CHAPTER 5

Handling Input to Move Our Camera

IN THIS CHAPTER

We created a camera in the last chapter that does not move. We did not allow any input (other than exiting the demo with the GamePad's Back button, which the template provided for us). We are going to change that now. You are going to learn how to use input devices (keyboard, mouse, and game pad controller) by working with the camera. To accomplish this, we are going to start a game library that we can utilize in our demos and games. Let's get started!

Creating a Game Service

Instead of just throwing input and camera code inside of another demo, we are going to take the time to create a library (even if it is just a small one). Before we dig into that, however, we need to discuss game services and how they are used with game components. This will greatly simplify our library architecture.

We have a camera we have been using up to this point. The camera code is actually inside of the game object, which is perfectly acceptable for smaller demos. However, we're going to take the time in this chapter to move the camera into its own game component. We have already moved our FPS code into its own game component, and we will end up putting that game component into the library we are creating. In this chapter, we are also going to create an input handler that's a game service.

You learned how to create a game component in the last chapter using our FPS game component. Having a game component makes it easier to manage our code because items are broken out into logical pieces. However, we can run into a potential issue when multiple game components we create need to access the same piece of functionality (or each other). We can see this more easily when thinking about input.

We want to put our code that handles the input into its own game component—this is the XNA way after all. So let's assume we have our input game component done and now we have a player game component that needs access to the input device so that the player game component can react to the gamer as he or she uses the input device. To enable our player game component (and any other future game component we make) to access the input handler game component, we will turn the input handler game component into a game service. A *game service* is simply a game component wrapped in a unique interface that allows other game components and services to interact with it.

NOTE

Remember what you learned from the last chapter: Too many game components can hurt performance. In such cases we can use a "manager" game component that handles its own objects. The example we discussed last chapter involved creating enemies. Some XNA beginners will create an enemy game component for every enemy they have on the screen (or even a bullet game component). They then try to add and remove or set visible and invisible each and every game component. This can bring a system to a crawl because of the overhead associated with the game components. They're just not meant to be used in that way. A better approach is to create an `EnemyManager` or `BulletManager` game component and allow the manager to handle which enemies or bullets are being displayed.

The `Game` object holds collections of services. We have actually been using one already. The `GraphicsDevice` is a game service built into XNA. When we created our FPS game component, we needed access to the `GraphicsDevice`. Therefore, we wrote the following code in the constructor for the FPS code:

```
GraphicsDeviceManager graphics =
    (GraphicsDeviceManager)Game.Services.GetService(
    typeof(IGraphicsDeviceManager));
```

Game components always have a `Game` object passed to them, so every game component can obtain access to any game service associated with the game. The preceding code does just that. It creates a reference to the game's `GraphicsDeviceManager` memory. When we create a new solution by selecting a game template, the code generated for us includes a variable called `graphics` that holds the graphics device manager. Although we could have made the `graphics` variable public and accessed the data that way, it is much more elegant to use a service. Otherwise, we would probably be creating many publicly accessible properties, which would hurt our game design.

Because the XNA Framework team used an interface for the graphics device manager object and because they added it to the services collection of the game object, we are able to access it from any piece of code that has access to the game object. You can see this if you look at the preceding code again. It calls out to the game object to which we have a reference and calls its `Services.GetService` method. This method takes in a type and returns an object. We pass in the type of the interface and get returned the graphics device object. We perform a cast on it so we have a strongly typed object that allows us to work with it intelligently.

You have seen how we used a game service that XNA created for us; now you'll learn how to create a game service. To get started, copy the Transformations project we created in the last chapter and name the new project InputDemo. Also rename the namespace in the projects and modify the GUID in the AssemblyInfo.cs file.

We will create the input game service now inside of this project and then move it along with our FPS game component when we create our library in the next section. We need to add a new game component file to our project, which we will call InputHandler. The first step to making this a game service is to create an interface.

The game services collection can only have one of any particular interface. This is how it knows which object to return when a request is made. This means that if we have more than one object of the same interface and we need to access them all as a game service, we would need to create a different interface for each object. Fortunately, an interface can be empty. This means through inheritance we can quickly make our "duplicate" object inherit from the original object and from the new interface that does not require any additional implementation.

To add our interface we need to use the following code:

```
public interface IInputHandler { };
```

We can create a new code file to store this interface or we can put it directly in the input handler file. Currently, our input handler interface is blank, but we will be adding properties to it soon. Now we need our game component to inherit from this interface (as well as the `GameComponent` object it is already inheriting from). We do this by changing our class declaration to the following:

```
public class InputHandler
    : Microsoft.Xna.Framework.GameComponent, IInputHandler
```

We also need to make sure our input handler is using the XNA Framework's `Input` namespace:

```
using Microsoft.Xna.Framework.Input;
```

To finish making this game component into a game service, we need to add it to the game's services collection. We do this in the game component's constructor with the following code:

```
game.Services.AddService(typeof(IInputHandler), this);
```

Adding a service is much like getting a service. We pass in the type of the interface for the object and then we actually pass in our object. We always want to pass in the type of our interface and not the type of our actual class. This way, when other objects access the game service, they can simply do it through the interface without having to explicitly use the object. Of course if needed, we can cast to the actual object. We should be able to compile the code at this point. It currently does nothing differently than it did in the last chapter. However, we are laying a foundation so we can get started on our library. Let's do that now!

Starting a Library

We have two game components currently (well, one is a skeleton) and are planning on adding more. We should put these into a library that we can easily access from multiple demos and games. We are going to set up a library project to hold our FPS game component as well as the start of the input handler class we just created. To do this, we will create a new Windows Game Library project, which we'll call XELibrary.

Now that we have our solution set up with the two projects, we can start building our library. To begin, we add the FPS.cs file from our last project. Let's cut the actual FPS.cs and InputHandler.cs files from our InputDemo project and paste them into our XELibrary project (after saving InputHandler.cs of course). At this point we have started a library that contains our fully made FPS game component and a skeleton of our InputHandler game component. We can remove the Class1.cs file from our project. We should also make sure our namespace is consistent across the library and set it to `XELibrary`. We should be able to build the library project with no compilation errors.

