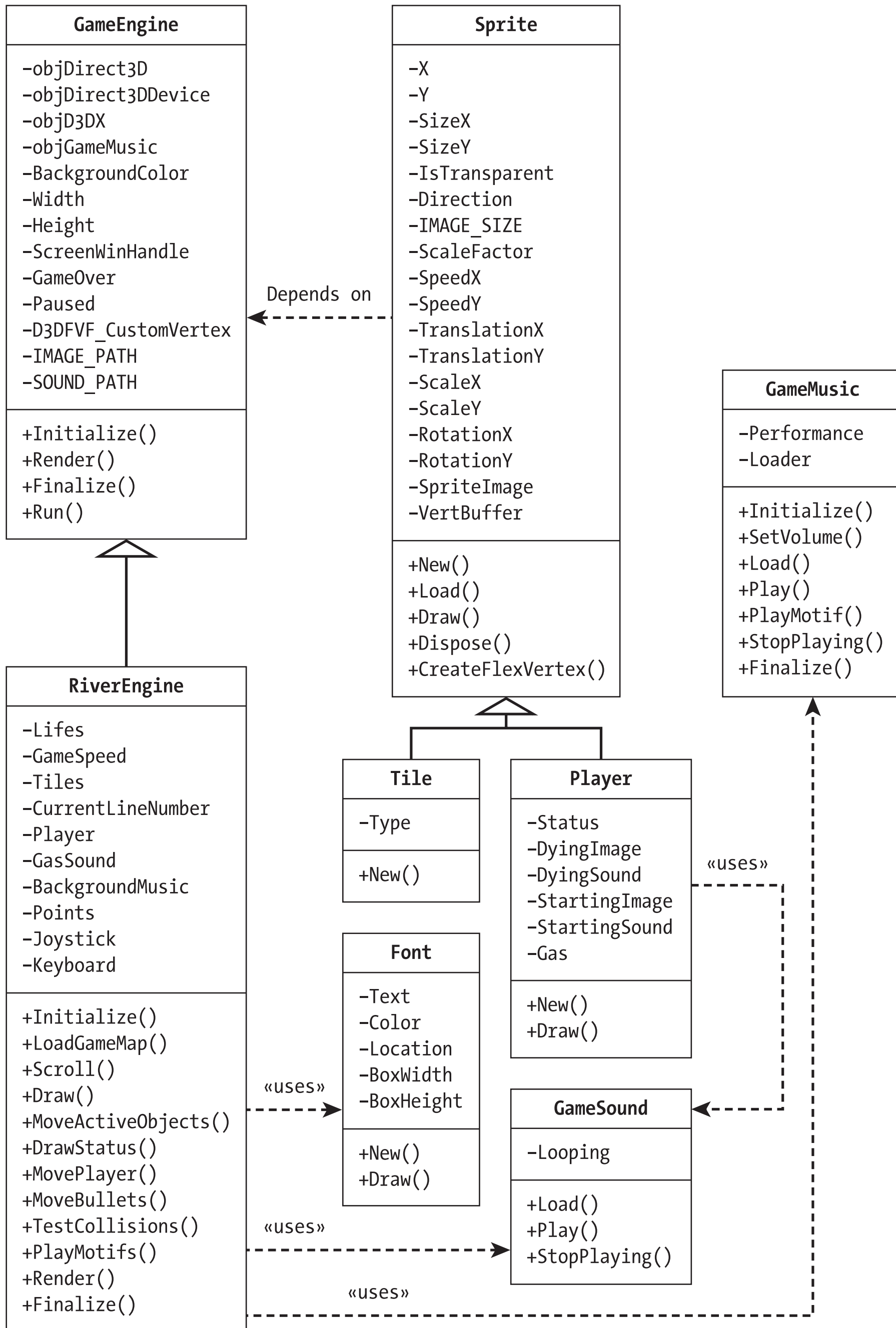

Figure 4-16. The River Pla.Net game class diagram

The following tables describe the properties and methods for each of the game classes, starting with the `Tile` class in Table 4-4.

Table 4-4. The Tile Class

TYPE	NAME	DESCRIPTION
Property	`Type`	A member of an internal enumeration that should store the type of the tile (land, water, enemies, etc.)
Method	`New`	An overloaded method for each `New` method of the `Sprite` class that simply adds an extra parameter for the `Type` property

Table 4-5 shows the description of the methods and properties for the `Player` class.

Table 4-5. The Player Class

TYPE	NAME	DESCRIPTION
Property	`Status`	The current status for the player: flying, dying, or starting a new life
Property	`Gas`	The fuel tank value by percentage
Property	`DyingImage`	Image (image file loaded to create textures) to show when the player dies
Property	`StartingImage`	Image to show when the player is in invincible mode (starting a new life)
Property	`DyingSound`	Object that stores the sound to be played when the player dies
Property	`StartingSound`	Object that stores the sound to be played when the player is in invincible mode (starting a new life)
Property	`GameSound`	Object that will initialize the sound components and make them ready to play sounds
Method	`New`	An overloaded method that will load all player images
Method	`Draw`	An overloaded method that will draw the player image based on the current status

The `RiverEngine` class includes members that are additions to the `GameEngine` class to make it fit our needs. In Table 4-6 we present the descriptions for each class member. As we discussed in earlier chapters, this class interface is a result of many interactions over game project refinements.

Table 4-6. The RiverEngine class

PROPERTY	NAME	DESCRIPTION
Property	`Lifes`	Specifies the number of lives of the player.
Property	`GameSpeed`	Indicates the scrolling speed of the game.
Property	`Tiles`	Represents an array with all the tiles of the game.
Property	`CurrentLineNumber`	Specifies the number of the current line at the bottom of the screen. This property is used to control the tiles we should draw.
Property	`Player`	Indicates an object of the `Player` class.
Property	`GasSound`	Indicates an object of the `GameSound` class, which will be played when the player passes over a gas barrel.
Property	`BackgroundMusic`	Indicates an object of the `GameMusic` class, which will play the game's background music.
Method	`Initialize`	Calls the `Initialize` method of the base class, loads the game map, and creates all game objects (`Player`, `GasSound`, and `BackgroundMusic`).
Method	`LoadGameMap`	Loads the game map text file and populates the `Tiles` array.
Method	`Render`	Overrides the `Render` empty method from the base class, and will be called in the game loop (on the `Run` base class method). All the drawing functions, physics tests, and sound-playing functions should be called from here.
Method	`Scroll`	Scrolls the game field by calculating the game world translation matrix.
Method	`Draw`	Draws the visible tiles of the game field, based on the `CurrentLineNumber` property and the `Height` of the screen.
Method	`MovePlayer`	Moves and draws the player, based on the input received from the keyboard.
Method	`TestCollisions`	Tests the collision of the player against the obstacles of the game field.
Method	`PlayMotifs`	Plays the random motifs over the background music to add a little variance to it.

