SAMPLE CHAPTER

Windows Phone 7 IN ACTION

Timothy Binkley-Jones
Massimo Perga
Michael Sync

MANNING

Windows Phone 7 in Action

by Timothy Binkley-Jones
Massimo Perga
Michael Sync

Chapter 8

brief contents

8 Using sensors

This chapter covers

- Sensor API design
- Interpreting sensor data
- Using sensors in the emulator
- Moving with the motion sensor

Some really cool applications can be built combining sensors with other features of the phone. Applications might respond to a user shaking the device by randomly selecting an object or clearing the screen. Games can use the device's movement as an input mechanism, basically turning the whole phone into a game controller. Another class of applications augments the real world with computer-generated information. Augmented reality apps can show you the location of friends nearby relative to your current location. Astronomy applications determine the position of your device and identify the stars in the night sky. A tourist application might be able to identify nearby landmarks.

All of these applications require sensor input from the physical world. The phone's accelerometer, compass, and gyroscope sensors capture input from the real world and serve the data to applications through the Windows Phone SDK's Sensor API. When combined with location data from the phone's Location Service,

stunning augmented reality applications are possible. We discuss the location service in chapter 13.

Dealing with raw data from the sensors can be tricky, especially when trying to calculate the direction in which a device is pointed. The `Motion` sensor takes input from each of the other sensors, performs several complex calculations, and provides data related to motion and a device's relative position in the real world.

In this chapter you're going to build two similar sample applications. The first sample application uses the `Accelerometer`, `Compass`, and `Gyroscope` to demonstrate how the sensors are similar, and how they differ. The second sample application demonstrates how the `Motion` sensor is a wrapper around the three other sensors.

Before we dive into the first sample application, we introduce the common Sensor API that's the foundation for the sensors exposed by the Windows Phone SDK.

8.1 Understanding the sensor APIs

Whereas the `Accelerometer`, `Compass`, `Gyroscope`, and `Motion` sensors each return different types of data, they each implement the same pattern for reporting their data. Over the next several pages, you'll learn techniques that are useful for reading data from any of the sensors. We show you these techniques as you build the foundation of the sample application.

The classes and interfaces that comprise the Sensor API are found in the `Microsoft.System.Devices` namespace which is implemented in the assembly with the same name. Projects that use the Sensor API must include a reference to the `Microsoft.System.Devices` assembly, as well as the `Microsoft.Xna.Framework` assembly. The Sensor API uses the `Vector3`, `Matrix`, and `Quaternion` structs defined in the `Microsoft.Xna.Framework` namespace.

The `Accelerometer`, `Compass`, `Gyroscope`, and `Motion` sensors share a common base class named `SensorBase`. The `SensorBase` class is a generic class where the type parameter is a class that implements `ISensorReading`. We can look at how the `Accelerometer` class is declared in an example:

```
public sealed class Accelerometer : SensorBase<AccelerometerReading>
```

The `ISensorReading` interface is a marker interface defining a single `DateTimeOffset` property named `Timestamp`. The `SensorBase` class defines the common behavior of the sensor implementations. `SensorBase` declares one event and several properties and methods, which are described in table 8.1.

Table 8.1 SensorBase members

Member	Type	Description
`CurrentValue`	Property	The read-only `ISensorReading` containing the currently available sensor data.
`CurrentValueChanged`	Event	The event raised when the `CurrentValue` property changes.

Table 8.1 SensorBase members *(continued)*

Member	Type	Description
`Dispose`	Method	Releases the hardware and other resources used by the sensor.
`IsDataValid`	Property	A read-only property that returns `true` if `CurrentValue` contains valid data.
`Start`	Method	Enables the sensor and begins data collection.
`Stop`	Method	Disables the sensors and ends data collection.
`TimeBetweenUpdates`	Property	Specifies how often the sensor reads new data. The `CurrentValue` property will only change once every time interval.

An application obtains the current sensor reading by calling the `CurrentValue` method. Alternatively, an application can subscribe to the `CurrentValueChanged` event to receive a sensor reading only when the `CurrentValue` property changes. The `CurrentValue` can be read even when the sensor isn't started, but the value returned may not be valid and `IsDataValid` will return false.

> **NOTE** If the `ID_CAP_SENSORS` capability isn't present in the WMAppManifest .xml file, attempts to read the current value, set the time between updates, or start the sensor will result in a `UnauthorizedAccessException`.

Each of the sensor classes defines a static method named `IsSupported`. The `IsSupported` method allows a developer to determine whether the sensor hardware is installed on a particular device and whether the sensor is available to the application. If the `IsSupported` method returns false, attempts to read the current value, set the time between updates, or start the sensor will result in an `InvalidOperationException`.

The Sensor API handles fast application switching on its own. Developers don't need to unhook the sensors when the application is switched from the foreground. Unlike the camera, sensors automatically resume and don't provide an explicit restore method. When the application is resumed, the sensors and events are reconnected and data starts to flow again. Before you learn how to work with the data flowing from the sensors, you need to understand how the sensors report data in three dimensions.

8.1.1 Data in three dimensions

Each of the sensors reports data relative to the x, y, z coordinate system defined by the Windows Phone device. The device's coordinate system is fixed to the device, and moves as the phone moves. The x axis extends out the sides of the device, with positive x pointing to the right side of the device, and negative x pointing to the left side of the device. The y axis runs through the top and bottom of the device, with positive y pointing toward the top. The z axis runs from back to front, with positive z pointing

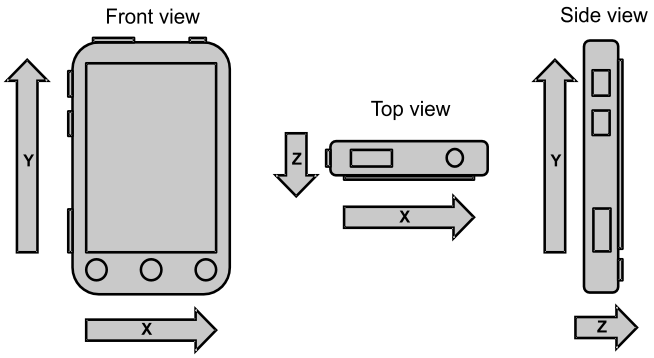


Figure 8.1 The x, y, z coordinate system as defined by a Windows Phone

out the front of the device. Figure 8.1 shows the x, y, and z axes from three different views of a phone.

The coordinate system used by the sensors doesn't necessarily match the coordinate system used by other APIs. One example is the coordinate system used by Silverlight. In portrait mode Silverlight, the y axis points in the opposite direction, with positive y pointing out the bottom of the device. Now that you understand the coordinate system used by the sensors, let's take a closer look at reading data from the sensors.

8.1.2 Reading data with events

Each of the sensors supports an event-driven interaction model with the `CurrentValueChanged` event. The `CurrentValueChanged` event sends a `SensorReadingEventArgs` instance to an event handler. `SensorReadingEventArgs` is a generic class where the type parameter is a class that implements `ISensorReading`. The type parameter matches the type parameter defined by the `SensorBase` class. The `SensorReadingEventArgs` class has a single property name `SensorReading`, which returns the data contained in the sensor's `CurrentValue` property at the time the event was raised.

The `CurrentValueChanged` event handler is called on a background thread. If the event handler updates the user interface, the update logic must be dispatched to the UI thread. The following code snippet shows an example that handles the `CurrentValueChanged` event from the motion sensor:

```
void sensor_CurrentValueChanged(object sender,
    SensorReadingEventArgs<MotionReading> e)
{
    MotionReading reading = e.SensorReading;
    Dispatcher.BeginInvoke(() =>
    {
        // add logic here to update the UI with data from the reading
        ...
    }
}
```

You'll use `CurrentValueChanged` later in the chapter when you build the second sample application. The first sample application will poll for data using the `CurrentValue` property.

