Beginning .NET Game Programming in C#

DAVID WELLER, ALEXANDRE SANTOS LOBÃO, AND ELLEN HATTON

Apress™

Beginning .NET Game Programming in C#

ISBN (pbk): 1-59059-319-7

Printed and bound in the United States of America 12345678910

Technical Reviewers: Andrew Jenks, Kent Sharkey, Tom Miller

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Julian Skinner, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Sofia Marchant

Copy Editor: Ami Knox

Production Manager: Kari Brooks

Production Editor: Janet Vail

Proofreader: Patrick Vincent

Compositor: ContentWorks

Indexer: Rebecca Plunkett

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

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

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

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

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

CHAPTER 5

Spacewar!

Back in December of 2001, Eric Gunnerson coded a C# implementation of the classic game Spacewar using the interop capabilities of .NET to access the DirectX 7 API. In May of 2002, he uploaded the code to help launch the .NET community Web site, GotDotNet (`http://www.gotdotnet.com`). Eric is a Microsoft Visual C#.NET program manager and author of the Apress book *A Programmer's Introduction to C#, Second Edition.* Eric also writes a C# column for MSDN Online called "Working with C#."

With the release of the Managed DirectX libraries, we decided it would be fun to update the Spacewar game. If you would like to see Eric's original Spacewar DirectX 7 source code, you can find it at the GotDotNet site in the sample code section. In addition, this version of the Spacewar game, shown in Figure 5-1, can be found on the GotDotNet site as Spacewar2D (`http://workspaces.gotdotnet.com/spacewar2d`).

Figure 5-1. Spacewar2D splash screen

About Spacewar

Spacewar was conceived in 1961 by Martin Graetz, Stephen Russell, and Wayne Wiitanen. It was first realized on the PDP-1 at MIT in 1962 by Stephen Russell, Peter Samson, Dan Edwards, and Martin Graetz, together with Alan Kotok, Steve Piner, and Robert A. Saunders (see Figure 5-2). It's widely credited as the first video game. Although the graphics are primitive compared to today's standards, the game play is still outstanding even after more than 40 years! (Figure 5-3 shows the coin-operated version.)

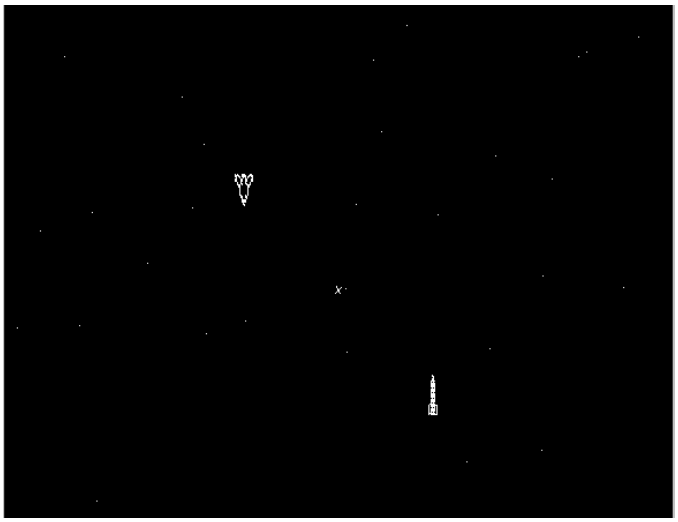

Figure 5-2. The original Spacewar game

Figure 5-3. The coin-operated version of Spacewar

Deciding What to Change

Although we wanted to bring the game up to date with respect to the source code, we didn't want to alter the gameplay. In fact, we've worked pretty hard to make the game faithful to the coin-operated version. Scott Haynie, a game developer with great Managed DirectX experience, came to us about porting the Spacewar game to Managed DirectX. After looking over Eric's source, we made a list of the changes to make to the code. The obvious first choice was to remove any DirectX 7 interop calls and replace them with Managed DirectX methods and classes. For network play, Eric used the System.Net.Sockets namespace. Although we could have left the networking piece alone, we felt that using DirectPlay (which was covered in the previous chapter) was a more logical choice for the upgrade. The same applies to the input and audio classes. Instead of the custom KeyEvent handlers, we could now use DirectInput and the Managed DirectSound namespace.

Those are the major changes, but we made a few others design decisions along the way:

- *Eliminating the use of pointers:* Eric used the pointers in the original version to increase speed in the network routines, but using DirectPlay eliminates that problem. In addition, using "unsafe" programming in C# should always be a last resort.

- *Scoring:* Scott was up late one night working on converting the player update code, and was getting a headache trying to figure out how the points got passed in the ship update packets. He cut them out of the ship updates and made each score update a separate, guaranteed delivery packet. In the process, he introduced an interesting side effect: It reduced the network load because it only sends a 1-byte score update message when someone dies. A very nice side effect!

