CHAPTER 15

Working with the Real World

Desktops, laptops, iPhones, and iPads are all physical devices existing in the real world—either on your desk, on your lap, or in your hand. For a long time, your apps were largely confined to your computer, and weren't able to do much with the outside world besides instructing a printer to print a document.

Starting with iOS and OS X 10.6, however, things began to change, and your code is now able to learn about the user's location, how the device is moving and being held, and how far away the computer is from landmarks.

In this chapter, you'll learn about how your programs can interact with the outside world. Specifically, you'll learn how to use Core Location to determine where your computer or device is on the planet, how to use MapKit to show and annotate maps, how to use Core Motion to learn about how the user is holding the device, how to use the printing services available on OS X and iOS to work with printers, how to connect game controllers into your apps, and how to make sure your apps don't excessively drain the user's battery.

Most of the technology discussed in this chapter works on both OS X and iOS. Some of the technologies have identical APIs on both platforms (Core Location, MapKit, and Game Controllers), some have different APIs on the two platforms (print services), and some are only available on iOS (Core Motion) or OS X (App Nap). We'll let you know which technology is available where.

Working with Location

Almost every user of your software will be located on Earth.[1]

1. Unless, of course, you're taking your iPhone to space.

Knowing where the user is on the planet is tremendously useful because it enables you to provide more relevant information. For example, while the business review and social networking site Yelp works just fine as a search engine for businesses and restaurants, it only becomes truly useful when it limits its results to businesses and restaurants near the user.

Location awareness is a technology that is at its most helpful on mobile devices (like an iPhone or iPad), because their locations are more likely to change. However, it's also applicable to a more fixed-location device (like a desktop) to know where it is in the world. A great example of this is the time-zone system in OS X—if you move your computer from one country to another, your time zone will have likely changed, and OS X uses its built-in location systems to work out how to set the clock to local time.

Location Hardware

There are a number of different techniques for determining where a computer is on the planet, and each requires different hardware. The ones in use by iOS and OS X are:

- GPS, the Global Positioning System
- WiFi base station lookups
- Cell tower lookups
- iBeacons

GPS

GPS devices first became popular as navigation assistants for cars, and later as features built into smartphones. Initially developed by the U.S. military, GPS is a constellation of satellites that contain extremely precise clocks and continuously broadcast time information. A GPS receiver can listen for these time signals and compare them to determine where the user is.

Depending on how many satellites the GPS receiver can see, GPS is capable of working out a location to less than one meter of accuracy.

The GPS receiver is only included on the iPhone, and on iPad models that contain 3G or 4G cellular radios. It's not included on any desktop, laptop, or iPod touch, or on WiFi-only iPads.

Since the introduction of the iPhone 4S and later models, iOS devices capable of receiving GPS signals are also capable of receiving signals from GLONASS,[2] a Russian

2. Because you asked, it stands for *GLObalnaya NAvigatsionnaya Sputnikovaya Sistema*.

satellite navigation system, all transparently handled and combined with GPS to give you a better location.

WiFi base station lookups

While a device that uses WiFi to access the Internet may move around a lot, the base stations that provide that connection generally don't move around at all. This fact can be used to determine the location of a user if a GPS receiver isn't available.

Apple maintains a gigantic database of WiFi hotspots, along with rough coordinates that indicate where those hotspots are. If a device can see WiFi hotspots and is also connected to the Internet, it can tell Apple's servers, "I can see hotspots A, B, and C." Apple's servers can then reply, "If you can see them, then you must be near them, and therefore you must be near location X." The device keeps a subset of this database locally, in case a WiFi lookup is necessary when the device has no access to the Internet.

Usually, this method of locating the user isn't terribly precise, but it can get within 100 meters of accuracy in urban areas (where there's lots of WiFi around). Because it uses hardware that's built into all devices that can run OS X and iOS, this capability is available on every device.

Cell tower lookups

If a device uses cell towers to communicate with the Internet, it can perform a similar trick with the towers as with WiFi base stations. The exact same technique is used, although the results are slightly less accurate—because cell towers are less numerous than WiFi stations, cell tower lookups can only get within a kilometer or so of accuracy.

Cell tower lookups are available on any device that includes a cell radio, meaning the iPhone, and all models of the iPad that have a cell radio. They're not available on iPods, because they don't have any cell radio capability.

iBeacons

iBeacons are a new means of determining location using low-energy Bluetooth devices. By constantly broadcasting their existence via a unique identifier, they can be detected by an iOS device, allowing you to determine where you are based on the iBeacon's location. iBeacon location and accuracy is much more subjective than any of the other location methods: instead of pinpointing your position on the planet, iBeacons can tell you when you are near or far from them. iBeacons are designed more to determine the store you're near in a shopping mall or the artwork you're close to in a museum, rather than to work out where you are for navigation or tracking.

iBeacons are available on any device capable of running iOS 7 or later. Additionally, these devices can also be set up to act as an iBeacon themselves.

The Core Location Framework

As you can see, not every piece of location-sensing hardware is available on all devices. Because it would be tremendously painful to have to write three different chunks of code for three different location services and then switch between them depending on hardware availability, OS X and iOS provide a single location services API that handles all the details of working with the location hardware.

Core Location is the framework that your applications use to work out where they are on the planet. Core Location accesses whatever location hardware is available, puts the results together, and lets your code know its best guess for the user's location. It's also able to determine the user's altitude, heading, and speed.

When you work with Core Location, you work with an instance of `CLLocationManager`. This class is your interface to the Core Location framework—you create a manager, optionally provide it with additional information on how you want it to behave (such as how precise you want the location information to be), and then provide it with a delegate object. The location manager will then periodically contact the delegate object and inform it of the user's changing location.