The Main Program

This main program will show the splash screen and, after closing it, execute the steps we saw before when presenting the `GameEngine` class. The pseudo-code for this will be very simple, as shown in the following sample:

```
Create object from RiverPlanet class
Create a window to be the game screen
Show splash screen
Initialize RiverPlanet object
Show the game window
Run the game (execute method RUN from RiverEngine object)
' The Run is a synchronous method, it will return when the game ends
Destroy the RiverEngine object
Dispose the game window
```

And now we are ready to start looking at the code details.

The Coding Phase

In order to follow a "progressive disclosure" technique, let's do the coding in discrete steps, so we can better test and understand our code. The functionalities of each step will be as follows:

1. First draft: Code the `Tile` Class and load the game map and the `Draw` method to draw the tiles in the `Render` method.

2. Second draft: Make the game field scroll.

3. Third draft: Create the `Player` class to draw the player's plane, and make it move according to the keyboard input.

4. Fourth draft: Code the collision detection.

5. Final version: Play background music and sound effects.

In the next section we start coding the first draft of our game.

First Draft: Loading and Drawing the Game Field

Our main objective in this first draft is to load the text file and convert it into a graphical game field. For this, we'll code the `Tile` class, some basic methods of the `RiverEngine` class, and our main program.

The `Tile` class, as we saw in the game project phase, will be very simple and will only add a specific `Type` attribute to the base class `Sprite`, as shown in the following code sample:

```
Imports Microsoft.DirectX.Direct3D
Public Class ClsTile
    Inherits clsSprite

    Enum enType
        Background = 0
        Water = 1
        Land = 2
        Gas = 3
        Ship = 4
        Plane = 5
        Bridge = 6
    End Enum
    Public Type As enType

    Sub New(strImageName As String, startPoint As POINT, intType As enType)
        MyBase.New(strImageName, startPoint)
        Type = intType
    End Sub

    Sub New(strImageName As String, colorKey As Integer, _
            startPoint As POINT,  intType As enType)
        MyBase.New(strImageName, colorKey, startPoint)
        Type = intType
    End Sub
End Class
```

As for the `RiverEngine` class, let's define the whole interface and then code only the functions we need at the moment:

```
Imports System.IO
Imports Microsoft.DirectX
Imports Microsoft.DirectX.Direct3D
Public Class ClsRiverEngine
    Inherits clsGameEngine
```

```
    Private BackgroundMusic As ClsGameMusic

    Public Lifes As Integer = 5463
    Public gameSpeed As Integer = 10

    '  Define the array that will store the reference to the tiles on the game
    '  field.  Since we don't know its size a priori, we'll REDIM it when reading
    '  the game map file
    Private tiles As ClsTile(,)
    Private CurrentLineNumber As Int32 = 0
    Public Player As ClsPlayer
    Private GasSound As ClsGameSound

    Overrides Sub Render()
    Sub Scroll()
    Sub Draw()
    Sub MovePlayer()
    Sub PlayMotifs()
    Function TestCollision() As Boolean
    Public Shadows Function Initialize(Owner as Windows.Forms.Control) As Boolean
    Function LoadGameMap() As Boolean
    Protected Overrides Sub Finalize()
End Class
```

The `Imports` clause in the header of the class file allows us to include a namespace reference to the current file so we can use its members directly. In the preceding sample code, we import the `System.IO` namespace, which includes classes and enumerations for manipulating files.

NEW IN .NET

The fastest way to read and write text files in Visual Basic .NET is to use the `StreamReader` *and* `StreamWriter` *classes. We'll use the first one, which has some methods dedicated to reading files, including the* `ReadLine` *method, which reads one text line until the new line char (just what we'll need here). The files are opened when creating the object (in the* `New` *method), and closed using the* `Close` *method.*

Coding the LoadGameMap Method

The first problem that comes to mind when we start thinking about how the `LoadGameMap` function will be implemented is that we read a text file from the first line to the last, but our game field needs to be presented to the player in the reverse order. So, if we open the game map file in Notepad, the last line of the file will be the first one to be drawn at the bottom of the screen, and then we'll draw the next ones over it, to make the game scroll until we reach the first line of the file.

A possible approach to solving this problem is to read the entire file to a string array and then run through this array to create the `Tiles` array. We'll include a first line in the text file with the number of lines of the array, so we can `Redim` it accordingly. It's a good idea to create a separate function for translating the chars of each line, so we can isolate the code that is responsible for accessing the file and read lines from the code that will be creating the tiles.