- *Game configuration:* We created a GameSettings class to allow you to change the way the game plays without recompiling. This made it easy to add some of the options from the coin operated game, such as

  - Variable gravity

  - Inverse gravity

  - Variable game speed

  - Bouncing off the game boundaries

  - Black hole (invisible sun)

Originally, we considered converting the ship graphics over to sprites and just animating them, but the clever line drawing classes Eric put together for the font and the ships looked pretty darn good, and added to the retro feel that we wanted to preserve in the game, so they remain the same as he originally coded them.

Methodology: Challenges of Working with Someone Else's Code

Working with someone else's code is always interesting. You get to see where they found a really clever solution to a problem, and also where they just decided to hack in what works. Nothing can be more thrilling, and sometimes more frustrating, than trying to figure out the original programmer's mindset when they wrote a particular block of code. Did they know a better way? Is this a new technique that should be learned? Or was it a late night hack? Converting Spacewar taught us a couple of tricks, and hopefully you'll pick up on them as you go along. One of the really nice benefits of Spacewar is that it was originally written by a C# veteran (Eric) inside of Microsoft, which gives you some great insight into how Microsoft intended you to use C#.

You'll also find that everyone has their own coding style—their naming conventions, where they put the braces, etc. The completed source code reflects the style used in this book, but you'll see a slightly different style used in Eric's original code. You can see it most notably in brace alignment and property declarations. When working on your own, use whatever style suits you, but if you're working on a team, or you intend to publish your code, make sure the style you use is consistent and readable. (One of the authors, many years ago, had to debug a 5000-line code program written by a programmer who used *no* indentation whatsoever!)

Where to Start?

There are two choices when you start to update the code. Either open Eric's solution file and just start replacing the parts that needed updating, or start a brand new solution, and gradually add in the classes. Although it's tempting to dive right in and work with existing code, it becomes very difficult to grok all the code at once, especially when you have many class files interacting with one another. Starting fresh lets you build up the solution methodically, importing classes as they are needed. It also allows you to start out with the DirectX 9 Visual C# Wizard—a great time-saver.

Using the Application Wizard

If you haven't used the DirectX 9 Application Wizard, you should read this section. The DirectX 9 SDK sets up the Application Wizard during its installation. This is a great way to kick start your DirectX application. To start a DirectX project in Visual Studio .NET 2003, select New Project from the File ➤ New menu to bring up the dialog box shown in Figure 5-4.

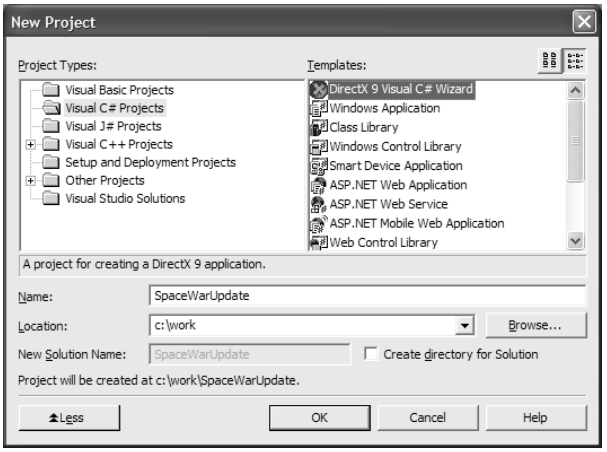

Figure 5-4. Creating a new DirectX application

Enter the project name and location, and click OK. In the wizard, select Project Settings, and then choose DirectDraw, DirectInput, and DirectPlay as shown in Figure 5-5.

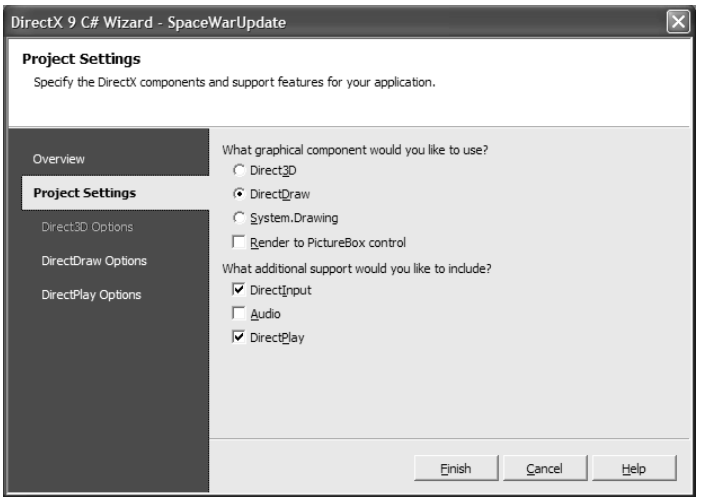

Figure 5-5. Choosing the application options

What Is DirectDraw?

DirectDraw is a programming API that was originally designed to make it easy to create 2-D graphics applications. Because DirectX 8 was released in 2001, most of the DirectDraw API was incorporated in new Direct3D interfaces. The Managed DirectX API offers a DirectDraw class hierarchy in order to help code migrations from earlier versions of DirectX, but you should use Direct3D now, even when doing 2-D graphics (see the discussion in Chapter 4 regarding the Sprite class).

