View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Springer - Publisher Connector

Chapter 9

Sensors

Sensors enable your BlackBerry 10 device to collect information about the outside world and to react to physical events. With some imagination, you can use the sensors API to build highly immersive apps that respond to the device's position, accelerations, and rotations. Gaming is an obvious area that benefits from using sensors, but the majority of apps have yet to tap into the potential of using sensors. There are really no limits to what you can achieve, and as mobile devices continue adding new types of sensors, the number of applications that use sensor data will experience exponential growth in the years to come.

Cascades leverages the Qt Mobility module for the sensors API (this is a good example of how BlackBerry 10 is built using a layered architecture where Cascades uses the underlying Qt modules when necessary; see Chapter 1). As illustrated in Figure 9-1, the sensors architecture is designed around a front end and a back end. The front end, a `QSensor` instance or subclass, is what you call to access data provided by the back end (which can be considered as a low-level wrapper to the actual hardware sensor; in other words, a glorified device driver). The advantage of splitting sensors into a back end and front end is that you can use a common abstraction to access data, regardless of the sensor type. I will show you how to use `QSensor` in a generic way. In most cases, I will directly instantiate a subclass to do the actual data reading (the data is returned to the application as an instance of `QSensorReading` or one of its subclasses).

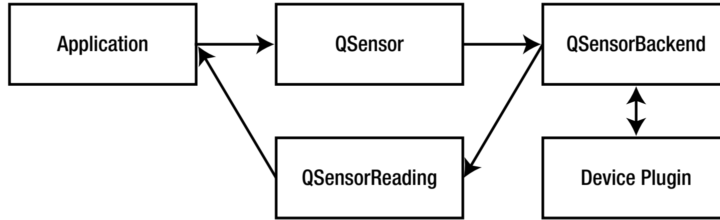

Figure 9-1. Sensor architecture

Finally, you can also directly access sensors from QML, which is important if you want to design sensor-aware applications entirely in QML/JavaScript.

The purpose of this chapter is to give you an overview of the BlackBerry 10 sensor types, as well as show you how to use them in your own applications. You will also learn how to handle sensor readings in C++ and QML/JavaScript.

Sensor Types

At the time of writing, the following sensors types are supported by the BlackBerry 10 platform. (Note that for a given device, not all sensor types are supported. The next section will show you how to detect the availability of a given sensor type at runtime. You can also check the BlackBerry web site for device specifications, which also lists supported sensors).

- *Ambient light sensor*: Returns a constant representing the current brightness of the external environment. You can use it to adjust the backlight, thus optimizing battery power consumption.
- *Light sensor*: Returns a value representing the light intensity measured in lux.
- *Accelerometer*: Returns the device acceleration in three dimensions. You can also specify which acceleration component should be reported by the sensor (gravity, user, or combined). For example, only the gravity component is relevant if you want to detect if the device is falling.
- *Compass*: Returns the device's azimuth, which is the angle between the device's current orientation when it is pointing toward the horizon and the magnetic north (the sensor reading is a clockwise angle measured in degrees).
- *Gyroscope*: Returns the device's angular velocity in three dimensions measured in degrees per second.
- *Holster sensor*: Returns a Boolean value indicating whether the device is in the holster or not.
- *Proximity sensor*: Returns a Boolean value, which indicates whether an object is close to the device.
- *Infrared proximity sensor*: Returns the measured reflectance, which is a percentage of the emitted infrared light returned by an object. Note that in practice it is easier to use the proximity sensor than to try to detect an object's presence with the infrared proximity sensor.

- *Magnetometer*: Returns the current magnetic field measured in Teslas.
- *Orientation*: Reports the device orientation. For example, you can use this sensor to detect whether the device is pointing up or down.
- *Rotation*: Returns a reading containing three angles—measured in angles—that define the orientation of the device in space (the device coordinate system will be explained shortly).