NOTE

As we add items to one project (Windows), we need to add them to the other project (Xbox 360) as well (after we create the additional platform project like we did in Chapter 2, "XNA Game Studio and the Xbox 360"). From here on, it will be assumed that as we create a project we will create the other platform's counterpart, and as we add files to one project it will be assumed that they are added to the other project as well.

Before we start on our input handler code, we can go ahead and create another game component called Camera. We are going to move the initial camera code we have created to this game component. We will also talk to our input handler game service inside this game component. Let's go ahead and set up a private member field to hold an instance of the input handler because we know we will need it. We do this by adding the following code to our Camera.cs file:

```
private IInputHandler input;
```

Now that we have the field created, we can initialize it inside of our constructor as follows:

```
input = (IInputHandler)game.Services.GetService(typeof(IInputHandler));
```

Now we can move our InitializeCamera method from the InputDemo Game1.cs code into our Camera class. We need to cut the call to this method from the Initialize method inside our game project. Now we need to paste the call into our camera's Initialize method. The updated code in our camera class should look like the following:

```
public override void Initialize()
{
    base.Initialize();
    InitializeCamera();
}

private void InitializeCamera()
{
    float aspectRatio = (float)Game.GraphicsDevice.Viewport.Width /
        (float)Game.GraphicsDevice.Viewport.Height;
    Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4, aspectRatio,
        1.0f, 10000.0f, out projection);

    Matrix.CreateLookAt(ref cameraPosition, ref cameraTarget,
        ref cameraUpVector, out view);
}
```

We did not modify this code; we simply moved it from our InputDemo game class into our XELibrary camera class. Notice we called InitializeCamera after calling base.Initialize. This is because we are utilizing the graphics device (from our component's Game object) and it is not available until the main game class finishes with its Initialize method.

Our InitializeCamera method is using some member fields we have not moved yet. We need to move the following code from our demo to our library:

```
private Matrix projection;
private Matrix view;
private Vector3 cameraPosition = new Vector3(0.0f, 0.0f, 3.0f);
private Vector3 cameraTarget = Vector3.Zero;
private Vector3 cameraUpVector = Vector3.Up;
```

Because we moved our view and projection fields from our demo, the demo will no longer compile until we change our demo's Draw method, where we set the effect's view and projection so it will display our objects correctly. We could leave these in the game class as public fields and have the Camera game component work with them there. A cleaner approach would be to leave the camera self-contained and access those properties from inside our demo. Let's create and initialize our camera game component inside of our demo. We can use the following code:

```
camera = new Camera(this);
Components.Add(camera);
```

We also need to set up the member field as follows:

```
private Camera camera;
```

We need to reference our game library inside of our game project. This can be done by right-clicking the InputDemo's project References tree node in the Solution Explorer. Then click Add Reference and select the Projects tab. Selecting XELibrary and clicking Add will make XELibrary a dependency of InputDemo. We need to make sure to reference the Windows library in the Windows demo and the Xbox 360 library in the Xbox 360 demo.

NOTE

It is usually a good idea to create the Xbox 360 copy of the project early on and keep it up to date as new files are added. It is much better to know how the code is performing on another platform during the development cycle instead of at the end.

After loading the library projects inside our demo solution, we now need to add a `using` statement to the top of our Game1.cs file in the demo project. We need to use `XELibrary`, as shown here:

```
using XELibrary;
```

If we compile at this point, the only errors we should see are inside our demo's `Draw` method, stating that it has no idea what `view` and `projection` are. We are going to fix that now by changing the code to use our camera's `view` and `projection` properties. We do not have any properties yet in our `Camera` object? Well, let's change that. We need to add the following code to our camera game component:

```
public Matrix View
{
    get { return view; }
}

public Matrix Projection
{
    get { return projection; }
}
```

Now we can access these properties inside of our demo's `Draw` method. Let's change the properties we set on our effect with the following code:

```
effect.Projection = camera.Projection;
effect.View = camera.View;
```

Now we can successfully compile our code again. Not only that, but we can run it and it will function just like before, but now we have started a library!

Working with Input Devices

We will be populating the stub we created for handling our input. We are going to create a stationary camera, a first-person camera, and a third-person camera. All these cameras will need to respond to user input. You'll learn how to handle input devices and utilize them to move our camera to fully see the worlds we create.

Keyboard

The first input device we will talk about is the keyboard. XNA provides helper classes that allow us to easily determine the state of our input devices. We determine the state of our keyboard by declaring a variable to store our keyboard state and then calling the `GetState` method on the `Keyboard` helper object. We can then query that variable to determine which key (or keys) is being pressed or released. We can start by adding a private member field to store our keyboard state. We do this via the following code inside of our InputHandler game component:

```
private KeyboardState keyboardState;
```

Then we can find the `Update` method that was stubbed out for us when the InputHandler game component was created:

```
keyboardState = Keyboard.GetState();
if (keyboardState.IsKeyDown(Keys.Escape))
    Game.Exit();
```

We can put this at the very beginning of the `Update` method. We are simply storing the state of the keyboard and then checking to see if the Escape key was pressed. If it was, we simply exit the program. It doesn't get much simpler than that! So now when we run the program we can exit by pressing the Escape key (instead of having to close the window with our mouse) .

Although being able to exit our game easily is important, we still haven't done anything exciting. Let's set up our camera so we swivel back and forth if the left and right keys are pressed. To do this we need to add and initialize a new private member field called `cameraReference` inside our Camera game component:

```
private Vector3 cameraReference = new Vector3(0.0f, 0.0f, -1.0f);
```

This camera reference direction will not change throughout the game, but we will be passing it in as a reference. Therefore, we cannot declare it as a `readonly` variable. Typically this value will be either (0,0,1) or (0,0,-1). We chose to have a negative z value so we can continue to have our camera face the same way.