All that is left is to click the Finish button, and the wizard creates a fully working DirectX application. You'll use this code as the base, and gradually import the Spacewar classes. Each time you add a file to the solution from Eric's version, you compile the code and see what was missing. Then you replace the code that accessed the old DirectX7 interface, and try the compile again. This kind of trial-and-error approach isn't ideal for a real software project, but it's a reasonable approach for this specific problem.

One thing you don't do is tell the wizard to create DirectSound interfaces for the game. This was a design decision, because we found Eric's SoundHandler class to be perfectly sufficient, and converting it to use the managed interface was very simple.

Let's look at some of the code. We won't go over every class, but both the original DX7 version and the updated DX9 version are available for you to review.

Main Class

The Main class is the entry point for the application. It creates a form to render your graphics, sets up double-buffering so your game won't flicker, initializes the Game class, and then calls the main game loop over and over until it receives an exit request.

```
public MainClass() {
   //
   // Required for Windows Form Designer support.
   //
   InitializeComponent();
   target = this;
```

```
   // Add event handlers to enable moving and resizing.
   this.MouseDown += new MouseEventHandler(OnMouseDown);
   this.MouseMove +=new MouseEventHandler(OnMouseMove);
   this.MouseUp += new MouseEventHandler(OnMouseUp);
   this.Resize += new System.EventHandler(OnMoveResize);
   this.Move +=new EventHandler(OnMoveResize);


   // Set up double buffering to eliminate flicker.
   SetStyle(ControlStyles.DoubleBuffer, true);
   SetStyle(ControlStyles.AllPaintingInWmPaint, true);
   SetStyle(ControlStyles.UserPaint, true);

   // Initialize the main game class.
   game = new GameClass(this, target);
   game.Initialize(this.Bounds);

   // Show your game form.
   this.Show();

   // Start the main game loop.
   StartLoop();
}
```

The StartLoop() method is a very tight loop that calls your game's logic and rendering method. The call to Application.DoEvents() processes all of the Windows event messages that have queued up. The Thread.Sleep(2) line allows other processes on the computer to execute so that your game isn't completely saturating the processor. Try running the game with the Task Manager open to the Performance tab both with and without the Thread.Sleep() call to see the difference.

```
private void StartLoop() {
   while(Created) {
      game.MainLoop();  // Execute the game logic.
      Application.DoEvents();  // Take care of all the windows event messages.
      Thread.Sleep(2);  // Yield some CPU time to other applications.
   }
}
```

Why Is the Loop Different?

The observant reader will notice that this game loop construct is different from the Space Donuts example in Chapter 4. There are actually several different ways to do game loops, each with trade-offs between simplicity and performance. The original Spacewar game also used a completely different approach from the example shown here.

GameClass

The GameClass is where the interesting things happen. It creates the DirectDraw graphics device, instantiates the input, network, and sound handlers, creates the ships, sets up the game options form, starts the frame timer, and handles drawing the scene.

The following is the contructor for this class:

```
public GameClass(MainClass mainClass, Control owner) {
   gameState = GameStates.Loading;
   this.owner = owner;
   this.mainClass = mainClass;
   splash = new SplashScreen(this);
   splash.ShowDialog();
   gameSettings = new SpaceWar.GameSettings(this);
   gameSettings.Location = new Point(owner.Bounds.Right,owner.Bounds.Top);
   gravity = gameSettings.Gravity;
   gameSpeed = gameSettings.GameSpeed;
   bounceBack = gameSettings.Bounce;
   inverseGravity = gameSettings.InverseGravity;
   blackHole = gameSettings.BlackHole;

   localDevice = new Microsoft.DirectX.DirectDraw.Device();
   localDevice.SetCooperativeLevel(owner,
   Microsoft.DirectX.DirectDraw.CooperativeLevelFlags.Normal);

   DXUtil.Timer(DirectXTimer.Start);

   SpaceWar.RotatableShape.CreateShapes();

   input = new InputClass(this.owner);

   soundHandler = new SoundHandler(this.owner);
```

```
   try {
      netPeer = new PlayClass(this);
   }
   catch(DirectXException e) {
      MessageBox.Show(owner,e.ToString());
   }
}
```