So let's see the code for the `LoadGameMap` function and the helper function `LoadLine`:

```
Function LoadGameMap(strGameMapFileName As String) As Boolean
    ' Define the string array that will store the lines read from the map file
    Dim GameMap As String()
    ' Define the streamreader to read the game map file
    Dim GameMapFile As StreamReader
    Dim strLine As String
    Dim i As Int32

    LoadGameMap = True
    Try
        ' Opens the game map text file
        GameMapFile = New StreamReader(Application.StartupPath & "\" & _
                      strGameMapFileName)
        ' reads the first line of the game map, which holds the size of the map
        GameMapSize = Convert.ToInt32(GameMapFile.ReadLine())
        ' Creates the game map array, including tiles and active objects
        ReDim GameMap(GameMapSize)
        ReDim tiles(Width, GameMapSize)

        ' Load the game map array
        For i = 1 To GameMapSize
            GameMap(GameMapSize - i) = GameMapFile.ReadLine()
            ' The game map file ends within the FOR loop
            '   if the game map size read from the file is wrong
```

```
            If GameMap(GameMapSize - i) Is Nothing Then
                MessageBox.Show("Incorrect game map size in " & _
                   strGameMapFileName & _
                   " - Expected size: " & GameMapSize & " / Real Size: " & i, _
                   "Critical error in game map")
                LoadGameMap = False
                Exit Function
            End If
        Next
        ' Checks to see if there are more lines in the game map file
        strLine = GameMapFile.ReadLine()
        If Not (strLine Is Nothing) Then
            ' Informs the user that we missed the last line(s)
            MessageBox.Show("Incorrect game map size in " & strGameMapFileName _
                & " - One or more lines after the " & GameMapSize & _
                "th line were ignored.", "Critical error in game map")
            LoadGameMap = False
            Exit Function
        End If
        GameMapFile.Close()
    Catch
        LoadGameMap = False
        Exit Function
    End Try

    ' Load all the game map lines from the GameMap array into the Tiles array
    For i = 0 To GameMapSize - 1
        If Not LoadLine(GameMap(i)) Then
            LoadGameMap = False
            Exit Function
        End If
    Next
    ' frees the memory used by the game map
    GameMap = Nothing
End Function

Function LoadLine(strLine As String) As Boolean
    Dim x As Integer
    Dim strSpriteFileName As String
    Dim Type As ClsTile.enType
    Static LineNumber As Integer = 0
```

```
LoadLine = True
For x = 0 To Width - 1
    Select Case strLine.Chars(x)
        Case "1"
            strSpriteFileName = "borderN"
            Type = ClsTile.enType.Land
        Case "2"
            strSpriteFileName = "borderNE1"
            Type = ClsTile.enType.Land
       ... < CASES for the other borders > ...
        Case "T"
            If Rnd() * 10 < 5 Then
                strSpriteFileName = "Tree1"
            Else
                strSpriteFileName = "Tree2"
            End If
            Type = ClsTile.enType.Land
        Case "M"
            If Rnd() * 10 < 5 Then
                strSpriteFileName = "Mountain1"
            Else
                strSpriteFileName = "Mountain2"
            End If
            Type = ClsTile.enType.Land
        Case "S"
            strSpriteFileName = "Ship"
            Type = ClsTile.enType.Ship
        Case "P"
            strSpriteFileName = "EnemyPlane"
            Type = ClsTile.enType.Plane
        Case "G"
            strSpriteFileName = "Gas"
            Type = ClsTile.enType.Gas
        Case "("
            strSpriteFileName = "BridgeBorderW"
            Type = ClsTile.enType.Land
        Case ")"
            strSpriteFileName = "BridgeBorderE"
            Type = ClsTile.enType.Land
        Case "-"
            strSpriteFileName = "Road"
            Type = ClsTile.enType.Land
```

```
            Case "="
                strSpriteFileName = "Bridge"
                Type = ClsTile.enType.Bridge
            Case "." ' Green background
                ' Do nothing
                strSpriteFileName = ""
                Type = ClsTile.enType.Background
            Case "_"
                strSpriteFileName = "Water"
                Type = ClsTile.enType.Water
            Case Else
                ' Should never happen
                strSpriteFileName = "InvalidTile"
                Type = ClsTile.enType.Land
        End Select
        Try
            If Type <> ClsTile.enType.Background Then
                 tiles(x, LineNumber) = New ClsTile(strSpriteFileName & ".bmp", _
                         New POINT(x, LineNumber), Type)
            Else
                tiles(x, LineNumber) = Nothing
            End If
        Catch e As Exception
            LoadLine = False
            MessageBox.Show("Unpredicted Error when loading game sprites: " & _
                   e.Message, "River Pla.Net", MessageBoxButtons.OK, _
                     MessageBoxIcon.Stop)
            Exit Function
        End Try
    Next
    ' Increments the line number counter
    LineNumber += 1
End Function
```

Visual Basic .NET is far stricter for automatic conversions between data types than the previous versions. To support conversions between types, each data type is treated as a class and has a set of converting methods; but we also have a `Convert` *class in the* `System` *namespace that has many conversion methods, such as the* `ToInt32` *method used in the previous code sample.*

As we can see, the `LoadGameMap` function simply uses the `StreamReader` methods to run through the text file, just checking the number of lines against the informed game map size (first line). The `LoadLine` function is just a big `Select Case`, which will set the image filename to be loaded and the tile type, which will be used to create the `Tiles` array elements. Each time we call the `LoadLine` function, we process a full line read from the file and create a new line on the array, composed of a set of 20 tiles (the width of the game field, as expressed in the next code listing). The `LineNumber` static variable, incremented at the end of the function, controls the number of lines already read to index the `Tiles` array properly.

As mentioned before, there'll be two types of mountains and trees, so we add a `rnd` function that will choose one image name or another, with a 50 percent chance for each. Roads, bridge borders, trees, and mountains are all defined as `Land` tile types, because we don't have any special treatment in code for them.

To allow us to see the map loaded, we have to code the `Initialize`, `Render`, and `Draw` methods, as presented in the next sections.

Coding the Initialize Method

In the `Initialize` method, we'll call the `LoadGameMap` function and the base class `Initialize` method, which will initialize Direct3D.

```
Private Const GAME_MAP As String = "GameMap.txt"
Public Shadows Function Initialize(Owner As Windows.Forms.Control) _
       As Boolean
    Dim WinHandle As IntPtr = Owner.Handle
    Dim i As Integer
    Randomize()
    Initialize = True

    ' Sets the background color to green
    BackgroundColor = Color.FromArgb(255, 0, 255, 0)
    ' Sets the width and height of the game field
    Width = 20
    Height = 15

    ' Loads the game map (into GameMap array)
    If Not LoadGameMap(GAME_MAP) Then
        Initialize = False
        Exit Function
    End If

    ' If the game map was loaded without errors, start Direct3D
    If Not MyBase.Initialize(WinHandle) Then
        Initialize = False
        Exit Function
    End If
End Function
```

Coding the Render Method

The `Render` function will, for now, only call the `Draw` method. This function will be called from the base class `Run` method.

```
Public Overrides Sub Render()
    Draw()
End Sub
```

Coding the Draw Method

The `Draw` method will be very simple too, because all the complexity for loading textures, initializing Direct3D, and manipulating vertex buffers is in the base classes. All we need to do is call the `Draw` method for each member of the `Tiles` array, starting from the line prior to the current line (which must have a default value of 1) and going through the height of the screen plus 1, so that we'll draw only the visible tiles, with a little margin to avoid any problems.

```
Public Sub Draw()
    Dim x As Integer, y As Integer
    Dim LineCount As Integer = 0

   ' Draw the game field
    y = CurrentLineNumber - 1

    ' We will draw a line below the current line number and a line above
    '   the last line on screen (CurrentLineNumber + Height)
    Do While LineCount < Height + 2
        For x = 0 To Width - 1
            If Not (tiles(x, y) Is Nothing) Then
                tiles(x, y).Draw()
            End If
        Next
        LineCount += 1
        y += 1
    Loop
End Sub
```

The last thing we must do is code the main procedure.