Now that we have our camera reference direction set up, we need to apply any movement and rotation to our view of the world. This way, if the player wants to look left and right, we can adjust our view matrix accordingly so that the view of the world is from a certain angle. To look left and right, we need to rotate around the y axis. We can add the following code to our `Update` method right before our call to the base object's `Update` method (we are still in our `Camera` class) :

```
Matrix rotationMatrix;
Matrix.CreateRotationY(MathHelper.ToRadians(cameraYaw), out rotationMatrix);
// Create a vector pointing the direction the camera is facing.
Vector3 transformedReference;
Vector3.Transform(ref cameraReference, ref rotationMatrix,
    out transformedReference);
// Calculate the position the camera is looking at.
Vector3.Add(ref cameraPosition, ref transformedReference, out cameraTarget);

Matrix.CreateLookAt(ref cameraPosition, ref cameraTarget, ref cameraUpVector,
    out view);
```

Because we know we will be rotating, we need to create a matrix to hold our rotation. With a helper function from XNA, we will create a matrix with the appropriate values to rotate our camera on the y axis by a certain number of degrees. We store that value in a variable we set with our input devices. You'll see how we set that value with our keyboard soon. Once we have our matrix with all the translations, scaling, and rotations that we need (in this case we are only rotating), we can create a transform vector that allows us to change the target of the camera. We get this transformed vector by using another XNA helper method, `Vector3.Transform`. We then add the transformed camera reference to our camera position, which will give us our new camera target. To do this, we could have used the plus (+) operator like in the following code:

```
cameraTarget = cameraPosition + transformedReference;
```

However, it is more efficient to use the built-in static `Add` method of the `Vector3` struct because it allows us to pass our vectors by reference instead of having to copy the values to memory. Finally, we reset our view with our new camera target. We also need to set the following private member field that we used in the code:

```
private float cameraYaw = 0.0f;
```

Now we are to the point where we can compile the code, but our newly added code does not do anything for us. This is because we are never changing our rotation angle. It stays at 0 on every frame. Let's change that now with our keyboard. Inside our `InputHandler` class, we need to update our `IInputHandler` interface to expose the keyboard state we retrieved earlier. Let's add the following code inside our interface:

```
KeyboardState KeyboardState { get; }
```

Now, we need to implement that property inside the class. An easy way to do this is to right-click IInputHandler where we derived from it and select Implement Interface. We will be doing this a couple times, and each time a region will be created. Therefore, we will have to clean up the code and remove the extra regions the IDE provides for us. Now we need to change the `get` value to retrieve our internal `keyboardState` value, as follows:

```
public KeyboardState KeyboardState
{
    get { return(keyboardState); }
}
```

Now that we have exposed our keyboard state, we can utilize it inside our `Camera` object. We need to add the following code to our `Update` method right above the previous code inside of our camera object:

```
if (input.KeyboardState.IsKeyDown(Keys.Left))
    cameraYaw += spinRate;
if (input.KeyboardState.IsKeyDown(Keys.Right))
    cameraYaw -= spinRate;

if (cameraYaw > 360)
    cameraYaw -= 360;
else if (cameraYaw < 0)
    cameraYaw += 360;
```

We also need to make sure camera is using the XNA Framework's `Input` namespace because we are accessing the `Keys` enumeration:

```
using Microsoft.Xna.Framework.Input;
```

Finally, we add this constant to the top of our camera class:

```
private const float spinRate = 2.0f;
```

In our update code we utilized the current keyboard state we already captured and checked to see if either the left arrow or right arrow on the keyboard was pressed. If the player wants to rotate to the left, we add our spin rate constant to our current camera yaw angle. (*Yaw*, *pitch*, and *roll* are terms borrowed from flight dynamics. Because we are using a right-handed coordinate system, this means that yaw is rotation around the y axis, pitch is rotation around the x axis, and roll is rotation around the z axis.) If the player wants to rotate to the right, we subtract our spin rate constant from our current camera yaw angle. Finally, we just check to make sure we do not have an invalid rotation angle.

There is one last thing we need to do before we can run our code again. We need to add our input handler game component to our game's collection of components. We can declare our member field as follows:

```
private InputHandler input;
```

Now we can initialize that variable and add it to the collection inside our constructor with the following code:

```
input = new InputHandler(this);
Components.Add(input);
```

NOTE

The preceding code should be added before the camera component is processed. This is because the camera component uses the input component. Otherwise, you will get a null reference exception.

We can compile and run the code, and the left- and right-arrow keys will rotate the camera. When running the code, you can see that our objects are just flying by. We are turning so fast that they seem to be blinking. This is because we are calling our `Update` statement as fast as possible. We can modify the game code where we are initializing our `fps` variable to use a fixed time step:

```
fps = new FPS(this, false, true);
```

For the preceding code to work, we need to add another constructor to our FPS code. We are doing this so we don't need to actually pass in our target elapsed time value if we want it to be the default.

```
public FPS(Game game, bool synchWithVerticalRetrace, bool isFixedTimeStep)
    : this(game, synchWithVerticalRetrace, isFixedTimeStep,
           game.TargetElapsedTime) { }
```

Now when you run the code you should see the objects move by at a consistent and slower rate. This is because the `Update` method is now only getting called 60 times a second instead of whatever rate your machine was running at.

You will notice, however, as the rectangles are rendered that our screen is "choppy." The reason is that we are not letting XNA only draw during our monitor's vertical refresh. If we were to set the second parameter to true, we would see the screen rotate at a nice even pace with the screen drawing nicely. However, a better way to handle this is by utilizing the elapsed time between calls to our methods. We need to retrieve the elapsed time since the last time our `Update` method was called and then multiply our `spinRate` by this delta of the time between calls. Change the camera code snippet to match the following:

```
float timeDelta = (float)gameTime.ElapsedGameTime.TotalSeconds;


if (input.KeyboardState.IsKeyDown(Keys.Left))
    cameraYaw += (spinRate * timeDelta);
```

```
if (input.KeyboardState.IsKeyDown(Keys.Right))
    cameraYaw -= (spinRate * timeDelta);
```

We can modify our game code to call the default constructor again:

```
fps = new FPS(this);
```

Now we are just creeping along. This is because our spin rate is so low. We had it low because we were relying on the update method to be called 60 times per frame, so we were basically rotating our camera 120 degrees per second. To get the same effect we simply set our `spinRate` to 120. The reason is we are now multiplying it by the time difference between calls. At this point we can safely set our units and know they will be used on a per-second basis. Now that we have our `spinRate` utilizing the delta of the time between calls, we are safe to run at any frame rate and have our objects drawn correctly based on the amount of time that has elapsed.