All sensors essentially work in the same way, as follows:

1. Instantiate a `QSensor` or one of its subclasses.
2. Set the sensor's properties according to your application's requirements. For example, you can specify that the sensor should not send you duplicate values or that it should not be active when the application is running in the background.
3. Optionally, add filters to the sensor in order to provide a more efficient way of notifying data changes. (For example, the accelerometer readings are very susceptible to noise. You can use a filter to smooth out the noisy signal and notify your application when a reading has truly changed).
4. Connect the `QSensor::readingChanged()` signal to a slot in your application in order to receive sensor readings.
5. Once the initial setup has been completed, you can start the sensor readings with a call to `QSensor::start()`.
6. Handle the sensor data using the slot you have configured for the `QSensor::readingChanged()` signal.
7. When you are done using the sensor, call `QSensor::stop()` to end data notifications.

Sensors in C++

Determining Sensors Types

Not all of the sensors described in the previous section are available on a given device. You will therefore have to determine the availability of a sensor by using the `QSensor::sensorTypes()` method, which returns a list of sensors. For example, Listing 9-1 shows you how to check for the presence of an accelerometer.

Listing 9-1. Sensors Check

```
bool checkForAccelerometer(){
    QList<QByteArray> sensorTypes = QSensor::sensorTypes();
    return sensorTypes.contains(QAccelerometer::type);
}
```

You need to add the following two lines to your application's `.pro` file in order to use sensors:

```
Config += mobility
MOBILITY += sensors
```

You can access the Sensors project presented in this chapter by cloning the BB10Apress repository (https://github.com/aludin/BB10Apress).

Using Sensors in C++

The sensors API blends in with the rest of the QtCore APIs, and as usual in the world of Qt, it is all about connecting signals to slots. To illustrate how sensors work in practice, let us put together a very simple application displaying multiple sensor values. The application illustrated in Figure 9-2 combines acceleration readings with light readings.

***Figure 9-2.** Sensors view*

When the Start button is touched, the application starts receiving data from the accelerometer and light sensors, and updates the corresponding UI text fields. The Stop button interrupts the data flow from the sensors. The corresponding QML document is show in Listing 9-2.

Listing 9-2. main.qml

```
import bb.cascades 1.2
Page {
    Container {
        leftPadding: 10
        rightPadding: 10
        Label {
            text: "Hello Sensors"
            textStyle.base: SystemDefaults.TextStyles.BigText
```

```
        horizontalAlignment: HorizontalAlignment.Center
    }
    Container {
        bottomMargin: 50
        layout: StackLayout {
            orientation: LayoutOrientation.LeftToRight
        }
        Label {
            text: "Accel x:"
            verticalAlignment: VerticalAlignment.Center
        }
        TextField {
            text: _app.sensor.accelX
        }

    }
    Container {
        bottomMargin: 50
        layout: StackLayout {
            orientation: LayoutOrientation.LeftToRight
        }
        Label {
            text: "Accel y:"
            verticalAlignment: VerticalAlignment.Center
        }
        TextField {
            text: _app.sensor.accelY
        }
    }
    Container {
        bottomMargin: 50
        layout: StackLayout {
            orientation: LayoutOrientation.LeftToRight
        }
        Label {
            text: "Accel z:"
            verticalAlignment: VerticalAlignment.Center
        }
        TextField {
            text: _app.sensor.accelZ
        }
    }
    Container {
        bottomMargin: 50

        layout: StackLayout {
            orientation: LayoutOrientation.LeftToRight
        }
        Label {
            text: "Light    :"
            verticalAlignment: VerticalAlignment.Center
        }
```

```
            TextField {
                id: light
                text: _app.sensor.lux
            }
        }
        Container {
            layout: StackLayout {
                orientation: LayoutOrientation.LeftToRight
            }
            horizontalAlignment: HorizontalAlignment.Center
            Button {
                text: "start"
                onClicked: {
                    _app.sensor.start();
                }
            }
            Button {
                text: "stop"
                onClicked: {
                    _app.sensor.stop();
                }
            }
        }
    }
}
```

The preceding QML code is fairly straightforward. Three text fields are used to display the device's current acceleration in three-dimensional space and a fourth text field displays the current luminosity. Note how the text fields' text properties have been bound to corresponding `_app.sensor` properties. You will see shortly how the `_app.sensor` variable is defined.