8.1.3 Polling for data

An application doesn't need to wait for the sensor to raise an event to ask for data. Each sensor exposes data through the `CurrentValue` property. The `CurrentValue` property can be read whenever the application data determines it needs new data. For example, the reading might be initiated from a button click, a timer tick event, or a background worker:

```
if (Compass.IsSupported && compassSensor.IsDataValid)
{
    CompassReading reading = compassSensor.CurrentValue;
    // add logic here to use the data from the reading
    ...
}
```

You'll read sensor data from a timer tick event in your first sample application. Before we can show you the sensors in action, you need to create a new project and prepare the application to display sensor data.

8.2 Creating the sample application

Open Visual Studio and create a new Windows Phone Application named Sensors. The sample application will read values from the `Accelerometer`, the `Compass`, and the `Gyroscope`. By default, Windows Phone Application projects don't reference either the Sensors or the XNA Framework assemblies, and you need to add references to Microsoft.Devices.Sensors.dll and Microsoft.Xna.Framework.dll.

The sample application, shown in figure 8.2, displays a set of colored bars for each of the sensors. Each set of bars displays sensor readings for the x, y, and z coordinates. At the bottom of the screen, the application displays a legend and informational messages about the sensors.

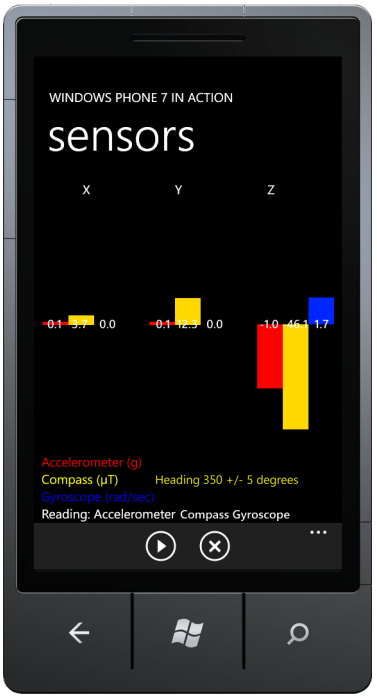


Figure 8.2 The Sensors sample application

When a sensor's value is positive, a bar will be drawn to scale above the effective zero line. A negative sensor value results in a bar drawn below the zero line. Since the range of possible values differs between each sensor, the height of the bar is transformed from the sensors value into a pixel height using a scaling factor. We'll talk more about each sensor's range of values throughout the chapter. First, you'll create a reusable control to display the positive and negative bars.

8.2.1 Creating a reusable Bar control

To simplify your application, you'll use a reusable control that allows you to set a scale factor and a sensor value. When the scale or value properties change, the control should draw the appropriate positive or negative bar, and display the value with a label. You'll implement the control using the Windows Phone User Control item template. Name the new item `Bar`. The XAML markup for the new control is shown in the next listing.

Listing 8.1 Markup for the Bar control

```
<Grid x:Name="LayoutRoot">
    <Grid.RowDefinitions>
        <RowDefinition Height="1*" />
        <RowDefinition Height="1*" />
    </Grid.RowDefinitions>
    <Rectangle x:Name="positiveBar" VerticalAlignment="Bottom" />
    <Rectangle x:Name="negativeBar" Grid.Row="1" VerticalAlignment="Top" />
    <TextBlock x:Name="label" VerticalAlignment="Center"
        Grid.RowSpan="2" Text="0" TextAlignment="Center" />
</Grid>
```

❶ Divide control into two rows

❷ Center label

The grid is divided into two halves ❶ with each half containing a `Rectangle`. The first `Rectangle` displays positive values and the other represents negative values. A label is placed right in the middle ❷ to show the bar's value.

Pages that host a `Bar` control need the ability to set different fill colors for the rectangles. Add a new property named `BarFill` to Bar.xaml.cs code behind file:

```
public Brush BarFill
{
    get { return positiveBar.Fill; }
    set
    {
        positiveBar.Fill = value;
        negativeBar.Fill = value;
    }
}
```

The setter for the `BarFill` property assigns the specified `Brush` to both the `positiveBar` and `negativeBar` rectangles.

> **NOTE** If you were building a proper reusable Silverlight control, the `BarFill` property and the other properties you're going to create would be dependency properties. You'd declare template parts and your XAML markup would be the default template. See the book *Silverlight 5 in Action* for more details on building reusable Silverlight controls.

Next you create properties to set the scale and value for the bar. Since you don't know the full range of values, you need the caller to tell the control how to scale the value to the height of the rectangles. Let's say you need the bar to display a value between 2 and -2, and the `Bar` control is 200 pixels high. A value of 2 would require the positive bar to be a hundred pixels high, whereas a value of -1 would require the negative bar

to be 50 pixels high. The next listing details how the bar height is calculated using the `Scale` and `Value` properties.

Listing 8.2 Calculating bar height with the `Scale` and `Value` properties

```
private int scale;
public int Scale
{
    get { return scale; }
    set
    {
        scale = value;
        Update();                                   ❶ Recalculate when properties change
    }
}

private float barValue;
public float Value
{
    get { return barValue; }
    set
    {
        barValue = value;
        Update();
    }
}

private void Update()
{
    int height = (int)(barValue * scale);           ❷ Calculate height of bar
    positiveBar.Height = height > 0 ? height : 0;
    negativeBar.Height = height < 0 ? height * -1 : 0;   ❸ Invert negative height
    label.Text = barValue.ToString("0.0");
}
```

You implement both the `Scale` and the `Value` properties with backing fields and simple getters and setters. Inside the setter of each property, you call the `Update` method ❶ to recalculate the height of the bar rectangles and update the user interface. Inside the `Update` method you multiply the `scale` and `barValue` fields ❷, and the resulting value is the number of pixels high the bar should be drawn. If the calculated height value is greater than 0, the `positiveBar`'s `Height` is updated to the new value. If the calculated height value is less than zero, you invert the calculated value ❸ before assigning the `negativeBar`'s height. Finally, you use the `ToString` method with a formatting string to set the label's `Text` property.

Now that you have a bar control, you can create your application's user interface. You need to add an XML namespace to MainPage.xaml so that you can use your new bar control:

```
xmlns:l="clr-namespace:Sensors"
```

You're now ready to use the `Bar` control in the MainPage's XAML markup. You need to design the MainPage to have three `Bar` controls for each sensor, for a total of nine `Bar` controls.

8.2.2 Designing the main page

If you look at the screenshot in figure 8.2, you'll notice that MainPage.xaml is divided into three rows and several columns. The markup for the `ContentPanel` of MainPage .xaml is shown in the next listing.

Listing 8.3 Markup for MainPage.xaml

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <Grid.RowDefinitions>
    <RowDefinition Height="25" />
    <RowDefinition Height="400" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="48" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="48" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
    <ColumnDefinition Width="40" />
  </Grid.ColumnDefinitions>

  <TextBlock Text="X" Grid.Column="1" />
  <TextBlock Text="Y" Grid.Column="5" />
  <TextBlock Text="Z" Grid.Column="9" />

  <l:Bar x:Name="accelX" Grid.Row="1" Grid.Column="0"
    BarFill="Red" Scale="100" />
  <l:Bar x:Name="accelY" Grid.Row="1" Grid.Column="4"
    BarFill="Red" Scale="100" />
  <l:Bar x:Name="accelZ" Grid.Row="1" Grid.Column="8"
    BarFill="Red" Scale="100" />

  <l:Bar x:Name="compassX" Grid.Row="1" Grid.Column="1"
    BarFill="Yellow" Scale="4" />
  <l:Bar x:Name="compassY" Grid.Row="1" Grid.Column="5"
    BarFill="Yellow" Scale="4" />
  <l:Bar x:Name="compassZ" Grid.Row="1" Grid.Column="9"
    BarFill="Yellow" Scale="4" />

  <l:Bar x:Name="gyroX" Grid.Row="1" Grid.Column="2"
    BarFill="Blue" Scale="32" />
  <l:Bar x:Name="gyroY" Grid.Row="1" Grid.Column="6"
    BarFill="Blue" Scale="32" />
  <l:Bar x:Name="gyroZ" Grid.Row="1" Grid.Column="10"
    BarFill="Blue" Scale="32" />