We'll be following the guidelines of the pseudo-code written in the game project, and adding a splash screen (just a screen with a nondynamic image) that will close when the player presses any key, as shown in the following code:

```
Sub frmSplash_KeyDown(sender As Object, e As KeyEventArgs) Handles MyBase.KeyDown
        Me.Dispose()
End Sub
```

The intro screen is shown in Figure 4-17.

Figure 4-17. The game splash screen

The final code for the main program is as follows:

```
Public RiverPlanet As ClsRiverEngine

Sub main()
    Dim winGameWindow As New GameWindow()
    Dim winSplash As New frmSplash()

    winSplash.ShowDialog()

    RiverPlanet = New ClsRiverEngine()
```

```
    If Not RiverPlanet.Initialize(winGameWindow) Then
        MessageBox.Show("Error initializing the game", "Critical Error", _
           MessageBoxButtons.OK, MessageBoxIcon.Stop)
        Exit Sub
    End If
    winGameWindow.Show()
    ' The run procedure will return only when the game is over
    RiverPlanet.Run()

    ' Destroying the object calls the finalize method
    RiverPlanet = Nothing
    winGameWindow.Close()
End Sub
```

We can run our game now, and see the resulting screen, which is exactly the same as our visual prototype, shown in Figure 4-15.

In the next section we'll implement the scrolling of our game field.

Second Draft: Scrolling

To see our game field scrolling, all we need to do is to add code to the `Scroll` method and call it on the `Render` procedure before calling the `Draw` method.

The new `Render` method will be as follows:

```
Public Overrides Sub Render()
    Scroll()
    Draw()
End Sub
```

To appropriately control the scrolling, the `Scroll` function must store `Matrix` as a static variable that will maintain the current translation to be applied to the `Transform.World` matrix of the `objDirect3DDevice`, created in the base class. To implement the translation, we must use the concept we learned in Chapter 3: Multiplying matrices has the same effect as adding the transformations of each one.

Since the base `GameEngine` class uses an orthogonal view of the textures, everything is automatically translated to pixel coordinates, so we can control the current line number by simply counting the number of pixels translated and, when the sum of pixels exceeds the tile size (the `clsSprite.IMAGE_SIZE` constant), add 1 to the current line property.

To avoid our game running at full (and unplayable) speed, we must control the frame rate using the technique we learned in Chapter 3 to control the walking man's speed: We store the system time using `System.Environment.TickCount` and only do the scroll processing when a given time has passed.

Taking all this into consideration, we can code a fully working scroll routine. Spend some time analyzing the following code to make sure you understand the basic concept here, which will be used later to move the player's plane.

```
Public Sub Scroll()
    Static ScrollMatrix As Matrix = Matrix.Identity
    Static LastTick As Integer
    Static PixelCount As Integer

    ' Force a Frame rate of 'GameSpeed' frames to second on maximum
    If System.Environment.TickCount - LastTick >= 1000 / gameSpeed Then
        LastTick = System.Environment.TickCount
        ' Scrolls the game field (translation on the Y axis)
        '  since the Y axis increases when going up on the screen,
        '  we use a negative transation value to make the tiles scroll down
        ScrollMatrix = _
                Matrix.Multiply(ScrollMatrix, _
                Matrix.Translation(0, -gameSpeed, 0))

        ' updates the current line number, used to control the screen drawing
        PixelCount = PixelCount + gameSpeed
        If PixelCount > clsSprite.IMAGE_SIZE Then
            CurrentLineNumber += 1
            PixelCount -= clsSprite.IMAGE_SIZE
        End If
    End If
    objDirect3DDevice.Transform.World = ScrollMatrix
End Sub
```

A set of scrolling images is shown in Figure 4-18.

Figure 4-18. Testing the scrolling game field

In the next section we'll code the player's plane, including the controls for using the keyboard.

Third Draft: Coding the Player

To add a player to our scrolling game field, we'll need to follow four steps:

1. Code the `Player` class.
2. Code the keyboard event of the game window to gather the player's input and set the appropriate values for doing the translations (moving) the player's plane.
3. Code the `MovePlayer` method of the `RiverEngine` class.
4. Call the `MovePlayer` method from within the `Render` procedure.

We won't code the collision detection now, but we can already include the images for dying and starting a new life in the `Player` class, as defined in the game project and shown in the next sample.

```
Imports Microsoft.DirectX.Direct3D

Public Class ClsPlayer
    Inherits clsSprite
    Public Status As enPlayerStatus
    Enum enPlayerStatus
        Flying = 0
        Dying = 1
        Starting = 2
        Shooting = 3
    End Enum

    Protected DyingImage As Direct3DTexture8
    Protected StartingImage As Direct3DTexture8
    Private DyingSound As ClsGameSound
    Private StartingSound As ClsGameSound

    Public Gas As Single = 100

    Sub New()
    Shadows Sub Draw()
End Class
```

The images for the player's plane are shown in Figure 4-19.