HybridSensor

The `HybridSensor` class encapsulates the sensors reading logic (see Listing 9-3).

Listing 9-3. ApplicationUI.hpp

```
#ifndef HYBRIDSENSOR_H_
#define HYBRIDSENSOR_H_

#include <QObject>
#include <QtSensors/QAccelerometer>
#include <QtSensors/QLightSensor>

class HybridSensor : public QObject {
    Q_OBJECT
    Q_PROPERTY(qreal accelX READ accelX NOTIFY accelChanged)
    Q_PROPERTY(qreal accelY READ accelY NOTIFY accelChanged)
    Q_PROPERTY(qreal accelZ READ accelZ NOTIFY accelChanged)
    Q_PROPERTY(qreal lux READ lux NOTIFY luxChanged)
```

```
public:
    HybridSensor(QObject* parent = 0);
    virtual ~HybridSensor();

signals:
    void accelChanged();
    void luxChanged();

public slots:
    void start();
    void stop();
    void onAccellerationChanged();
    void onLightChanged();

public:
    double accelX();
    double accelY();
    double accelZ();
    double lux();

private:
    QtMobility::QAccelerometer* m_accelerometer;
    QtMobility::QLightSensor* m_lightSensor;

    double m_accelX;
    double m_accelY;
    double m_accelZ;
    double m_lux;
};

#endif /* HYBRIDSENSOR_H_ */
```

As illustrated in Listing 9-3, the `HybridSensor` class declares four properties intended to be accessed from QML (`accelX`, `accelY`, `accelZ`, and `lux`). These are the same properties that will be bound to the corresponding QML text fields. The `m_accelerometer` and `m_lightSensor` member variables provide the actual sensor readings (`m_accelerometer` is an instance of the `QAccelerometer` class and `m_lightSensor` an instance of `QLightSensor`). Both variables are initialized in the `HybridSensor` class constructor, which is shown in Listing 9-4. The `start()` and `stop()` slots are used respectively for initiating and halting sensor readings. The `onAccelerationChanged()` slot is called by the accelerometer sensor when a new reading is available, and the `onLightChanged()` slot is called by the light sensor when a new light reading is available (as you will see shortly, the slots "propagate" the sensor signals using the corresponding `HybridSensor` notify signals in order to update the QML bindings).

Listing 9-4. HybridSensor Constructor

```
HybridSensor::HybridSensor(QObject* parent) :
                QObject(parent),
                m_accelerometer(new QAccelerometer(this)),
                m_lightSensor(new QLightSensor(this)),
                m_accelX(0), m_accelY(0), m_accelZ(0), m_lux(0) {
```

```
        m_accelerometer->setAccelerationMode(QAccelerometer::User);
        m_accelerometer->setSkipDuplicates(true);
        m_accelerometer->setAlwaysOn(false);
        m_accelerometer->setAxesOrientationMode(QAccelerometer::FixedOrientation);

        bool result = QObject::connect(m_accelerometer, SIGNAL(readingChanged()), this,
                        SLOT(onAccellerationChanged()));
        Q_ASSERT(result);

        result = QObject::connect(m_lightSensor, SIGNAL(readingChanged()), this,
                        SLOT(onLightChanged()));
        Q_ASSERT(result);
}
```

As usual, you need to handle memory management correctly by setting the "parent-child" ownerships of all dynamically allocated member variables (in the code shown in Listing 9-4, the parent object is the `HybridSensor` instance). There are a few interesting points to consider in the way the accelerometer sensor is initialized. Setting `QAccelerometer::setSkipDuplicates()` to `true` results in the sensor notifying the application only when data has changed. This eliminates duplicate updates when successive readings are identical or very similar. Setting `QAccelerometer::setAlwaysOn()` to `false` ensures that the application will not receive sensor data when it's running in the background (this is the default behavior, but I prefer making it explicit in the code). You should be aware that if you decide to override the default behavior, running sensors such as the accelerometer in the background will drain the device's power quickly.

