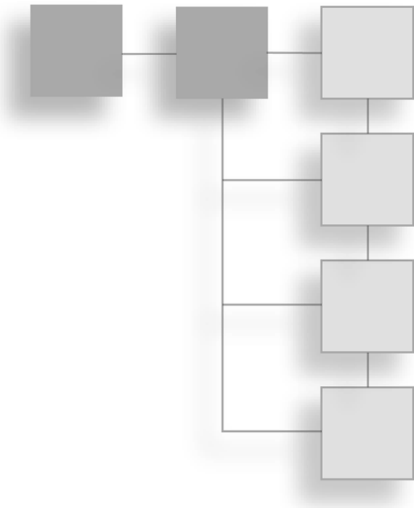
CHAPTER 7



Using XNA Input for Controllers

Games are useless unless the player can interact with them. In fact, there's a special name for a game that doesn't accept input—it's called a screen saver.

The XNA Framework provides support for three categories of user input devices: the Xbox 360 Controller, the keyboard, and the mouse. This chapter, and the two that follow, describes the differences between each, and presents code and design guidelines to help you use player input in your game to maximum effect. In this chapter, we will develop a simple game project that graphically shows the state of all the buttons of the Xbox 360 Controller and turns the rumble motors on and off to give the player tactile feedback.

In this chapter, you will learn how to:

- Process input from the Xbox 360 Controller (on Windows and the Xbox 360 game console)
- Use the vibration motors of the controller to present tactile feedback to the player
- Manage many in-game objects, each with independent states
- Break large game tasks into smaller, more manageable tasks

GamePad Overview

The XNA Framework has support for 14 digital buttons, 4 analog buttons, and 2 vibration motors when using the Xbox 360 Controller. The media button is not supported. Up to four actively connected controllers are supported, and the Xbox 360 Controller is supported on both the Windows and the Xbox 360 game console platforms. The number and variety of input sources on an Xbox 360 Controller make it the ideal input device for your games.

Note

The only type of controller that is supported by the XNA Framework is the Xbox 360 Controller. If you're writing a Windows game, and you want your game to support another type of controller (like the Microsoft Sidewinder Controller), you will need to access that controller via some other API.

Polling

Input from the controller is collected by the XNA Framework via polling. During every frame of your game, you will ask the controller what the current state of its buttons is. This state information is wrapped in an XNA Framework structure known as the `GamePadState`, and it represents a snapshot of the state of all the controller's buttons at the time the request was made. This state information can be saved from frame to frame (so that you can determine which buttons were pressed or released since the last frame), and you can pass the state as a parameter to your methods (so that you don't need to poll the controller repeatedly, running the risk of retrieving conflicting information).

If you've done any Windows programming in the past, you've dealt with event-driven programming. Polling may seem inefficient by comparison, and you might think that it would be easy to miss player input between `GetState` calls. But, you need to keep in mind that your game is running at 15, 30, 60, or more frames per second. When you're gathering input data 60 times a second, it's not very likely that a player can press and release a button between any two polling requests. If you want to test this out, run the example that we develop in this chapter and try to press and release a button without the screen registering your action.

Note

Per the XNA Framework documentation, the state of a controller is undefined when the controller is disconnected. The API will allow you to poll for the state of a disconnected controller, and there is a boolean property in the `GamePadState` structure that indicates whether the controller is connected or not. All of the other controller state information is unreliable, and should not be trusted. You may want to automatically pause your game whenever there are no controllers connected.