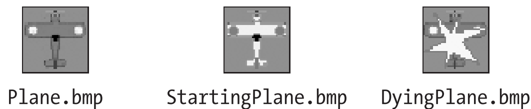

Figure 4-19. Images used for the player's plane

Coding the New Method

In the `New` method, we'll have to initialize the game variables. Calling the `Load` method of the base class, we can load the default `Image` property for the `Player` class, and we'll have to add special code for loading the `DyingImage` and `StartingImage` properties. We can simply copy and paste the code from the previously defined `Sprite` class, which loads a file to a texture.

```
Sub New()
    Dim colorKey As Integer

    colorKey = Color.FromArgb(255, 255, 0, 255)
    IsTransparent = True
    X = 0 : Y = 0
    If Not Load("plane.bmp", colorKey.ToArgb) Then _
        Err.Raise(vbObjectError + 1, "clsPlayer", _
                  "Could not create the player textures")
    Try
       DyingImage(i - 1) = TextureLoader.FromFile(objDirect3DDevice, _
                    Application.StartupPath & "\" & IMAGE_PATH & _
                    "\dyingPlane" & i & ".bmp", _
                    64, 64, D3DX.Default, 0, Format.Unknown, Pool.Managed, _
                    Filter.Point, Filter.Point, colorKey.ToArgb)
      StartingImage(i - 1) = TextureLoader.FromFile(objDirect3DDevice, _
                     Application.StartupPath & "\" & IMAGE_PATH & _
                    "\startingPlane" & i & ".bmp", _
                    64, 64, D3DX.Default, 0, Format.Unknown, Pool.Managed, _
                    Filter.Point, Filter.Point, colorKey.ToArgb)
    Catch
        MsgBox("Could not create the player textures", MsgBoxStyle.Critical)
    End Try
End Sub
```

In the next section we'll code the `Draw` method, which will enable us to see on screen the images loaded by the `New` method.

Coding the Draw Method

As for the `Draw` method, we'll have to shadow the base property `Draw`, including a `select Case` that will choose which image must be drawn based on the current player status. This player status must be set to `Dying` by the game engine when a collision occurs or when the plane runs out of fuel.

We'll need to control the time for displaying the images for dying and starting a new life on screen; for this we'll employ the same technique we have used before for controlling the frame rate (getting the current clock tick and controlling the milliseconds for each image).

The next code sample shows our `Draw` procedure, including a `Gas` property that is decremented as the plane flies. If we run out of gas, the player status will be automatically set to `Dying`.

```
Shadows Sub Draw()
   Static LastTick As Integer = 0
    '  Turn on alpha blending only if the sprite has transparent colors
    If IsTransparent Then
        objDirect3DDevice.RenderState.AlphaBlendEnable = True
    End If

    Select Case Status
        Case enPlayerStatus.Flying
            objDirect3DDevice.SetTexture(0, SpriteImage)
            objDirect3DDevice.SetStreamSource(0, VertBuffer, 0)
            objDirect3DDevice.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2)
            ' when flying, subtracts the gas counter every half second
            If System.Environment.TickCount - LastTick >= 500 Then
                LastTick = System.Environment.TickCount
                Gas -= 0.5
                ' if the tank is empty, destroy the plane
                If Gas < 0 Then
                    Status = enPlayerStatus.Dying
                End If
            End If
            LastTick = System.Environment.TickCount
        Case enPlayerStatus.Dying
             DyingSound.Play()
            objDirect3DDevice.SetTexture(0, DyingImage)
            objDirect3DDevice.SetStreamSource(0, VertBuffer, 0)
            objDirect3DDevice.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2)
            ' start a new life after 2 seconds
            If System.Environment.TickCount - LastTick >= 3000 Then
                LastTick = System.Environment.TickCount
                Status = enPlayerStatus.Starting
            End If
        Case enPlayerStatus.Starting
            objDirect3DDevice.SetTexture(0, StartingImage)
            objDirect3DDevice.SetStreamSource(0, VertBuffer, 0)
            objDirect3DDevice.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2)
            ' restore the flying status after 3 seconds
            If System.Environment.TickCount - LastTick >= 3000 Then
                Status = enPlayerStatus.Flying
            End If
    End Select
```

```
    '  Turn off alpha blending  if the sprite has transparent colors
    If IsTransparent Then
        objDirect3DDevice.RenderState.AlphaBlendEnable = False
    End If
End Sub
```

That's all for the `Player` class. Now we want to create the player object on the `GameEngine` class, and code the `MovePlayer` method, which will move the player across the screen, as described in the next sections.

Creating the Player Object

We'll create the player object inside the `Initialize` method of the `GameEngine` class. Since the `New` method of the `Player` class takes no parameters, all we need to do is create the object and check if this has been done correctly.

```
        Player = New ClsPlayer()
        If Player Is Nothing Then
            Initialize = False
            Exit Function
        End If
```

In the next section we'll include the code for the `GameEngine` class that will call the `Draw` method of the `Player` class, allowing us to see the player's plane on the screen and move it.

Coding the MovePlayer Method

Before coding the keyboard control and the `MovePlayer` procedure, let's take a step back and review the concepts for applying transformations to objects.

As we were reminded when coding the scrolling feature of the game, to perform transformations on an object defined by its vertices stored in the vertex buffer, all we need to do is set the `Transform.World` property of the device object.

But what if we want to apply different transformations to a certain object? In our specific case, we want the game field to scroll and the player's plane to move according to the keys pressed by the player.

In such a case, all we need to do is to set different world matrices for each object according to the effect we want to have. For this example, we'll loop through the following steps:

1. Set the world matrix to perform the translation to implement the scrolling.

2. Draw the tiles.

3. Set the world matrix to perform the translation, according to the keys pressed, to implement the player's movements (overwriting the previous world matrix).

4. Draw the player.

As we did in the `Scroll` procedure, to control the player we'll use a static `Matrix` to store the player's movements. Looking at the `New` procedure shown previously, we can see that the player coordinates were set to (0,0); and although the vertices will all stay in the position in which they were first created, we'll see the player's plane moving if we set the correct translations.

The value for the translations to be applied in each direction will be set at the `KeyDown` event of the game window, by simply setting the `SpeedX` or `SpeedY` properties (defined in the base class `Sprite`) to the current game speed using negative values where appropriate.

```
Sub GameWindow_KeyDown(sender As Object, e As KeyEventArgs) _
                                         Handles MyBase.KeyDown
     Select Case e.KeyCode
         Case Keys.Right
             RiverPlanet.Player.SpeedX = RiverPlanet.gameSpeed
         Case Keys.Left
             RiverPlanet.Player.SpeedX = -RiverPlanet.gameSpeed
         Case Keys.Up
             RiverPlanet.Player.SpeedY = RiverPlanet.gameSpeed
         Case Keys.Down
             RiverPlanet.Player.SpeedY = -RiverPlanet.gameSpeed
    End Select
End Sub
```