Initializing the Game Class

The GameClass Initialize() method positions the sun in the middle of the game screen, creates your ship, sets up the background star field, and checks the network status to see if you should enable the game settings form controls. If you're connected to another host, your controls are locked out, and only the host can set the game options. Notice the call to CreateSurfaces(). You create the DirectDraw surfaces in a separate routine so that you can call it again without resetting the game. Any time your DirectDraw device is unable to access the surface's memory, it throws a SurfaceLost exception. You catch this in your drawing routine, and call CreateSurfaces() again as follows:

```
public void Initialize(Rectangle bounds) {
   owner.Bounds = bounds;
   this.GameState = GameStates.Config;
   this.windowBounds = bounds;
   CreateSurfaces();

   sunLocation.X = windowBounds.Left + windowBounds.Width / 2;
   sunLocation.Y = windowBounds.Top + windowBounds.Height / 2;

   ship = new Ship(this);
   Random random = new Random((int) DateTime.Now.Ticks);
   ship.ScreenBounds = bounds;
   ship.SetRandomPosition(true, sunLocation);

   if ((null != localPlayer.Name) && (localPlayer.Name.Length > 0))
      ship.HostName = localPlayer.Name.ToUpper();
   else
      ship.HostName = System.Environment.MachineName.ToUpper();

   stars = new Stars(bounds, Constants.NumStars);
   sun = new Sun(sunLocation, Constants.SunSize);
```

```
            gameSettings.ControlsEnabled = true;
            if (netPeer.InSession) {
                if (netPeer.IsHost) {
                    gameSettings.ControlsEnabled = true;
                    netPeer.SendGameState(GameStates.Running);
                }
                else {
                    gameSettings.ControlsEnabled = false;
                }
            }
            gameState = GameStates.Running;
        }
```

Here in the CreateSurfaces() method, you create two surfaces: your primary, which is what is visible, and the secondary in which you accumulate all of the objects being drawn. You'll see when you get to the render method that you draw everything on the secondary surface, and then draw (copy) the whole buffer at once to the primary surface.

```
        private void CreateSurfaces() {
            SurfaceDescription desc = new SurfaceDescription();
            SurfaceCaps caps = new SurfaceCaps();

            localClipper = new Clipper(localDevice);
            localClipper.Window = owner;

            desc.SurfaceCaps.PrimarySurface = true;
            if (null != surfacePrimary)
                surfacePrimary.Dispose();
            surfacePrimary = new Surface(desc, localDevice);
            surfacePrimary.Clipper = localClipper;

            desc.Clear();
            desc.SurfaceCaps.OffScreenPlain = true;
            desc.Width = surfacePrimary.SurfaceDescription.Width;
            desc.Height = surfacePrimary.SurfaceDescription.Height;

            if (null != surfaceSecondary)
                surfaceSecondary.Dispose();
            surfaceSecondary = new Surface(desc, localDevice);
            surfaceSecondary.FillStyle = 0;
        }
```

The Main Game Loop

MainLoop() is a public method that gets called from the Main class StartLoop() method. This is where all of the per-frame logic happens.

The first thing it does is check to see how much time has elapsed since the last frame. If not enough time has passed, it returns. The DXUtil class has a static method named Timer that makes it really easy to track frame times. You started the timer back in the GameClass constructor, so DXUtil.Timer(DirectXTimer.GetElapsedTime) will return the elapsed time since the last GetElapsedTime call. The minimum amount of time between frames is adjustable via the GameSpeedSlider control on the GameSettings form. If you don't control the time between frames, all of the ships in the game will move and rotate at different speeds. On fast machines, the ships become almost impossible to control. Try setting minFrameTime to 0 and see for yourself.

NOTE *Limiting the frame rate is usually not the best option in your games. Basically you are throwing away CPU and GPU time that could be used for other things such as more AI calculations, better physics, more advanced effects, etc.*

But if you don't limit the frame rate, how do you ensure that all players are able to move at the same speed?

*The secret is to multiply all movement and rotation by the elapsed time. Instead of moving your ship x units, you should move it x*elapsedTime units. That way if one player has a really slow machine, the elapsed time between frames on that machine will be larger, but the player in question will still move the same amount as the other players. Players with fast machines just enjoy a smoother game.*

That said, limiting the frame rate is the best way to go in Spacewar because you don't have an AI, and all of the graphics are based on rotating lines a set integer number of steps, not on arbitrary floating-point angles.

Here is the first part of the MainLoop() method:

```
public void MainLoop() {
   float minFrameTime = gameSettings.GameSpeed * 0.005f;

   if (lastFrameTime < minFrameTime) {
      lastFrameTime += DXUtil.Timer(DirectXTimer.GetElapsedTime);
      return;
   }
   lastFrameTime = 0.0f;
   …
```

If you get past the frame time check, the application sets the lastFrameTime to 0, and then makes sure that the game isn't paused. If the game is paused, it displays the "PAUSED" message near the ship names. Then you check for input with the HandleKeys() method, and the ship's sound flags are cleared. If the game is in any state except GameStates.Running, you don't need to update the screen, so you return. If the game is running, you fill the buffer with black, and update the ship's position and state. Notice that you enclose the whole drawing section in a try/catch block so that you can intercept the SurfaceLost exceptions.

```
try {
   if (gameState == GameStates.Paused) {
      Word paused = new Word("PAUSED", Constants.LetterSize * 1.5f);
      paused.Draw(surfaceSecondary, Color.White.ToArgb(),
                  Constants.LetterSpacing * 2,
                  new Point(windowBounds.Left + 50, windowBounds.Top + 50));
      surfacePrimary.Draw(surfaceSecondary, DrawFlags.DoNotWait);
   }
   // Clear the ship's sound flags.
   ship.Sounds = (Sounds) 0;

   // Process input.
   HandleKeys();


   if (gameState != GameStates.Running) return;

   surfaceSecondary.ColorFill(Color.Black);
   surfaceSecondary.DrawWidth = 1;

   // Update my position, and tell others about it...
   ship.UpdatePosition();

   // Update my state, and draw myself...
   ship.UpdateState();
```