`GamePadState` information is retrieved using the `GetState` member of the `GamePad` class. Only four Xbox 360 Controllers are supported, and each must be polled separately. The following code shows how you would get the state of the controller used by player one.

```
// Poll the current controller state for player one
GamePadState padState1 = GamePad.GetState(PlayerIndex.One);
```

Digital Buttons

The Xbox 360 Controller has 14 (accessible) digital buttons. The 10 most obvious digital buttons are the A, B, X, Y, Start, Back, left shoulder, and right shoulder buttons. The four directions of the DPad are actually four separate digital buttons (one for each direction), and the left and right thumbsticks can be pressed, each behaving as a digital button. Digital buttons have only two states—pressed and released—so they can be represented with Boolean values in your code. The following code demonstrates how you would see if player one is pressing the A button on his controller.

```
// Poll the current controller state for player one
GamePadState padState1 = GamePad.GetState(PlayerIndex.One);

// make sure the controller is connected
if (padState1.IsConnected)
{
   if (padState1.Buttons.A == ButtonState.Pressed)
   {
      // player one is pressing the A button
   }
}
```

Analog Buttons

Unlike their digital siblings, analog buttons can report a range of values. The Xbox 360 Controller has two triggers on the back side of the controller. The state of each trigger is represented by a float, ranging from 0.0f (not pressed) to 1.0f (fully pressed). The controller also sports two directional thumbsticks. Each thumbstick has an x- and a y-axis. Each axis is represented by a float ranging from –1.0f to 1.0f. For the x-axis, –1.0f indicates that the player is pressing the stick fully to the left, 1.0f indicates that the player is pressing the stick fully to the right, and 0.0f indicates that the stick is not being used at all. Similarly for the y-axis, –1.0f represents down, 1.0f represents up, and 0.0f represents no action.

The following code demonstrates how you would see if player one is using the left thumbstick on his controller.

```
// Poll the current controller state for player one
GamePadState padState1 = GamePad.GetState(PlayerIndex.One);

// make sure the controller is connected
if (padState1.IsConnected)
{
   if (padState1.ThumbSticks.Left.X != 0.0f)
   {
      // player one is directing the left thumbstick to the left or right
   }
   if (padState1.ThumbSticks.Left.Y != 0.0f)
   {
      // player one is directing the left thumbstick up or down
   }
}
```

When processing analog input from the thumbsticks, there's a concept known as *dead zone processing* that you should be aware of. There are minor variances in manufacturing from controller to controller; and over time, movable parts can wear with use. When at rest, thumbsticks will almost always be slightly off center. To account for this, the `GetState` method automatically disregards values that are below a certain threshold. That threshold is known as the dead zone. If the dead zone weren't taken into account, you would see what's commonly referred to as "drift" in your game—phantom user actions, like moving left even though you're not pressing any buttons.

The overloaded version of the `GetState` API provides three methods of dead zone processing, each represented by a member of the `GamePadDeadZone` enumeration.

- **`IndependentAxes`**—The X and Y components of the thumbstick position are processed against the dead zone independently. This is the default behavior of the API if you don't identify a specific dead zone processing method. For most applications, this is the best, general-purpose processing mode.

- **`Circular`**—The X and Y components of the thumbstick position are combined before processing the dead zone. In some applications, this will provide better control.

- **None**—The raw values reported by the controller are returned to your game, and you can process the dead zone using your own logic.

Vibration

It's amazing how adding simple little details to a game can make it so much more immersive. Vibration effects are an effective way to draw your player into the experience. When his car hits a wall, or a grenade explodes just a few feet away, or he's just fallen to his death from a high-rise apartment complex, shake the controller to provide tactile feedback.

The Xbox 360 Controller has two vibration motors. The left is a low-frequency rumble motor, and the right is a high-frequency rumble motor. Each motor can be activated independently of the other. And each motor supports varying speeds. You can turn the motors on and off, and adjust their speeds, using the `SetVibration` method of the `GamePad` class. The following snippet demonstrates how you would enable the vibration motors.

```
// Poll the current controller state for player one
GamePadState padState1 = GamePad.GetState(PlayerIndex.One);

// make sure the controller is connected
if (padState1.IsConnected)
{
   if (padState1.Buttons.A == ButtonState.Pressed)
   {
      // shake the controller if the A button is pressed
      GamePad.SetVibration(PlayerIndex.One, 1.0f, 1.0f);
   }
   else
   {
      // otherwise, disable the motors
      GamePad.SetVibration(PlayerIndex.One, 0f, 0f);
   }
}
```

Note

The preceding example simply set each motor to its maximum speed, but you can create all sorts of effects by combining various motor speeds and varying the settings over time. Imagine enabling and disabling the motors in rapid succession to simulate riding over a gravel road or firing a machine gun. Or think about using the low-frequency motor to provide constant, pulsing ambient feedback when the player enters a room filled with humming alien technology.