  <StackPanel Grid.Row="2" Grid.ColumnSpan="11">
    <TextBlock Foreground="Red" Text="Accelerometer (g)" />
    <TextBlock x:Name="heading" Foreground="Yellow" Text="Compass (uT)"/>
```

1 Eleven columns

2 Three bars for each sensor

3 Legend and messages

```
    <TextBlock Foreground="Blue" Text="Gyroscope (rad/sec)" />
    <TextBlock x:Name="messageBlock" Text="Press Start" />
  </StackPanel >
</Grid>
```

You start by dividing the `ContentPanel` into three rows and eleven columns ❶. The first row contains three `TextBlocks` serving as the titles for the x, y, and z coordinates. The second row shows three bars ❷ for each of the `Accelerometer`, `Compass`, and `Gyroscope` sensors. The `Bar` controls are 400 pixels high divided into positive and negative sections of 200 pixels each. Allowing for three columns for each sensor, and two spacer columns, you need a total of eleven columns. The last row ❸ contains a legend and messages.

Each `Bar` control is assigned a `BarFill` color—red for accelerometer values, yellow for compass values, and blue for gyroscope values. Each `Bar` control is also assigned a scale value. We'll describe how the scale factors were calculated in our detailed discussion of each sensor later in the chapter.

The last things you need for your application are application bar buttons to start and stop the sensors:

```
<phone:PhoneApplicationPage.ApplicationBar>
  <shell:ApplicationBar IsVisible="True" IsMenuEnabled="False">
    <shell:ApplicationBarIconButton Text="start" Click="start_Click"
      IconUri="/Images/appbar.transport.play.rest.png" />
    <shell:ApplicationBarIconButton Text="stop" Click="stop_Click"
      IconUri="/Images/appbar.cancel.rest.png" />
  </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>
```

Create a project folder named Images and add the two images to the project. The images can be found in the Icons folder of the Windows Phone 7.1 SDK. Be sure to set the image's build property to `Content`.

8.2.3 Polling sensor data with a timer

In the sample application you update the screen with data from three different sensors. To simplify the logic, you're not going to use the `CurrentValueChanged` events for the sensors, and will use a polling method instead. You'll use a `DispatchTimer` to update the user interface about 15 times a second. Add a `DispatchTimer` field:

```
DispatcherTimer timer;
```

You initialize the timer field inside the `MainPage` constructor. You ask the timer to `Tick` every 66 milliseconds or about 15 times a second. You'll poll each of the sensors inside the `timer_Tick` method:

```
public MainPage()
{
    InitializeComponent();

    timer = new DispatcherTimer();
    timer.Tick += timer_Tick;
```

```
    timer.Interval = TimeSpan.FromMilliseconds(66);
}
```

The timer is started in the `start_Click` method. When the timer is started, you update the message displayed in the `TextBlock` named `messageBlock` to let the user know which sensors have been started:

```
private void start_Click(object sender, EventArgs e)
{
    if (!timer.IsEnabled)
    {
        string runningMessage = "Reading: ";
        timer.Start();
        messageBlock.Text = runningMessage;
    }
}
```

You'll add additional code to the `start_Click` method as you hook up the `Accelerometer`, `Compass`, and `Gyroscope` later in the chapter.

The timer is stopped in the `stop_Click` method. When the timer is stopped, you update the message displayed in the user interface:

```
private void stop_Click(object sender, EventArgs e)
{
    timer.Stop();
    messageBlock.Text = "Press start";
}
```

You'll also update the `stop_Click` method as you hook up the sensors in later sections of the chapter.

You could run the application now to check that the form is laid out as you expect, but the application doesn't do anything interesting yet. You'll remedy that by adding in the accelerometer sensor.

8.3 Measuring acceleration with the accelerometer

The accelerometer can be used in games and applications that use phone movement as an input mechanism or for controlling game play. The Accelerometer tells you when the device is being moved. It can also tell you whether the device is being held flat, at an angle, or straight up and down.

The accelerometer measures the acceleration component of the forces being applied to a device. Note that acceleration due to gravity isn't reported by the sensor. Unless the device in is free fall, forces are always being applied to a device. The `Accelerometer` reports numbers in terms of the constant g, which is defined as the acceleration due to Earth's gravity at sea level. The value of g is -9.8 m/s^2.

When a device is at rest lying on a table, the table is exerting a force on the device that offsets the pull of gravity. The `Accelerometer` measures the acceleration of the force the table applies. When the device is falling, the accelerometer reports zero acceleration.

Now consider when you push the device along the surface of the table. Other forces are now in play such as the force being applied by your hand and the force

due to friction between the device and the table. The `Accelerometer` measures all of these forces.

> **NOTE** In the Windows Phone 7.0 SDK, the Accelerometer class didn't inherit from SensorBase and was updated in the 7.1 SDK to conform to the SensorBase API. To avoid breaking changes between 7.0 and 7.1, Microsoft left the original API intact. The obsolete members include the accelerometer's `ReadingChanged` event and the `AccelerometerReadingEventArgs` class.

When a user shakes a phone, the x, y, and z acceleration values will rapidly change from one extreme to another in a fairly random pattern. By examining the x, y, and z values of the accelerometer, and how they change from one reading to the next, you can determine whether the device is in motion, and how the device is being held. Before we get into the details about exactly what the accelerometer measures and what the reported values mean, you'll hook up the sensor to the bars displayed in the user interface of your application.

8.3.1 Hooking up the sensor

The sample application you built in the previous section is designed to show how the data returned by the sensors changes as the user moves the phone. To see how the accelerometer data changes, you just need to read the `CurrentValue` property from an `Accelerometer` instance and update the three `Bar` controls allocated for the accelerometer's x, y, and z values. Before you can hook up a sensor, you need to declare a member field to reference the `Accelerometer` instance:

```
Accelerometer accelSensor;
```

In the `MainPage` constructor, initialize the field and set the `TimeBetweenUpdates`. You only set the property if the sensor is supported by the phone, so you check the `IsSupported` property. You set the `TimeBetweenUpdates` to match the tick interval of the `DispatchTimer` you use to trigger user interface updates:

```
if(Accelerometer.IsSupported)
{
    accelSensor = new Accelerometer();
    accelSensor.TimeBetweenUpdates = TimeSpan.FromMilliseconds(66);
}
```

Next you start the sensor in the `start_Click` method. Add the following snippet right before the line where the timer's `Start` method is called:

```
if (Accelerometer.IsSupported)
{
    accelSensor.Start();
    runningMessage += "Accelerometer ";
}
```

You're adding the string *Accelerometer* to the message you display in the user interface, informing the user that the accelerometer was started. You only start the sensor if it's supported.

You also must stop the sensor in the `stop_Click` method when the Stop button is clicked:

```
if (Accelerometer.IsSupported)
    accelSensor.Stop();
```

The final step is to read the accelerometer data when the timer ticks and the `timer_Tick` event handler is called. You're going to isolate the code that reads the accelerometer data into a method named `ReadAccelerometerData`. The timer tick method calls the `ReadAccelerometerData` method, which is shown in the next listing.

Listing 8.4 Reading acceleration

