Bonus Chapter 5

More Windows Apps Features

WHAT'S IN THIS CHAPTER?

- Using the camera
- Accessing geolocation information
- Using the MapControl
- Using sensors

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The Wrox.com code downloads for this chapter are found at `http://www.wrox.com` on the Download Code tab. The source code is also available at `https://github.com/ProfessionalCSharp/ProfessionalCSharp7` in the directory `MoreWindows`.

The code for this chapter is divided into these major samples:

- Camera Sample
- Map Sample
- Sensor Sample
- Rolling Marble

OVERVIEW

This chapter gives you information about programming Windows apps using built-in hardware from your devices and special controls.

You access the camera of your Windows 10 device with the `CameraCaptureUI` control. This control can be used to not only take photos but also to record videos.

To access location information, the GPS sensor (if it is available) is used from `GeoLocator`. If the GPS sensor is not available, location information is accessed with the help of the Wi-Fi network or the IP address. The geocoordinates you receive from the location information can be used with `MapControl`. This control that is part of the Windows Runtime offers several views. You can display streets or an aerial view. With some locations, a street-level experience is available, which enables users to "walk through" the streets using nice 3D images.

This bonus chapter concludes with information about accessing sensors that are available with many Windows 10 devices. The `LightSensor` returns information about available light. The `Compass` uses a magnetometer to measure the magnetic and the geographic north. With the `Accelerometer` you can measure g-forces. These are just a few samples of the many sensors demonstrated in this chapter.

With the rolling marble sample app, the `Accelerometer` will be used to move an `Ellipse` across the screen. The Microsoft Surface devices include many of the sensors used. Notebooks and tablets from other vendors offer them as well. You need to try this out with your devices.

CAMERA

As apps are becoming more and more visual, and more devices offer one or two cameras built-in, using the camera is becoming a more and more important aspect of apps—and it is easy to do with the Windows Runtime.

> **NOTE** *Using the camera requires that you configure the* Webcam *capability in the Manifest Editor. For recording videos, you need to configure the* Microphone *capability as well.*

Photos and videos can be captured with the `CameraCaptureUI` class (in the namespace `Windows.Media.Capture`). First, you need to configure the photo and video settings to use the `CaptureFileAsync` method. The first code snippet captures a photo. After instantiating the `CameraCaptureUI` class, `PhotoSettings` are applied. Possible photo formats are JPG, JPGXR, and PNG. It is also possible to define cropping where the UI for the camera capture directly asks the user to select a clipping from the complete picture based on the cropping size. For cropping, you can define either a pixel size with the property `CroppedSizeInPixels` or just a ratio with `CroppedAspectRatio`. After the user takes the photo, the sample code uses the returned `StorageFile` from the method `CaptureFileAsync` to store it as a file inside a user-selected folder with the help of the `FolderPicker` (code file `CameraSample/MainPage.xaml.cs`)

```
private async void OnTakePhoto(object sender, RoutedEventArgs e)
{
  var cam = new CameraCaptureUI();
  cam.PhotoSettings.AllowCropping = true;
  cam.PhotoSettings.Format = CameraCaptureUIPhotoFormat.Png;
  cam.PhotoSettings.CroppedSizeInPixels = new Size(300, 300);

  StorageFile file = await cam.CaptureFileAsync(CameraCaptureUIMode.Photo);
  if (file != null)
  {
    var picker = new FileSavePicker();
    picker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;
    picker.FileTypeChoices.Add("Image File", new string[] { ".png" });
    StorageFile fileDestination = await picker.PickSaveFileAsync();
    if (fileDestination != null)
    {
      await file.CopyAndReplaceAsync(fileDestination);
    }
  }
}
```

The second code snippet is used to record a video. As before, you first need to take care of the configuration. Besides the `PhotoSettings` property, the `CameraCaptureUI` type defines the `VideoSettings` property. You can restrict the video recording based on the maximum resolution (using the enumeration value `CameraCaptureUIMaxVideoResolution.HighestAvailable` allows the user to select any available resolution) and the maximum duration. Possible video formats are WMV and MP4 (code file `CameraSample/MainPage.xaml.cs`)

```
private async void OnRecordVideo(object sender, RoutedEventArgs e)
{
  var cam = new CameraCaptureUI();
  cam.VideoSettings.AllowTrimming = true;
  cam.VideoSettings.MaxResolution =
    CameraCaptureUIMaxVideoResolution.StandardDefinition;
  cam.VideoSettings.Format = CameraCaptureUIVideoFormat.Wmv;
  cam.VideoSettings.MaxDurationInSeconds = 5;

  StorageFile file = await cam.CaptureFileAsync(
    CameraCaptureUIMode.Video);
  if (file != null)
  {
    var picker = new FileSavePicker();
    picker.SuggestedStartLocation = PickerLocationId.VideosLibrary;
    picker.FileTypeChoices.Add("Video File", new string[] { ".wmv" });
    StorageFile fileDestination = await picker.PickSaveFileAsync();
    if (fileDestination != null)
    {
      await file.CopyAndReplaceAsync(fileDestination);
    }
  }
}
```

In cases where the user should be offered the option to capture either a video or a photo, you can pass the parameter `CameraCaptureUIMode.PhotoOrVideo` to the method `CaptureFileAsync`.

Because the camera also records location information, when the user runs the app for the first time, he or she is asked if recording location information should be allowed (see Figure BC5-1).

Running the application, you can record photos and videos.