Next, we proceed by specifying the way the sensor should report the data to the application: the call to `QAccelerometer::setAccelerationMode(QAccelerometer::User)` tells the sensor to only report the acceleration caused by the user moving the device (i.e., the effect of gravity is discarded). The call to `QAccelerometer::setAxesOrientation(QAccelerometer::FixedOrientation)` fixes the coordinate system so that axes are not reoriented when the device orientation changes (I will tell you more about coordinate systems shortly).

Next, you connect the accelerometer's `readingChanged()` signal to HybridSensor's `onAccelerationChanged()` slot. As mentioned previously, the accelerometer sensor will call the slot when a new reading is available. In a similar way, the light sensor's `readingChanged()` signal is connected to the application's `onLightChanged()` slot. Finally, the code for HybridSensor's slots is given in Listing 9-5.

Listing 9-5. HybridSensor Slots

```
void HybridSensor::start() {
    m_accelerometer->start();
    m_lightSensor->start();
}

void HybridSensor::stop() {
    m_accelerometer->stop();
    m_lightSensor->stop();
}
```

```
void HybridSensor::onAccellerationChanged() {
    QAccelerometerReading* reading = m_accelerometer->reading();

    double x = reading->x();
    double y = reading->y();
    double z = reading->z();

    if(x*x+y*y+z*z > 0.1){
            m_accelX = x;
            m_accelY = y;
            m_accelZ = z;
            emit accelChanged();
    }
}

void HybridSensor::onLightChanged() {
    QLightReading* reading = m_lightSensor->reading();
    m_lux = reading->lux();
    emit luxChanged();
}
```

The code is relatively self-explanatory. The `start()` and `stop()` slots call the corresponding sensor methods. The `onAccelerationChanged()` slot is triggered by the accelerometer when a new reading is available: the method retrieves a pointer to a `QAccelerometerReading` instance and uses the `x`, `y`, and `z` components to update the corresponding `HybridSensor` member variables. The QML bindings are also updated with the new acceleration values when the `accelChanged` signal is emitted (note that the `accelChanged` signal is emitted only if the reading's magnitude is higher than a predefined threshold, which is defined by `x*x+y*y+z*z > 0.1`). The `onLightChanged()` slot works in a similar way by retrieving a pointer to a `QLightReading` instance.

The Application Delegate

You still need to access a `HybridSensor` instance from QML. The application delegate takes care of this by providing a QML property for the `HybridSensor` instance (see Listing 9-6).

Listing 9-6. ApplicationUI.hpp

```
class ApplicationUI : public QObject
{
    Q_OBJECT
    Q_PROPERTY(HybridSensor* sensor READ sensor CONSTANT)
public:
    ApplicationUI(bb::cascades::Application *app);
    virtual ~ApplicationUI() { }
private:
    HybridSensor* sensor();
    HybridSensor* m_hybridSensor;
};
```

The application delegate's constructor proceeds by registering the `HybridSensor` class with the QML type system. (The constructor also sets the application delegate as a QML document context property. The sensor property will therefore be accessible as `_app.sensor` from QML. See Listing 9-7.)