`CLLocationManager` is actually a slightly incomplete name for the class, because it doesn't just provide geographic location information. It also provides heading information (i.e., the direction the user is facing relative to magnetic north or true north). This information is only available on devices that contain a magnetometer, which acts as a digital compass. At the time of writing, all currently shipping iOS devices contain one, but devices older than the iPhone 3GS, iPod touch 3rd generation, and iPad 2 don't.

To work with Core Location, you create the `CLLocationManager` delegate, configure it, and then send it the `startUpdatingLocation()` message. When you're done needing to know about the user's location, you send it the `stopUpdatingLocation()` message.

You should always turn off a `CLLocationManager` when you're done, as location technologies are CPU-intensive and can require the use of power-hungry hardware. Think of the user's battery!

To work with Core Location, you provide it with an object that conforms to the `CLLocationManagerDelegate` protocol. The key method in this protocol is `locationManager(manager: didUpdateLocations:)`, which is sent periodically by the location manager.

This method receives both the user's current location (as far as Core Location can tell) and his previous location. These two locations are represented as `CLLocation` objects, which contain information like the latitude and longitude, altitude, speed, and accuracy.

Core Location may also fail to get the user's location at all. If, for example, GPS is unavailable and neither WiFi base stations nor cell towers can be found, no location information will be available. If this happens, the `CLLocationManager` will send its delegate the `locationManager(manager: didFailWithError:)` message.

Working with Core Location

To demonstrate Core Location, we'll create a simple application that attempts to display the user's location. This will be an OS X application, but the same API applies to iOS.

To get started with Core Location, create a new Cocoa application called Location.

Now that we have a blank application we'll build the interface. The interface for this app will be deliberately simple—it will show the user's latitude and longitude coordinates, as well as the radius of uncertainty that Core Location has about the user.

No location technology is completely precise, so unless you're willing to spend millions of dollars on (probably classified) technology, the best any consumer GPS device will get is about 5 to 10 meters of accuracy. If you're not using GPS, which is the case when using a device that doesn't have it built in, Core Location will use less-accurate technologies like WiFi or cell tower triangulation.

This means that Core Location is always inaccurate to some extent. When Core Location updates its delegate with the location, the latitude and longitude provided are actually the center of a circle that the user is in, and the value of the `CLLocation` property `horizontalAccuracy` indicates the radius of that circle, represented in meters.

The interface of this demo application, therefore, will show the user's location as well as how accurate Core Location says it is:

1. Open *MainMenu.xib* and select the window.
2. Now add the latitude, longitude, and accuracy labels. Drag in three labels and make them read `Latitude`, `Longitude`, and `Accuracy`. Lay them out vertically.
3. Drag in another three labels, and lay them out vertically next to the first three.
4. Finally, drag in a circular progress indicator and place it below the labels.

When you're done, your interface should look like Figure 15-1.

You'll now connect the interface to the app delegate. Open *AppDelegate.swift* in the assistant and Control-drag from each of the labels on the right. Create outlets for each of them called `longitudeLabel`, `latitudeLabel`, and `accuracyLabel`, respectively. Control-drag from the progress indicator, and create an outlet for it called `spinner`.

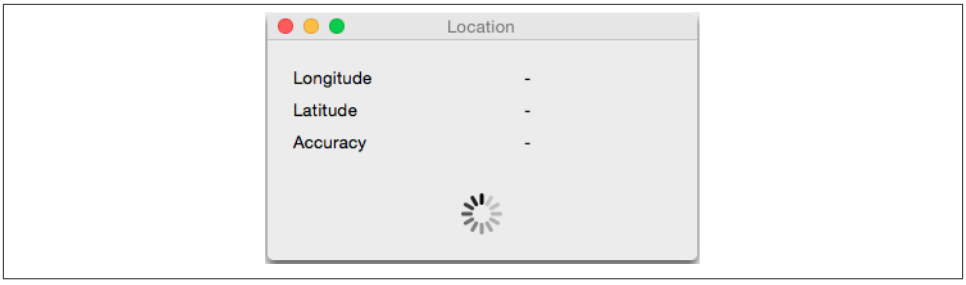

Figure 15-1. The completed interface for our Location app

Now make the app delegate conform to the `CLLocationManagerDelegate` protocol and finally, but very importantly, import the Core Location framework. When you're done, the top of *AppDelegate.swift* should look like the following code:

```
import Cocoa
import CoreLocation

class AppDelegate: NSObject, NSApplicationDelegate,CLLocationManagerDelegate {

    @IBOutlet weak var window: NSWindow!
    @IBOutlet weak var longitudeLabel: NSTextField!
    @IBOutlet weak var latitudeLabel: NSTextField!
    @IBOutlet weak var accuracyLabel: NSTextField!
    @IBOutlet weak var spinner: NSProgressIndicator!

    var locationManager = CLLocationManager()
```

Now we need to tell the `CLLocationManager` to start finding the users location. Update the `applicationDidFinishingLaunching` method in *AppDelegate.swift* to look like the following:

```
    func applicationDidFinishLaunching(aNotification: NSNotification?) {
        self.locationManager.delegate = self
        self.locationManager.startUpdatingLocation()
        self.spinner.startAnimation(nil)
    }
```

This code does the following things:

- Sets the delegate for the `CLLocationManager`; in this case we are setting the delegate to be the app delegate.
- Tells the location manager to start updating the user's location.
- Finally, it instructs the progress indicator to start animating.

Now we need to implement two `CLLocationManagerDelegate` methods—`locationManager(manager: didUpdateLocations:)` and `locationManager(manager: didFailWithError:)`—to handle when we get a location and when we fail to get a location.