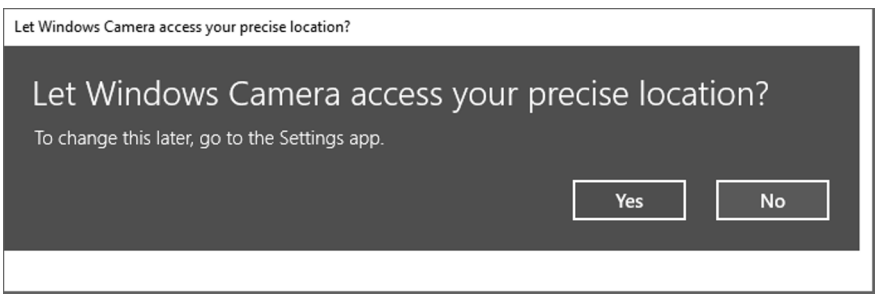

FIGURE BC5-1

GEOLOCATION AND MAPCONTROL

Knowing the location of the user is an important aspect of apps, whether it's an app to show a map, an app that shows the weather of the area of the user, or an app for which you need to decide in what nearest cloud center the data of the user should be saved. When ads are used in the app, the user location can be important to show ads from the near area (if available).

With Windows apps you can also show maps. With Windows 10, a `MapControl` is available as part of the Windows API, and you don't need to use additional libraries for doing this.

The sample app uses both the `Geolocator` (namespace `Windows.Devices.Geolocation`), to give information about the address of the user, and the `MapControl` (namespace `Windows.UI.Xaml.Controls.Maps`). Of course, you can also use these types independent of each other in your apps.

Using the MapControl

With the sample app, a `MapControl` is defined in the `MainPage` where different properties and events bind to values from the `MapsViewModel` that is accessed via the `ViewModel` property of the page. This way, you can dynamically change some settings in the app and see different features available with the `MapControl` (code file `MapSample/MainPage.xaml`):

```
<maps:MapControl x:Name="map"
  Center="{x:Bind ViewModel.CurrentPosition, Mode=OneWay}"
  MapTapped="{x:Bind ViewModel.OnMapTapped, Mode=OneTime}"
  Style="{x:Bind ViewModel.CurrentMapStyle, Mode=OneWay}"
  ZoomLevel="{x:Bind Path=ViewModel.ZoomLevel, Mode=OneWay}"
  DesiredPitch="{x:Bind Path=ViewModel.DesiredPitch, Mode=OneWay}"
  TrafficFlowVisible="{x:Bind checkTrafficFlow.IsChecked, Mode=OneWay,
  Converter={StaticResource nbtob}}"
  BusinessLandmarksVisible="{x:Bind checkBusinessLandmarks.IsChecked,
  Mode=OneWay, Converter={StaticResource nbtob}}"
  LandmarksVisible="{x:Bind checkLandmarks.IsChecked, Mode=OneWay,
  Converter={StaticResource nbtob}}"
  PedestrianFeaturesVisible="{x:Bind checkPedestrianFeatures.IsChecked,
  Mode=OneWay, Converter={StaticResource nbtob}}" />
```

The sample app defines controls to configure the `MapControl` within the Pane of the `SplitView` that is positioned on the right side. The `MapControl` is defined within the content of the `SplitView`.

With the code-behind file, the `ViewModel` property is defined, and a `MapsViewModel` is instantiated by passing the `MapControl` to the constructor. Usually it's best to avoid having Windows controls directly accessible to the view model, and you should only use data binding to map. However, when you use some special features, such as street-side experience, it's easier to directly use the `MapControl` in the `MapsViewModel` class. Because this view model type is not doing anything else and cannot be used on anything other than Windows devices anyway, it's a compromise for passing the `MapControl` to the constructor of the `MapsViewModel` (code file `MapSample/MainPage.xaml.cs`):

```
public sealed partial class MainPage: Page
{
  public MainPage()
  {
    this.InitializeComponent();
    ViewModel = new MapsViewModel(map);
  }
  public MapsViewModel ViewModel { get; }
}
```

The constructor of the `MapsViewModel` initializes some properties that are bound to properties of the `MapControl`, such as the position of the map to a location within Vienna, the map style to a road variant, the pitch level to 0, and the zoom level to 12 (code file `MapSample/ViewModels/MapsViewModel.cs`):

```
public class MapsViewModel: BindableBase
{
  private readonly CoreDispatcher _dispatcher;
  private readonly Geolocator _locator = new Geolocator();
  private readonly MapControl _mapControl;

  public MapsViewModel(MapControl mapControl)
  {
    _mapControl = mapControl;
    StopStreetViewCommand = new DelegateCommand(
    StopStreetView, () => IsStreetView);
    StartStreetViewCommand = new DelegateCommand(
    StartStreetViewAsync, () => !IsStreetView);

    if (!DesignMode.DesignModeEnabled)
    {
      _dispatcher = CoreWindow.GetForCurrentThread().Dispatcher;
    }

    _locator.StatusChanged += async (s, e) =>
    {
      await _dispatcher.RunAsync(CoreDispatcherPriority.Low, () =>
        PositionStatus = e.Status);
    };

    // intialize defaults at startup
    CurrentPosition = new Geopoint(
      new BasicGeoposition { Latitude = 48.2, Longitude = 16.3 });
    CurrentMapStyle = MapStyle.Road;
    DesiredPitch = 0;
    ZoomLevel = 12;
  }
```

Upon starting the app with the initial configuration, you can see the maps loaded with a location in Vienna as defined by the `BasicGeoposition`, the controls on the right side for managing the `MapControl`, and textual information about the loading status of the map (see Figure BC5-2).

When you zoom in, change the pitch level, and select landmarks and business landmarks to be visible, you can see famous buildings such as the Stephansdom in Vienna, as shown in Figure BC5-3.

When you switch to the Aerial view, you can see real images, as shown in Figure BC5-4.

Some locations also show nice images with the Aerial3D view, as shown in Figure BC5-5.