Next you check to see if you have a network session, and if so, send your ship data to the other players. Then you draw the scores, your star background, and your ship onto the back buffer.

```
// If there are other players, send them your ship info.
if (netPeer.InSession  && otherPlayers.Count > 0)
   SendMyPlayerUpdate();
```

```
WriteScores();
stars.Draw(surfaceSecondary);

int shipColor = Color.White.ToArgb();
int shotColor = Color.White.ToArgb();
ship.Draw(surfaceSecondary, shipColor, shotColor);
```

Now that you've drawn your own ship, you need to loop through all of the other player ships, draw them, and do your collision detection routines.

```
// Handle other ships.
// Walk through all other players. For each player
// 1) Draw the ship.
// 2) Check to see whether the other ship has killed you.
// 3) Figure the score.
// 4) See if you need to time-out this ship.
int shipIndex = 0;
Sounds otherShipSounds = (Sounds) 0;
DateTime now = DateTime.Now;
lock (otherPlayers) {
   foreach (RemotePlayer player in otherPlayers.Values) {
      if (!player.Active)
         continue;


player.Ship.Draw(surfaceSecondary,shipColors[shipIndex].ToArgb(),shotColor);
      shipIndex = (shipIndex + 1) % shipColors.Length;
      ship.TestShip(player);
      otherShipSounds |= player.Ship.Sounds;

      // If you haven't gotten an update in a while,
      // mark the player as inactive...
      TimeSpan delta = now - player.UpdateTime;
      if (delta.Seconds > Constants.RemoteTickTimeout) {
         player.Active = false;
      }
   }
}
```

After checking and drawing all of the other players, you draw the sun if necessary, and then draw the whole secondary buffer up to the primary so you can see it.

```
   …
   // Draw the sun only if the "Black Hole" option isn't enabled.
   if (!blackHole)
      sun.Draw(surfaceSecondary);

   surfacePrimary.Draw(surfaceSecondary, DrawFlags.DoNotWait);
   PlaySounds(otherShipSounds);
}
catch(SurfaceLostException) {
   // The surface can be lost if power saving
   // mode kicks in, or any other number of reasons.
   CreateSurfaces();
}
```

That's it for the main loop. Let's take a quick look at the HandleKeys() method you call to process the input. If you have started an application with the DirectX Wizard, you may notice that it's structured a little differently from how the wizard sets it up for you. When you use the wizard and check the option to include DirectInput, it will create a class file called dinput.cs for you, and add a delegate to your main class for sending input messages to your application. When it's set up this way, all of your key checking logic goes in InputClass. It's better to do the key checks in the game class, so you'll create a public method called GetKBState() in InputClass that will return the current keyboard state. This way all key checks will live in the HandleKeys() method of the game class, and the input class is more generic and can be used in other projects. Here is the GetKBState method. Notice you add the Thread.Sleep(2) calls again. These prevent your game from choking the processor by trying to reacquire the keyboard device too often.

```
public KeyboardState GetKBState() {
   KeyboardState state = null;
   try {
      state = localDevice.GetCurrentKeyboardState();
   }
   catch(InputException) {
      do {
         Application.DoEvents();
         try{ localDevice.Acquire(); }
         catch (InputLostException) {
            Thread.Sleep(2);
            continue;
         }
         catch(OtherApplicationHasPriorityException) {
            Thread.Sleep(2);
            continue;
```

```
            }

            break;

         }while( true );
      }
      return state;
   }
```

HandleKeys() is just a simple series of conditionals to see if any of the keys you care about are currently pressed.

```
private void HandleKeys() {
   KeyboardState keyboardState = input.GetKBState();


   if (null == keyboardState)
      return;

   if (keyboardState[Key.LeftArrow])
      ship.RotateLeft();
   if (keyboardState[Key.RightArrow])
      ship.RotateRight();

   ship.SetThrust(keyboardState[Key.UpArrow]);

   if (keyboardState[Key.LeftControl]  || keyboardState[Key.RightControl])
      ship.Shoot();

   if (keyboardState[Key.Space])
      ship.EnterHyper();

   // Game configuration / Pause key.
   // The configuration controls are disabled if you are connected to another host.
   if (keyboardState[Key.F2]) {
      Pause();

      if (!netPeer.InSession || netPeer.IsHost)
         gameSettings.ControlsEnabled = true;
      else
         gameSettings.ControlsEnabled = false;

      gameSettings.Show();
   }
```

```
        // Sound keys.
        if (keyboardState[Key.F5]) {
            ship.Sounds |= Sounds.Taunt;
        }

        if (keyboardState[Key.F6]) {
            ship.Sounds |= Sounds.Dude1;
        }

        if (keyboardState[Key.F7]) {
            ship.Sounds |= Sounds.Dude2;
        }

        if (keyboardState[Key.F8]) {
            ship.Sounds |= Sounds.Dude3;
        }

        if (keyboardState[Key.F9]) {
            ship.Sounds |= Sounds.Dude4;
        }

        if (keyboardState[Key.F10]) {
            ship.Sounds |= Sounds.Dude5;
        }

        // Exit if Escape key is pressed.
        if (keyboardState[Key.Escape]) {
            End();
        }

    }
```