Add the following code to the app delegate:

```
func locationManager(manager: CLLocationManager!,
    didUpdateLocations locations: [AnyObject]!)
{
    // collecting the most recent location from the array of locations
    if let newLocation = locations.last as? CLLocation
    {
        self.longitudeLabel.stringValue = NSString(format: "%.2f",
            newLocation.coordinate.longitude)
        self.latitudeLabel.stringValue = NSString(format: "%.2f",
            newLocation.coordinate.latitude)
        self.accuracyLabel.stringValue = NSString(format: "%.1fm",
            newLocation.horizontalAccuracy)
        self.spinner.stopAnimation(nil);
    }
    else
    {
        println("No location found")
    }
}

func locationManager(manager: CLLocationManager!,
    didFailWithError error: NSError!)
{
    // locationManager failed to find a location
    self.longitudeLabel.stringValue = "-"
    self.latitudeLabel.stringValue = "-"
    self.accuracyLabel.stringValue = "-"
    self.spinner.startAnimation(nil)
}
```

These two methods do the following:

- Inside `locationManager(manager: didUpdateLocations:)`, a `CLLocation` object holding all the user's location information is created from the array of locations the location manager found. Then the labels are updated with the relevant information and the spinner is stopped.
- Inside `locationManager(manager: didFailWithError:)`, the labels are updated to show dashes, and the spinner is started again.

It's possible for the location manager to successfully determine the user's location and then later fail (or vice versa). This means that a failure isn't necessarily the end of the line—the location manager will keep trying, so your application should keep this in mind.

Now run the application. On its first run, it will ask the user if it's allowed to access his location. If the user grants permission, the application will attempt to get the user's location. If it can find it, the labels will be updated to show the user's approximate location, and how accurate Core Location thinks it is.

Geocoding

When you get the user's location, Core Location returns a latitude and longitude coordinate pair. This is useful inside an application and great for showing on a map, but isn't terribly helpful for a human being. Nobody looks at the coordinates "-37.813611, 144.963056" and immediately thinks, "Melbourne, Australia."

Because people deal with addresses, which in America are composed of a sequence of decreasingly precise place names[3] ("1 Infinite Loop," "Cupertino," "Santa Clara," "California," etc.), Core Location includes a tool for converting coordinates to addresses and back again. Converting an address to coordinates is called *geocoding*; converting coordinates to an address is called *reverse geocoding*.

Core Location implements this via the `CLGeocoder` class, which allows for both forward and reverse geocoding. Because geocoding requires contacting a server to do the conversion, it will only work when an Internet connection is available.

To geocode an address, you create a `CLGeocoder` and then use one of its built-in geocoding methods. You can provide either a string that contains an address (like "1 Infinite Loop Cupertino California USA") and the geocoder will attempt to figure out where you mean, or you can provide a dictionary that contains more precisely delineated information. Optionally, you can restrict a geocode to a specific region (to prevent confusion between, say, Hobart, Minnesota and Hobart, Tasmania).

We're going to add reverse geocoding to the application, which will show the user her current address. To do this, we'll add a `CLGeocoder` to the `AppDelegate` class. When Core Location gets a fix on the user's location, we'll ask the geocoder to perform a reverse geocode with the `CLLocation` provided by Core Location.

When you reverse geocode, you receive an array that contains a number of `CLPlacemark` objects. An array is used because it's possible for the reverse geocode to return with a number of possible coordinates that your address may resolve to. `CLPlacemark` objects contain a number of properties that contain address information. Note that not all of the properties may contain information; for example, if you reverse geocode a location that's in the middle of a desert, you probably won't receive any street information.

The available properties you can access include:

3. This is the case in most Western locales, but isn't the case everywhere on the planet. Japanese addresses, for example, often go from widest area to smallest area.

- The name of the location (e.g., “Apple Inc.”)
- The street address (e.g., “1 Infinite Loop”)
- The locality (e.g., “Cupertino”)
- The sublocality—that is, the neighborhood or name for that area (e.g., “Mission District”)
- The administrative area—that is, the state name or other main subdivision of a country (e.g., “California”)
- The subadministrative area—that is, the county (e.g., “Santa Clara”)
- The postal code (e.g., “95014”)
- The two- or three-letter ISO country code for that country (e.g., “US”)
- The country name (e.g., “United States”)

Some placemarks may contain additional data, if relevant:

- The name of the inland body of water that the placemark is located at or very near to (e.g., “Derwent River”)
- The name of the ocean where the placemark is located (e.g., “Pacific Ocean”)
- An array containing any additional areas of interest (e.g., “Golden Gate Park”)

You can use this information to create a string that can be shown to the user.

We’ll start by creating a label that will display the user’s address.

Open *MainMenu.xib* and drag in a new label under the current set of labels. The updated interface should look like Figure 15-2. Then open *AppDelegate.swift* in the assistant and Control-drag from the new label into the app delegate. Create a new outlet for the label called `addressLabel`.