Wrapper Class

It's a good idea to wrap the `GamePad` and `GamePadState` functionalities within your own custom helper class. That way, you can make global changes to how your game processes input by changing one source code file. Imagine that you've written a game that uses the controller. This game has several types of levels—each programmed as a separate C# class and each accessing the controller via the XNA Framework's `GamePad` class. After play testing, you decide to provide an option so that the player can reverse the controls for looking up and down (a common option in most first-person shooter games).

Without the wrapper class, you'll need to edit and test every class that directly accesses the controller APIs. With the wrapper class, you edit one source code file, and the change is inherited by all your custom C# game classes that use it. Of course, you'll also want to provide a way to temporarily return to your default controller mappings for menus, but there is great benefit in centralizing your controller logic.

Let's consider another scenario. Imagine that you've written your game solely for the Xbox 360 game console. When you're done writing it, you decide that you would also like to release it as a Windows game. Not every Windows gamer will have an Xbox 360 Controller connected to his PC, so you decide to add keyboard support. Touching every game screen that accepts controller input will be a pain. You don't want to punish the Windows gamers with Xbox 360 Controllers by simply replacing your controller logic with keyboard logic, so you'll need to make sure that your game logic gracefully combines input from both sources. If you're using a wrapper class, you can intercept keyboard input, map it to controller input, and inject phantom button presses into the controller state that your game uses. In fact, we will develop such a beast in a later chapter.

If you do decide to write your own custom wrapper to gather user input, you should be aware of the fact that the `GamePadState` structure contains a handy little member named `PacketNumber` that will let you know if the controller state has changed since the last time you polled it. Many times, there will be relatively long spans where the player is pressing the same button or combination of buttons. In a racing game, the player may be on a straightaway, pressing the accelerator all the way in. In a first-person shooter, the player may be crouched in a corner, waiting to snipe one of his buddies. On a pause screen, the player has likely put the controller down and isn't pressing anything.

The Example

We will develop a simple ''game'' that exercises each of the features of the controller. Each digital button will be represented on the screen with a graphic, and when the player presses a button, its graphic will change to reflect the change in state. The screen will also contain ''slider'' images where the state of each of the analog buttons will be presented. In addition to the button images, there will be a small viewport where a ship flies around on a sheet of graph paper. As the player uses any of the directional controls to move left or right, the ship will rotate. When the player presses the triggers or uses any of the directional controls to move up or down, the ship will move.

Figure 7.1 shows a screenshot of the final game.

The Source Images

I created two graphics for each digital button, one for each state (pressed and released). To lay out the screen, I played around with the images in a paint program, moving them around until I was happy with the rough design. Figures 7.2, 7.3, and 7.4 show the source graphics. I divided the collection of images into three files, but this separation is fairly arbitrary.

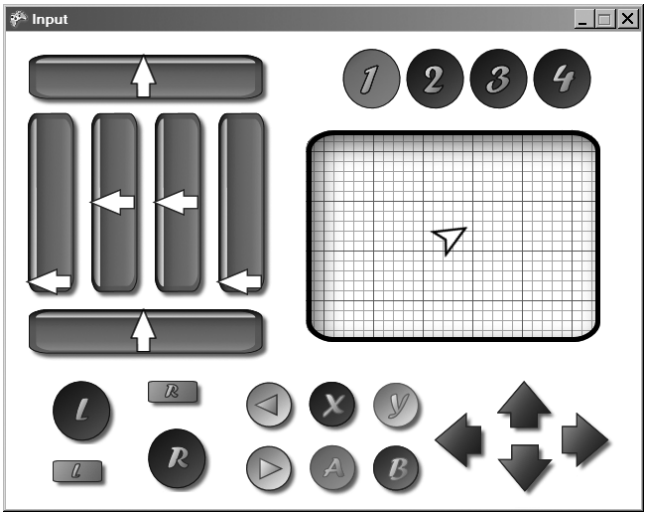


Figure 7.1
A screenshot of the example game that we will develop in this chapter.