That's all there is to the input section. If you want to make the game more user friendly, you could expand the GameSettings form to allow users to change the control keys. You would set up an enumeration describing the function of the keys, and map your default keys to the enumeration. Say you created your enumeration and called it ControlKeys. Then instead of the following:

```
if (keyboardState[Key.Space])
    ship.EnterHyper();
```

you would check the input with this:

```
if (keyboardState[ControlKeys.HyperSpace])
   ship.EnterHyper();
```

Allowing configurable input is always a good idea, as everyone has a different opinion on the best keys to use. You'll notice that the keys used in this version of Spacewar are different from Eric's original version.

Direct Play

One of the more extensive renovations in the upgrade was replacing the System.Net.Sockets namespace with DirectX.DirectPlay. The DirectX 9 Visual C# Wizard creates the PlayClass for you and configures it to send 1-byte messages back to the Main class. With this to build on, you move the PlayerCreated, PlayerDestroyed, and MessageReceived event handlers into GameClass, and add methods to the PlayClass for sending out your player and score updates. Let's take a look at the PlayClass constructor (found in dplay.cs):

```
public PlayClass(GameClass Game) {
   this.game = Game;
   this.peerObject = peerObject;

   // Initialize your peer-to-peer network object.
   peerObject = new Peer();

   // Set up your event handlers (you only need events for the ones you care about)
   peerObject.PlayerCreated +=
      new PlayerCreatedEventHandler(game.PlayerCreated);
   peerObject.PlayerDestroyed +=
      new PlayerDestroyedEventHandler(game.PlayerDestroyed);
   peerObject.Receive +=
      new ReceiveEventHandler(game.DataReceived);
   peerObject.SessionTerminated +=
      new SessionTerminatedEventHandler(SessionTerminated);

   // Use the DirectPlay Connection Wizard to create your join sessions.
   Connect = new ConnectWizard(peerObject, AppGuid, "SpacewarDX9");
   Connect.StartWizard();

   inSession = Connect.InSession;

   if (inSession) {
      isHost = Connect.IsHost;
   }
}
```

The constructor sets up event handlers for the events you care about, and then calls the DirectPlay Connection Wizard StartWizard() method. The wizard takes care of enumerating the service providers and searching for hosts, so all you have to do is check to see if you're connected, and if so, whether you host the session.

You'll see the event handlers in the GameClass later in this section. Right now, let's examine the message-sending functions in the PlayClass. Initially, there were four different message types planned for the game: a game parameter update, a paused game state, a running game state, and a player update. However, you want to add separate messages to support updating the game score. All of the methods are very similar. They first make sure that you have a network session, and then they create a new network packet. The first byte you write to the network packet is your message type, so that when your DataReceived event handler receives the packet, it knows how to decode the message that follows. To simplify sending the message type, you create an enumeration as follows:

```
public enum MessageType:byte {
   PlayerUpdateID,
   GameParamUpdateID,
   GamePaused,
   GameRunning,
   Add1ToScore,
   Add2ToScore
}
```

You can see how you use it here in the SendGameParamUpdate() method:

```
public void SendGameParamUpdate(GameParamUpdate update) {
   if (inSession) {
      NetworkPacket packet = new NetworkPacket();
      packet.Write(MessageType.GameParamUpdateID);
      packet.Write(update);
      peerObject.SendTo((int)PlayerID.AllPlayers, packet, 0,
        SendFlags.Guaranteed | SendFlags.NoLoopback);
   }
}
```

After you write out the MessageType byte, the actual update data gets written to the packet. You add a GameParamUpdate struct to make handling the data easier. The last line of the method tells DirectPlay to send the message. The first parameter of the SendTo() method is the recipient. You want to update all of the players with the new game parameters, so you use PlayerID.AllPlayers. Then you specify the network packet, the timeout, and the send flags. Here you use the SendFlags.Guaranteed flag because you want to make sure every client

receives the game update, and the SendFlags.NoLoopback flag, because you are the host—you already know the new game parameters. If you don't include this flag, you would receive the update message and process it because you are included in PlayerID.AllPlayers.