Location Information with Geolocator

Next, you need to get the actual position of the user with the help of the `Geolocator` instance `_locator`. The method `GetPositionAsync` returns the geolocation by returning a `Geoposition` instance. The result is applied to the `CurrentPosition` property of the view model that is bound to the center of the `MapControl` (code file `MapSample/ViewModels/MapsViewModel.cs`):

```
public async void GetCurrentPositionAsync()
{
  try
  {
```

```
      Geoposition position = await _locator.GetGeopositionAsync(
        TimeSpan.FromMinutes(5), TimeSpan.FromSeconds(5));
      CurrentPosition = new Geopoint(new BasicGeoposition
      {
        Longitude = position.Coordinate.Point.Position.Longitude,
        Latitude = position.Coordinate.Point.Position.Latitude
      });
    }
    catch (UnauthorizedAccessException ex)
    {
      await new MessageDialog(ex.Message).ShowAsync();
    }
  }
```

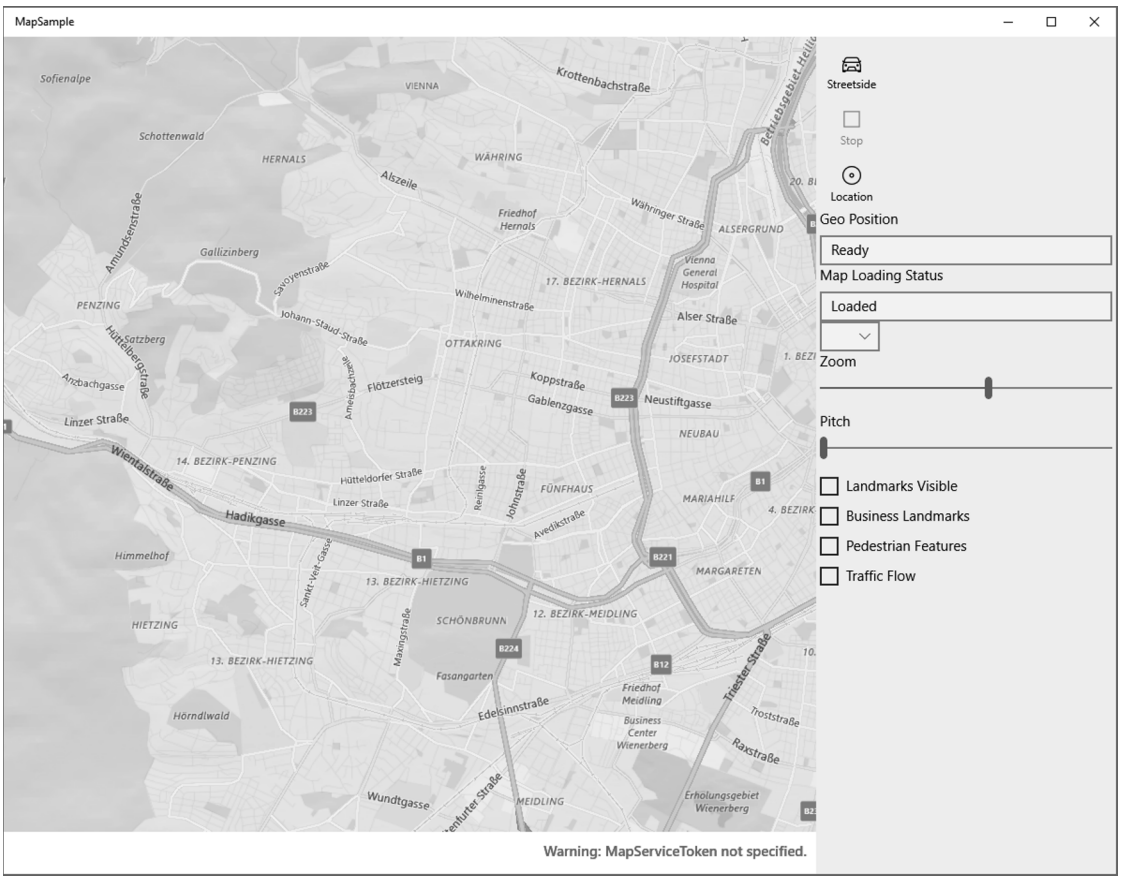

FIGURE BC5-2

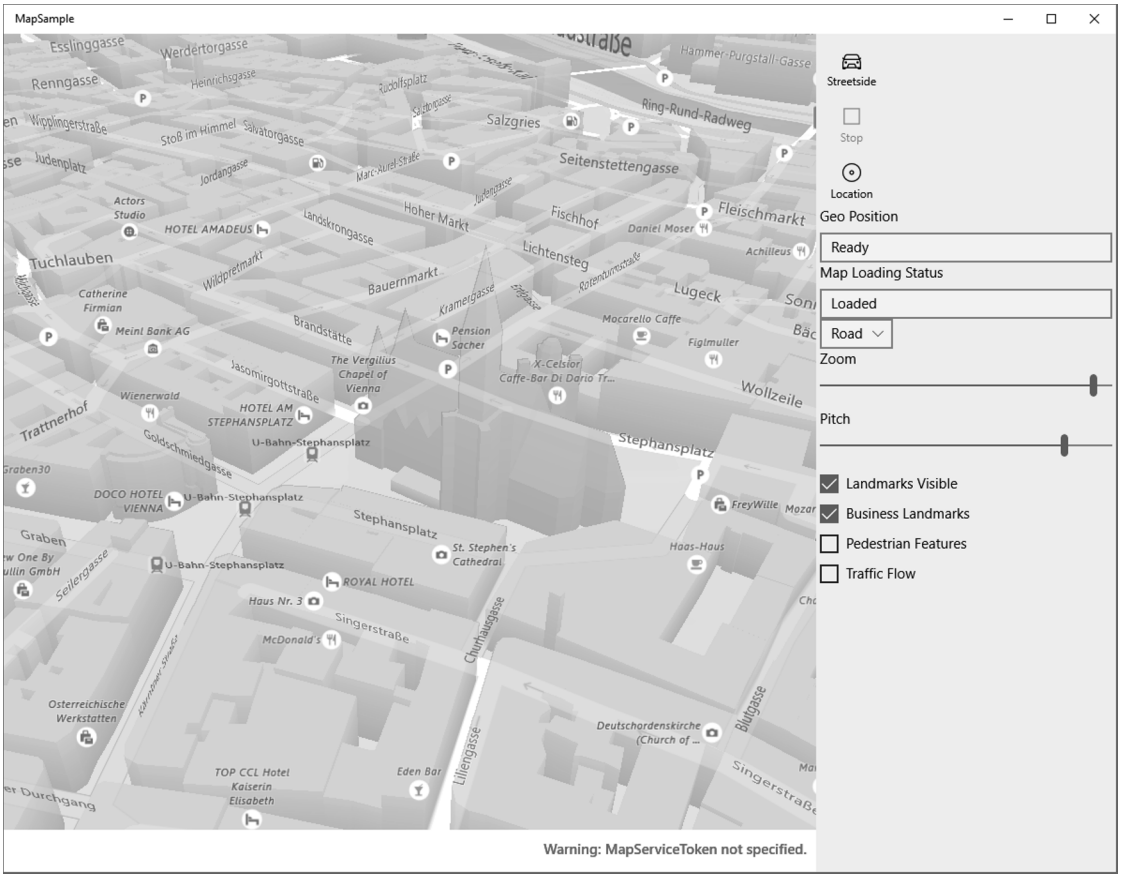

FIGURE BC5-3

The `Geoposition` instance returned from `GetGeopositionAsync` lists information about how the `Geolocator` determined the position: using a cellular network with a phone, satellite, a Wi-Fi network that is recorded, or an IP address. When you configure the `Geolocator`, you can specify how accurate the information should be. By setting the property `DesiredAccuracyInMeters`, you can define how exact the location should be within a meter range. Of course, this accuracy is what you hope for, but it might not be possible to achieve. If the location should be more exact, GPS information from accessing satellite information can be used. Depending on the technology needed, more battery is used, so you shouldn't specify such accuracy if it isn't necessary. Satellite or cellular