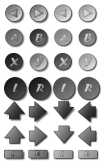


Figure 7.2
The sprites that represent the pressed and released states for each of the 14 digital buttons.

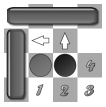


Figure 7.3
The sprites for the sliders to show the state of the six analog buttons, as well as the sprites that show which controller ports are active.

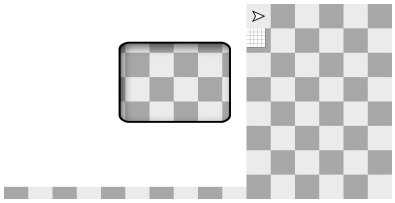


Figure 7.4
The background sprite that will be layered on top of the scrolling grid. The game screen is 640 × 480, but the image is 1024 × 512 (dimensions as powers of 2).

The first source image, shown in Figure 7.2, is named "buttons," and it contains the pressed and released graphics for each of the 14 digital buttons.

The second image, shown in Figure 7.3, is named "analog," and it contains the slider bars and arrows that I use to render the states of the analog buttons. It also contains the images that I use to indicate which of the four Xbox 360 Controllers are currently connected. Rather than having eight separate images of the controller connection states (four controllers, each with two states), I decided to build a composite image at runtime by layering the images that you see here. The actual source image is transparent, but I've added a simple checkerboard pattern so that you can see how the individual components of the image line up with each other.

The final image, shown in Figure 7.4, is named "background," and it contains the graph paper graphic that is tiled across the entire game screen, as well as the ship, and the 640-by-480-pixel background image, which includes the viewport

(complete with drop shadow). The actual source image is transparent, but I've added a simple checkerboard pattern so that you can see how the individual components of the image line up with each other.

The ButtonSprite Class

The individual graphics are lined up in the source image so that I can render each of the button state graphics to the same X and Y coordinates on the screen. When it's time to draw the button on the screen, I select the appropriate graphic based on the current state of the button it represents. I wanted to be able to easily move the buttons around on the screen in case I decided to change my layout, so I created a simple C# class to represent the on-screen button.

For each button, I need references to two `Texture2D` objects—one for the pressed state and one for the released state. Since my source images contain multiple graphics, I can use the same texture for many of the buttons, but I'll still need some way to remember where the individual graphics are within the larger source image. To keep track of their locations, I'll use a `Rectangle` for each state.

```
private Texture2D m_TextureNormal;
public Texture2D TextureNormal
{
    get { return m_TextureNormal; }
    set { m_TextureNormal = value; }
}

private Rectangle m_RectNormal = Rectangle.Empty;
public Rectangle RectNormal
{
    get { return m_RectNormal; }
    set { m_RectNormal = value; }
}

private Texture2D m_TexturePressed;
public Texture2D TexturePressed
{
    get { return m_TexturePressed; }
    set { m_TexturePressed = value; }
}

private Rectangle m_RectPressed = Rectangle.Empty;
public Rectangle RectPressed
```

```
{
    get { return m_RectPressed; }
    set { m_RectPressed = value; }
}
```

I also need to know where to draw the button on the screen. To store the X and Y coordinates, I'll use a `Vector2D` structure.

```
public Vector2 Location = Vector2.Zero;
```

Using this basic class, I can place the `ButtonSprite` anywhere on the screen, and not worry about the details of how it's rendered. Each `ButtonSprite` will be rendered the same way—using code similar to that found in the following snippet.

```
// bat is an XNA SpriteBatch class, btn is our custom ButtonSprite class
bat.Draw(btn.TexturePressed, btn.Location, btn.RectPressed, Color.White);
```