We can have it always run at 60 fps in release mode and run as fast as possible in debug mode by modifying our game code as follows:

```
#if DEBUG
    fps = new FPS(this);
#else
    fps = new FPS(this, true, false);
#endif
```

This allows us to run as fast as we can in debug mode while consistently moving our objects no matter the frame rate. It allows us to force XNA to only update the screen during the monitor's vertical retrace, which would drop us to 60 fps or whatever rate the monitor is refreshing at.

Making a game update itself consistently regardless of the frame rate can make the game more complex. We need to calculate the elapsed time (time delta) since the last frame and use that value in all our calculations. With the fixed step mode that XNA provides, we could cut down on development time and rely on the fact that the update code will be called 60 times a second. This is not sufficient if we are writing a library that can be plugged into games, because those games might not run at a consistent frame rate.

Game Pad

The Microsoft Xbox 360 wired controller works on the PC. The wireless Xbox 360 controller will also work on the PC, but the Xbox 360 Wireless Gaming Receiver for Windows is required. The XNA Framework provides us with helper classes that make it very easy to determine the state of our game pad. The template already provided one call to the game pad helper class. This call is also in the `Update` method. Let's take a look at what is provided for us already.

```
if (GamePad.GetState(PlayerIndex.One).IsButtonDown(Buttons.Back))
    this.Exit();
```

The template is calling the built-in XNA class `GamePad` and calling its `GetState` method, passing in the specific player's controller to check. The template then checks the Back button on that controller to see if it has been pressed. If the controller's Back button has been pressed, the game exits. Now, that was pretty straightforward. To be consistent we can use our input class to check for the condition.

Before we can do that, we need to update our interface and add the appropriate property. We also need to get the game pad state just like we did for our keyboard. Let's jump to our input handler code and do some of these things. We can start by adding a property to get to our list of game pads in our interface:

```
GamePadState[] GamePads { get; }
```

Now we can create the member field and property to get that field, as in the following code:

```
private GamePadState[] gamePads = new GamePadState[4];
public GamePadState[] GamePads
{
    get { return(gamePads); }
}
```

We need to initialize each game pad state. We can do that in the `Update` method of the input handler object:

```
gamePads[0] = GamePad.GetState(PlayerIndex.One);
gamePads[1] = GamePad.GetState(PlayerIndex.Two);
gamePads[2] = GamePad.GetState(PlayerIndex.Three);
gamePads[3] = GamePad.GetState(PlayerIndex.Four);
```

Now, let's remove the code that checks to see if the Back button is pressed on player one's game pad from our demo. We can add this code in the `Update` method of our `InputHandler` game component to get the same effect:

```
if (gamePads[0].Buttons.Back == ButtonState.Pressed)
    Game.Exit();
```

Let's update our yaw rotation code inside the camera game component so that we can get the same result with our controller. We can modify our existing code that checks for left and right to also handle input from our controller. Thus, we modify our two conditional statements that set the `cameraYaw` to also check the right thumb stick state of the game pad we are examining:

```
if (input.KeyboardState.IsKeyDown(Keys.Left) ||
    (input.GamePads[0].ThumbSticks.Right.X < 0))
{
    cameraYaw += (spinRate * timeDelta);
}
if (input.KeyboardState.IsKeyDown(Keys.Right) ||
    (input.GamePads[0].ThumbSticks.Right.X > 0))
{
    cameraYaw -= (spinRate * timeDelta);
}
```

The thumb stick x and y axes provide a float value between –1 and 1. A value of 0 means there is no movement. A value of –0.5 means the stick is pushed to the left halfway. A value of 0.9 means the stick is pushed to the right 90% of the way.

We did not change the keyboard code; instead, we simply added another “or” condition to handle our controller. You can see it is very simple—we only needed to check the `ThumbSticks.Right` property. We check the x axis of that joystick, and if it is less than zero the user is pushing the stick to the left. We check to see if it is positive (user pushing to the right) in the second condition. We leave our `cameraYaw` variable to be set by our spin rate (taking into account our time delta). At this point, regardless of whether the players are using the keyboard or the game pad, they will get the same result from the game: The camera will rotate around the y axis. Compile and run the program to try it out. You can also try the game on the Xbox 360 because we have hooked up our game pad code.

At this point you know how to get access to any of the buttons (they are treated the same way as the Back button) and either thumb stick, but we have not discussed the D-pad yet. The D-pad is actually treated like buttons. If we want to allow the player to rotate the camera left or right by using the D-pad, we could add the following as part of our condition:

```
input.GamePads[0].DPad.Left == ButtonState.Pressed
```

However, the XNA Framework has the helper methods `IsButtonDown` and `IsButtonUp`. We can get the same result by using the following condition:

```
input.GamePads[0].IsButtonDown(Buttons.DPadLeft)
```

We could replace the thumb stick condition by using `IsButtonDown` as well. Even though the movement on the thumb stick is not really a button, the XNA Framework can treat it like one and do the check for us. Therefore, the following condition to check whether the right thumb stick is pushed to the left is valid:

```
input.GamePads[0].IsButtonDown(Buttons.RightThumbStickLeft)
```

We can add those conditions to our code. Compile and run the code again and make sure the demo allows you to look left or right by using the keyboard, thumb stick, and D-pad. The code should now look like the following:

```
if (input.KeyboardState.IsKeyDown(Keys.Left) ¦¦
    input.GamePads[0].IsButtonDown(Buttons.RightThumbstickLeft) ¦¦
    input.GamePads[0].IsButtonDown(Buttons.DPadLeft))
{
    cameraYaw += (spinRate * timeDelta);
}
if (input.KeyboardState.IsKeyDown(Keys.Right) ||
    input.GamePads[0].IsButtonDown(Buttons.RightThumbstickRight) ||
    input.GamePads[0].IsButtonDown(Buttons.DPadRight))
{
    cameraYaw -= (spinRate * timeDelta);
}
```

The shoulder buttons are just that—buttons—and you know how to handle those already. We can determine when the left or right thumb stick is pressed down because both are also considered buttons. In fact, every input on a game pad is mapped to a button. However, the final item we'll discuss regarding our game pad controller is the triggers. Now, a good use of our triggers for this demo would be to turn on vibration!