Listing 9-7. ApplicationUI.cpp

```
#include <bb/cascades/Application>
#include <bb/cascades/QmlDocument>
#include <bb/cascades/AbstractPane>
#include "applicationui.hpp"

using namespace bb::cascades;

ApplicationUI::ApplicationUI(bb::cascades::Application *app) :
        QObject(app), m_hybridSensor(new HybridSensor(this))
{
    qmlRegisterType<HybridSensor>();
    // Create scene document from main.qml asset, the parent is set
    // to ensure the document gets destroyed properly at shut down.
    QmlDocument *qml = QmlDocument::create("asset:///main.qml").parent(this);

    qml->documentContext()->setContextProperty("_app", this);

    // Create root object for the UI
    AbstractPane *root = qml->createRootObject<AbstractPane>();

    // Set created root object as the application scene
    app->setScene(root);
}

HybridSensor* ApplicationUI::sensor(){
        return m_hybridSensor;
}
```

Filters

Some sensors, such as the accelerometer, are particularly sensible to a noisy signal. You can therefore recourse to a filter as a way of removing spikes out of the signal. A filter permits you to do the following:

- Modify the reading values.
- Suppress the reading altogether.
- Process readings in a pipeline. The filters will be called in turn by the sensor and each filter can modify the current reading.

Filters must subclass the QSensorFilter class and implement the following pure virtual method:

- bool QSensorFilter::filter(QSensorReading* reading)=0: This function is called by the sensor when the reading changes. If the filter returns true, the next filter in the chain will handle the reading; otherwise, the reading will be dropped. When the last filter in the chain returns true, the readingChanged signal is emitted.

Note that you can greatly optimize your application by using filters and avoiding triggering the readingChanged signal unnecessarily. Also, instead of subclassing QSensorFilter directly, you can use one of its subclasses corresponding to a particular sensor type. For example, you can subclass the QAccelerometerFilter class for accelerometer readings, as follows:

```
bool QAccelerometerFilter::filter(QAccelerometerReading* reading) = 0.
```

Finally, you can add a filter to a sensor using the QSensor::addFilter(QSensorFilter* filter) method.

To illustrate the previous points, let's modify HybridSensor by adding filtering capabilities to the class (see Listing 9-8).

Listing 9-8. HybridSensor.hpp

```
class HybridSensor : public QObject, public QtMobility::QAccelerometerFilter{
Q_OBJECT
// properties omitted
public:
    virtual bool filter(QtMobility::QAccelerometerReading *reading);
// remaining class members
};
```

Next, you need to update the HybridSensor constructor (see Listing 9-9).

Listing 9-9. HybridSensor.cpp

```
HybridSensor::HybridSensor(QObject* parent) :
                QObject(parent), m_accelerometer(new QAccelerometer(this)),
                m_lightSensor(new QLightSensor(this)), m_accelX(0), m_accelY(0), m_accelZ(0),
                m_lux(0) {
        // code omitted. See Listing 9-4
        m_accelerometer->addFilter(this);
}
```

And finally, Listing 9-10 gives the filter method.

Listing 9-10. HybridSensor.hpp

```
bool HybridSensor::filter(QAccelerometerReading *reading) {
    double x = reading->x();
    double y = reading->y();
    double z = reading->z();
    if (x * x + y * y + z * z > 0.1) {
        return true;
```

```
    } else {
        return false;
    }
}
```

Sensors in QML

Using sensors in QML is deceptively simple. All you need to do is declare the sensor as an `attachedObject` property of a control in the scene graph. You can then handle the sensor's `readingChanged` signal in the usual QML way by defining an `onReadingChanged` slot. To illustrate this, I have rewritten the QML document from Listing 9-1 so that it uses sensors directly (see Listing 9-11).