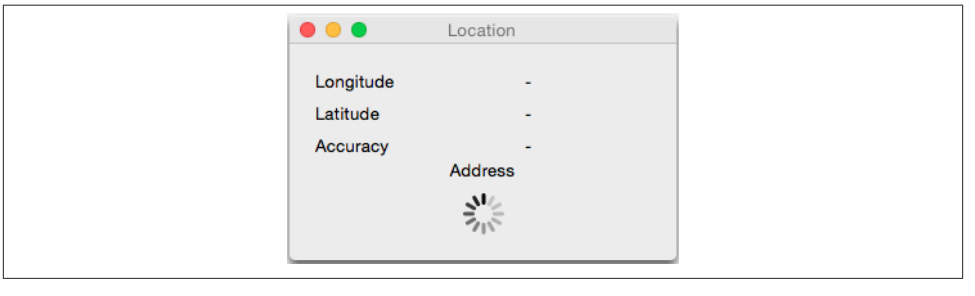

Figure 15-2. The completed interface for our Location app with geocoding

Then add `CLGeocoder` to the app delegate by updating *AppDelegate.swift* to have the following property:

```
@IBOutlet weak var addressLabel: NSTextField!
var geocoder = CLGeocoder()
```

When the user's location is determined, perform a reverse geocode by updating the `locationManager(manager: didUpdateLocations:)` method with the following code:

```
func locationManager(manager: CLLocationManager!,
    didUpdateLocations locations: [AnyObject]!)
{
    // collecting the most recent location from the array of locations
    if let newLocation = locations.last as? CLLocation
    {
        self.longitudeLabel.stringValue = NSString(format: "%.2f",
            newLocation.coordinate.longitude)
        self.latitudeLabel.stringValue = NSString(format: "%.2f",
            newLocation.coordinate.latitude)
        self.accuracyLabel.stringValue = NSString(format: "%.1fm",
            newLocation.horizontalAccuracy)
        self.spinner.stopAnimation(nil);

        self.geocoder.reverseGeocodeLocation(newLocation)
        {
            (placemarks, error) in
            if error == nil
            {
                let placemark = placemarks[0] as CLPlacemark
                let address = NSString(format: "%@ %@, %@, %@ %@",
                    placemark.subThoroughfare,
                    placemark.thoroughfare,
                    placemark.locality,
                    placemark.administrativeArea,
                    placemark.country)

                self.addressLabel.stringValue = address
            }
            else
            {
                // failed to reverse geocode the address
                self.addressLabel.stringValue = "Failed to find an address"
            }
        }
    }
    else
    {
        NSLog("No location found")
    }
}
```

Now run the application. Shortly after the user's location is displayed, the approximate address of your location will appear. If it doesn't, check to make sure that you're connected to the Internet.

Region Monitoring and iBeacons

Depending on what your app's goals are, it might be more useful to know when your user enters an area as opposed to knowing their precise location. To help with this, Core Location provides region monitoring.

There are two types of region monitors: geographical and iBeacon. Region monitoring lets you set up virtual boundaries around the Earth and be informed via delegate callbacks when the user enters or exits one of the regions; iBeacon monitoring lets your app be informed when a user is near an iBeacon region represented by a low energy Bluetooth signal.

Currently iBeacons are only available on iOS. Sorry, Mac developers.

Monitoring when the user enters and exits a geographical region is straightforward. If you wanted to add region monitoring to our existing location app, all you'd need to do is add the following to the bottom of the `applicationDidFinishLaunching` method:

```
let location = CLLocationCoordinate2DMake(-42.883317, 147.328277)
let region = CLCircularRegion(  center: location,
                                radius: 1000,
                            identifier: "Hobart")
locationManager.startMonitoringForRegion(region)
```

This does several things: first, it creates a location to use as the center of the region (in this case, the city of Hobart, in Australia). It then creates a region around this center point with a radius of 1,000 meters, and gives it an identifier to help you differentiate it later. Finally, it tells the location manager to start monitoring whether a user has entered or exited a region.

The way `CLRegion` works on OS X is a little bit different than on iOS. `CLRegion` is an abstract class in iOS that is meant to be subclassed for the specific different types of regions. So to get the same functionality in iOS, we would instead use the `CLRegion` subclass `CLCircularRegion`, the region subclass designed for monitoring circular regions.

To know when a user has entered or exited a region, there are two delegate methods: `locationManager(manager: didExitRegion:)` and `locationManager(manager: didEnterRegion:)`. Both of these callbacks pass in a `CLRegion` with a string identifier property that you can use to determine the region that the user has entered or exited:

```
func locationManager(manager: CLLocationManager!, didEnterRegion
    region: CLRegion!) {
    // have entered the region
    NSLog("Entered %@", region.identifier)
}
func locationManager(manager: CLLocationManager!, didExitRegion
    region: CLRegion!) {
    // have exited the region
    NSLog("Exited %@", region.identifier)
}
```

iBeacon regions function a little bit differently when working with circular regions. Because of the nature of Bluetooth radio devices, you will never be able to guarantee a radius for each beacon. Additionally, a device might encounter multiple beacons within a single area that might not all be relevant to your app. Therefore, a radius and an identifier are not enough to be able to use iBeacons. The principle is the same though —you create a region to represent the beacon and you tell your `CLLocationManager` to start looking for that region. Say you wanted to create an iBeacon region for a particular painting in a gallery:

```
// the UUID string was generated using the uuidgen command
let uuid = NSUUID(UUIDString:"F7769B0E-BF97-4485-B63E-8CE121988EAF")

let beaconRegion = CLBeaconRegion(proximityUUID: uuid,
    major: 1,
    minor: 2,
    identifier: "Awesome painting");
```

This code does several things. First, it creates a `CLBeaconRegion` with both an identifier string that works exactly the same as it does in the circular region, and a proximity UUID. The proximity UUID is a unique identifier to be used by all the beacons in your app. In the case of a gallery, the UUID would be the same for all iBeacons in the gallery or any other gallery that your app works in. The major and minor properties are numbers that can be used to help identify exactly which beacon the user is near. In the case of a gallery, the major property could represent a section of the museum and the minor property a particular artwork.

The UUID and major and minor properties of the beacon region must match the settings you put into your actual hardware iBeacons. How to set these will change from iBeacon to iBeacon.