information cannot be used if the device doesn't offer these features. In those cases, you can use only the Wi-Fi network (if available) or an IP address. Of course, the IP address can be imprecise. Maybe you're getting the geolocation of an IP provider instead of the user. With the device and network I'm using, I get an accuracy of 55 meters. The source of the position is Wi-Fi. The result is very accurate. You can see the map in Figure BC5-6.

FIGURE BC5-4

Street-Side Experience

Another feature offered by the `MapControl` is street-side experience. This feature is not available with all devices. You need to check the `IsStreetsideSupported` property from the `MapControl` before using it. In cases where street view is supported by the device, you can try to find nearby street-side places using the static method `FindNearbyAsync` of the `StreetsidePanorama` class. Street-side experience is available only for some locations. You can test to find out whether it is available in your location. If `StreetsidePanorama` information is available, it can be passed to the `StreetsideExperience` constructor and assigned to the `CustomExperience` property of the `MapControl` (code file `MapSample/ViewModels/MapsViewModel.cs`):

```
public async void StartStreetViewAsync()
{
  if (_mapControl.IsStreetsideSupported)
  {
    var panorama = await StreetsidePanorama.FindNearbyAsync(CurrentPosition);
```

```
        if (panorama == null)
        {
          var dlg = new MessageDialog("No streetside available here");
          await dlg.ShowAsync();
          return;
        }
        IsStreetView = true;
        _mapControl.CustomExperience = new StreetsideExperience(panorama);
      }
    }
```

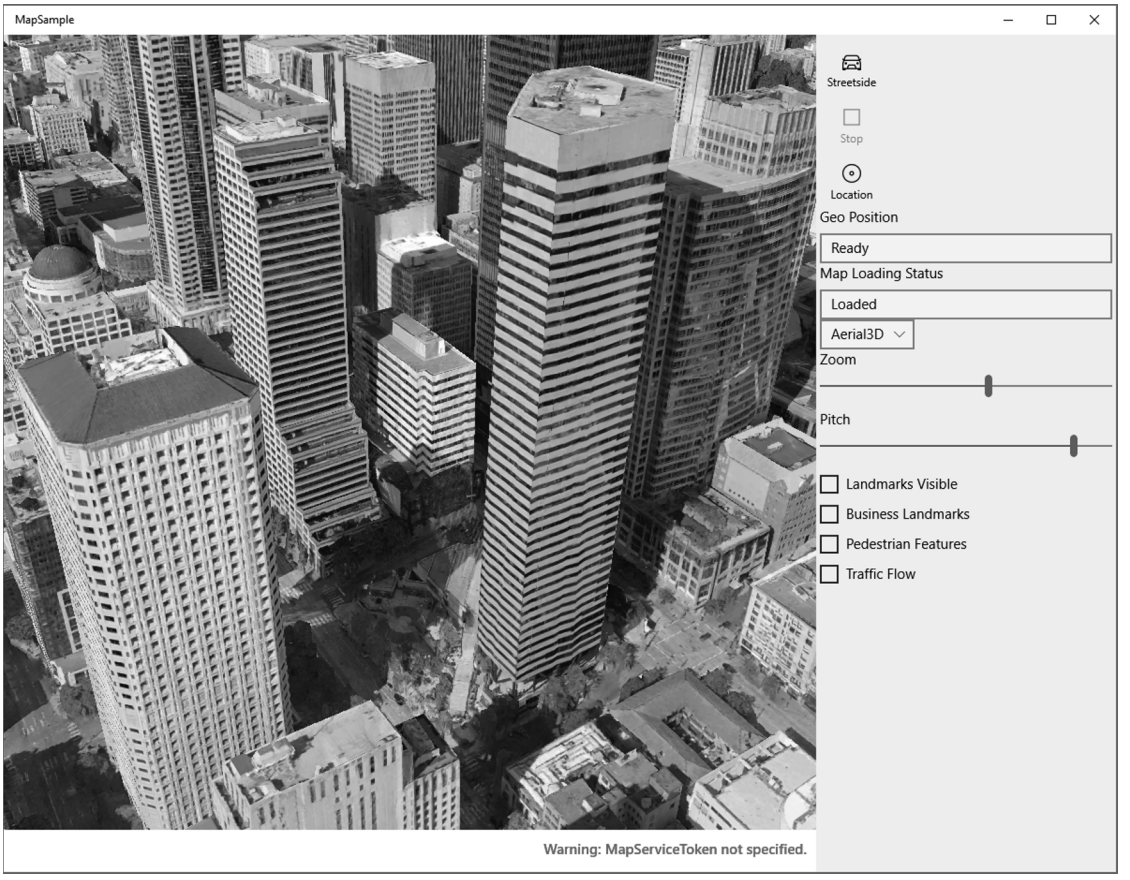

FIGURE BC5-5

Street-side experience looks like what's shown in Figure BC5-7.