When you send a score update, you don't need to send any more than a single byte to the remote player, telling that player to increment their score by one or two. Your scoring system awards every other player 1 point every time your ship dies, and 2 additional points to a player if their shot is what killed you. Here you can see the methods that send the points:

```
public void SendScorePointToAll() {
   if (inSession) {
      NetworkPacket packet = new NetworkPacket();
      packet.Write(MessageType.Add1ToScore);
      peerObject.SendTo((int)PlayerID.AllPlayers, packet, 0,
         SendFlags.Guaranteed | SendFlags.NoLoopback);
   }
}

public void SendTwoPointsToPlayer(int player) {
   if (inSession) {
      NetworkPacket packet = new NetworkPacket();
      packet.Write(MessageType.Add2ToScore);
      peerObject.SendTo(player, packet, 0, SendFlags.Guaranteed);
   }
}
```

Now that you've looked at how to send the messages, let's take a look at what happens when you receive an update from another player.

When the PlayClass receives a message, it generates a DataReceived event. In the PlayClass constructor, you instructed it to use the DataReceived event handler in the GameClass for this event.

The first thing you do is check to see if you received a message before you were ready. If so, discard the message and return.

```
public void DataReceived(object sender, ReceiveEventArgs rea) {
   int senderID = rea.Message.SenderID;

   // Ignore messages received before you are initialized.
   if ((gameState == GameStates.Loading) || (gameState == GameStates.Config)) {
      rea.Message.ReceiveData.Dispose();
      return;
   }
   …
```

If the game is running, then you need to determine what kind of message you've received so you can extract the values and assign them to the correct fields. A network message is nothing more than an array of bytes, so if you don't know what kind of message you've received, you won't know how to interpret the message. This is why you need to make sure that the first byte of the message is always the MessageType. Here you see the code to read 1 byte from the message. You then use a switch() statement to route the execution to the appropriate block to decode the message.

```
…
byte mType = (byte)rea.Message.ReceiveData.Read(typeof(byte));
MessageType messageType = (MessageType)mType;
switch (messageType) {
   case MessageType.PlayerUpdateID: {

      PlayerUpdate update = (PlayerUpdate)rea.
         Message.ReceiveData.Read(typeof(PlayerUpdate));
      ShotUpdate shotUpdate = new ShotUpdate();
      shotUpdate.ShotPosition = new Vector2[Constants.NumShots];
      shotUpdate.ShotAge = new int[Constants.NumShots];

      for (int i = 0; i < Constants.NumShots; i++) {
         shotUpdate.ShotPosition[i] =
            (Vector2)rea.Message.ReceiveData.Read(typeof(Vector2));
         shotUpdate.ShotAge[i] =
            (int)rea.Message.ReceiveData.Read(typeof(int));
      }

      rea.Message.ReceiveData.Dispose();



      lock (otherPlayers) {
         object playerObject = otherPlayers[senderID];
         if (null == playerObject)
            return;
         RemotePlayer player = (RemotePlayer) playerObject;

         Shot[] shotArray = new Shot[Constants.NumShots];
         for (int i = 0; i < Constants.NumShots; i++) {
            shotArray[i] = new Shot();
            shotArray[i].Position = shotUpdate.ShotPosition[i];
            shotArray[i].Age = shotUpdate.ShotAge[i];
```

```
            }
                  player.Ship.ShotHandler.SetShotArray(shotArray);

            player.Ship.Position = update.ShipPosition;
            player.Ship.Outline = update.Outline;
            player.Ship.Velocity = update.ShipVelocity;
            player.Ship.State = update.State;
            player.Ship.WaitCount = update.WaitCount;
            player.Ship.DeathCount = update.DeathCount;
            player.Ship.FlameIndex = update.FlameIndex;
            player.Ship.Sounds = (Sounds)update.Sounds;
            player.Ship.Score = update.Score;

            player.UpdateTime = DateTime.Now;
            player.Active = true;

            otherPlayers[senderID] = player;
         }

         break;
      }
      case MessageType.GameParamUpdateID: {
         GameParamUpdate update = (GameParamUpdate)rea.
            Message.ReceiveData.Read(typeof(GameParamUpdate));
         rea.Message.ReceiveData.Dispose();
         gravity = update.Gravity;
         gameSpeed = update.GameSpeed;

         if (update.BounceBack != 0)
            bounceBack = true;
         else
            bounceBack = false;

         if (update.InverseGravity != 0)
            inverseGravity = true;
         else
            inverseGravity = false;

         if (update.BlackHole != 0)
            blackHole = true;
         else
            blackHole = false;
```

```
            Size newWindowSize = update.WindowSize;
            Rectangle newBounds = new
               Rectangle(this.windowBounds.Location, newWindowSize);
            // Initialize(newBounds);
            break;
         }
         case MessageType.Add1ToScore: {
            rea.Message.ReceiveData.Dispose();
            ship.Score += 1;
            break;
         }
         case MessageType.Add2ToScore: {
            rea.Message.ReceiveData.Dispose();
            ship.Score += 2;
            break;
         }

         case MessageType.GamePaused: {
            rea.Message.ReceiveData.Dispose();
            gameState = GameStates.Paused;
            break;
         }
         case MessageType.GameRunning: {
            rea.Message.ReceiveData.Dispose();
            if (gameState == GameStates.Paused)
               gameState = GameStates.Running;
            break;
         }
      }
   }
```

Although the DataReceived event is the most common, you ask DirectPlay to notify you of a few other events as well.