Just like with geographical regions, your app gets delegate callbacks for any beacon regions that your app enounters. Unlike geographical regions, however, you have a couple of options when using iBeacons; your app can receive the delegate callbacks for circular regions `locationManager(manager: didEnterRegion:)` and `locationManager(manager: didExitRegion:)` by calling the standard `startMonitoringForRegion(region:)` on your `CLLocationManager`. Alternatively, you can call `startRangingBeaconsInRegion(region:)` on your location manager. This will result in the delegate method `locationManager(manager: didRangeBeacons: inRegion:)` being called, with all the beacons in range being passed in as an array of `CLBeacon` objects as well as the region object that was used to locate the beacons. The closest beacon will have a proximity property equal to the system-defined `CLProximityNear` value.

When testing your iBeacon apps, you need an iBeacon to make sure that your app is doing what it needs to do when it finds a beacon. Buying lots of iBeacons just for testing is going to be a bit expensive, but Apple has thankfully provided a way to turn your iOS devices into iBeacons. It is quite easy to do this, but first you need to add a `CBPeripheralManager`, an object for managing Bluetooth peripherals, to your app and make your app conform to the `CBPeripheralDelegate` protocol:

```
import CoreLocation
import CoreBluetooth

class ViewController: UIViewController,CBPeripheralManagerDelegate{
    var bluetoothManager : CBPeripheralManager?

    override func viewDidLoad() {
        super.viewDidLoad()

        self.bluetoothManager = CBPeripheralManager(delegate: self, queue: nil)
    }
```

Then to create an iBeacon for a specific region, you need to create a `CLBeaconRegion` that matches the beacon you want to simulate. Finally you need to tell the Bluetooth manager to start advertising the iBeacon:

```
    func peripheralManagerDidUpdateState(peripheral: CBPeripheralManager!) {

        if (self.bluetoothManager?.state == CBPeripheralManagerState.PoweredOn)
        {
            // the UUID string was generated using the uuidgen command
            let uuid = NSUUID(UUIDString:"F7769B0E-BF97-4485-B63E-8CE121988EAF")

            let beaconRegion = CLBeaconRegion(proximityUUID: uuid,
                major: 1,
                minor: 2,
                identifier: "Awesome painting");

            var beaconData = beaconRegion.peripheralDataWithMeasuredPower(nil)
            self.bluetoothManager?.startAdvertising(beaconData)
```

```
    }
}
```

Locations and Privacy

The user's location is private information, and your application must be granted explicit permission by the user on at least the first occasion that it tries to access it.

People are understandably wary about software knowing where they are—it's a privacy issue and potentially a safety risk. To avoid problems, follow these guidelines in your application:

- Be very clear about what your application is using location information for.
- Never share the user's location with a server or other users unless the user has explicitly given permission.
- The user can always see when an application is accessing location information because a small icon appears at the top of the screen (on both iOS and OS X). Once your app has performed the task that location information is needed for, turn off Core Location—both to save power and to let the user know that they are no longer being tracked.

Maps

The MapKit framework provides an everything-but-the-kitchen-sink approach to maps, allowing you to create, annotate, overlay, and adjust maps as needed for your applications. It plays very well with the location tracking and region monitoring technology that Apple provides.

Previously, MapKit was only available on iOS, but since the release of OS X 10.9, MapKit is now available on OS X as well and the framework functions in virtually the same way on both platforms.

The base of all maps is the `MKMapView`, which is the actual view containing map data that your application can use. The map data that the map view displays comes from Apple Maps data, although in the past the view used Google Maps data.

Using Maps

Getting a map up and running is quite straightforward, so let's get cracking!

1. Create a new, single view iPhone application and call it Maps.
2. Next, create the interface. Open the *Main.storyboard* file. Drag in a map view and make it take up the entire view.

3. Then connect the interface. Open *ViewController.swift* in the assistant. Control-drag from the map view to the view controller and make a new outlet for the map view. Call it `mapView`.
4. You then need to extend the view controller. Open *ViewController.swift*, import the MapKit framework, and set the controller to be a map view delegate. Finally, add the MapKit framework into the project.

   When you are done, *ViewController.swift* should look like this:

```
import MapKit

class ViewController: UIViewController,MKMapViewDelegate {

    @IBOutlet weak var mapView: MKMapView!
```

5. Finally, configure the map. By default, the map view will be centered over Apple's headquarters in Cupertino, so we want to move that somewhere a bit different. Update `viewDidLoad()` to look like this:

```
    override func viewDidLoad() {
        super.viewDidLoad()

        self.mapView.delegate = self;
        let center = CLLocationCoordinate2DMake(-37.813611, 144.963056)
        let span = MKCoordinateSpanMake(2, 2);
        self.mapView.region = MKCoordinateRegionMake(center, span)
```

Now if you run the app, your map should be centered over Melbourne. The span is how many degrees of longitudinal and latitudinal delta the view should cover.

Annotating Maps

Just having a map all by itself isn't a lot of fun. What if we want to see where something interesting is? Luckily for us, Apple has included an easy way of annotating the map, which you may have seen in the built-in Maps app as little red, green, and purple pins. These additions to the map are called *annotations*.

The annotations are broken up into two object types: `MKAnnotation`, which represents the annotation data, including its location and any other data such as a name; and `MKAnnotationView`, which will be the view that the map displays when needed.

Apple has provided a built-in annotation class for when all you need is a simple pin called `MKPointAnnotation`. This class has built-in properties for a title and subtitle, and will automatically draw a pin view when needed, making it very easy to use. MapKit on both OS X and iOS supports custom annotation and custom views for the annotations, but for this example we will stick to the built-in pin annotations.