FIGURE BC5-6

Continuously Requesting Location Information

Instead of getting the location just once using the `Geolocator`, you can also retrieve the location based on a time interval or the movement of the user. With the `Geolocator`, you can set the `ReportInterval` property to a minimum time interval in milliseconds between location updates. Updates can still happen more often—for example, if another app requested geo information with a smaller time interval. Instead of using a time interval, you can specify that the movement of the user fire location information. The property `MovementThreshold` specifies the movement in meters.

After setting the time interval or movement threshold, the `PositionChanged` event is fired every time a position update occurs:

```
private void OnGetContinuousLocation(object sender, RoutedEventArgs e)
{
  locator = new Geolocator();
  locator.DesiredAccuracy = PositionAccuracy.High;
  // locator.ReportInterval = 1000;
  locator.MovementThreshold = 10;
  locator.PositionChanged += (sender1, e1) =>
```

```
    {
      // position updated
    };
    locator.StatusChanged += (sender1, e1) =>
    {
      // status changed
    };
  }
```

FIGURE BC5-7

> **NOTE** *Debugging apps with position changes does not require that you now get into a car and debug your app while on the road. Instead, the simulator is a helpful tool.*

SENSORS

For a wide range of sensors, the Windows Runtime offers direct access. The namespace `Windows.Devices.Sensors` contains classes for several sensors that can be available with different devices.

Before stepping into the code, it helps to have an overview of the different sensors and what they can be used for with the following table. Some sensors are very clear with their functionality, but others need some explanation.

SENSOR	FEATURES
`LightSensor`	The light sensor returns the light in lux. This information is used by Windows to set the screen brightness.
`Compass`	The compass gives information about how many degrees the device is directed to the north using a magnetometer. This sensor differentiates magnetic and geographic north.
`Accelerometer`	The accelerometer measures G-force values along x, y, and z device axes. This could be used by an app that shows a marble rolling across the screen.
`Gyrometer`	The gyrometer measures angular velocities along x, y, and z device axes. If the app cares about device rotation, this is the sensor that can be used. However, moving the device also influences the gyrometer values. It might be necessary to compensate the gyrometer values using accelerometer values to remove moving of the device and just work with the real angular velocities.
`Inclinometer`	The inclinometer gives number of degrees as the device rotates across the x-axis (pitch), y-axis (roll), and z-axis (yaw). An example of when this could be used is an app showing an airplane that matches yaw, pitch, and roll.
`OrientationSensor`	The orientation uses data from the accelerometer, gyrometer, and magnetometer and offers the values both in a quaternion and a rotation matrix.
`Barometer`	The barometer measures atmospheric pressure.
`Altimeter`	The altimeter measures the relative altitude.
`Magnetometer`	The magnetometer measures the strength and direction of a magnetic field.
`Pedometer`	The pedometer measures the steps taken. Usually you're not walking with your desktop PC, which doesn't have such a sensor, maybe you walk with a mobile device turned on. This is probably more likely than you're using a Windows phone.
`ProximitySensor`	The proximity sensor measures the distance of nearby objects. It uses an electromagnetic field or infrared sensor to measure the distance.

Depending on your device, only a few of these sensors are available. Many of these sensors are used only within mobile devices. For example, counting your steps with a desktop PC might not result in the number of steps you should reach during a day.

An important aspect with sensors that return coordinates is that it's not the display orientation coordinate system that is used with Windows apps. Instead, it's using device orientation, which can be different based on the device. For example, for a Surface Pro that is by default positioned horizontally, the x-axis goes to right, y-axis to top, and the z-axis away from the user.

The sample app for using the sensors shows the results of several sensors in two ways: You can get the sensor value once, or you can read it continuously using events. You can use this app to see what sensor data is available with your device and see what data is returned as you move the device.

For each of the sensors shown in the app, a `RelativePanel` that contains two `Button` and two `Textblock` controls is added to the `MainPage`. The following code snippet defines the controls for the light sensor (code file `SensorSample/MainPage.xaml`):

```
<Border BorderThickness="3" Margin="12" BorderBrush="Blue">
  <RelativePanel>
    <Button x:Name="GetLightButton" Margin="8" Content="Get Light"
      Click="{x:Bind LightViewModel.OnGetLight}" />
    <Button x:Name="GetLightButtonReport" Margin="8"
      RelativePanel.Below="GetLightButton" Content="Get Light Report"
      Click="{x:Bind LightViewModel.OnGetLightReport}" />
    <TextBlock x:Name="LightText" Margin="8"
      RelativePanel.RightOf="GetLightButtonReport"
      RelativePanel.AlignBottomWith="GetLightButton"
      Text="{x:Bind LightViewModel.Illuminance, Mode=OneWay}" />
    <TextBlock x:Name="LightReportText" Margin="8"
      RelativePanel.AlignLeftWith="LightText"
      RelativePanel.AlignBottomWith="GetLightButtonReport"
      Text="{x:Bind LightViewModel.IlluminanceReport, Mode=OneWay}" />
  </RelativePanel>
</Border>
```