Before we determine how to use the triggers, we can look at another property of the game pad state we have access to—whether or not the controller is actually connected. We can check this by getting the value of the `IsConnected` property.

Even though we can determine whether or not a trigger is pressed (just like we can determine whether the right thumb stick is pushed to the left), there are times we need more information. We may need to know how far in the trigger is pressed. All the way? Ten percent? Ninety percent? Fortunately, the trigger values return a float between 0 and 1 to signify how much the trigger is pressed (0 = not pressed; 1 = fully pressed). The Xbox 360 controller has two motors that create its vibration. The motor on the left is a low-frequency motor, whereas the motor on the right is a high-frequency motor. We can set the values on both motors in the same method call. We do this by calling the `GamePad.SetVibration` method. Because this is just something we are doing for our demo and not really a part of the library, we will put this code in our Game1.cs file inside the `Update` method:

```
if (input.GamePads[0].IsConnected)
{
    GamePad.SetVibration(PlayerIndex.One, input.GamePads[0].Triggers.Left,
        input.GamePads[0].Triggers.Right);
}
```

The first thing we are doing with this new code is checking to see whether the game pad is actually connected. If it is connected, we set the vibration of both the left and right motors based on how much the left and right triggers are being pressed. We'll call the GamePad's static `SetVibration` method. There is currently no benefit in wrapping that into a method inside of our input handler.

We can also change the information being displayed in our window title bar to include the left and right motor values. This can help you determine what values you should use as you implement vibration in your games! The following is the code to accomplish that task:

```
this.Window.Title = "left: " +
    input.GamePads[0].Triggers.Left.ToString() + "; right: " +
    input.GamePads[0].Triggers.Right.ToString();
```

Go ahead and add this debug line inside of the `IsConnected` condition and compile and run the code to check the progress. We will no longer be able to see our frame rate with this code, so we could just comment out the fps object until we are ready to check on our performance.

Mouse (Windows Only)

This input device is only available for Windows, so if you are deploying the game for the Xbox 360, you will need to put the `XBOX360` compilation directive check around any code that references the mouse as an input device (refer to Chapter 2). Therefore, we will create a private member field with this preprocessor check inside our `InputHandler` class. We need to set up a private member field to hold our previous mouse state and another one to hold our current mouse state:

```
#if !XBOX360
    private MouseState mouseState;
    private MouseState prevMouseState;
#endif
```

Then in the constructor we tell XNA that we want the mouse icon visible in the window and we store the current mouse state:

```
#if !XBOX360
    Game.IsMouseVisible = true;
    prevMouseState = Mouse.GetState();
#endif
```

In our `Update` method of the input handler game component, we need to set the previous state to what our current state is and then reset our current state as follows:

```
#if !XBOX360
    prevMouseState = mouseState;
    mouseState = Mouse.GetState();
#endif
```

Now we need to expose these internal fields so our camera (and any other objects) can get their values. First we need to add the properties to our interface as follows:

```
#if !XBOX360
    MouseState MouseState { get; }
    MouseState PreviousMouseState { get; }
#endif
```

Now we can implement those properties in our class:

```
#if !XBOX360
    public MouseState MouseState
    {
        get { return(mouseState); }
    }

    public MouseState PreviousMouseState
    {
        get { return(prevMouseState); }
    }
#endif
```

In our camera's `Update` method, we want to get the latest state of our mouse and compare the current `X` value to the previous `X` value to determine whether we moved the mouse left or right. We also want to check whether the left mouse button is pushed before updating our `cameraYaw` variable. Of course, all this is wrapped in our compilation preprocessor condition, as follows:

```
#if !XBOX360
    if ((input.PreviousMouseState.X > input.MouseState.X) &&
        (input.MouseState.LeftButton == ButtonState.Pressed))
    {
        cameraYaw += (spinRate * timeDelta);
    }
    else if ((input.PreviousMouseState.X < input.MouseState.X) &&
        (input.MouseState.LeftButton == ButtonState.Pressed))
    {
        cameraYaw -= (spinRate * timeDelta);
    }
#endif
```

We can compile and run the code and test the latest functionality on Windows. If the preprocessor checks are in place correctly, we should be able to deploy this demo to the Xbox 360, although it will not do anything different than it did before we added the mouse support.

Creating a Stationary Camera

Now that you know how to utilize our input, we can get working on implementing our stationary camera. We actually have most of this done, but we need to add pitching in addition to the yaw. One use for a stationary camera is to look at an object and follow it by rotating as needed. This is commonly used in racing game replay mode.

Before we dig into the camera changes, though, let's add a few more rectangles to our world. We can do this by adding the following code to the end of our demo's `Update` method:

```
world = Matrix.CreateTranslation(new Vector3(8.0f, 0, -10.0f));
DrawRectangle(ref world);
```

```
world = Matrix.CreateTranslation(new Vector3(8.0f, 0, -6.0f));
DrawRectangle(ref world);

world = Matrix.CreateRotationY(MathHelper.ToRadians(180f)) *
    Matrix.CreateTranslation(new Vector3(3.0f, 0, 10.0f));DrawRectangle(ref world);
```

We should also change our cull mode to `None` so that as we rotate around, we will always see our rectangles. We can do that by calling the following code at the top of our game's `Draw` method:

```
graphics.GraphicsDevice.RenderState.CullMode = CullMode.None;
```

To get our camera updated, we need to modify our camera class a little bit. First we need to declare a private member field as follows:

```
private float cameraPitch = 0.0f;
```

Now we can modify the `Update` method to set our camera pitch. Remember, *pitching* refers to rotating around the x axis. To calculate this, we simply take the code we used for calculating the yaw and replace our reference to the y axis with the x axis. The following is the code to check our keyboard and game pad:

```
if (input.KeyboardState.IsKeyDown(Keys.Down) ||
    input.GamePads[playerIndex].IsButtonDown(Buttons.RightThumbstickDown) ||
    input.GamePads[playerIndex].IsButtonDown(Buttons.DPadDown))
{
    cameraPitch -= (spinRate * timeDelta);
}
if (input.KeyboardState.IsKeyDown(Keys.Up) ||
    input.GamePads[playerIndex].IsButtonDown(Buttons.RightThumbstickUp) ||
    input.GamePads[playerIndex].IsButtonDown(Buttons.DPadUp))
{
    cameraPitch += (spinRate * timeDelta);
}
```

