9

Introduction to Augmented Reality

The latest generation of smart phone platforms, including the iPhone, has allowed augmented reality to step out of the Computer Science laboratory and into the consumer market place for the first time. Augmented Reality has become one of the killer applications for the iPhone platform.

Location Aware Augmented Reality

One of the first augmented reality applications to go live in the App Store was acrossair's Nearest Tube application (see Figure 9-1).

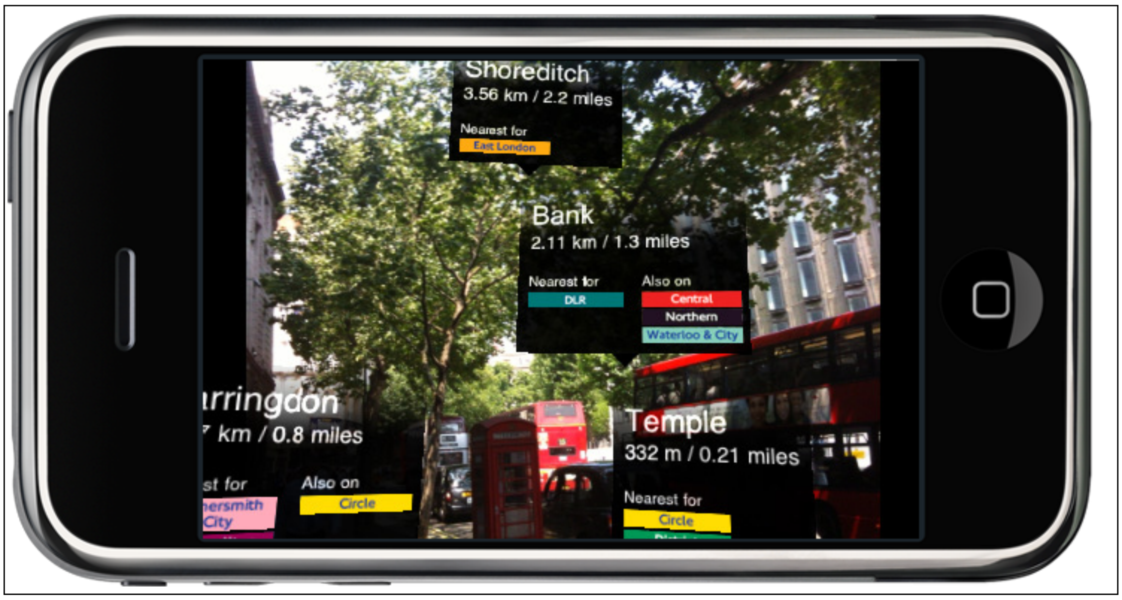

Figure 9-1. The Nearest Tube application

The Nearest Tube application typifies location-aware augmented reality, where objects are injected into the real-world view based their location relative to your own position, as opposed to marker-based augmented reality where the real-world view is interpreted in real or near to real time and objects are placed in the view based on markers or other characteristics of the image. We'll talk about marker based augmented reality in the next chapter.

Building an AR Toolkit

> At the time of writing the only iPhone, iPod touch or iPad device that had and accelerometer, magnetometer (digital compass) and a rear-facing camera was the iPhone 3GS. The code in this chapter is therefore only applicable to this device.

In this chapter we're going to go ahead and build a simple location-aware AR toolkit. When we're done we'll have a number of classes that you can extend and reuse in your own projects.

> The code in this chapter is based previous work by Zac White (http://www.zacwhite.com/) and Niels Hansen (http://www.agilitesoftware.com/). A fully working Xcode project containing the code from this chapter is available on the website supporting the book at http://programmingiphonesensors.com/code/.

Open Xcode and start a new project, choose a Window-based Application template for the iPhone from the new Project Window and name it **`ARView`** when prompted. Expand the groups in the Groups & Files pane in Xcode and right click on the Frameworks group and select Add→Existing Frameworks... from the popup menu. We'll need to make use of the Core Location framework, so choose *CoreLocation.framework* in the drop-down menu when it presents itself to add it to your project.