Note

Rather than writing new classes to support the other (non-digital) buttons, I reuse the properties of this class. Where I deviate from the obvious functionality, I've included comments in the source code. For example, the left trigger is an analog control that has no "pressed" state. The `TextureNormal` and `RectNormal` properties of the `ButtonSprite` are used to define the texture and bounds of the vertical slider graphic. Since there is no pressed state, I use the `RectPressed` structure to denote the bounds of the "usable" area of the slider—the subset of the graphic that excludes the rounded edges, the transparent areas, and the drop shadow. This inner rectangle is used to place and constrain the vertical bar arrow graphic.

Draw Methods

The final game screen that you see is made up of many independent components. I've broken the task of rendering all of these components into separate, specialized drawing methods.

- `DrawButton` renders a graphic based on a digital button's state (pressed or released).
- `DrawHBar` renders a slider bar and its arrow to reflect the state of one of the thumbstick's x-axis values.
- `DrawVBar` renders a slider bar and its arrow to reflect the state of one of triggers or one of the thumbstick's y-axis values. Since triggers have a range of values from zero to one, and thumbsticks report values between negative one and positive one, this method assumes that the vertical bar

represents a trigger. By passing it a value of –1 for the optional `min` parameter, the bar can represent the full range of a thumbstick.

- `DrawGraph` fills the screen with the graph paper tile, rendered relative to the ship's current location. The results of this render step will be overlayed with the background image and the graph paper will show through the viewport. You could save some processing by only rendering the part of the paper that will be visible through the viewport, but filling the entire screen means that you can move your viewport to another part of the screen in your graphics editor without editing the rendering code.
- `DrawCursor` draws the ship, at its current rotation.

For details on the implementation of these methods, keep reading. They're described in greater detail in the section entitled, "Drawing the Game." You can also find the full source code for this game on the CD-ROM that came with the book.

Update()

The `Update` method of this game includes logic to poll the controllers and update the ship's location based on the player's input. The ship itself doesn't actually move on the screen. The paper texture below the ship moves to show the relative direction and speed of the ship.

The `Update` method is where the code that tracks the state of each of the controller buttons is housed. And this is also where the vibration motors on the controller are set to rumble as long as the A button is pressed on the controller.