So let's start dropping those little pins. Open *ViewController.swift* and add the following to the bottom of the `viewDidLoad` method:

```
// creating an new annotation
var annotation = MKPointAnnotation()
annotation.coordinate = center
annotation.title = "Melbourne"
annotation.subtitle = "Victoria"
// adding the annotation to the map
self.mapView.addAnnotation(annotation);
```

This code creates a new point annotation at Melbourne, sets a title and subtitle string, and then finally adds it to the map. If you run the app, a red pin will be on top of Melbourne and if you tap it, you'll get a little window with the title and subtitle strings.

Maps and Overlays

Annotations are good and all, but sometimes you need a bit more than what a pin can provide. This is where overlays come in handy. Unlike an annotation, they don't represent a single point but instead cover an area on the map.

Overlays, much like annotations, are broken up into two parts: an `MKOverlay` object representing the data, and a view object called `MKOverlayRender`. Also much like annotations, Apple has provided some prebuilt overlays to make life simpler, including overlays and associated renderers for circles, lines, and polygons—and if you want, you can build your own.

With that understood, let's add a 100 km circle around the pin we dropped before. Add the following to the bottom of the `viewDidLoad()` method:

```
// creating and adding a new circle overlay to the map
var overlay = MKCircle(centerCoordinate: center, radius: 50000)
self.mapView.addOverlay(overlay)
```

This creates a circular region with a radius of 50 km around the same location where we added the pin, and then it adds the overlay to the map. Nothing will be displayed, though, until the map view's delegate provides an appropriate renderer for that overlay.

Add the following map view delegate method to *ViewController.swift*:

```
func mapView(
    mapView: MKMapView!, rendererForOverlay
    overlay: MKOverlay!) -> MKOverlayRenderer! {
    if (overlay.isKindOfClass(MKCircle))
    {
        var circleRenderer = MKCircleRenderer(overlay: overlay)
        circleRenderer.strokeColor = UIColor.greenColor()
        circleRenderer.fillColor = UIColor(
            red: 0,
            green: 1.0,
            blue: 0,
            alpha: 0.5)

        return circleRenderer
    }
    return nil
}
```

Terra, as you might know, is a sphere, and your screen is a rectangle, so there are going to be some issues when trying to squish the planet into a rectangle. Apple has used what is known as the Mercator projection to overcome this. The Mercator projection is just one of many different projections, and they all have their own strengths and weaknesses.

One of the side effects of the Mercator projection is if you add a lot of overlays spread all over the place, they might look different if you zoom the map all the way out than they do up close. Thankfully, most of the time you are using maps, you will likely be zoomed close enough in that you won't notice this.

Now if you run the app, you should see a rather large green circle hovering ominously over Melbourne, as seen in Figure 15-3.

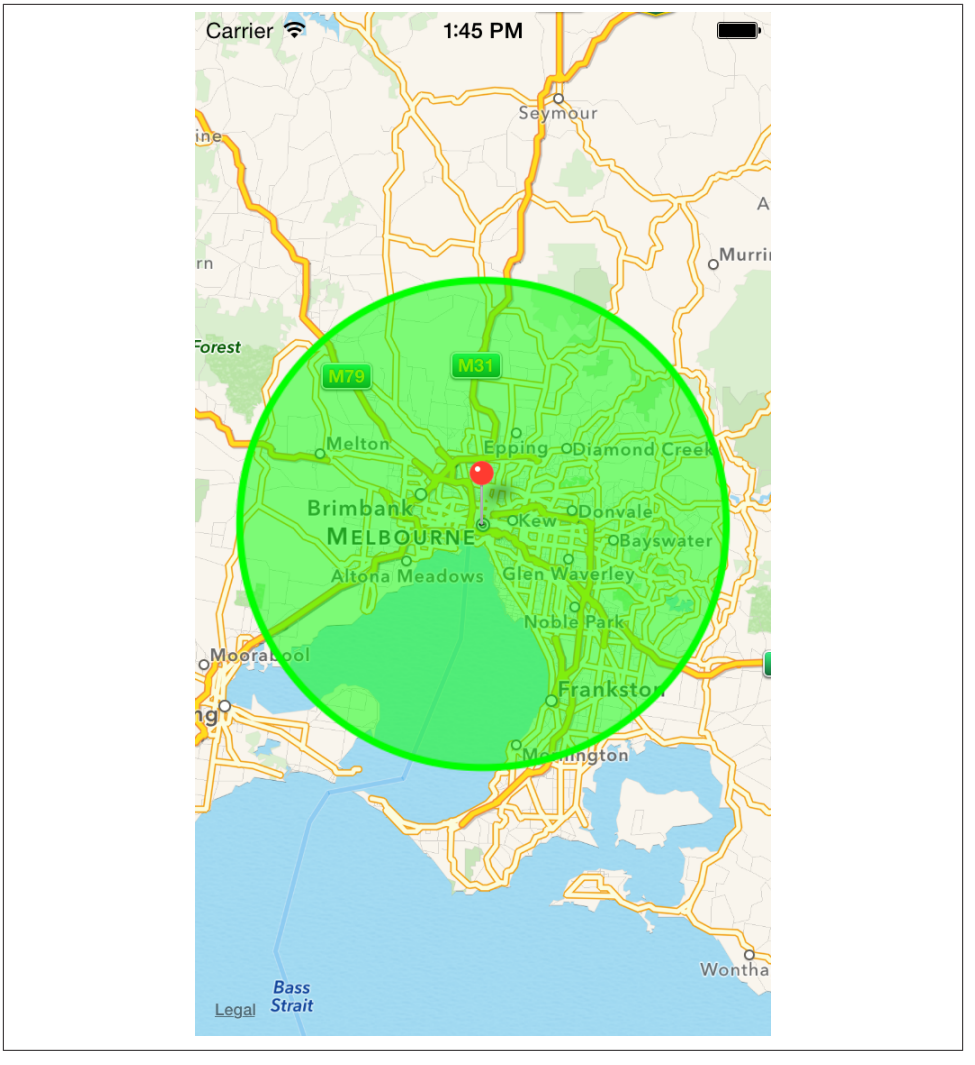

Figure 15-3. The map, with an overlay drawn over it

Device Motion

An iOS device is often held in the user's hands, which means that it's subject to movement and rotation. iOS allows your code to receive information about how the device is moving, how it's being rotated, and how it's oriented. All of this information is available through the Core Motion framework, which provides a unified interface to the various relevant sensors built into the device.