Listing 9-11. main.qml

```
import bb.cascades 1.0
import QtMobility.sensors 1.3
Page {
    Container {
        leftPadding: 10
        rightPadding: 10

        Label {
            text: "Hello Sensors"
            textStyle.base: SystemDefaults.TextStyles.BigText
            horizontalAlignment: HorizontalAlignment.Center
        }
        Container {
            bottomMargin: 50
            layout: StackLayout {
                orientation: LayoutOrientation.LeftToRight
            }
            Label {
                text: "Accel x:"
                verticalAlignment: VerticalAlignment.Center
            }
            TextField {
                id: x
            }

        }
        Container {
            bottomMargin: 50
            layout: StackLayout {
                orientation: LayoutOrientation.LeftToRight
            }
            Label {
                text: "Accel y:"
                verticalAlignment: VerticalAlignment.Center
            }
```

```
        TextField {
            id: y
        }
    }
    Container {
        bottomMargin: 50
        layout: StackLayout {
            orientation: LayoutOrientation.LeftToRight
        }
        Label {
            text: "Accel z:"
            verticalAlignment: VerticalAlignment.Center
        }
        TextField {
            id: z
        }
    }
    Container {

        bottomMargin: 50

        layout: StackLayout {
            orientation: LayoutOrientation.LeftToRight
        }
        Label {
            text: "Light    :"
            verticalAlignment: VerticalAlignment.Center
        }
        TextField {
            id: light
        }
    }
    Container {
        layout: StackLayout {
            orientation: LayoutOrientation.LeftToRight
        }
        horizontalAlignment: HorizontalAlignment.Center
        Button {
            id: start
            text: "start"
            onClicked: {
                accel.start();
                lux.start();
            }
        }
        Button {
            id: stop
            text: "stop"
```

```
                onClicked: {
                    accel.stop();
                    lux.stop();
                }
            }
        }
        attachedObjects: [
            Accelerometer {
                id: accel
                active: false
                // Don't change sensor axis on screen rotation.
                axesOrientationMode: Accelerometer.FixedOrientation
                // Remove gravity, detect only user movement.
                accelerationMode: Accelerometer.User
                skipDuplicates: true
                // Called when a new accel reading is available.
                onReadingChanged: {
                    if(reading.x*reading.x+reading.y*reading.y+reading.z*reading.z > 0.1)
                    {
                       x.text = reading.x;
                       y.text = reading.y;
                       z.text = reading.z;
                    }
                }
            },
            LightSensor {
                id: lux
                active: false
                onReadingChanged: {
                    light.text = reading.lux;
                }
            }
        ]
    }
}
```

Before referencing sensors in QML, you need to import the `QtMobility.sensors` namespace (this is achieved with the second import statement). You also have to declare the sensor objects as `attachedObjects` properties of the root container. Note that the signal handlers are similar to their C++ counterparts and behave in exactly the same way.

Sensors Coordinate System

Sensors such as the accelerometer, gyroscope, and magnetometer use a right-handed coordinate system to report their readings. The x-axis, or *abscissa*, increases as you move toward the right of the screen, and the y-axis, or *ordinate*, increases as you move toward the top of the screen. Finally, the z-axis is perpendicular to the screen (see Figure 9-3).

***Figure 9-3.** Right-handed coordinate system (image source: BlackBerry web site)*

Sensors inheriting from `QOrientableSensorBase` (such as the accelerometer) can react to screen orientation changes. Therefore, these sensors can report their readings differently according to the screen's orientation. Their reporting behavior is controlled by the `QOrientableSensorBase::axesOrientationMode` property, which can take the following values:

- `QOrientableSensorBase::FixedOrientation`: This is the default behavior and the readings remain unaffected by the screen's orientation change. When the screen orientation changes, the application will have to "compensate" the returned values in order to take into account the new screen orientation (the application will also need to detect screen orientation changes).
- `QOrientableSensorBase::AutomaticOrientation`: The sensor readings are automatically remapped based on the current screen orientation. Therefore, the application need not worry about screen orientation changes (this is the recommended value to use in your application).
- `QOrientatableSensorBase::UserOrientation`: This is similar to the previous setting except that the readings are rotated by fixed angles of 0, 90, 180, and 270 degrees (no intermediate values).

Notice that applying the device rotation to the sensor readings is equivalent to rotating the coordinate system when the screen orientation changes.