The `MovePlayer` method will follow the basic structure of the `Scroll` method, but it'll use the properties set for the player in the preceding code to build the translation matrix.

```
Public Sub MovePlayer()
    ' Initializes the player position in the middle of screen (x-axis)
    '   and 3 tiles up (y-axis)
    Static PlayerMatrix As MATRIX = Matrix.Translation( _
              10 * clsSprite.IMAGE_SIZE, 3 * clsSprite.IMAGE_SIZE, 0)

    If Player.Status = Player.enPlayerStatus.Flying Or _
       Player.Status = Player.enPlayerStatus.Starting Then
       ' Draw the player sprite, moving acording to the arrow keys pressed
       If Player.SpeedX <> 0 Or Player.SpeedY <> 0 Then
           PlayerMatrix = Matrix.Multiply(PlayerMatrix, _
               Matrix.Translation(Player.SpeedX, Player.SpeedY, 0))
       End If
       ' Reset the speed of the sprite to prevent the plane from moving
       '  after the player releases the arrow keys
       RiverPlanet.Player.SpeedX = 0
       RiverPlanet.Player.SpeedY = 0
    End If
    objDirect3DDevice.Transform.World = PlayerMatrix
    Player.Draw()
End Sub
```

All we need to do now is include a call to the `MovePlayer` method in the `Render` procedure, and we are ready to test our plane by moving it around the screen. By this stage, our game designer should have developed a larger game field, so we can fly around over whole new backgrounds, as shown in Figure 4-20.

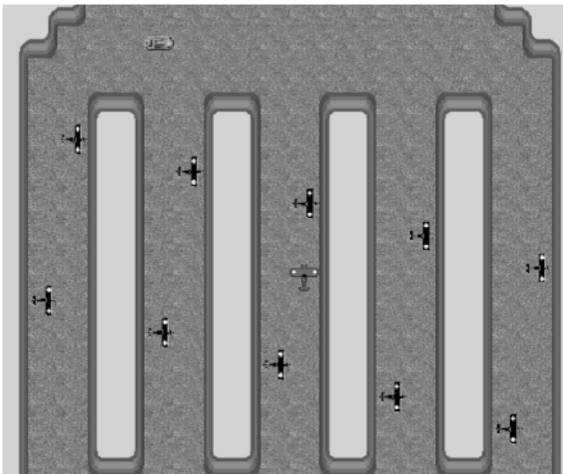

Figure 4-20. Our plane flying over trouble waters

If you were to test this game at this point, one thing you'd discover is that you can drive your plane off the screen. Although you can come back later, it's not a good game practice. So we'd better include some testing on our movement procedure to avoid this, just after the line in which we set the `PlayerMatrix` variable, inside the `if` command:

```
'    the m41 element represents the translation on the X axis
If PlayerMatrix.m41 < clsSprite.IMAGE_SIZE Then _
    PlayerMatrix.m41 = clsSprite.IMAGE_SIZE
If PlayerMatrix.m41 > (Width - 1) * clsSprite.IMAGE_SIZE Then _
    PlayerMatrix.m41 = (Width - 1) * clsSprite.IMAGE_SIZE

'    the m42 element represents the translation on the Y axis
If PlayerMatrix.m42 < clsSprite.IMAGE_SIZE Then _
    PlayerMatrix.m42 = clsSprite.IMAGE_SIZE
If PlayerMatrix.m42 > (Height - 1) * clsSprite.IMAGE_SIZE Then _
    PlayerMatrix.m42 = (Height - 1) * clsSprite.IMAGE_SIZE
```

We can now control the plane within the screen limits.

In next section we'll code the collision detection functions, so the first version of our game will be almost finished.

Fourth Draft: Collision Detection

The collision detection in our game will be fairly simple: We'll use an algorithm that will provide approximate results to make the code simpler. Although it's not very accurate, it'll suffice for fair game play.

The basic idea here is to check the current player position, convert it to (x,y) coordinates of the `Tiles` array, and then check the tile array element we are over, to see if we are colliding. There'll be three types of collisions:

- If we are over water, we aren't colliding.
- If we are over a gas barrel, we aren't colliding, but we'll need to destroy the gas barrel tile, fill our tank with some gas, and create a new tile (with water) to replace the gas tile.
- If we are over a bridge, a ship, or a plane, we are colliding.

The `TestCollision` method will return a Boolean indicating if we are colliding or not, so the `Render` procedure will deal with the collision as appropriate.

One last point before looking at the code for this procedure: As mentioned in the previous draft, when coding the player's movements, the player vertices will always be at the original positions they were created; what we'll do is change the world matrix to see the player in different positions. So, to allow the `TestCollision` procedure to get the current player position, we'll need to update the `X` and `Y` properties of the player as he or she moves. All we need to do is to add the next lines of code to the `MovePlayer` method, just before the lines in which we set the `PlayerMatrix` transformation matrix:

```
' Updates the player location (used in collision detection)
 Player.X = PlayerMatrix.m41
 Player.Y = PlayerMatrix.m42
```

The complete code for the `TestCollision` procedure is shown in the following sample:

```
Private Function TestCollision() As Boolean
    Dim x As Integer, y As Integer
    Dim i As Integer

    x = Player.X / 32
    y = (Player.Y + 16) / 32 + CurrentLineNumber

    ' If we are over water or over a gas barrel, we are not colliding
    If Not (tiles(x, y) Is Nothing) Then
        If tiles(x, y).Type = ClsTile.enType.Water Then
            TestCollision = False
        ElseIf tiles(x, y).Type = ClsTile.enType.Gas Then
            ' Remove the gas barrel from screen
            tiles(x, y).Dispose()
            tiles(x, y) = New ClsTile("water.bmp", _
                 New POINT(x, y), ClsTile.enType.Water)
            TestCollision = False
            Player.Gas = Player.Gas + 30
            If Player.Gas > 100 Then Player.Gas = 100
        Else
            ' If we collide with a ship or a plane, destroy it...
            If tiles(x, y).Type = ClsTile.enType.Plane Or _
               tiles(x, y).Type = ClsTile.enType.Ship Or _
               tiles(x, y).Type = ClsTile.enType.Bridge Then
                tiles(x, y).Dispose()
                tiles(x, y) = New ClsTile("water.bmp", _
                     New POINT(x, y), ClsTile.enType.Water)
```

```
                End If
                TestCollision = True
            End If
        Else
                TestCollision = True
        End If
    End Function
```

The code for the `Render` procedure will have to deal with the results of the `TestCollision` procedure, changing the player status and removing one life from the game's `Lifes` property, as shown in the following code sample:

```
Public Overrides Sub Render()
    ' Scrolls the game field and moves the player
    Scroll()
    Draw()
    MovePlayer()

    ' Only tests for collision if flying
    If Player.Status = Player.enPlayerStatus.Flying Then
        ' If there's a collision, set player status to dying
        If TestCollision() Then
            Player.Status = Player.enPlayerStatus.Dying
            Lifes -= 1
            If Lifes = 0 Then
                GameOver = True
            End If
        End If
    End If
End Sub
```

We can now run the game's new version, flying more carefully because now we are flying at lower altitude, as shown in Figure 4-21.