No surprises there, and we need to do the same thing with our mouse code. Inside our `#if !XBOX360` compilation directive, add the following code:

```
if ((input.PreviousMouseState.Y > input.MouseState.Y) &&
    (input.MouseState.LeftButton == ButtonState.Pressed))
{
    cameraPitch += (spinRate * timeDelta);
}
else if ((input.PreviousMouseState.Y < input.MouseState.Y) &&
    (input.MouseState.LeftButton == ButtonState.Pressed))
{
    cameraPitch -= (spinRate * timeDelta);
}
```

We want to clamp our values so we do not rotate over 90 degrees in either direction:

```
if (cameraPitch > 89)
    cameraPitch = 89;
if (cameraPitch < -89)
    cameraPitch = -89;
```

Finally, we need to update our rotation matrix to include our pitch value. Here is the updated calculation:

```
Matrix rotationMatrix;
Matrix.CreateRotationY(MathHelper.ToRadians(cameraYaw), out rotationMatrix);
//add in pitch to the rotation
rotationMatrix = Matrix.CreateRotationX(MathHelper.ToRadians(cameraPitch)) *
    rotationMatrix;
```

The last statement is the only thing we added. We just added our pitch to the rotation matrix that was already being used to transform our camera. The full `Update` code can be found in Listing 5.1.

LISTING 5.1 Our Stationary Camera's **`Update`** Method

```
public override void Update(GameTime gameTime)
{
    float timeDelta = (float)gameTime.ElapsedGameTime.TotalSeconds;

    if (input.KeyboardState.IsKeyDown(Keys.Left) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.RightThumbstickLeft) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.DPadLeft))
    {
        cameraYaw += (spinRate * timeDelta);
    }
    if (input.KeyboardState.IsKeyDown(Keys.Right) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.RightThumbstickRight) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.DPadRight))
    {
        cameraYaw -= (spinRate * timeDelta);
    }

    if (input.KeyboardState.IsKeyDown(Keys.Down) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.RightThumbstickDown) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.DPadDown))
    {
        cameraPitch -= (spinRate * timeDelta);
    }
    if (input.KeyboardState.IsKeyDown(Keys.Up) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.RightThumbstickUp) ¦¦
```

```
            input.GamePads[playerIndex].IsButtonDown(Buttons.DPadUp))
        {
            cameraPitch += (spinRate * timeDelta);
        }


    #if !XBOX360
        if ((input.PreviousMouseState.X > input.MouseState.X) &&
            (input.MouseState.LeftButton == ButtonState.Pressed))
        {
            cameraYaw += (spinRate * timeDelta);
        }
        else if ((input.PreviousMouseState.X < input.MouseState.X) &&
            (input.MouseState.LeftButton == ButtonState.Pressed))
        {
            cameraYaw -= (spinRate * timeDelta);
        }

        if ((input.PreviousMouseState.Y > input.MouseState.Y) &&
            (input.MouseState.LeftButton == ButtonState.Pressed))
        {
            cameraPitch += (spinRate * timeDelta);
        }
        else if ((input.PreviousMouseState.Y < input.MouseState.Y) &&
            (input.MouseState.LeftButton == ButtonState.Pressed))
        {
            cameraPitch -= (spinRate * timeDelta);
        }
    #endif

        //reset camera angle if needed
        if (cameraYaw > 360)
            cameraYaw -= 360;
        else if (cameraYaw < 0)
            cameraYaw += 360;

        //keep camera from rotating a full 90 degrees in either direction
        if (cameraPitch > 89)
            cameraPitch = 89;
        if (cameraPitch < -89)
            cameraPitch = -89;

        Matrix rotationMatrix;
        Matrix.CreateRotationY(MathHelper.ToRadians(cameraYaw),
            out rotationMatrix);
        //add in pitch to the rotation
        rotationMatrix = Matrix.CreateRotationX(MathHelper.ToRadians(cameraPitch))
            * rotationMatrix;
        // Create a vector pointing the direction the camera is facing.
        Vector3 transformedReference;
        Vector3.Transform(ref cameraReference, ref rotationMatrix,
            out transformedReference);
        // Calculate the position the camera is looking at.
        Vector3.Add(ref cameraPosition, ref transformedReference, out cameraTarget);

        Matrix.CreateLookAt(ref cameraPosition, ref cameraTarget, ref cameraUpVector,
            out view);

        base.Update(gameTime);
    }
```

Creating a First-person Camera

We can build on our stationary camera by adding a first-person camera. The main thing we want to do is to add in a way to move back and forth and to each side. Before we start, we should create a new camera class and inherit from the one we have. We can call this new class `FirstPersonCamera`. The following is the `Update` method for this new class:

```
public override void Update(GameTime gameTime)
{
    //reset movement vector
    movement = Vector3.Zero;

    if (input.KeyboardState.IsKeyDown(Keys.A) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.LeftThumbstickLeft))
    {
        movement.X—;
    }
    if (input.KeyboardState.IsKeyDown(Keys.D) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.LeftThumbstickRight))
    {
        movement.X++;
    }

    if (input.KeyboardState.IsKeyDown(Keys.S) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.LeftThumbstickDown))
    {
        movement.Z++;
    }
    if (input.KeyboardState.IsKeyDown(Keys.W) ¦¦
        input.GamePads[playerIndex].IsButtonDown(Buttons.LeftThumbstickUp))
    {
        movement.Z—;
    }

    //make sure we don"t increase speed if pushing up and over (diagonal)
    if (movement.LengthSquared() != 0)
        movement.Normalize();

    base.Update(gameTime);
}
```

The conditional logic should look familiar. It is identical to our stationary camera, except we changed where we were reading the input and what values it updated. We are reading the A, S, W, D keys and the left thumb stick. We are not looking at the mouse for movement. The value we are setting is a movement vector. We are only setting the X (left and right) and Z (back and forth) values. At the end of the conditions we are normalizing our vector as long as the length squared of the vector is not zero. This makes sure that we are not allowing faster movement just because the user is moving diagonally. There is no more code in the `FirstPersonCamera` object. The rest of the changes were made back in our original camera object.