Finally, angular displacements around the coordinate system's axes are also reported as right-hand rotations. You can visualize this by imagining that you are holding an imaginary screwdriver in your hand along a coordinate system axis. Positive rotations along an axis are then defined by using the screwdriver so that an imaginary screw would move toward increasing values along the axis (see Figure 9-4).

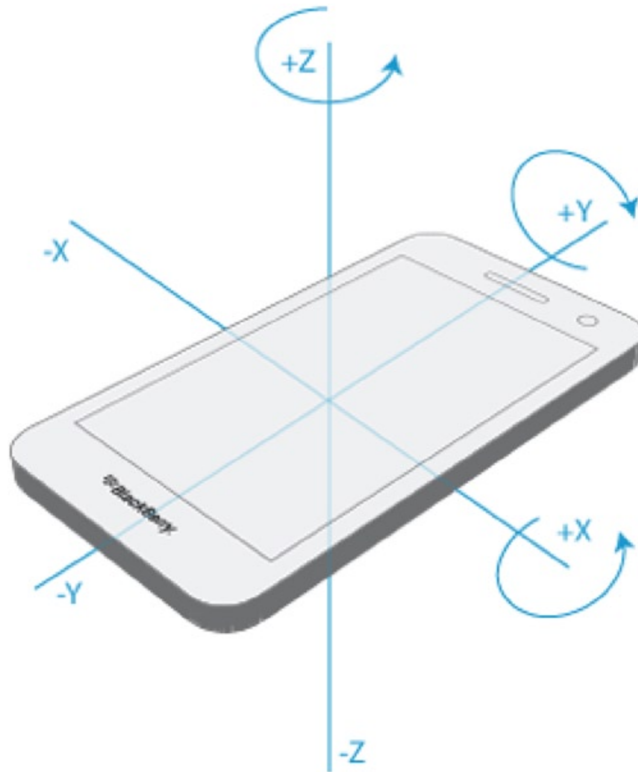

***Figure 9-4.** Right-handed rotations around coordinate system (image source: BlackBerry web site)*

Accelerometer and Gyroscope

Before finishing this chapter, I want to give you some tips on how to process the data readings provided by the accelerometer and gyroscope sensors. As you noticed throughout the chapter, receiving sensor readings is quite simple. The difficulty lies in the handling and interpretation of the data. I don't intend to give you a comprehensive treatment of the data processing, but hopefully this section will put you on the right track should you need to implement more advanced techniques in your own applications.

Accelerometer

As implied by its name, an accelerometer measures acceleration; in our case, it measures your device's *linear* acceleration in three-dimensional space. So how do you define acceleration exactly? You might recall from high-school physics that acceleration is a *vector* giving the rate of change of velocity per unit of time (a vector is a quantity having direction and magnitude). Velocity in turn is the rate at which an object changes position per unit of time. Expressing this mathematically, we can write the following:

$$a = \frac{dv}{dt} = \frac{d}{dt}\left(\frac{dx}{dt}\right) = \frac{d^2x}{dt^2}$$

An accelerometer can therefore be used in order to measure

- Velocity and displacement by integrating the accelerometer readings.
- A vibration or impact indicator (for example, when you shake or jolt the device).

So how should you proceed to integrate accelerometer values to obtain the device's velocity and position in practice? You will first need to capture accelerometer readings at regular time intervals, as previously illustrated using the `QTimer` technique. You will then need to integrate twice. The first integration step is acceleration with respect to time in order to obtain the device's velocity. You will then integrate velocity with respect to time in order to obtain the device's displacement. To illustrate this, let us consider the acceleration readings given in Figure 9-5.