Light

As soon as you know how to work with one sensor, the other ones are very similar. Let's start with the `LightSensor`. First, an object is accessed invoking the static method `GetDefault`. You can get the actual value of the sensor by calling the method `GetCurrentReading`. With the `LightSensor`, `GetCurrentReading` returns a `LightSensorReading` object. This reading object defines the `IlluminanceInLux` property that returns the luminance in lux (code file `SensorSample/ViewModels/LightViewModel.cs`):

```
public class LightViewModel: BindableBase
{
  public void OnGetLight()
  {
    LightSensor sensor = LightSensor.GetDefault();
    if (sensor != null)
    {
      LightSensorReading reading = sensor.GetCurrentReading();
      Illuminance = $"Illuminance: {reading?.IlluminanceInLux}";
    }
    else
    {
      Illuminance = "Light sensor not found";
    }
  }

  private string _illuminance;
  public string Illuminance
  {
    get => _illuminance;
    set => Set(ref _illuminance, value);
  }
  //...
}
```

For getting continuous updated values, the `ReadingChanged` event is fired. Specifying the `ReportInterval` property specifies the time interval that should be used to fire the event. It may not be lower than `MinimumReportInterval`. With the event, the second parameter `e` is of type

`LightSensorReadingChangedEventArgs` and specifies the `LightSensorReading` with the `Reading` property:

```
public class LightViewModel: BindableBase
{
  //...
  public void OnGetLightReport()
  {
    LightSensor sensor = LightSensor.GetDefault();
    if (sensor != null)
    {
      sensor.ReportInterval = Math.Max(sensor.MinimumReportInterval, 1000);
      sensor.ReadingChanged += async (s, e) =>
      {
        LightSensorReading reading = e.Reading;
        await CoreApplication.MainView.Dispatcher.RunAsync(
          CoreDispatcherPriority.Low, () =>
          {
             IlluminanceReport =
               $"{reading.IlluminanceInLux} {reading.Timestamp:T}";
          });
      };
    }
  }

  private string _illuminanceReport;
  public string IlluminanceReport
  {
    get => _illuminanceReport;
    set => Set(ref _illuminanceReport, value);
  }
}
```

Compass

The compass can be used very similarly. The `GetDefault` method returns the `Compass` object, and `GetCurrentReading` retrieves the `CompassReading` representing the current values of the compass. `CompassReading` defines the properties `HeadingAccuracy`, `HeadingMagneticNorth`, and `HeadingTrueNorth`.

In cases where `HeadingAccuracy` returns `MagnometerAccuracy.Unknown` or `Unreliable`, the compass needs to be calibrated (code file `SensorSample/ViewModels/CompassviewModel.cs`):

```
public class CompassViewModel: BindableBase
{
  public void OnGetCompass()
  {
    Compass sensor = Compass.GetDefault();
    if (sensor != null)
    {
      CompassReading reading = sensor.GetCurrentReading();
      CompassInfo = $"magnetic north: {reading.HeadingMagneticNorth} " +
        $"real north: {reading.HeadingTrueNorth} " +
        $"accuracy: {reading.HeadingAccuracy}";
    }
    else
    {
      CompassInfo = "Compass not found";
    }
  }
  private string _compassInfo;
```

```
  public string CompassInfo
  {
    get => _compassInfo;
    set => Set(ref _compassInfo, value);
  }
  //...
}
```

Continuous updates are available with the compass as well:

```
public class CompassViewModel: BindableBase
{
  //...
  public void OnGetCompassReport()
  {
    Compass sensor = Compass.GetDefault();
    if (sensor != null)
    {
      sensor.ReportInterval = Math.Max(sensor.MinimumReportInterval, 1000);
      sensor.ReadingChanged += async (s, e) =>
      {
        CompassReading reading = e.Reading;
        await CoreApplication.MainView.Dispatcher.RunAsync(
        CoreDispatcherPriority.Low, () =>
        {
          CompassInfoReport =
            $"magnetic north: {reading.HeadingMagneticNorth} " +
            $"real north: {reading.HeadingTrueNorth} " +
            $"accuracy: {reading.HeadingAccuracy} {reading.Timestamp:T}";
        });
      };
    }
  }

  private string _compassInfoReport;
  public string CompassInfoReport
  {
    get => _compassInfoReport;
    set => Set(ref _compassInfoReport, value);
  }
}
```

Accelerometer

The accelerometer gives information about the g-force values along x-, y-, and z-axes of the device. With a landscape device, the x-axis is *horizontal, the y-axis is vertical, and the z-axis* is oriented in direction from the user. For example, if the device stands upright at a right angle on the table with the Windows button on bottom, the x has a value of –1. When you turn the device around to have the Windows button on top, x has a value of +1.

Like the other sensors you've seen so far, the static method `GetDefault` returns the `Accelerometer`, and `GetCurrentReading` gives the actual accelerometer values with the `AccelerometerReading` object. `AccelerationX`, `AccelerationY`, and `AccererationZ` are the values that can be read (code file `SensorSample/ViewModels/AccelerometerViewModel.cs`):

```
public class AccelerometerViewModel: BindableBase
{
  public void OnGetAccelerometer()
  {
    Accelerometer sensor = Accelerometer.GetDefault();
    if (sensor != null)
    {
```

```
      AccelerometerReading reading = sensor.GetCurrentReading();
      AccelerometerInfo = $"X: {reading.AccelerationX} " +
        $"Y: {reading.AccelerationY} Z: {reading.AccelerationZ}";
    }
    else
    {
      AccelerometerInfo = "Compass not found";
    }
  }

  private string _accelerometerInfo;
  public string AccelerometerInfo
  {
    get => _accelerometerInfo;
    set => Set(ref _accelerometerInfo, value);
  }
  //...
}
```

You get continuous values from the accelerometer by assigning an event handler to the `ReadingChanged` event. As this is the same as with the other sensors that have been covered so far, the code snippet is not shown in the book. However, you get this functionality with the code download of this chapter. You can test your device and move it continuously while reading the accelerometer values.