```
void timer_Tick(object sender, EventArgs e)
{
    ReadAccelerometerData();
}

private void ReadAccelerometerData()
{
    if (Accelerometer.IsSupported)
    {
        AccelerometerReading reading = accelSensor.CurrentValue;
        Vector3 acceleration = reading.Acceleration;
        accelX.Value = acceleration.X;
        accelY.Value = acceleration.Y;
        accelZ.Value = acceleration.Z;
    }
}
```

Get acceleration vector ❶

❷ Update user interface

First you check whether the `Accelerometer` is supported before getting the current `AccelerometerReading` value from the sensor. Next you obtain the acceleration vector from the reading ❶. The acceleration vector reports acceleration values in the three directions of the phone's coordinate system. Finally, you update the `Bar` controls in the user interface with the x, y, and z properties reported by the acceleration vector ❷.

When you created the `Bar` controls for the accelerometer in MainPage.xaml, you set the `Scale` property to 100. The `Bar` controls are 400 pixels high allowing for positive and negative sections of 200 pixels each. The maximum value of the acceleration vector is ±2. Using this information, you can determine that the scale factor for the bar should be 100, or 200/2.

At this point, you should be able to run the application. If you run the application on a physical device, you should see the bars grow and shrink as you move the device about. Tilt it front to back, tilt it side to side, lay it down flat, hold it upside down. You can mimic all of these movements in the emulator using the accelerometer tool.

8.3.2 Acceleration in the emulator

With the Sensors sample application open in Visual Studio, run the application on the emulator and tap the application bar button labeled Start to begin collecting acceleration data. The emulator's default position is standing in portrait layout and the

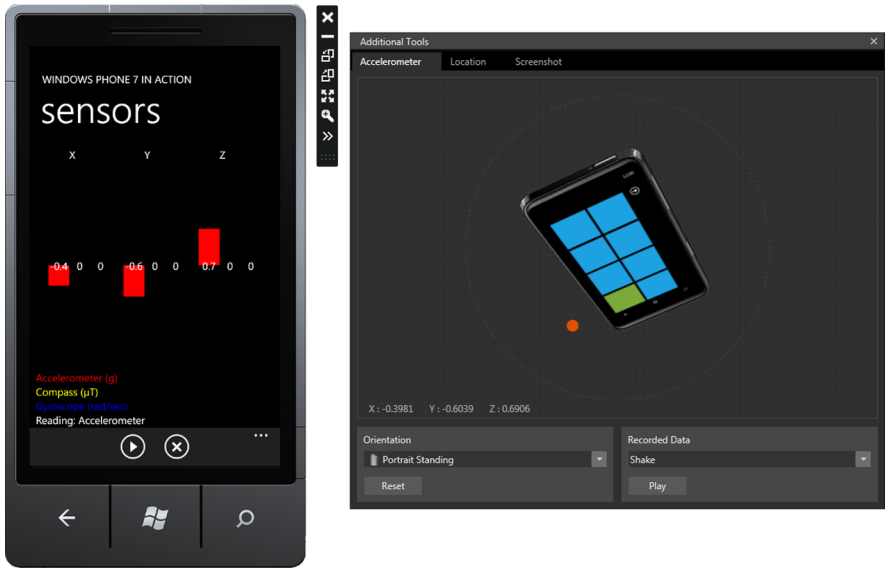


Figure 8.3 Controlling acceleration with the emulator's Accelerometer Tool

accelerometer reports an acceleration of -1 along the y axis. Open the Additional Tools windows using the expander button on the emulator's control bar. The Accelerometer Tool, shown in figure 8.3, is found in the first tab of the Additional Tools window.

The Accelerometer Tool allows you to move the device by dragging the orange dot. The device can also be changed from the Portrait Standing orientation to Portrait Flat, Landscape Standing, and Landscape Flat. The Accelerometer also plays a canned script that mimics shaking the device.

With the Sensors application running in the emulator, you should see the bars grow and shrink as you move the device about with the orange dot. Play the Shake script and watch how the acceleration bars bounce up and down as the data changes. Now that you have a better idea of the numbers reported by the `Accelerometer`, let's take a closer look at exactly what the numbers mean.

8.3.3 Interpreting the numbers

The accelerometer sensor in the phone senses the acceleration due to forces applied to the phone but ignores the acceleration due to gravity. When a device is at rest lying on a table, the table is exerting a force on the device that offsets the pull of gravity. The accelerometer measures the acceleration of the force that the table applies to the device. When the device is falling, the accelerometer reports zero acceleration. Figure 8.4 demonstrates the values reported by the accelerometer when a phone is held in various positions.

If the number reported by the device is related to the force a surface exerts on the device, why is the number negative instead of positive? Remember that the number

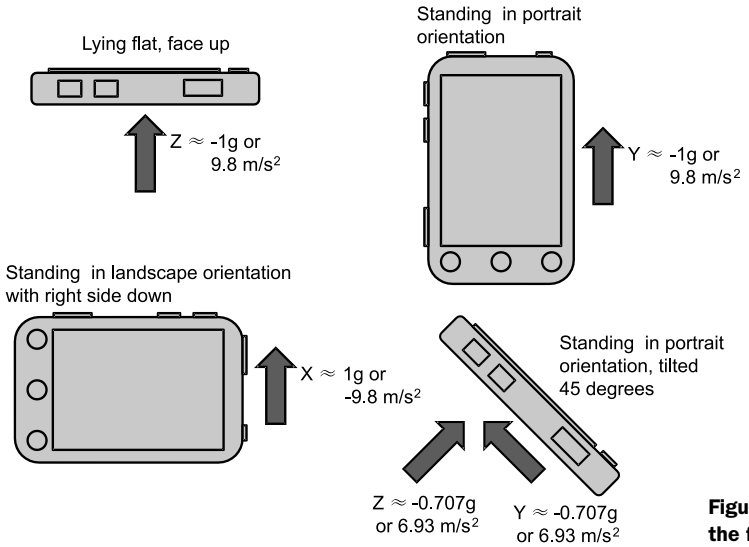


Figure 8.4 Acceleration from the forces on a device at rest

reported is in terms of g or gravity, and g equals -9.8 m/s^2, a negative number. When the accelerometer reports a -1 (or a vector 1 pointing down), it really means a vector with the value 9.8 pointing up. Table 8.2 lists the approximate x, y, z values reported by the `Accelerometer` when the device is at rest in various positions.

Table 8.2 Accelerometer readings with the device at rest

Position	X	Y	Z
In free fall	0g	0g	0g
Flat on back, lying on a surface	0g	0g	-1g or 9.8 m/s^2
Flat on face	0g	0g	1g or -9.8 m/s^2
Standing portrait, bottom is down	0g	-1g or 9.8 m/s^2	0g
Standing portrait, top is down	0g	1g or -9.8 m/s^2	0g
Standing landscape, left is down	-1g or 9.8 m/s^2	0g	0g
Standing landscape, right is down	1g or -9.8 m/s^2	0g	0g

When you move a device, you apply a force along the direction you want the device to move. The force causes the device to accelerate, acceleration changes the velocity of the device, and it starts to move. Let's say your phone is resting flat face up on the surface of a table at point A. Now you give your device a modest push so that it slides along the surface to point B. The initial push is a moderate force in the positive X axis. After you release the device, allowing it to slide, your initial force stops, and the force due to friction begins to slow down the device until it stops moving. Figure 8.5 shows the values reported by the accelerometer in this scenario. The

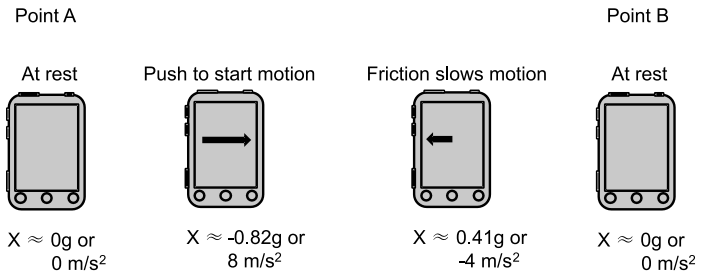


Figure 8.5 Acceleration due to motion

numbers are somewhat contrived, as real numbers will vary based on how hard the initial push is, and the amount of friction between the phone and the surface it's resting upon.

Again, note that the numbers reported by the accelerometer are opposite what you might expect. You push the device in the direction of the positive x axis, but the number reported is a negative value. Remember that the number reported is in terms of g or gravity, and g equals -9.8 m/s^2.

The figure demonstrates the forces involved with pushing a phone across a table. This is probably not something you do often. The same concepts can be applied when the device is moving in a user's hand. When motion begins, the user's hand is applying a force to the device in the direction of motion. When motion ends, the user's hand is applying force in the direction opposite the motion. When the user is moving the device, there may be a period between start and stop when the device is moving at a constant rate, and the acceleration in the direction of motion is zero.

By detecting changes in acceleration values, an application can determine when the device is being moved. The acceleration data can also tell you whether the device is being held flat, at an angle, or straight up and down. What the accelerometer can't tell you is which direction the device is pointed. If you need to know the direction the device is pointed, use the `Compass`.

8.4 Finding direction with the Compass

The `Compass` is useful when an application needs to know which direction a device is pointed relative to the real world. The direction is reported relative to the North Pole. This information is useful for applications such as the astronomy application we discussed earlier, where the application updates the display based on the direction the user is facing. The Compass is also useful in motion-sensitive applications that need to know when a device is rotated.

The `Compass` senses the strength and direction of the Earth's magnetic field. The `Compass` sensor decomposes the magnetic field into x, y, and z vectors, and reports the magnitude of the vectors in microteslas (μT). A tesla is the standard unit of measurement for a magnetic field.

The vertical intensity of the magnetic field (the z value when holding the device flat on a level surface) varies based on the latitude you're in. If you're near the equator, the

vertical intensity will be nearly 0 µT. If you're in the far north (or south) the vertical intensity may be as high as 56 µT. The horizontal intensity varies inversely with latitude. At the equator, the horizontal intensity will be somewhere in the neighborhood of 32 µT. In the far north (or south) the horizontal intensity may be around 4 µT. The absolute intensity, which is the combination of horizontal and vertical intensity, varies as well. Figure 8.6 depicts the horizontal and vertical intensity vectors for a device held parallel to the Earth's surface.

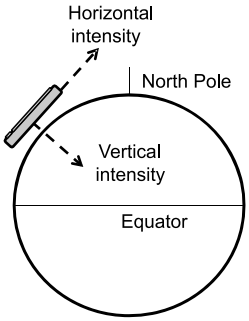


Figure 8.6 The vertical intensity vector points down into the Earth, whereas the horizontal intensity vector points at the magnetic North Pole.

The horizontal intensity vector points to magnetic north. The `Compass` is able to use this information to report heading, or the direction the device is pointing. The `Compass` reports heading compared to magnetic north and a true heading relative to the geographic north. The `Compass` reports heading as the number of degrees the device is turned clockwise from north.

> **NOTE** The Compass API was introduced in the Windows Phone 7.1 SDK. Many of the original 7.0 phones shipped with a compass but didn't ship with appropriate compass driver software. When the Windows Phone 7.5 update shipped, most phones also received updated drivers from the manufacturer to enable compass support. The Compass may not be supported on these phones since some phone models didn't receive new drivers, whereas other phones didn't successfully apply the driver update.

The `Compass` reports information with the `CompassReading` structure. The device direction is read with the `MagneticHeading` and `TrueHeading` properties. The magnetic intensity vectors are available from the `MagnetometerReading` property. Before we look closer at the `CompassReading`, you'll hook up the sensor to the bars displayed in the user interface of your application.

8.4.1 Hooking up the sensor

This section is going to look a whole lot like the section where you hooked up the accelerometer. You need to initialize the sensor in the constructor, start and stop it in the click event handlers, and create a method to read the sensor data. You start by defining a field in the `MainPage` class:

```
Compass compassSensor;
```

Next you initialize the sensor in the `MainPage` constructor. Before constructing the sensor, you first check whether the `Compass` is supported. After constructing the sensor you set the `TimeBetweenUpdates`:

```
if (Compass.IsSupported)
{
    compassSensor = new Compass();
```

```
    compassSensor.TimeBetweenUpdates = TimeSpan.FromMilliseconds(66);
}
```

You start the sensor in the `start_Click` method:

```
if (Compass.IsSupported)
{
    compassSensor.Start();
    runningMessage += "Compass ";
}
```

You stop the sensor in the `stop_Click` method:

```
if (Compass.IsSupported)
    compassSensor.Stop();
```

Finally you create the `ReadCompassData` method in order to update the user interface with the sensor's `CurrentValue`. The following listing contains the implementation of the `ReadCompassData` method. Don't forget to call the new `ReadCompassData` method from the `timer_Tick` event handler.

Listing 8.5 Reading compass data

```
void ReadCompassData()
{
    if (Compass.IsSupported)
    {
        CompassReading reading = compassSensor.CurrentValue;
        Vector3 magnetic = reading.MagnetometerReading;

        compassX.Value = magnetic.X;
        compassY.Value = magnetic.Y;
        compassZ.Value = magnetic.Z;

        heading.Text = string.Format(
            "Compass (µT) : Heading {0} +/- {1} degrees",
            reading.TrueHeading, reading.HeadingAccuracy);
    }
}
```

1 **Get reading**

2 **Update user interface**

3 **Report heading values**

You start by retrieving the current `CompassReading` value ❶ from the sensor and you assign the `MagnetometerReading` to a `Vector3` variable named `magnetic`. Next you update the user interface with the x, y, and z values of the `magnetic` vector ❷. Finally you update the message displayed near the bottom of the screen to show the values of the `TrueHeading` and `HeadingAccuracy` properties ❸. The `HeadingAccuracy` is the amount of potential error, in degrees, of the reported heading.

The `Scale` property of the compass-related bar controls declared in MainPage.xaml has been set to the value 4. The approximate maximum value from the `magnetic` vector is ±50 µT. Since the `Bar` controls are 400 pixels high and are divided into positive and negative halves, you set the `Scale` property to 4 or 200/50.

Now you're ready to run the application. You must run the application on a physical device because the `Compass` isn't supported on the emulator. When running the application, you should see the bars grow and shrink as you move the device about.

Can you figure out where north is by interpreting the numbers reported in the `Bar` controls?

8.4.2 Interpreting the numbers

We mentioned that the Earth's magnetic field can be represented as horizontal and vertical intensities. The `MagnetometerReading` represents the magnetic field as three different vectors, one aligned with each of the device's three axes. When you hold the device flat with the top of the device pointed to the north, the vertical intensity is aligned with the device's z axis, and the horizontal intensity is aligned with the y axis. If you spin the device so that it points to the northeast, the horizontal intensity is split between the x and y axes. Figure 8.7 shows the `MagnetometerReading` values as a device lying flat is pointed north, east, south, and west. The numbers in the figure are approximate for a location on the Earth's surface where the horizontal intensity is near 15 μT.

If you held the device standing vertical instead of flat, then the x and z axes would report the horizontal intensity, whereas the y axis would report the vertical intensity of the magnetic field. A device held at an angle relative to the ground would have a mixture of vertical and horizontal intensities reported along each axis.

No matter the angle the device is held, the heading properties will continue to report how far from north the device is pointed. The `HeadingAccuracy` should always have a small value. When the accuracy value grows, the sensor needs to be calibrated.

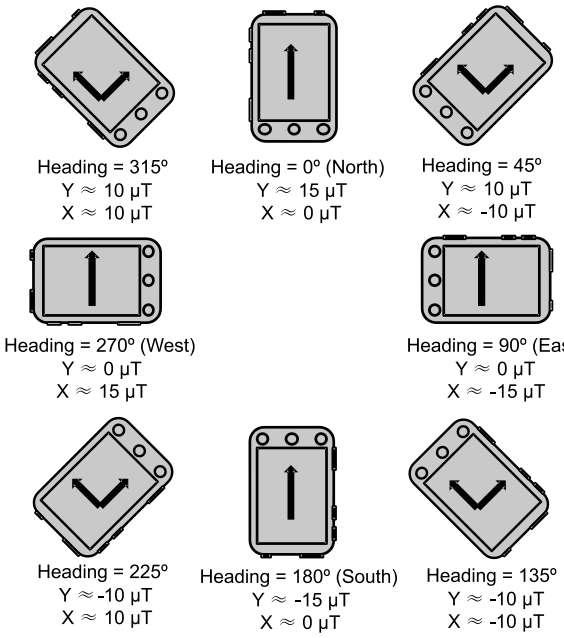


Figure 8.7 Examples of horizontal intensity vectors when lying flat

8.4.3 Calibrating the sensor

The Earth's magnetic field is relatively weak. Items in the local environment, such as strong magnets or large metal objects, will have an effect on the accuracy of the `Compass`. When the `HeadingAccuracy` is off by more than 20 degrees, the phone will raise the `Calibrate` event. When this happens, the phone wants the user to wave the device about in the air for a bit until it gets a grip on reality again.

You can ignore this event and nothing will happen, other than that the device will continue to report a large accuracy value. The event is only raised once for each instance of the `Compass`. If you'd like to be a responsible citizen, you should inform the user to wave the phone about in the air until the heading accuracy is more reasonable. You'll add support for the `Calibrate` event to your sample application.

In the `MainPage` constructor, subscribe to the `Calibrate` event right after the `Compass` is constructed:

```
compassSensor.Calibrate += compassSensor_Calibrate;
```

Now define the `Calibrate` event handler. Your sample application displays a message to the user asking them to wave the phone around in the air until the heading accuracy drops below 20 degrees. Other than displaying a message, the application can't do anything to help perform the calibration:

```
void compassSensor_Calibrate(object sender, CalibrationEventArgs e)
{
    Dispatcher.BeginInvoke(() =>
        MessageBox.Show("The compass sensor needs to be calibrated.
➥ Wave the phone around in the air until the heading accuracy
➥ value is less than 20 degrees")
    );
}
```

The `CalibrationEventArgs` class doesn't define any properties and is just a simple derivation of the base `EventArgs` class.

The `Compass` is useful when an application needs to know which direction the device is pointed relative to the real world. If the device is turned or rotated, an application can determine how much the device was turned by comparing the current heading with a previous heading. The `Compass` isn't very useful if your application needs to be notified while the device is turning, or how quickly the device is turning. The `Gyroscope` is ideal for applications that respond when the device is rotated.

8.5 Pivoting with the Gyroscope

The `Gyroscope` sensor reports how quickly the device is turning on one or more of its axes. The rotational velocity is reported in radians per second, and when a device is rotated in a complete circle it'll rotate 2π radians or 6.28 radians. The values are reported with counterclockwise being the positive direction.

NOTE The gyroscope is optional hardware for Windows Phones and isn't supported on many phones. The `Gyroscope` class's `IsSupported` static property should be checked before using the gyroscope sensor.

The gyroscope only reports turning motion around an axis and if the device is held still, the sensor will report values of zero. If the device is moved from point A to point B without any twisting motion, the gyroscope will also report zero.

The `Gyroscope` reports values with the `GyroscopeReading` struct. Rotational velocities are read from the `GyroscopeReading` through the `RotationRate` property, a `Vector3` that breaks absolute movement into rotation about the x, y, and z axes. You'll now hook up the `Gyroscope` sensor to the user interface in your sample application so you can see the numbers for yourself.

8.5.1 Hooking up the sensor

The sensor APIs are intentionally similar, and hooking up the `Gyroscope` in your sample application is nearly identical to hooking up the `Accelerometer` and `Compass`. You start by declaring a field to reference the `Gyroscope` instance:

```
Gyroscope gyroSensor;
```

You then construct and initialize the field in the `MainPage` constructor:

```
if (Gyroscope.IsSupported)
{
    gyroSensor = new Gyroscope();
    gyroSensor.TimeBetweenUpdates = TimeSpan.FromMilliseconds(66);
}
```

The sensor is started in the `start_Click` method:

```
if (Gyroscope.IsSupported)
{
    gyroSensor.Start();
    runningMessage += "Gyroscope ";
}
```

The sensor is stopped in the sample application's `stop_Click` method:

```
if (Gyroscope.IsSupported)
    gyroSensor.Stop();
```

As with the accelerometer and the compass sensors, you create a new method to read the `Gyroscope`'s `CurrentValue` and update the user interface. The new method is named `ReadGyroscopeData`, and is called from the `timer_Tick` method. The code for `ReadGyroscopeData` is shown in the following listing.

Listing 8.6 Reading gyroscope data

```
void ReadGyroscopeData()
{
    if (Gyroscope.IsSupported)
    {
```

```
            GyroscopeReading reading = gyroSensor.CurrentValue;
            Vector3 rotation = reading.RotationRate;

            gyroX.Value = rotation.X;
            gyroY.Value = rotation.Y;
            gyroZ.Value = rotation.Z;
        }
    }
```

❶ Get reading

❷ Update user interface

You start by retrieving the current `GyroscopeReading` value ❶ from the sensor and assign the `RotationRate` to a `Vector3` variable named `rotation`. Next you update the user interface with the x, y, and z values of the `rotation` vector ❷.

When you created the `Bar` controls for the gyroscope in MainPage.xaml, you set the `Scale` property to 32. The positive and negative bars are each 200 pixels each. You assume the maximum rotation rate is a full spin once per second or $\pm 2\pi$ radians per second. With 2π equal to approximately 6.25, you calculated the scale of the `Bar` control at 32 or 200/6.25.

What can you do to see the gyroscope bars move in the application? Let's get dizzy. Do you have a spinning office chair? If so, you can hold the device flat in your hand and spin back and forth in your chair. You should see the Z-bar move up and down as you spin. Another example is to hold the device in your hand so that it's standing up in portrait mode. Now tilt the phone back until it's lying flat in your hand. You should see the X-bar move down and report a negative value. Tilt the phone back up, and the bar should move up and report a positive value.

We're now finished with the Sensors sample application. You've seen how each of the hardware sensors is exposed by classes and structures in the Sensors API. The sensors each return individual sets of data that can be used in various ways to build interesting applications. Each of the sensors tell you different bits of information about how the device is held, how the device is moving, and which direction the device in pointed in. Correlating this information across sensors can be tricky, and involves a solid understanding of physics, mathematics, and three-dimensional coordinate spaces. The Windows Phone SDK provides the `Motion` class to perform these calculations for you.

8.6 Wrapping up with the motion sensor

Unlike the other sensors we've covered so far in this chapter, the motion sensor isn't a hardware-based sensor. The motion sensor, represented in the Windows Phone SDK as the `Motion` class, is a wrapper around the `Accelerometer`, `Compass`, and `Gyroscope`. Instead of sensing data from hardware, the motion sensor consumes data from the other sensors and performs some convenient number crunching.

> **NOTE** The `Motion` class is supported if a phone has an accelerometer and compass. The motion sensor is supported even if a phone doesn't have gyroscope hardware installed. When the gyroscope isn't installed, the data provided by the Motion class may not be as accurate as the data provided when the gyroscope is present.

The `Motion` class analyzes the data provided by the `Accelerometer`, `Compass`, and `Gyroscope`. The `Motion` class reports the results of its data analysis in the `MotionReading` class. The `Motion` class separates motion-based acceleration from gravity. Motion-based acceleration is reported in `MotionReading`'s `DeviceAcceleration` property, whereas the `Gravity` property reports acceleration due to gravity. The `DeviceRotationRate` property reports rotational velocities obtained from the gyroscope.

The `Motion` class also provides tools for mapping device coordinates into real-world coordinates with the `AttitudeReading` class. An instance of the `AttitudeReading` class is returned from the `Attitude` property of the `MotionReading` class. The `AttitudeReading` class reports `Yaw`, `Pitch`, and `Roll` values. The `AttitudeReading` class also provides both a rotation `Matrix` and a `Quaternion` that can be used for coordinate mapping. We'll show you how to use the `AttitudeReading` to map coordinates in a new `MotionSensor` sample application.

Next you'll create a new sample application to demonstrate how the motion sensor works and how to use the numbers provided by the `MotionReading` class.

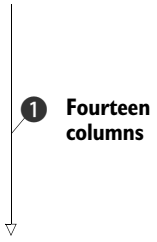
8.6.1 Building a motion enabled sample application

You're going to create a new sample application to show off the motion sensor. This new sample application is similar to the Sensors sample application you just finished. Create a new Windows Phone Application named MotionSensor, and add references to the Microsoft.Devices.Senors.dll and Microsoft.Xna.Framework.dll assemblies. The MotionSensor application will also use the `Bar` control you created in the Sensors application. Copy the Bar.xaml and Bar.xaml.cs files into the MotionSensor project and change the `Bar` class's namespace to `MotionSensor`.

MainPage.xaml is assembled similarly to the MainPage.xaml.cs used in the Sensors application. It's similar enough that you may wish to start by copying the markup for the Sensors application's `ContentPanel` and modifying it. The next listing shows the markup for the MotionSensor application's `ContentPanel`.

Listing 8.7 MainPage markup for the motion sensor sample application

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
  <Grid.RowDefinitions>
    <RowDefinition Height="25" />
    <RowDefinition Height="400" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="48" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
```



```
    <ColumnDefinition Width="48" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
    <ColumnDefinition Width="30" />
  </Grid.ColumnDefinitions>

  <TextBlock Text="X" Grid.Column="1" />
  <TextBlock Text="Y" Grid.Column="6" />
  <TextBlock Text="Z" Grid.Column="11" />

  <l:Bar x:Name="accelX" Grid.Row="1" Grid.Column="0"
    BarFill="Red" Scale="67" />
  <l:Bar x:Name="accelY" Grid.Row="1" Grid.Column="5"
    BarFill="Red" Scale="67" />
  <l:Bar x:Name="accelZ" Grid.Row="1" Grid.Column="10"
    BarFill="Red" Scale="67" />

  <l:Bar x:Name="gravityX" Grid.Row="1" Grid.Column="1"
    BarFill="Yellow" Scale="200" />
  <l:Bar x:Name="gravityY" Grid.Row="1" Grid.Column="6"
    BarFill="Yellow" Scale="200" />
  <l:Bar x:Name="gravityZ" Grid.Row="1" Grid.Column="11"
    BarFill="Yellow" Scale="200" />

  <l:Bar x:Name="gyroX" Grid.Row="1" Grid.Column="2"
    BarFill="Blue" Scale="32" />
  <l:Bar x:Name="gyroY" Grid.Row="1" Grid.Column="7"
    BarFill="Blue" Scale="32" />
  <l:Bar x:Name="gyroZ" Grid.Row="1" Grid.Column="12"
    BarFill="Blue" Scale="32" />

  <l:Bar x:Name="attitudeX" Grid.Row="1" Grid.Column="3"
    BarFill="Violet" Scale="64" />
  <l:Bar x:Name="attitudeY" Grid.Row="1" Grid.Column="8"
    BarFill="Violet" Scale="128" />
  <l:Bar x:Name="attitudeZ" Grid.Row="1" Grid.Column="13"
    BarFill="Violet" Scale="32" />

  <StackPanel Grid.Row="2" Grid.ColumnSpan="14">
    <TextBlock Foreground="Red" Text="Acceleration (g)" />
    <TextBlock Foreground="Yellow" Text="Gravity (g)" />
    <TextBlock Foreground="Blue" Text="Rotation (rad/sec)" />
    <TextBlock x:Name="point" Foreground="Violet" Text="Attitude" />
  </StackPanel >
</Grid>
```