Core Motion is only available on iOS, as laptops and desktops don't generally get shaken around while being used. While some Mac laptops include an accelerometer, it isn't available to your code through any publicly accessible APIs.

Core Motion provides *device motion* information to your application by measuring data from the sensors available to the device:

- The accelerometer, which measures forces applied to the device
- The gyroscope, which measures rotation
- The magnetometer, which measures magnetic fields

The first iOS devices only included an accelerometer, which is actually sufficient for getting quite a lot of information about device motion. For example, based on the forces being applied to the device over time, you can determine the direction of gravity, which gives you information about how the device is being held, as well as determine if the device is being shaken. If a gyroscope is available, you can refine this information and determine if the device is rotating around an axis of gravity. If a magnetometer is available, it's possible to determine a frame of reference, so that it can determine which way north is—something that's not possible when all you have is relative rotation information.

Core Motion collects data from all the available sensors and provides direct access to the sensor information. You can therefore ask Core Motion to give you the angles that define the device's orientation in space, as well as get raw accelerometer information.

Raw sensor data can lead to some very cool tricks. For example, the magnetometer measures magnetic fields, which means that the device can actually be used as a metal detector by measuring changes in the magnetic field.

Working with Core Motion

Core Motion works in a very similar manner to Core Location: it uses a manager object that provides periodic updates on device motion. However, the means by which the manager object provides these updates differs from how `CLLocationManager` does—instead of providing a delegate object, you instruct the motion manager to call a block that you provide. In this block, you handle the movement event.

The iOS Simulator doesn't simulate any of the motion hardware that Core Motion uses. If you want to test Core Motion, you need to use a real iOS device.

The motion manager class is called `CMMotionManager`. To start getting motion information, you create an instance of this class and instruct it to begin generating motion updates. You can also optionally ask for only accelerometer, gyroscope, or magnetometer information.

You can also get information from the `CMMotionManager` by querying it at any time. If your application doesn't need to get information about device motion very often, it's more efficient to simply ask for the information when it's needed. To do this, you send the `CMMotionManager` object the `startDeviceMotionUpdates()` method (or the `startAccelerometerUpdates()` or `startGyroUpdates()` methods), and then, when you need the data, you access the `CMMotionManager`'s `accelerometerData`, `gyroData`, or `deviceMotion` properties.

The fewer devices that Core Motion needs to activate in order to give you the information you need, the more power is saved. As always, consider the user's battery!

Core Motion separates out "user motion" from the sum total of forces being applied to the device. There's still only one accelerometer in there, though, so what Core Motion does is use a combination of low- and high-pass filtering to separate out parts of the signal, with the assistance of the gyroscope, to determine which forces are gravity and which forces are "user motion"—forces like shaking or throwing your device. (Note: The authors do not recommend throwing your device, no matter how much fun it is. The authors specifically do not recommend making an awesome app that takes a photo at the peak of a throw. You will break your phone.)

You can also configure how often the `CMMotionManager` updates the accelerometer and gyro—the less often it uses it, the more power you save (and, as a trade-off, the more imprecise your measurements become).

To work with Core Motion, you'll need to add the Core Motion framework to your project. We'll now build a small iPhone app that reports on how it's being moved and how the device is oriented:

1. Create a new, single view iPhone application and call it Motion.

2. Add the Core Motion framework. Select the project at the top of the project navigator. The project settings will appear in the main editor; select the `Motion` target.

   Scroll down to Linked Frameworks and Libraries, and click the + button.

   The frameworks sheet will appear; browse or search for `CoreMotion.framework` and add it to the project.

3. Once the framework has been added, we'll begin building the interface for the app. This will be similar to the Core Location app: it will report on the numbers being sent by Core Motion. However, you can (and should!) use the information for all kinds of things—game controls, data logging, and more. The authors once used Core Motion to build an app that tracks human sleeping activity. It's a tremendously flexible framework.

   Open *Main.storyboard*. First, we'll create the labels that display the user motion.

   Drag in three labels and lay them out vertically on the lefthand side of the screen. Make their text `X`, `Y`, and `Z`, respectively.

   Drag in another three labels and lay them out vertically to the right of the first three. Next, we'll create the labels that display orientation.

   Drag in another three labels and lay them out vertically on the lefthand side of the screen, under the existing set of labels. Make their text `Pitch`, `Yaw`, and `Roll`, respectively.

   Drag in a final set of three labels and lay them out vertically and to the right.

   Once you're done, your interface should look like Figure 15-4.

4. Connect the interface to the view controller. Open *ViewController.swift* in the assistant.

   Control-drag from each of the labels on the right and create outlets for each of them. From top to bottom, the labels should be called `xLabel`, `yLabel`, `zLabel`, `pitchLabel`, `yawLabel`, and `rollLabel`.

   While you have *ViewController.swift* open, import `CoreMotion`.