Inclinometer

The inclinometer is for advanced orientation; it gives yaw, pitch, and roll values in degrees with respect to gravity. The resulting values are specified by the properties `PitchDegrees`, `RollDegrees`, and `YawDegrees` (code file `SensorSample/ViewModels/InclinometerViewModel.cs`):

```
public class InclinometerViewModel: BindableBase
{
  public void OnGetInclinometer()
  {
    Inclinometer sensor = Inclinometer.GetDefault();
    if (sensor != null)
    {
      InclinometerReading reading = sensor.GetCurrentReading();
      InclinometerInfo = $"pitch degrees: {reading.PitchDegrees} " +
        $"roll degrees: {reading.RollDegrees} " +
        $"yaw accuracy: {reading.YawAccuracy} " +
        $"yaw degrees: {reading.YawDegrees}";
    }
    else
    {
      InclinometerInfo = "Inclinometer not found";
    }
  }

  private string _inclinometerInfo;
  public string InclinometerInfo
  {
    get => _inclinometerInfo;
    set => Set(ref _inclinometerInfo, value);
  }
  //...
}
```

Gyrometer

The `Gyrometer` gives angular velocity values for the x-, y-, and z-device axes (code file `SensorSample/ViewModels/GyrometerViewModel.cs`):

```
public class GyrometerViewModel: BindableBase
{
  public void OnGetGyrometer()
  {
    Gyrometer sensor = Gyrometer.GetDefault();
    if (sensor != null)
    {
      GyrometerReading reading = sensor.GetCurrentReading();
      GyrometerInfo = $"X: {reading.AngularVelocityX} " +
        $"Y: {reading.AngularVelocityY} Z: {reading.AngularVelocityZ}";
    }
    else
    {
      GyrometerInfo = "Gyrometer not found";
    }
  }

  private string _gyrometerInfo;
  public string GyrometerInfo
  {
    get => _gyrometerInfo;
    set => Set(ref _gyrometerInfo, value);
  }
  //...
}
```

Orientation

The `OrientationSensor` is the most complex sensor because it takes values from the accelerometer, gyrometer, and magnetometer. You get all the values in either a quaternion represented by the `Quaternion` property or a rotation matrix (`RotationMatrix` property).

Try the sample app to see the values and how you move the device (code file `SensorSample/ViewModels/OrientationViewModel.cs`):

```
public static class OrientationSensorExtensions
{
  public static string Output(this SensorQuaternion q) =>
    $"x {q.X} y {q.Y} z {q.Z} w {q.W}";

  public static string Ouput(this SensorRotationMatrix m) =>
    $"m11 {m.M11} m12 {m.M12} m13 {m.M13} " +
    $"m21 {m.M21} m22 {m.M22} m23 {m.M23} " +
    $"m31 {m.M31} m32 {m.M32} m33 {m.M33}";
  }

public class OrientationViewModel: BindableBase
{
  public void OnGetOrientation()
  {
    OrientationSensor sensor = OrientationSensor.GetDefault();
    if (sensor != null)
    {
```

```
        OrientationSensorReading reading = sensor.GetCurrentReading();
        OrientationInfo = $"Quaternion: {reading.Quaternion.Output()} " +
          $"Rotation: {reading.RotationMatrix.Ouput()} " +
          $"Yaw accuracy: {reading.YawAccuracy}";
      }
      else
      {
        OrientationInfo = "Compass not found";
      }
    }

    private string _orientationInfo;
    public string OrientationInfo
    {
      get => _orientationInfo;
      set => Set(ref _orientationInfo, value);
    }
    //...
  }
```

When you run the app, you can see sensor data as shown in Figure BC5-8.