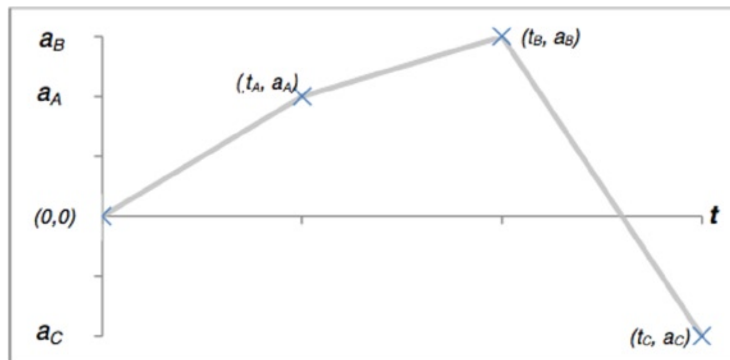

***Figure 9-5.** Acceleration readings with linear interpolation*

You will notice that I am using linear interpolation for acceleration, which also makes the integration trivial. The velocity's value at time t_A is therefore given by:

$$v_a = \int_0^{t_A} \frac{a_A}{t_A} t\,dt = \frac{a_A}{t_A}\frac{t^2}{2}\Bigg]_0^{t_A} = \frac{a_A}{2} t_A$$

Repeating the same procedure at time t_B, we get (I am going to consider here that the time samples are equally spaced and $t_B = 2t_A$) the following:

$$v_b - v_a = \int_{t_A}^{t_B} \left[\frac{a_B - a_A}{t_A} t + 2a_A - a_B \right] dt = \frac{1}{2} t_A (a_B + a_A)$$

In the general case, the following recursion stands:

$$v_n = v_{n-1} + \frac{1}{2} t_A (a_n + a_{n-1})$$

In other words, you can calculate your device's velocity at any time by sampling the acceleration and applying this recursive relation.

You can measure displacement applying the same technique, but this time by integrating velocity, as follows:

$$x_A = \int_0^{t_A} \frac{v_A}{t_A} t\, dt = \frac{a_A}{2} \frac{t^2}{2} \Big]_0^{t_A} = \frac{a_A}{4} t_A^2$$

You will then also get a recursive relation of the following form:

$$x_n = x_{n-1} + \frac{1}{2} t_A (v_n + v_{n-1})$$

Gyroscope

A gyroscope measures angular velocity. By integrating the gyroscope readings with respect to time, you will get the device's angular position (note that you will need to integrate along all three axes of the coordinate system to get a complete view of the device's rotations). The gyroscope's angular velocity is given by:

$$\omega = \frac{d\theta}{dt}$$

And the angular position is given by:

$$\theta = \int_0^t \omega \ dt \cong \sum_0^N \omega \Delta t$$

If you want to use the relation in recursive form, it is given by:

$$\theta_n = \theta_{n-1} + \omega_n \Delta t$$

Combining Readings

In practice, you will combine the gyroscope and accelerometer readings to measure your device's displacement using six degrees of freedom (i.e., three translations measured by the accelerometer and three rotations measured by the gyroscope).

The first application that comes to mind is gaming. For example, let us consider the infamous first person shooter: you could use the gyroscope in order to "aim" with your weapon at various targets. A tap on the screen would fire that weapon, and then jolting the device would reload the weapon.

Summary

This chapter introduced you to the rich world of sensors and their applications in mobile computing. I showed you how to write sensor-aware applications by using the QtMobility module, which is part of the BlackBerry 10 platform. You also saw how easily you could obtain sensor readings in C++ and QML by using the sensor types supported by BlackBerry 10. I emphasized the fact that obtaining those readings is extremely simple and that the real difficulty lies in the data post-processing.

The obvious application of sensors is in game programming by combining the accelerometer and gyroscope. However, as the BlackBerry 10 platform evolves and new sensor types are introduced in the future, the potential applications will grow exponentially. Applications in domains such as personal health management have huge potential. For example, imagine an application using sensors capable of monitoring your heart and stress levels and capable of playing a specific playlist on your device in order to lower your stress.

Sensor-aware applications are a largely untapped market at the moment and this is something you should definitely consider when designing your next BlackBerry 10 killer app.