While the state of each of the four controllers is polled, only the controller in port one can affect the state of the button images on the screen. The other controller states are only used to detect when those controllers are added or removed, updating the port connection indicator images.

```
/// run logic such as updating the world
protected override void Update(GameTime gameTime)
{
    // capture pad state once per frame
    m_pad1 = GamePad.GetState(PlayerIndex.One);
    m_pad2 = GamePad.GetState(PlayerIndex.Two);
    m_pad3 = GamePad.GetState(PlayerIndex.Three);
    m_pad4 = GamePad.GetState(PlayerIndex.Four);
```

```
// only process input from player one, and only if
// the controller is connected
if (m_pad1.IsConnected)
{
    // combine states to rotate left, true if any are pressed
    bool bLeft = m_pad1.DPad.Left == ButtonState.Pressed;
    bLeft |= m_pad1.ThumbSticks.Left.X < 0;
    bLeft |= m_pad1.ThumbSticks.Right.X < 0;
    if (bLeft) { m_angle -= 5.0f; }

    // combine states to rotate right, true if any are pressed
    bool bRight = m_pad1.DPad.Right == ButtonState.Pressed;
    bRight |= m_pad1.ThumbSticks.Left.X > 0;
    bRight |= m_pad1.ThumbSticks.Right.X > 0;
    if (bRight) { m_angle += 5.0f; }

    // distance to travel per frame, split into X and Y
    float dx = (float)Math.Cos(m_angle * ToRadians);
    float dy = (float)Math.Sin(m_angle * ToRadians);

    // check button states to determine thrust
    float fMove = 0.0f; // assume no movement

    // is the player moving the ship?
    if (m_pad1.ThumbSticks.Left.Y != 0.0f)
    {
        fMove = m_pad1.ThumbSticks.Left.Y;
    }
    else if (m_pad1.ThumbSticks.Right.Y != 0.0f)
    {
        fMove = m_pad1.ThumbSticks.Right.Y;
    }
    else if (m_pad1.Triggers.Right != 0.0f)
    {
        fMove = m_pad1.Triggers.Right;
    }
    else if (m_pad1.Triggers.Left != 0.0f)
    {
        fMove = -m_pad1.Triggers.Left;
    }
    else if (m_pad1.DPad.Up == ButtonState.Pressed)
    {
        // treat as max thumbstick Y
```

```
            fMove = 1.0f;
        }
        else if (m_pad1.DPad.Down == ButtonState.Pressed)
        {
            // treat as min thumbstick Y
            fMove = -1.0f;
        }

        // ship's thrust is relative to analog button states
        m_GraphOrigin.X -= dx * fMove;
        m_GraphOrigin.Y -= dy * fMove;

        // make sure that 0 <= graph origin x <= 50
        while (m_GraphOrigin.X < 0.0f)
        {
            m_GraphOrigin.X += 50.0f;
        }
        while (m_GraphOrigin.X > 50.0f)
        {
            m_GraphOrigin.X -= 50.0f;
        }

        // make sure that 0 <= graph origin y <= 50
        while (m_GraphOrigin.Y < 0.0f)
        {
            m_GraphOrigin.Y += 50.0f;
        }
        while (m_GraphOrigin.Y > 50.0f)
        {
            m_GraphOrigin.Y -= 50.0f;
        }

        // shake the controller while the A button is pressed
        if (m_pad1.Buttons.A == ButtonState.Pressed)
        {
            GamePad.SetVibration(PlayerIndex.One, 1.0f, 1.0f);
        }
        else
        {
            GamePad.SetVibration(PlayerIndex.One, 0f, 0f);
        }
    }
    base.Update(gameTime);
}
```

Drawing the Game

The standard `Draw` method of this game includes logic to build the screen, piece by piece. First, the graph paper is drawn via a call to `DrawGraph`. Then, the background image is rendered over the graph tiles so that they can show through the viewport. Then, all of the digital and analog buttons are drawn, along with the four controller connection states, via a call to `DrawButtons` (note the plural version). And finally, the ship is drawn via a call to `DrawCursor`.

DrawGraph

The `DrawGraph` method is very similar to the logic that you see in Chapter 8, "Using XNA Input for Keyboards." A small image is tiled to form an array of images that cover the screen.

Covering the screen is a bit of overkill in this example, since most of the graph is obscured by the rest of the interface. But, by covering the entire screen, we can change the size or location of the window in the interface (that lets the graph show through) without worrying whether there will be any visible gaps in the tiles.

```
// render the graph paper
private void DrawGraph(SpriteBatch batch)
{
    // a single graph tile is only 50-by-50, so repeat
    // it as many times as needed to cover the entire
    // game screen. since the paper is overlaid with the
    // background image, we don't need to worry too much
    // about the edges.

    // temp variable for tiling
    Vector2 vSquare = new Vector2();

    // round to nearest pixel
    float oy = (float)Math.Round(m_GraphOrigin.Y);
    float ox = (float)Math.Round(m_GraphOrigin.X);

    for (float y = oy; y < SCREEN_HEIGHT; y += 50.0f)
    {
        // row by row
        vSquare.Y = y;
        for (float x = ox; x < SCREEN_WIDTH; x += 50.0f)
        {
            // column by column
```

```
            vSquare.X = x;
            batch.Draw(m_texBackground, vSquare,
               m_rectGraph, Color.White);
        }
    }
}
```