Before we plunge headlong into the code lets think about the architecture of the application we're going to build, see Figure 9-2. We want to make it reusable, so we're not going to tie it to a specific view controller. Instead we'll sub-class `NSObject` and create an `ARController` class that will present the `UIImagePickerController` modally and manage and update the augmented reality overlay view. The overlay will consist of a number of `ARMarker` objects, which are representations of an `ARCoordinate` object. For convenience we'll also create an `ARGeoCoordinate` class, which will be a sub-class of an `ARCoordinate` that will allow us to create `ARCoordinate` objects conveniently using `CLLocation` objects.

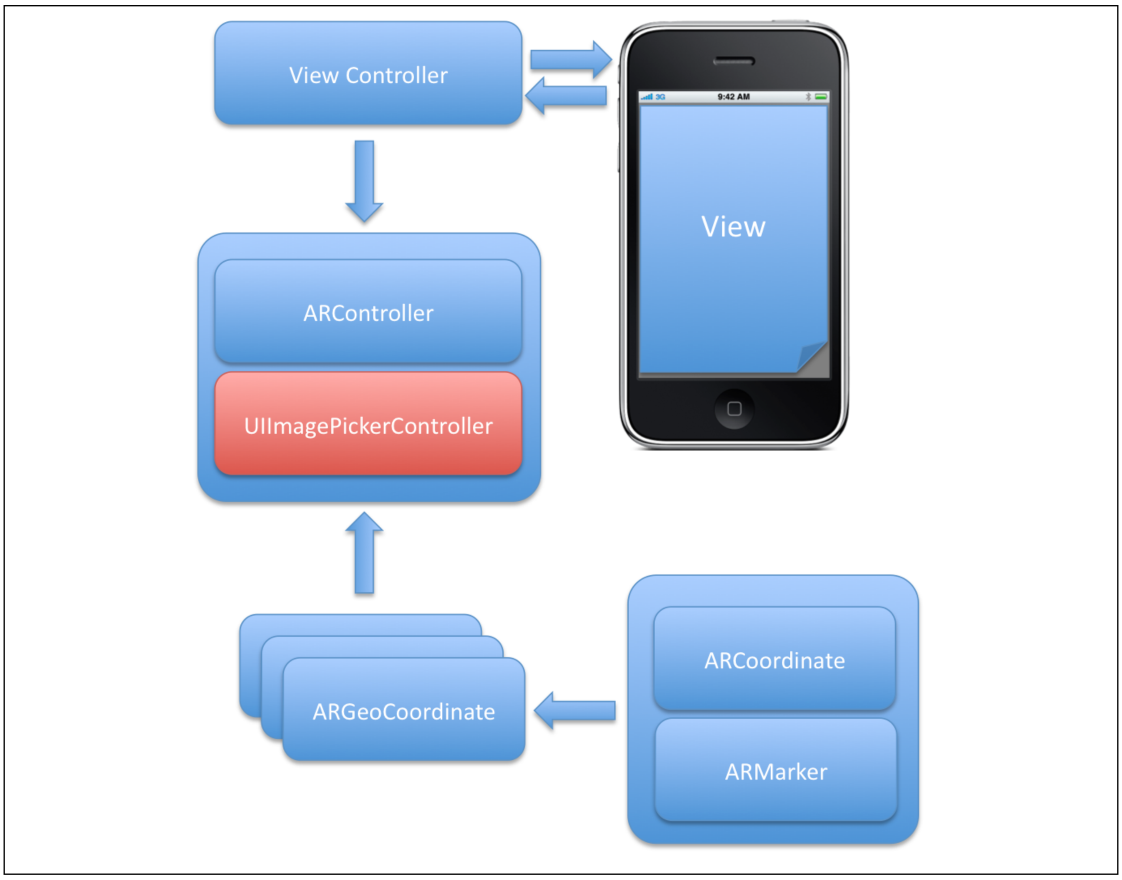

Figure 9-2. The architecture of the ARView application

With this architecture we can create an instance of the `ARController` class from a normal view controller object that may be controlling other views in our application.