Figure 4-21. The plane now collides with any solid obstacles—in this case, a bridge

Final Version: Music and Sound Effects

Since our base sound manipulation library is coded, the task of including sounds in our application is very simple. All we need to do is to create the sound objects and call them as appropriate. Since we want to play some motifs randomly over the background music, we'll code the `PlayMotifs` method, as defined in the game project, to do so.

The `BackgroundMusic` object and the `GasSound` object must be created in the `Initialize` method of the `RiverEngine` class, so they'll be accessible to all other methods. As for the background music, we can start playing it right after the object creation; it'll be looping until the game end.

```
' Start the background music
BackgroundMusic = New ClsGameMusic()
BackgroundMusic.Initialize(WinHandle)
If Not BackgroundMusic.Load("boidsd.sgt") Then
    MessageBox.Show("Error loading background music", "River Pla.Net")
End If
BackgroundMusic.Play()

' Initializes the gas filling sound effect
GasSound = New ClsGameSound(Owner)
If Not GasSound.Load("FillGas.wav") Then
    MessageBox.Show("Error loading Gas sound effect", "River Pla.Net")
End If
```

As for the player game effects, we need to add the object creation to the New procedure of the Player class:

```
' Initializes the sound effects
DyingSound = New ClsGameSound(Owner)
If Not DyingSound.Load("explosion.wav") Then
    MessageBox.Show("Error loading explosion sound effect", "River Pla.Net")
End If
StartingSound = New ClsGameSound(Owner)    If Not
StartingSound.Load("init.wav") Then
    MessageBox.Show("Error loading starting sound effect", "River Pla.Net")
End If
```

Once we have created the sound objects, all we need to do is call the Play method of each object where appropriate.

- In the TestCollision procedure, when the player collides with a gas barrel, we'll play the "gas bonus" sound.

```
GasSound.Play()
```

- In the Draw method of the Player class, we'll play the "dying" sound every time the player has a status of Dying.

```
DyingSound.Play()
```

- In this same method, we'll play the "starting a new life" sound every time the player has a status of Starting.

```
StartingSound.Play()
```

This will suffice to add music and sound effects to our game. And to add that little bit extra, for subtle variations in the background music from time to time, we'll code the PlayMotifs function. This function will be called with every frame that's drawn on the Render method, so we'll include two random choices: first, choosing a random time (let's say, between 5 and 15 seconds) to wait for the next motif to play, and choosing a random motif to play, using the PlayMotif method of our GameSound class and passing an index between zero and the value of the MotifCount property, as shown in the next code sample:

```
Sub PlayMotifs()
    Dim MotifIndex As Integer
    Static LastTick As Integer
    Static Interval As Integer

    ' Plays a random motif every 5 to 15 seconds
    If System.Environment.TickCount - LastTick >= Interval Then
        LastTick = System.Environment.TickCount
        ' Gets a new random interval (in miliseconds) to play the next motif
        Interval = (Rnd() * 10 + 5) * 1000
        MotifIndex = Rnd() * BackgroundMusic.MotifCount
        BackgroundMusic.PlayMotif(MotifIndex)
    End If
End Sub
```

And that's all for this chapter. The game is up to the standard described in the game project. But there are a lot of improvements we can make, as shown in the next section and in the next game version, in Chapter 5, when we'll introduce DirectInput and joystick control.

Adding the Final Touches

We'll code a second version of our game in the next chapter, with many improvements, but there is already some upgrading we can do right now, as shown in the next sections.

Including Player Animations

A good improvement would be to include some player animations for dying and starting a new life.

Animations are only a set of images that are presented, one at a time, using specific time intervals. To define an animation, we should take into account the total time we'll have to play the animation and the number of frames we want to display.

For the total time for each animation, we can simply check the duration of each sound effect: about 1 second for the explosion sound, and about 2 seconds for the sound of starting a new life.

To create an interesting explosion animation, we'll need as many images as possible. Figure 4-22 shows a minimal set for an explosion animation.

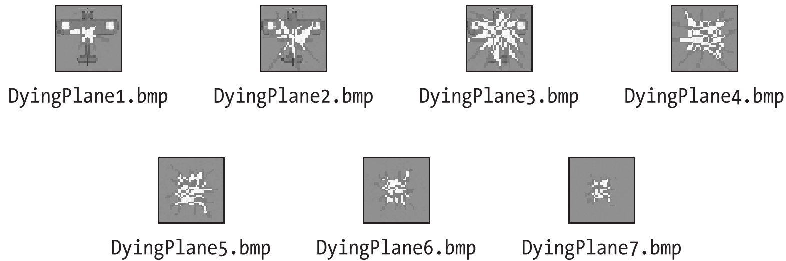

Figure 4-22. Explosion images for dying animation

Since we have seven images, we can calculate the desired interval between each image: about 0.15 seconds.

Figure 4-23 shows a second set of images that will be used to give the player a visual clue that the plane is invincible when starting a new life.

Figure 4-23. Flashing planes for starting a new life animation

In this case, we can use a different approach: Let's simply show the images from the first to the fourth, and then from the fourth down to the first, so the animation will appear to be flashing to the player.

To implement the animations, we'll need to change the `Player` class as follows:

- Change the `DyingImage` and the `StartingImage` properties from variables to arrays.
- Adjust the `New` method to dimension the arrays to the appropriated values.
- On the `New` method, load each of the images to the corresponding array position.
- On the `Draw` method, include the code for displaying the images one at a time, taking into account the specified interval between images.