DrawVBar and DrawHBar

The `DrawVBar` and `DrawHBar` methods are very similar, so I'll list the more general of the two. The value of an analog button is translated to a percentage (0.0f - 1.0f) of the button's full range, and that adjusted value is used to map the analog state to an on-screen pixel. A graphic that represents the button's full range of values is drawn, and an arrow is overlaid to show the current value.

```
// overload for DrawVBar, with default min
private void DrawVBar(SpriteBatch batch, ButtonSprite btn,
    float value)
{
    DrawVBar(batch, btn, value, -1.0f);
}

// draw the bar and the arrow
private void DrawVBar(SpriteBatch batch, ButtonSprite btn,
    float value, float min)
{
    // determine the X of the arrow
    // NOTE: btn.RectNormal describes the bounds of the image
    // btn.RectPressed describes the bounds of the bar itself
    m_btnVBarArrow.Location.X =
        btn.Location.X +
        btn.RectPressed.X +
        btn.RectPressed.Width / 2 -
        m_btnVBarArrow.RectNormal.Width / 2;

    if (min < 0.0f)
    {
        // value is between -1.0f and 1.0f. offset value
        // so that value is between 0.0f and 2.0f
        value += 1.0f;
        // then scale so that value is
        // between 0.0f and 1.0f
        value /= 2.0f;
    }
```

```
    // since value is now between 0 and 1, we can treat it
    // like a percentage. so, Y becomes value percent of
    // Height. NOTE: need to invert value since Y values
    // increase as you move down the screen. (see line with
    // "// bottommost" comment)
    m_btnVBarArrow.Location.Y =
       btn.Location.Y + btn.RectPressed.Y +  // topmost pixel
       btn.RectPressed.Height -               // bottommost
       btn.RectPressed.Height * value -      // scaled value
       m_btnVBarArrow.RectNormal.Height / 2; // arrow midpoint

    // draw bar
    batch.Draw(btn.TextureNormal, btn.Location,
        btn.RectNormal, Color.White);
    // draw arrow
    DrawButton(batch, m_btnVBarArrow, false);
}
```

DrawPort

The `DrawPort` method calls out to the `DrawButton` method to draw a green (if active) or gray (if inactive) button, and then overlays a number image to indicate the state and number of the port.

```
// draw the active port indicators
private void DrawPort(SpriteBatch batch, ButtonSprite btn,
    int index, bool active)
{
    // gray (inactive) or green (active) circle
    DrawButton(batch, btn, active);
    // port number
    batch.Draw(btn.TextureNormal, btn.Location,
        m_rectPortNum[index], Color.White);
}
```

DrawButton

The `DrawButton` method is used by nearly all of the other draw methods. It uses the data that's contained within our `ButtonSprite` class to draw an image at the proper location, with the proper dimensions, and (with the help of the `pressed` parameter) in the proper color.

```
// draw the button at its current location in its current state
```

```
private void DrawButton(SpriteBatch batch, ButtonSprite btn,
    bool pressed)
{
    if (pressed)
    {
        batch.Draw(btn.TexturePressed, btn.Location,
            btn.RectPressed, Color.White);
    }
    else
    {
        batch.Draw(btn.TextureNormal, btn.Location,
            btn.RectNormal, Color.White);
    }
}
```

LoadGraphicsContent()