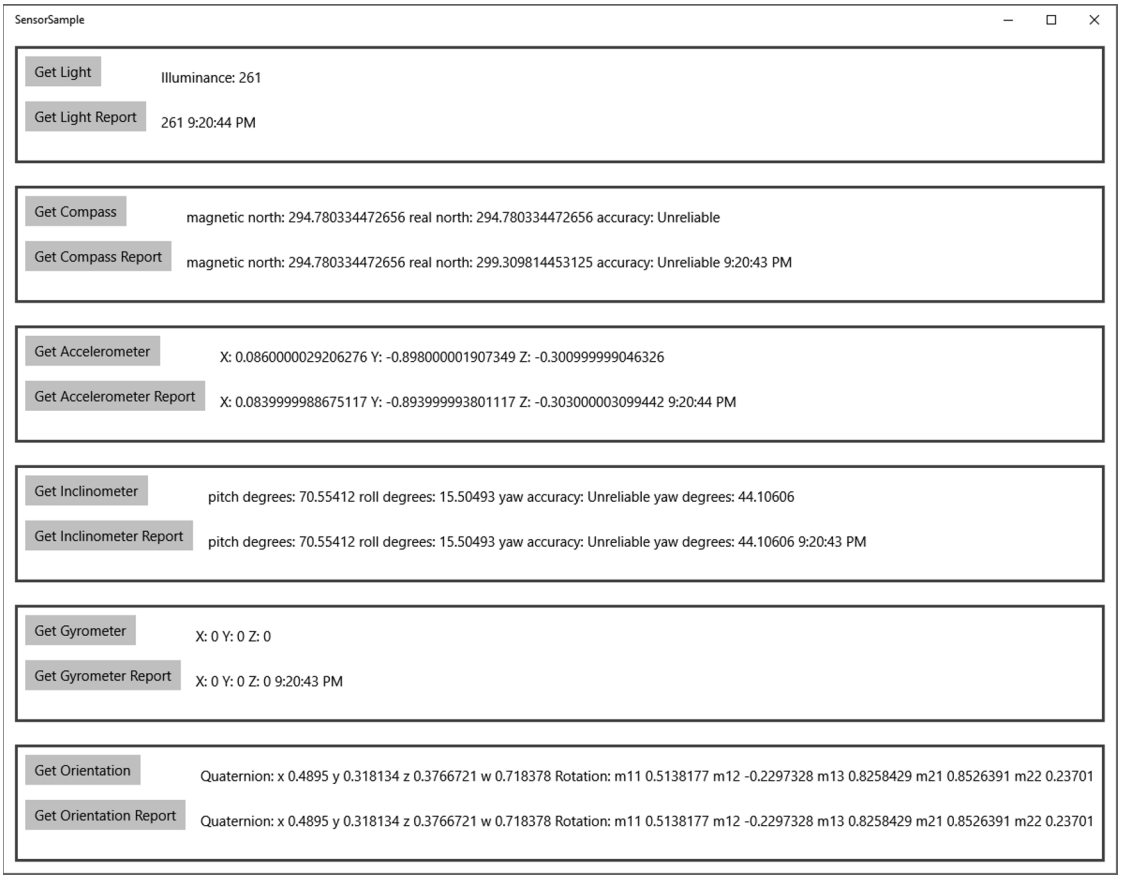

FIGURE BC5-8

Rolling Marble Sample

For seeing sensor values in action not only with result values in a `TextBlock` element, you can make a simple sample app that makes use of the `Accelerometer` to roll a marble across the screen.

The marble is represented by a red ellipse. Having an `Ellipse` element positioned within a `Canvas` element allows moving the `Ellipse` with an attached property (code file `RollingMarble/MainPage.xaml`):

```
<Canvas Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  <Ellipse Fill="Red" Width="100" Height="100" Canvas.Left="550"
    Canvas.Top="400" x:Name="ell1" />
</Canvas>
```

> **NOTE** *Attached properties and the* `Canvas` *panel are explained in Chapter 33.*

The constructor of the `MainPage` initializes the `Accelerometer` and requests continuous reading with the minimum interval. To know the boundaries of the window, with the `LayoutUpdated` event of the page, `MaxX` and `MaxY` are set to the width and height of the window minus size of the ellipse (code file `RollingMarble/MainPage.xaml.cs`):

```
public sealed partial class MainPage: Page
{
  private Accelerometer _accelerometer;
  private double _minX = 0;
  private double _minY = 0;
  private double _maxX = 1000;
  private double _maxY = 600;
  private double _currentX = 0;
  private double _currentY = 0;

  public MainPage()
  {
    InitializeComponent();
    _accelerometer = Accelerometer.GetDefault();
    if (_accelerometer != null)
    {
      accelerometer.ReportInterval = accelerometer.MinimumReportInterval;
      accelerometer.ReadingChanged += OnAccelerometerReading;
    }
    else
    {
      textMissing.Visibility = Visibility.Visible;
    }

    LayoutUpdated += (sender, e) =>
    {
      _maxX = this.ActualWidth-100;
      _maxY = this.ActualHeight-100;
    };
}
```

With every value received from the accelerometer, the ellipse is moved within the `Canvas` element in the event handler method `OnAccelerometerReading`. Before the value is set, it is checked according to the boundaries of the window:

```
private async void OnAccelerometerReading(Accelerometer sender,
  AccelerometerReadingChangedEventArgs args)
```

```
{
  currentX += args.Reading.AccelerationX * 80;
  if (currentX < _minX) currentX = _minX;
  if (currentX > _maxX) currentX = _maxX;

  currentY += -args.Reading.AccelerationY * 80;
  if (currentY < _minY) currentY = _minY;
  if (currentY > _maxY) currentY = _maxY;

  await this.Dispatcher.RunAsync(CoreDispatcherPriority.High, () =>
  {
    Canvas.SetLeft(ell1, _currentX);
    Canvas.SetTop(ell1, _currentY);
  });
}
```

Now you run the app and move the device to get the marble rolling as shown in Figure BC5-9.

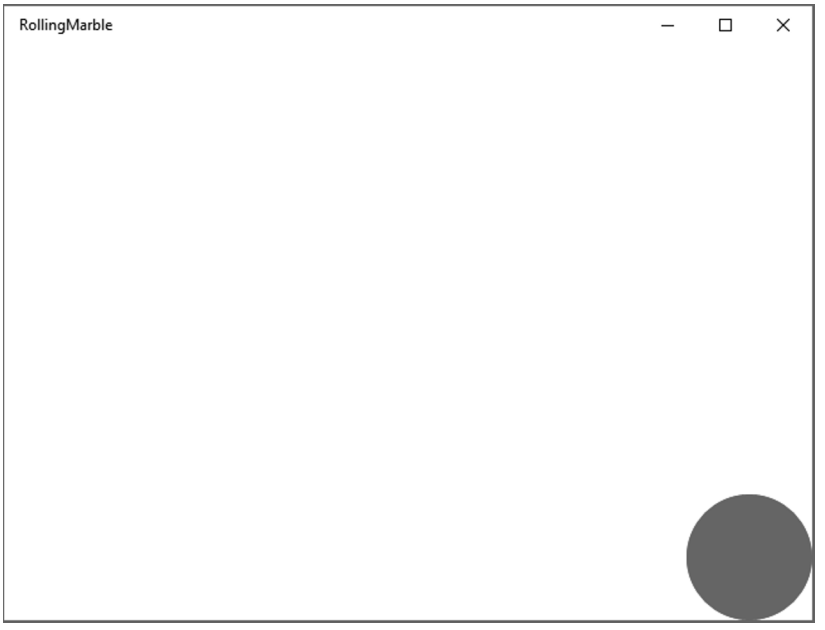

FIGURE BC5-9

SUMMARY

This chapter covered using several devices, including the camera for taking pictures and recording videos, a geolocator for getting the location of the user, and a bunch of different sensors for getting information about how the device moves.

You've also seen how to use the `MapControl` to show maps with landmarks, provide Aerial and Aerial3D views of maps, and include the street-side view experience.