❶ **Fourteen columns**

❷ **Three bars for each data point**

Legend and messages ❸

You start by dividing the `ContentPanel` into three rows and fourteen columns ❶. The first row contains three `TextBlocks` serving as the titles for the x, y, and z coordinates. The second row shows three bars ❷ for each of the linear acceleration, gravity, rotation, and attitude readings. Allowing for three columns for each reading, and two spacer columns, you need a total of fourteen columns, as displayed in figure 8.8. The last row ❸ contains a legend and messages.

When setting the scale factors for the acceleration, gravity, and rotation bars, you assume the maximum values returned by the sensor are 3, 1, and 2π respectively. The

scale values for the attitude bars reflect that the maximum values for `Pitch`, `Roll`, and `Yaw` are π, π/2 and 2π respectively. We'll examine each of the readings values in more detail in section 8.6.3.

The MotionSensor sample application also declares Start and Stop buttons on the application bar. Copy the application bar markup from the Sensors application. Don't forget to create the Images folder and copy the SDK icons.

The application isn't quite ready to run and doesn't compile yet. The application is missing the `Click` event handlers for the two application bar buttons. You'll implement the two missing methods after you hook up the motion sensor.

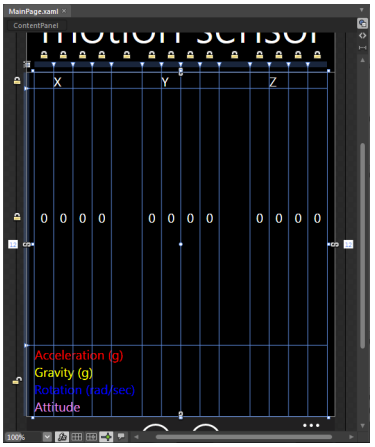


Figure 8.8 A screen shot of the ContentPanel in MainPage.xaml as displayed by the designer view in Microsoft Expression Blend

8.6.2 Hooking up the sensor

In the Sensors application you polled all three sensors using a `DispatchTimer` and the `CurrentValue` property of each sensor. In this sample application you're going to use the `CurrentValueChanged` event instead. Start by creating a class field for the motion sensor. Don't forget to add using statements for `Microsoft.Devices.Sensors`:

```
Motion sensor;
```

The sensor variable is constructed and initialized in the `MainPage` constructor. In addition to constructing the sensor, you set the `TimeBetweenUpdates` property and subscribe to the `CurrentValueChanged` and `Calibrate` methods. The next listing shows the implementation of the `MainPage` constructor.

Listing 8.8 Initializing the motion sensor

```
public MainPage()
{
    InitializeComponent();

    if (Motion.IsSupported)
    {
        sensor = new Motion();
        sensor.TimeBetweenUpdates = TimeSpan.FromMilliseconds(66);
        sensor.CurrentValueChanged += sensor_CurrentValueChanged;
        sensor.Calibrate += sensor_Calibrate;
    }
}
```

❶ Construct motion sensor

❷ Subscribe to value changed event

Before attempting to use the `Motion` class, you check the `IsSupported` static property to see whether the motion sensor is supported by the device running the application. Once you're sure that the motion sensor is supported you construct a new instance of the `Motion` class ❶. You then set the `TimeBetweenUpdates` property to 66 milliseconds so

that you receive updates approximately 17 times each second. Next you subscribe to the `CurrentValueChanged` event ❷, registering the method `sensor_CurrentValueChanged` as the event handler. Finally, you subscribe to the `Calibrate` event.

Because the `Motion` class wraps the compass sensor, you need to be aware of when the compass needs calibration. Copy the `Calibrate` event handler implementation from the Sensors program into a method named `sensor_Calibrate`.

Let's examine the `CurrentValueChanged` event handler. The `Motion` class raises the `CurrentValueChanged` event on a background thread. If the user interface is updated from the event handler, the update code must be executed on the UI thread. The next listing shows the implementation of the `sensor_CurrentValueChanged` method.

Listing 8.9 Handling the `CurrentValueChanged` event

```
void sensor_CurrentValueChanged(object sender,
    SensorReadingEventArgs<MotionReading> e)
{
    MotionReading reading = e.SensorReading;
    Dispatcher.BeginInvoke(() =>
    {
        Vector3 acceleration = reading.DeviceAcceleration;
        accelX.Value = acceleration.X;
        accelY.Value = acceleration.Y;
        accelZ.Value = acceleration.Z;

        Vector3 gravity = reading.Gravity;
        gravityX.Value = gravity.X;
        gravityY.Value = gravity.Y;
        gravityZ.Value = gravity.Z;

        Vector3 rotation = reading.DeviceRotationRate;
        gyroX.Value = rotation.X;
        gyroY.Value = rotation.Y;
        gyroZ.Value = rotation.Z;

        AttitudeReading attitude = reading.Attitude;
        attitudeX.Value = attitude.Pitch;
        attitudeY.Value = attitude.Roll;
        attitudeZ.Value = attitude.Yaw;

        Vector3 worldSpacePoint = new Vector3(0.0f, 10.0f, 0.0f);
        Vector3 bodySpacePoint =
            Vector3.Transform(worldSpacePoint, attitude.RotationMatrix);
        point.Text = string.Format("Attitude: Transform of (0.0, 10.0, 0.0)
➥ = ({0:F1}, {1:F1}, {2:F1})",
            bodySpacePoint.X, bodySpacePoint.Y, bodySpacePoint.Z);
    });
}
```

❶ Work on UI thread

❷ Update bar controls

❸ Transform coordinate point

The `CurrentValueChanged` event is raised on a background thread. In your event handler implementation, you update the user interface and use the `Dispatcher` object to run your code on the user interface thread ❶. The next section of the listing reads various properties of the provided `MotionReading` instance and updates the twelve

`Bar` controls ❷. Finally, you use the `AttitudeReading`'s `RotationMatrix` to convert a real-world coordinate to the coordinate system of the phone ❸.