Before we get started go ahead and add the following definitions to the *ARView_Prefix.pch*,

```
#define degreesToRadians(x) (M_PI * (x) / 180.0)
#define radiansToDegrees(x) ((x) * 180.0/M_PI)
```

to make these methods available throughout our code.

Let's start with the skeleton of the `ARController` class. In the Groups & File pane of the Xcode interface right-click on the Classes group and choose Add→New Group from the popup menu. Name the group **`ARToolkit`**. Now right-click on the ARToolkit group and choose Add→New File... from the menu. Choose to create an Objective-C class, which is a subclass of `NSObject`. Name the new class `ARController` when prompted.

Click on the *ARController.h* interface file to open it in the Xcode editor and modify it as below. We're going to need instance variables for the calling view controller, the `UIImagePickerController` that we're using to present the camera view, the `UIView` we're overlaying on the camera view as well as the `CLLocationManager` and `UIAccelerometer` instances.

```
#import <Foundation/Foundation.h>
#import <CoreLocation/CoreLocation.h>
#import <UIKit/UIKit.h>

@interface ARController : NSObject <UIAccelerometerDelegate,
CLLocationManagerDelegate>  {!!CO1!!

    UIViewController *rootController;
    UIImagePickerController *pickerController;
    UIView *overlayView;
    CLLocationManager *locationManager;
    UIAccelerometer*accelerometer;
}

@property (nonatomic, retain) UIViewController *rootController;
@property (nonatomic, retain) UIImagePickerController *pickerController;
@property (nonatomic, retain) UIView *overlayView;

@property (nonatomic, retain) CLLocationManager *locationManager;
@property (nonatomic, retain) UIAccelerometer *accelerometer;

- (id)initWithViewController:(UIViewController *)theView;!!CO2!!

@end
```

1. Our ARController class will be both a UIAccelerometer and CLLocationManagerDelegate class.
2. We're going to go ahead and override the init: method for the object and pass the current (calling) view controller into our ARController object during its initialization.

Save your changes and click on the *ARController.m* implementation file to open it in the Xcode editor, we need to go ahead and synthesize our properties and implement our `initWithViewController:` method.

```
#import "ARController.h"

@implementation ARController

@synthesize rootController;
@synthesize pickerController;
@synthesize overlayView;
@synthesize locationManager;
@synthesize accelerometer;

- (id)initWithViewController:(UIViewController *)theView {

    // Initialise the root and overlay views!!CO1!!
    self.rootController = theView;
    CGRect screenBounds = [[UIScreen mainScreen] bounds];
    self.overlayView = [[UIView alloc] initWithFrame: screenBounds];
    self.rootController.view = overlayView;

    // Initialise the UIImagePickerController!!CO2!!
    self.pickerController = [[[UIImagePickerController alloc] init] autorelease];
    self.pickerController.sourceType = UIImagePickerControllerSourceTypeCamera;
    self.pickerController.cameraViewTransform = CGAffineTransformScale(
                 self.pickerController.cameraViewTransform, 1.13f,  1.13f);

    self.pickerController.showsCameraControls = NO;
    self.pickerController.navigationBarHidden = YES;
    self.pickerController.cameraOverlayView = overlayView;

    // Initialise the CLLocationManager!!CO3!!
    self.locationManager = [[CLLocationManager alloc] init];
    self.locationManager.headingFilter = kCLHeadingFilterNone;
    self.locationManager.desiredAccuracy = kCLLocationAccuracyBest;
    self.locationManager.delegate = self;
    [self.locationManager startUpdatingHeading];
    [self.locationManager startUpdatingLocation];

    // Initalise the UIAccelerometer!!CO4!!
    self.accelerometer = [UIAccelerometer sharedAccelerometer];
    self.accelerometer.updateInterval = 0.25;
    self.accelerometer.delegate = self;

    // Listen for changes in device orientation!!CO5!!
    [[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(deviceOrientationDidChange:) name:
UIDeviceOrientationDidChangeNotification object:nil];
```

```
    [[UIDevice currentDevice] beginGeneratingDeviceOrientationNotifications];

    return self;
}

- (void)deviceOrientationDidChange:(NSNotification *)notification {!!CO6!!

   // Code goes here to handle device orientation changes

}

- (void)dealloc {!!CO7!!
    [[UIDevice currentDevice] endGeneratingDeviceOrientationNotifications];
    [rootController release];
    [pickerController release];
    [overlayView release];
    [locationManager release];
    [accelerometer release];
    [super dealloc];
}

@end
```