Figure 15-4. The interface for the accelerometer application

When you're done, the top of *ViewController.swift* should look like the following:

```
import UIKit
import CoreMotion

class ViewController: UIViewController {

    @IBOutlet weak var xLabel: UILabel!
    @IBOutlet weak var yLabel: UILabel!
    @IBOutlet weak var zLabel: UILabel!
    @IBOutlet weak var pitchLabel: UILabel!
    @IBOutlet weak var yawLabel: UILabel!
    @IBOutlet weak var rollLabel: UILabel!
```

5. Now that the view controller's header file has been set up, we'll write the code that actually creates the motion manager and then starts updating the labels as device motion updates arrive.

We'll store a reference to the `CMMotionManager` as an instance variable in the class, and start and stop the device motion updates when the view controller appears and disappears.

Update *ViewController.swift* so that it looks like the following code:

```
override func viewDidAppear(animated: Bool) {
    self.motionManager.startDeviceMotionUpdatesToQueue(
        NSOperationQueue.mainQueue(),
        withHandler: { (motion: CMDeviceMotion!, error: NSError!) -> Void in
          let xString = NSString(format:"%.1f", motion.userAcceleration.x)
          let yString = NSString(format:"%.1f", motion.userAcceleration.y)
          let zString = NSString(format:"%.1f", motion.userAcceleration.z)

          self.xLabel.text = xString
          self.yLabel.text = yString
          self.zLabel.text = zString

          //convert the pitch, yaw and roll to degrees
          let pitchDegrees = motion.attitude.pitch * 180 / M_PI
          let yawDegress = motion.attitude.yaw * 180 / M_PI
          let rollDegress = motion.attitude.roll * 180 / M_PI

          let pitchString = NSString(format: "%.1f", pitchDegrees)
          let yawString = NSString(format: "%.1f", yawDegress)
          let rollString = NSString(format: "%.1f", rollDegress)

          self.pitchLabel.text = pitchString
          self.yawLabel.text = yawString
          self.rollLabel.text = rollString
    })
}
```

Finally, connect an iOS device and run the application on it. Watch the numbers on the screen change as you shake and rotate the device.

Using the Built-in Altimeter

Some iOS devices have a built-in barometer, which can be used to get information about the current atmospheric pressure. This data can be analyzed to work out *relative altitude* (i.e., the change in altitude since recording began).

Not all devices have the hardware needed to use the altimeter. To find out, you ask the `CMAltimeter` class whether altitude data is available, using the `isRelativeAltitudeAvailable` method:

```
if CMAltimeter.isRelativeAltitudeAvailable() {
    println("Altimeter is available")
} else {
    println("Altimeter is not available")
}
```

To use the altimeter, you create an instance of the `CMAltimeter` class:

```
let altimeter = CMAltimeter()
var currentAltitude : Float = 0.0
```

Once that's done, you ask it to begin sending you information about changes in the user's altitude:

```
    let mainQueue = NSOperationQueue.mainQueue()
altimeter.startRelativeAltitudeUpdatesToQueue(mainQueue) { (data, error) in

        // data.relativeAltitude is the change in
        // altitude since the last time this closure
        // ran, measured in meters.

        // For example, you can keep track of the total
        // change in altitude, relative to where we started:
        self.currentAltitude += Float(data.relativeAltitude)

        self.altitudeLabel.text = "\(self.currentAltitude)m"
    }
```

Using the Pedometer

A pedometer is a piece of hardware that keeps track of the number of steps the user has walked or run. Your apps can access live updates from the pedometer (i.e., they can be updated in near real time as the user moves around) or they can ask the system for information about how the user has moved around over the last seven days.

Before you can access the pedometer, you need to first check to see if pedometer information is available:

```
if CMPedometer.isStepCountingAvailable() {
    println("Pedometer is available")
} else {
    println("Pedometer is not available")
}
```

If the pedometer is available, you create an instance of the `CMPedometer` class:

```
let pedometer = CMPedometer()
```

To query the system to find information about how many steps the user has taken over a date range—for example, in the last day—you use the `queryPedometerDataFromDate(_,toDate:)` method, and provide the date range as well as a handler closure that runs when data is available:

```
// Create NSDate objects that refer to 1. right now
// and 2. one day ago
let calendar = NSCalendar.currentCalendar()
let now = NSDate()
let oneDayAgo = calendar.dateByAddingUnit(NSCalendarUnit.DayCalendarUnit,
```

```
                                                value: -1,
                                                toDate: now,
                                                options: NSCalendarOptions())

        // Ask the pedometer to give us info about that date range
        pedometer.queryPedometerDataFromDate(oneDayAgo, toDate: now) {
            (pedometerData, error) in

            // This closure is called on a background queue,
            // so run any changes to the GUI on the main queue
            NSOperationQueue.mainQueue().addOperationWithBlock() {
                if let data = pedometerData {

                    self.stepsSinceLastDayLabel.text =
                        "\(data.numberOfSteps) steps"

                }
            }
        }
```

If you want to receive live updates on the user's step count, use the `startPedometerUpdatesFromDate` method instead:

```
        // Start getting live updates on step count, starting from now
        pedometer.startPedometerUpdatesFromDate(now) { (pedometerData, error)  in

            if let data = pedometerData {

                // Run the update on the background
                NSOperationQueue.mainQueue().addOperationWithBlock() {

                    self.stepsSinceAppStartLabel.text =
                        "\(data.numberOfSteps) steps"

                }
            }
        }
```

Printing Documents

Despite decades of promises, the "paperless office" has never really materialized. Users like having documents on paper, and both OS X and iOS provide ways of getting stuff printed on dead trees.

The APIs and methods for printing on OS X and iOS are completely different. On OS X, individual `NSViews` are printed, either directly or via an intermediary system. On iOS, you print via a separate system of print renderer and formatter objects.

We'll build two demo apps that show how to print documents on OS X and iOS.