You still need to create the `Click` event handlers for the Start and Stop application bar buttons. The `start_Click` method is simple. The method checks whether the `Motion` class is supported, and then calls the `Start` method:

```
private void start_Click(object sender, EventArgs e)
{
    if (Motion.IsSupported)
        sensor.Start();
}
```

The `stop_Click` method is just as simple:

```
private void stop_Click(object sender, EventArgs e)
{
    if (Motion.IsSupported)
        sensor.Stop();
}
```

With the two click event handlers implemented, the code for the sample application is complete. Deploy the application to your motion-sensor-enabled device and run the application. Start the sensor and examine the acceleration, gravity, rotation, and attitude values as you move your phone around. Don't forget to look at the attitude message line and how the coordinate point is transformed. Let's take a closer look at the readings reported by the `Motion` class and how to interpret the numbers.

8.6.3 Interpreting the numbers

We talked about acceleration earlier in the chapter, and how the data reported by the `Accelerometer` mixes both the acceleration due to gravity and the acceleration due to motion. There are many scenarios where it's important to separate the two types of acceleration. Consider an application or game that works by tilting the phone, such as one that moves a ball through a maze. This type of application is only interested in the acceleration due to gravity. Without the convenient `Gravity` property of the `MotionReading`, the maze application would have to somehow account for the acceleration due to motion. We demonstrate how to use the `Gravity` vector to control a simple game in chapter 15.

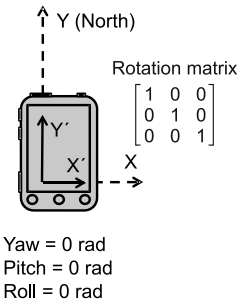


Figure 8.9 The device frame aligned with the world frame when the device is lying flat and pointed North

The other convenience provided by the `Motion` class is the calculation of the `Attitude` data. To understand the `Attitude` you need to understand the frame of reference, or coordinate system for both the real world and the device. The `Motion` class assumes a real-world coordinate system where y points due north, z points straight up, and x points due east. When the device is lying flat, face up, with the top of the device pointing north, the device's frame of reference matches the real world frame of reference. This is shown in figure 8.9.

The `Yaw`, `Pitch`, and `Roll` readings are all approximately zero, and the rotation matrix is the identity matrix. An object at point (0, 10, 0) in the world frame will have the same coordinates in the device frame. The device's y axis, labeled Y' in the figure, is pointing north and the x axis, labeled X', is pointing east.

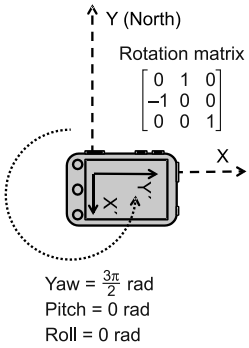


Figure 8.10 The device rotated 3/2 π radians or 270 degrees around the z axis

When the device is rotated, its frame of reference no longer matches the real world frame of reference. If the top of the device lying flat is rotated to point east, the device is considered to be rotated 270 degrees. The attitude reading will have a `Yaw` reading of 3/2 π radians, or 270 degrees. The `Yaw`, rotation about the z axis, is read as the counterclockwise angle between the two y axes. This is shown in figure 8.10.

Now the device's y axis is pointing east and the x axis is pointing south. Again, consider an object at the coordinate (0, 10, 0) in the world frame. This same object will have the coordinates (-10, 0, 0) in the device frame.

With the top of the device still pointed east, raise the top of the device until it's in the standing portrait orientation with the back of the device facing east. This is shown in figure 8.11. In this case you've rotated the device frame about the x axis and changed the `Pitch` of the device. The attitude reading will still have a `Yaw` reading of 3/2 π radians, but will now also have a `Pitch` reading of 1/2 π radians, or 90 degrees. The `Pitch`, or rotation about the x axis, is read as a counterclockwise angle.

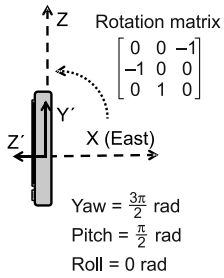


Figure 8.11 The device rotated 270 degrees around the z axis and 90 degrees around the x axis

Now the device's y axis is pointing up toward the sky, aligned with the world frame's z axis. The device's z axis is pointing to the west. The device's x axis is still pointing south. Again, consider an object at the coordinate (0, 10, 0) in the world frame. This same object will still have the coordinates (-10, 0, 0) in the device frame because changing the pitch didn't change the direction of the device's x axis.

When working with the `AttitudeReading`, you must remember that the `Yaw`, `Pitch`, and `Roll` values are order-dependent. To translate a point in one frame of reference to a point in another frame of reference, you must apply `Yaw` first, followed by `Pitch`, then by `Roll`.

Though we've referred to the `Motion` class as a motion sensor, it's really more of a service than a sensor. The `Motion` class makes use of a few different sources of data to provide a convenient service for detecting motion and position.

8.7 Summary

In this chapter we've covered three different hardware sensors, and one class that wraps the other sensors. The `Accelerometer` reports acceleration due to the forces acting on a device. The `Compass` reports the strength of local magnetic fields as well as the heading of the device relative to north. The `Gyroscope` reports the rotational velocity of the device. There aren't any sensors that report linear velocity or rotational acceleration. There is also not a sensor that reports exactly how far a phone has moved.

The `Motion` sensor uses data from the `Accelerometer`, `Compass`, and `Gyroscope` to perform a few complex calculations to separate acceleration into gravity and motion components. The motion sensor also provides information necessary to convert device coordinates into real-world coordinates.

Application developers should consider mixing one or more sensors with the location service to build applications that mesh real world with the digital world. Novel augmented reality applications can be built to show the user the location of nearby landmarks or the position of constellations in the night sky.

In the next chapter, we'll explore the networking features of the Windows Phone SDK. You'll learn how to determine network connection state and how to connect to web services. You'll also learn how to send notifications to a phone from a web service.

MOBILE TECHNOLOGY/WINDOWS

Windows Phone 7 IN ACTION

Binkley-Jones • Perga • Sync

Windows Phone 7 is a powerful mobile platform sporting the same Metro interface as Windows 8. It offers a rich environment for apps, browsing, and media. Developers code the OS and hardware using familiar .NET tools like C# and XAML. And the new Windows Store offers an app marketplace reaching millions of users.

Windows Phone 7 in Action is a hands-on guide to programming the WP7 platform. It zips through standard phone, text, and email controls and dives head-first into how to build great mobile apps. You'll master the hardware APIs, access web services, and learn to build location and push applications. Along the way, you'll see how to create the stunning visual effects that can separate your apps from the pack.

What's Inside

- Full introduction to WP7 and Metro
- HTML5 hooks for media, animation, and more
- XNA for stunning 3D graphics
- Selling apps in the Windows Store

Written for developers familiar with .NET and Visual Studio. No WP7 or mobile experience is required.

Timothy Binkley-Jones is a software engineer with extensive experience developing commercial IT, web, and mobile applications. **Massimo Perga** is a software engineer at Microsoft and **Michael Sync** is a solution architect for Silverlight and WP7.

To download their free eBook in PDF, ePub and Kindle formats, owners of this book should visit manning.com/WindowsPhone7inAction

Free eBook
SEE INSERT

"Definitely recommended!"
—Vipul Patel, Amazon.com

"Top resource for Windows Phone developers."
—Loïc Simon, Solent SAS

"A great handbook for climbing the WP7 ladder."
—Francesco Goggi
Magneti Marelli

"Gives you a kickstart in Windows Phone development."
—Mark Monster
Monster Consultancy

ISBN 13: 978-1-617290-09-1
ISBN 10: 1-617290-09-2
53999
9 781617 290091

MANNING $39.99 / Can $41.99 [INCLUDING eBOOK]