The standard `LoadGraphicsContent` method of this game includes the logic to load the three source images and to define the location of each graphic with the images. When you view the source code, you'll notice that whenever possible, many buttons will share the same `Texture2D` object. That way, we're not loading the same image into memory over and over.

```
/// Load your graphics content.
protected override void LoadGraphicsContent(bool loadAllContent)
{
    if (loadAllContent)
    {
        // local temp variables
        Texture2D texTemp;
        Rectangle recTemp;

        // initialize our sprite batch here
        m_batch = new SpriteBatch(graphics.GraphicsDevice);

        // background, cursor, and graph textures
        m_texBackground =
            content.Load<Texture2D>(@"media\background");

        // button textures
        texTemp = content.Load<Texture2D>(@"media\buttons");

        // init A
```

```
            m_btnA.TextureNormal = texTemp;
            m_btnA.RectNormal =
                new Rectangle(0, 64, 64, 64);
            m_btnA.TexturePressed = texTemp;
            m_btnA.RectPressed =
                new Rectangle(128, 64, 64, 64);

            // init B
            m_btnB.TextureNormal = texTemp;
            m_btnB.RectNormal =
                new Rectangle(64, 64, 64, 64);
            m_btnB.TexturePressed = texTemp;
            m_btnB.RectPressed =
                new Rectangle(192, 64, 64, 64);

            // init X
            m_btnX.TextureNormal = texTemp;
            m_btnX.RectNormal =
                new Rectangle(0, 128, 64, 64);
            m_btnX.TexturePressed = texTemp;
            m_btnX.RectPressed =
                new Rectangle(128, 128, 64, 64);

            // NOTE: listing edited to conserve space
            //       see accompanying CD for full listing

            // layout the UI
            PositionButtons();
        }
    }
```

Once all the images have been loaded and the bounds of the button graphics have been defined, `LoadGraphicsContent` makes a call to `PositionButtons` to define where each of the buttons will be drawn on the screen.

Ideally, groups of buttons would be positioned relative to each other so that they can be moved as a single unit (e.g., A, B, X, Y as a group, the DPad arrows as a group). But in the interest of brevity and clarity, I just hard-coded the X and Y values for each button. If you wanted to change this code so that the buttons were positioned relative to each other, or you wanted to devise some way to load the button locations from an XML or INI file, the `PositionButtons` method is where you would inject your changes.

```
public void PositionButtons()
{
    // horizontal analog bars
    m_btnHBarLThumb.Location.X = 15;
    m_btnHBarLThumb.Location.Y = 15;

    m_btnHBarRThumb.Location.X = 15;
    m_btnHBarRThumb.Location.Y = 271;

    // vertical analog bars
    m_btnVBarLTrigger.Location.X = 15;
    m_btnVBarLTrigger.Location.Y = 79;

    m_btnVBarLThumb.Location.X = 79;
    m_btnVBarLThumb.Location.Y = 79;

    m_btnVBarRThumb.Location.X = 143;
    m_btnVBarRThumb.Location.Y = 79;

    m_btnVBarRTrigger.Location.X = 207;
    m_btnVBarRTrigger.Location.Y = 79;

    // Left Thumbstick button
    m_btnLThumb.Location.X = 47;
    m_btnLThumb.Location.Y = 351;

    // Left Shoulder button
    m_btnShoulderLeft.Location.X = 47;
    m_btnShoulderLeft.Location.Y = 431;

    // NOTE: listing edited to conserve space
    //       see accompanying CD for full listing
}
```

Summary

In this section, you learned how to capture and process player input from an Xbox 360 Controller. You learned how to use this data to update objects in your game world. You saw how large tasks can be broken down into a series of smaller, more manageable tasks. You learned how to send feedback to the user through the controller using the vibration motors. You learned about dead zone processing and what support XNA Framework includes to manage dead zones. And

we discussed the importance of wrapping the standard `GamePad` APIs within your own centralized, custom class.

Review Questions

Think about what you've read in this chapter (and the included source code) to answer the following questions.

1. What is a thumbstick "dead zone"? How can you avoid "drift" in your games?
2. What are some reasons that you should write your own wrapper class for the `GamePad` APIs?
3. Can you use a standard Xbox 360 Controller on your PC? Can you use another PC joystick, like the Microsoft Sidewinder?
4. Does the XNA Framework provide support for novel input devices like light guns? If so, how? If not, why?
5. How many vibration motors does an Xbox 360 Controller have? How do they differ from each other?

Exercises

EXERCISE 1. Modify this code so that the maximum speed of the ship is double what it is now.

EXERCISE 2. Expand this example so that input from any attached Xbox 360 Controller will be reflected on the screen.

EXERCISE 3. Modify this code so that the controller starts vibrating when the player presses the A button, and continues to vibrate until the player presses the B button.