We declared the movement as a protected member field of type `Vector3` called `movement` inside our original `Camera` object. We also declared a constant value for our movement speed. Both of these are listed here:

```
protected Vector3 movement = Vector3.Zero;
private const float moveRate = 120.0f;
```

We also set the access modifier of our input field to protected so our `FirstPersonCamera` could access it:

```
protected IInputHandler input;
```

Finally, we updated the last part of our `Update` method to take the movement into account when transforming our camera:

```
//update movement (none for this base class)
movement *= (moveRate * timeDelta);

Matrix rotationMatrix;
Vector3 transformedReference;
Matrix.CreateRotationY(MathHelper.ToRadians(cameraYaw), out rotationMatrix);

if (movement != Vector3.Zero)
{
    Vector3.Transform(ref movement, ref rotationMatrix, out movement);
    cameraPosition += movement;
}

//add in pitch to the rotation
rotationMatrix = Matrix.CreateRotationX(MathHelper.ToRadians(cameraPitch)) *
    rotationMatrix;

// Create a vector pointing the direction the camera is facing.
Vector3.Transform(ref cameraReference, ref rotationMatrix,
    out transformedReference);
// Calculate the position the camera is looking at.
Vector3.Add(ref cameraPosition, ref transformedReference, out cameraTarget);

Matrix.CreateLookAt(ref cameraPosition, ref cameraTarget, ref cameraUpVector,
    out view);
```

Besides just moving our local variables closer together, the only things that changed are the items in bold type. We take our movement vector and apply our move rate to it (taking into account our time delta, of course). The second portion is the key. We transformed our movement vector by our rotation matrix. This keeps us from just looking in a direction but continuing to move straight ahead. By transforming our movement vector via our rotation matrix, we actually move in the direction we are looking! Well, the movement actually happens in the next statement when we add this movement vector to our current camera position. We wrapped all this in a condition to see if any movement happened because we do not want to take a performance hit to do the math if we did not move.

Another thing to note is that because we were creating a first-person camera, we only transformed our movement vector by the yaw portion of our rotation matrix. We did not include the pitch, because that would have allowed us to “fly.” If we did want to create a flying camera instead, we could simply move the following statement before the code in bold:

```
//add in pitch to the rotation
rotationMatrix = Matrix.CreateRotationX(MathHelper.ToRadians(cameraPitch)) *
    rotationMatrix;
```

In order to have our game actually use this new first-person camera, we need to replace the regular Camera component in our `InputDemo` class with the FirstPersonCamera component. Now when we compile and run the demo, we can use the left thumb stick or D-pad (or up and down arrows) to move through our world. Feel free to modify the `moveRate` value as desired.

Creating a Split Screen

Now that we know how to set up our camera and accept input, we can look into how our code will need to change to handle multiple players in a split-screen game. To start, we need to make a copy of the InputDemo project we just finished. We can rename the project SplitScreen. After we have our solution and projects renamed (complete with our assembly GUID and title) we can look at the code we will need to change to accomplish a split-screen mode of play.

To create a split screen, we need to two different viewports. We have only been using one up until now and we actually retrieved it in our camera's `Initialization` method. We simply grabbed the `GraphicsDevice.Viewport` property to get our camera's viewport. Because we want to display two screens in one we need to define our two new viewports and then let the cameras (we will need two cameras) know about them so we can get the desired effect. To start we will need to add the following private member fields to our Game1.cs code:

```
private Viewport defaultViewport;
private Viewport topViewport;
private Viewport bottomViewport;
private Viewport separatorViewport;
private bool twoPlayers = true;
private FirstPersonCamera camera2;
```

Then at the end of our `LoadContent` method we will need to define those viewports and create our cameras and pass the new values. We do this in the following code:

```
if (twoPlayers)
{
    defaultViewport = graphics.GraphicsDevice.Viewport;
    topViewport = defaultViewport;
    bottomViewport = defaultViewport;

    topViewport.Height = topViewport.Height / 2;

    separatorViewport.Y = topViewport.Height – 1;
    separatorViewport.Height = 3;

    bottomViewport.Y = topViewport.Height + 1;
    bottomViewport.Height = (bottomViewport.Height / 2) - 1;

    camera.Viewport = topViewport;

    camera2 = new FirstPersonCamera(this);
    camera2.Viewport = bottomViewport;
    camera2.Position = new Vector3(0.0f, 0.0f, -3.0f);
    camera2.Orientation = camera.Orientation;
    camera2.PlayerIndex = PlayerIndex.Two;
    Components.Add(camera2);
}
```

We discussed briefly that we would need more than one camera to pull this off. This is because we have our view and projection matrices associated with our camera class (which we should). It makes sense that we will have two cameras because the camera is showing what the player is seeing. Each player needs his or her own view into the game.

Our initial camera is still set up in our game's constructor, but our second camera will get added here. Our first camera gets the default viewport associated with it. The preceding code first checks to see if we are in a two-player game. For a real game, this should be determined by an options menu or something similar, but for now we have just initialized the value to true when we initialized the `twoPlayer` variable.

Inside the two-player condition the first thing we do is set our default viewport to what we are currently using (the graphic device's viewport). Then we set our top viewport to the same value. We also initialize our `bottomViewport` to our `defaultViewport` value. The final thing we do with our viewports is resize them to account for two players. We divide the height in two (we are making two horizontal viewports) on both. We then set our bottom viewport's Y property to be one more than the height of our top viewport. This effectively puts the bottom viewport right underneath our top viewport.

While still in the two-player condition, we change our first camera's viewport to use the top viewport. Then we set up our second camera by setting more properties. Not only do we set the viewport for this camera to the bottom viewport, but we also set a new camera position as well as the orientation of the camera. Finally, we set the player index.