The modifications of the `Player` class are shown in the following code listing:

```
Protected DyingImage() As Direct3DTexture8
Protected StartingImage() As Direct3DTexture8
Sub New()
    ReDim DyingImage(7)
    ReDim StartingImage(4)
    Dim colorKey As Integer
    Dim i As Integer

    colorKey = Color.fromARGB(255, 255, 0, 255)
     . . .
    Try
       For i = 1 To 7
           DyingImage(i - 1) = TextureLoader.FromFile(objDirect3DDevice, _
                         Application.StartupPath & "\" & IMAGE_PATH & _
                          "\dyingPlane" & i & ".bmp", _
                          64, 64, D3DX.Default, 0, Format.Unknown, Pool.Managed, _
                          Filter.Point, Filter.Point, colorKey.ToArgb)
       Next
       For i = 1 To 4
           StartingImage(i - 1) = TextureLoader.FromFile(objDirect3DDevice, _
                           Application.StartupPath & "\" & IMAGE_PATH & _
                          "\startingPlane" & i & ".bmp", _
                          64, 64, D3DX.Default, 0, Format.Unknown, Pool.Managed, _
                          Filter.Point, Filter.Point, colorKey.ToArgb)
       Next
    Catch
        MsgBox("Could not create the player textures", MsgBoxStyle.Critical)
    End Try
 . . .
End Sub

Shadows Sub Draw()
    Static CountAnim As Integer = 0
    Static LastTick As Integer = 0
    Static IncAnim As Integer = 1
    . . .
    Select Case Status
        Case enPlayerStatus.Flying
            . . .
        Case enPlayerStatus.Dying
            If CountAnim = 0 Then
                DyingSound.Play()
```

```
            End If
            ' Each frame will be shown for .15 seconds,
            '  the 7 frames of the explosion in about 1 second
            If System.Environment.TickCount - LastTick >= 150 Then
                LastTick = System.Environment.TickCount
                CountAnim += 1
            End If
            objDirect3DDevice.SetTexture(0, DyingImage( _
                   IIf(CountAnim - 1 < 0, 0, CountAnim - 1)))
            objDirect3DDevice.SetStreamSource(0, VertBuffer, 0)
            objDirect3DDevice.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2)
            ' The dying animation is 7 frames long
            If CountAnim = 6 Then
                CountAnim = 0
                Status = enPlayerStatus.Starting
            End If
        Case enPlayerStatus.Starting
            If CountAnim = 0 Then
                StartingSound.Play()
            End If
            objDirect3DDevice.SetTexture(0, StartingImage(CountAnim))
            objDirect3DDevice.SetStreamSource(0, VertBuffer, 0)
            objDirect3DDevice.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2)
            ' The starting animation is 4 frames long,
            '   and must run in a reverse loop
            If CountAnim = 3 Then IncAnim = -1
            If CountAnim = 0 Then IncAnim = 1

            ' Each frame will show a different frame of the animation
            CountAnim += IncAnim

            ' restore the flying status after 4 seconds
            If System.Environment.TickCount - LastTick >= 4000 Then
                CountAnim = 0
                Status = enPlayerStatus.Flying
                ' We have a new plane, fill the tank!
                Gas = 100
            End If
    End Select
   . . .
End Sub
```

Implementing a Neverending Game Map

So we managed to define a map with several hundreds of tiles. And what happens when the user reaches the end of the game map?

Since we'll have no ending screen, we can use a little trick to make our game field infinite in length.

Adding some code to reset the scroll translation matrix to the beginning of the game map when we reach the end will make the player loop forever on our game. To allow a smooth transition, we can copy the first 15 lines of the game to the end of the game map, so when we return to the beginning the player won't notice a difference.

We can add an extra degree of playability to our game by including the concept of different phases: Every time the player reaches the end of the map, we can increase the game speed, so that even though he or she starts the same game field, the game increases in difficulty.

To do this we'll need to change the code for the `Scroll` method, including a new test within the `if` command that increments the current line number counter, to reset the scroll matrix and increase the game speed (using a new constant, `gameSpeedIncrease`), as shown in the next code lines:

```
Private gameSpeedIncrease As Single = 1.3
. . .
' If we ended our game map, start it all over again, but with increasing speed
If CurrentLineNumber + Height = GameMapSize Then
    gameSpeed = gameSpeedIncrease * gameSpeed
    ' The maximum gameSpeed will be the size of a tile per frame
    If gameSpeed > 32 Then gameSpeed = clsSprite.IMAGE_SIZE
    ScrollMatrix = Matrix.Identity
    CurrentLineNumber = 0
End If
```

In the next chapter we'll see some more improvements when we code the second version of River Pla.Net.

Improving the Performance

Taking our sample game as an example, we can see that we are spending a lot of time drawing each tile by itself. Looking at the `Draw` method of the `Tile` class, we can see that for every tile we are calling three functions:

```
objDirect3DDevice.SetTexture(0, SpriteImage)
objDirect3DDevice.SetStreamSource(0, VertBuffer, 0)
objDirect3DDevice.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 2)
```

In commercial games we'll usually want a higher frame rate, so we need to set aside the simplicity and use higher performance algorithms.

A simple way to speed up the game is to group equal tiles together, in a big vertex buffer, so we could call these three functions only once for each texture. Since the `DrawPrimitives` function can receive the first vertex to draw and the number of primitives (triangle strips, in our case), all we need do is store a vertex number in the `Tile` class, so we can pick the first tile and the last tile of each type on screen and calculate the values for the `DrawPrimitives` function.

Since our main goal here is to introduce the gaming concepts, we didn't spend time on optimizations; in the next chapter we'll include extra features in our game, such as joystick control, but the game engine will remain basically the same.

Summary

In this chapter, we managed to use the Direct3D concepts discussed in the previous chapter to create an interesting new game, River Pla.Net. Among the many new points learned are the following:

- An introduction to DirectAudio library, including the basic concepts about music and sound reproduction through the DirectSound and DirectMusic interfaces.

- The creation of a new game library, including two graphic classes (`Sprite` and `GameEngine`) and two audio classes (`GameSound` and `GameMusic`).

- How to employ some advanced object-oriented concepts in programming, like the use of overrideable functions.

- The introduction of two new game concepts, tile-based game fields and scrolling games, and a practical example of their use.

In the next chapter, we'll include some enhancements in our game, introducing two new concepts indispensable in every game: input device control with DirectInput, including the use of force feedback in joysticks, and the practice of writing text on the device context screen used by Direct3D.