A PlayerCreated event is generated every time someone joins the session, including when you create or join it, so your PlayerCreated event handler tests to see if your client is the new player. From GameClass, here is your event handler:

```
public void PlayerCreated(object sender, PlayerCreatedEventArgs pcea) {
   Peer peer = (Peer) sender;
   int playerID = pcea.Message.PlayerID;
   PlayerInformation playerInfo = peer.GetPeerInformation(playerID);
```

```
   // See if the player that was just created is you.
   if (playerInfo.Local) {
      localPlayer.ID = playerID;
      localPlayer.Name = playerInfo.Name;
   }
      // If not, create a remote player.
   else {
      Ship newShip = new Ship(this);
      newShip.HostName = playerInfo.Name.ToUpper();
      newShip.State = (int)ShipState.Normal;
      newShip.ScreenBounds = this.windowBounds;
      RemotePlayer newPlayer =new RemotePlayer(playerID,playerInfo.Name,newShip);
      lock (otherPlayers) {
         otherPlayers.Add(playerID, newPlayer);
      }
   }
}
```

If another player is joining and not you, you set up a new ship for them, and add them to the list of other players. Notice that you need to lock the Other Players list before you add them. You do this to make sure that the list is thread safe. Unless you specifically lock DirectPlay down to a single thread, it will create its own worker threads to handle incoming and outgoing network messages. Because you have no way of knowing what you'll be doing when a "PlayerCreated" message comes in, you use the lock statement to make sure that only one thread is accessing the Other Players list at a time.

A PlayerDestroyed event is triggered every time someone leaves the DirectPlay session. All you have to do is test to see if you were the one leaving the session, and if not, remove the departing player from the list.

```
public void PlayerDestroyed(object sender, PlayerDestroyedEventArgs pdea) {
   // Remove this player from your list.
   // You lock the data here because it is shared across multiple threads.
   int playerID = pdea.Message.PlayerID;
   if (playerID != localPlayer.ID) {
      lock (otherPlayers) {
         otherPlayers.Remove(playerID);
      }
   }
}
```

The last event you care about in your game is the SessionTerminated event. When the session ends, you simply change your inSession and isHost values to false.

```
private void SessionTerminated(object sender, SessionTerminatedEventArgs stea) {
   inSession = false;
   isHost = false;
}
```

That wraps up most of the major changes made to this new version, but there is a lot to be learned from looking at the complete source code. Be sure to load up the different versions and look them over.

Debugging Hints

Before closing this chapter, we want to leave you with a few tips for debugging DirectX applications. Hopefully they will save you some time (and frustration) when you are debugging your programs. They don't necessarily have anything to do with Spacewar, but are useful in general.

- Turn on unmanaged debugging.

  Although DirectX9 has managed interfaces, it still won't return useful error codes from the DirectX DLLs unless you enable unmanaged debugging. In Visual Studio, select your project's properties from the Project menu. Click the Configuration Properties folder, and then select Debugging. Set Enable Unmanaged Debugging to true.

- Use the debug DirectX runtime, and turn up the debug output level.

  In the Windows Control Panel, you'll find a DirectX utility. Launch the utility, and click the Direct3D tab. Select the Use debug version of Direct3D option and move the slider to increase the level of debug output. You'll see the DirectX messages in your output window when you are debugging.

CAUTION *This will dramatically affect your performance, but sometimes it's the only way to determine why Direct3D won't render your graphics. Once you've found the error, be sure to set these back.*

- Run as a console application.

  Select your project properties again, click the Common Properties folder, and select General. Set the Output Type option to Console Application. Because you are using Windows Forms, the program will still create all of your forms, but when it runs it will pop up a console window that you can access using the Console.Write and WriteLine() methods. It's great for debugging things when you don't want to flip back to your development environment to look at the output window.

- Use the property grid.

  You can create a form with a property grid on it to run alongside your application when you're debugging. This is great for trying out RenderStates, CullModes, Clipping Planes, etc. Set propertyGrid.SelectedObject to whatever you want to have runtime control over, and the property grid control does all of the work for you. All you have to do is tweak the values. Very handy.

Summary

Hopefully we've given you a couple of ideas you can put to use in your own projects. If you're looking for some fun things to do with the Spacewar code, just dig in. There are bugs that you can fix, or enhancements to add. Here are some of the things you might want to try:

- Add a splash screen instead of starting out with the network configuration.
- Add mouse input.
- Allow moving the screen without a restart.
- Add shields to the ships.
- Create an AI player.
- Allow a network reconnection option after the session has been terminated.
- Allow the player to customize the keys used for gameplay.

There are plenty of things you can play with—just open the source code and have fun!

Acknowledgments

The authors are indebted to Scott Haynie, who contributed to this chapter and rewrote Eric Gunnerson's original C# Spacewar code.