None of these properties is exposed from our camera object, so we need to open our Camera.cs file and make some changes to account for this. First, we need to add a new private member field to hold our player index. We just assumed it was player 1 before. We can set up our protected index (so our `FirstPersonCamera` class can access it) as an integer, as follows:

```
protected int playerIndex = 0;
```

Now, we can modify the input code that controls our camera to use this index instead of the hard-coded value 0 for the game pads. In the camera's `Update` method we can change any instance of `input.GamePads[0]` to `input.GamePads[playerIndex]`. We also need to do the same for the `FirstPersonCamera` object. We did not update the keyboard code and will not for the sake of time. However, to implement multiple users where both can use the keyboard, we should create a mapping for each player and check accordingly. In general, it is a good practice to have a keyboard mapping so that if the gamer does not like the controls we have defined in our games, he or she has a way to change the controls so they work more logically for him or her. The same can be said about creating a mapping for the game pads, but many games simply give a choice of a couple of layouts. Because the code does not implement a keyboard mapping, the only way for us to control the separate screens differently is by having two game pads hooked up to the PC or Xbox 360.

After we have changed our camera to take the player index into consideration before reading values from the game pad, we can add the following properties to our Camera.cs code file:

```
public PlayerIndex PlayerIndex
{
    get { return ((PlayerIndex)playerIndex); }
    set { playerIndex = (int)value; }
}

public Vector3 Position
{
    get { return (cameraPosition); }
    set { cameraPosition = value; }
}

public Vector3 Orientation
{
    get { return (cameraReference); }
    set { cameraReference = value; }
}

public Vector3 Target
{
    get { return (cameraTarget); }
    set { cameraTarget = value; }
}
public Viewport Viewport
{
    get
    {
        if (viewport == null)
            viewport = Game.GraphicsDevice.Viewport;

        return ((Viewport)viewport);
    }
    set
    {
        viewport = value;
        InitializeCamera();
    }
}
```

We are simply exposing the camera's position, orientation (reference), and target variables. We are casting the player index property to a `PlayerIndex` enumeration type. The final property is the `Viewport` property. We first check to see if our viewport variable is null. If it is, we set it to the graphics device's viewport. When we set our `Viewport` property, we also call our `InitializeCamera` method again so it can recalculate its view and projection matrices. We need to set up a private member field for our viewport. We allow it to have a default null value so we can declare it as follows:

```
private Viewport? viewport;
```

Because we are utilizing the `Viewport` type, we need to make sure the following `using` statement is in our code:

```
using Microsoft.Xna.Framework.Graphics;
```

The only thing left for us to do now is to update our game's drawing code to draw our scene twice. Because we have to draw our scene twice (once for each camera), we need to refactor our `Draw` code into a `DrawScene` method and pass in a camera reference. Our new code for the new `Draw` method is shown here:

```
protected override void Draw(GameTime gameTime)
{
    graphics.GraphicsDevice.Viewport = camera.Viewport;
    DrawScene(gameTime, camera);

    if (twoPlayers)
    {
        graphics.GraphicsDevice.Viewport = camera2.Viewport;
        DrawScene(gameTime, camera2);
        //now clear the thick horizontal line between the two screens
        graphics.GraphicsDevice.Viewport = separatorViewport;
        graphics.GraphicsDevice.Clear(Color.Black);
    }

    base.Draw(gameTime);
}
```

We took all the code that was inside this method and put it into a new method, `DrawScene(GameTime gameTime, Camera camera)`. The code we put into the `DrawScene` method did not change from how it looked when it resided inside of the `Draw` method.

The first thing we do with the preceding code is set our graphics device's viewport to be what our camera's viewport is. We then draw the scene passing in our camera. Then we check to see if we have two players; if so, we set the viewport appropriately and finally draw the scene for that camera. Run the application and see that it is using a split screen! The screenshot of this split-screen demo can be seen in Figure 5.1.

Summary

Another chapter is behind us. You have learned about XNA game services and how they, along with game components, can really add benefit to our overall game architecture. We started a small library that currently handles our camera (which we can currently

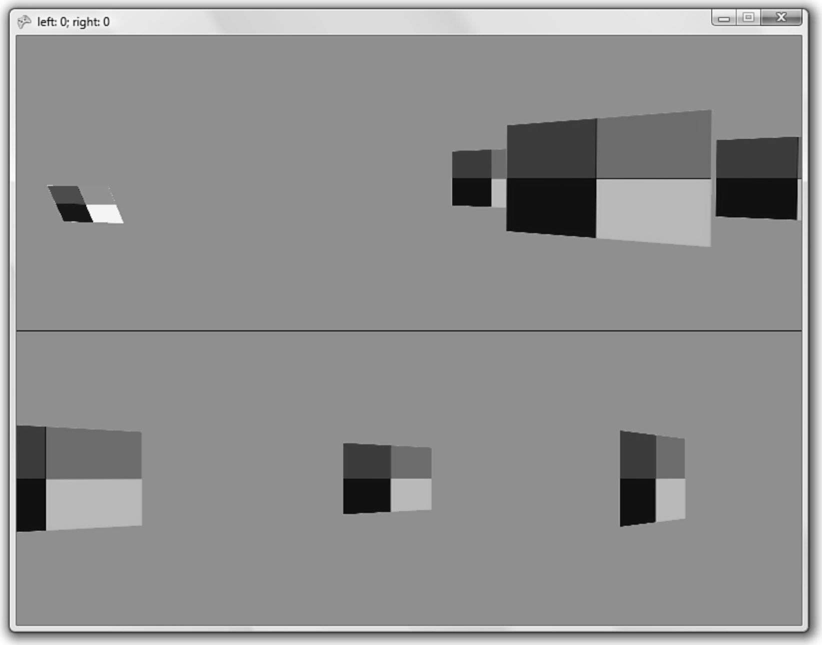

FIGURE 5.1 SplitScreen demo shows how you can create two cameras that can be controlled by two different game pads.

switch between stationary and first person), our input devices, and our frame rate counter component.

We discussed how to utilize the keyboard, game pad, and mouse to get input from our gamer to move our camera around. You learned specifically how to write a stationary camera that only rotates and created a functioning first-person camera.

We updated our camera functionality to handle two players. We added split-screen functionality by creating two different cameras and viewports the cameras could use.

In the next chapter, we are going to load 3D objects to the screen, which will allow us to move in a much better-looking world. Play some games, rest the mind, and come back strong as we jump right into working with the Content Pipeline.