1. Here we get a reference to the calling view controller and initialize our camera overlay view based on the current screen bounds.
2. In this block of code we initialize our camera controller, removing the default camera controls, resizing it to fill the entire screen and adding our new overlay view.
3. In this block we initialize the CLLocationManager, set the heading and location accuracy desired (the best available in both cases) and pointing it the current class as its delegates. We then ask the location manager to start updating the heading and location.
4. Here we get a reference to the shared accelerometer object and point to the current class as its delegate.
5. In this block we register the current class as an observer with the NSNotificationCenter for all UIDeviceOrientationDidChangeNotification messages, registering the deviceOrientationDidChange: method in the current class as the method that will handle these notifications. We then ask the UIDevice class to begin generating these methods.
6. A placeholder for the deviceOrientationDidChange: method that we will add code to later on in the chapter.
7. The dealloc: method, we need to stop generating UIDeviceOrientationDidChangeNotification messages and release our instance variables.

Save your changes to the `ARController` class implementation, and let's take stock. How are we going to test this controller? We need to add another method to our `ARController` class to present the image picker controller, with the associated augmented reality overlay view, to the user. So go ahead and add the following method to the `ARController` class implementation,

```
- (void)presentModalARControllerAnimated:(BOOL) animated {
    [self.rootController presentModalViewController:[self pickerController]
animated:animated];
    self.overlayView.frame = self.pickerController.view.bounds;
}
```

remembering to also declare the method in the corresponding interface file.

```
- (void)presentModalARControllerAnimated:(BOOL) animated;
```

Save your changes to the controller class. We're going to need a test harness in our project to present our `ARControler` class.

Let’s add a view controller to out project. Right click on the Classes group in the Groups & Files pane in Xcode and and choose Add→New File... from the menu. Choose to create a UIViewController subclass. We will be programmatically creating the view for this controller using our ARController class, so do not check the box to add a XIB file for the user interface Name the new class RootController when prompted.

Click on the *RootController.h* file in the Groups & Files pane to open it in the Xcode editor and add a instance variable to hold our ARController.

```
#import <UIKit/UIKit.h>

@class ARController;

@interface RootController : UIViewController {
    ARController *arController;
}

@property (nonatomic, retain) ARController *arController;

@end
```

Then in the corresponding implementation class modify the loadView: and viewDidAppear: methods as below to create and instance of our ARController class and then call our presentModalARControllerAnimated: method.

```
#import "RootController.h"
#import "ARController.h"

@implementation RootController

@synthesize arController;

- (void)loadView {
    self.arController = [[ARController alloc] initWithViewController:self];
}

- (void)viewDidAppear:(BOOL)animated {
    [super viewDidAppear:animated];
    [self.arController presentModalARControllerAnimated:NO];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:
(UIInterfaceOrientation)interfaceOrientation {
    return YES;!!CO1!!
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)dealloc {
    [arController release];
    [super dealloc];
}

@end
```

1. We want our controller to operating in both portrait and landscape mode.

Save your changes and go ahead and modify our application delegate interface,

```
#import <UIKit/UIKit.h>

@class RootController;

@interface ARViewAppDelegate : NSObject <UIApplicationDelegate> {
    UIWindow *window;
    RootController *viewController;
}
```

```
@property (nonatomic, retain) IBOutlet UIWindow *window;
@property (nonatomic, retain) IBOutlet RootController *viewController;

@end
```

and implementation,

```
#import "ARViewAppDelegate.h"
#import "RootController.h"

@implementation ARViewAppDelegate

@synthesize window;
@synthesize viewController;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:
(NSDictionary *)launchOptions {

    self.viewController = [[RootController alloc] init];
    [window addSubview:[viewController view]];
    [window makeKeyAndVisible];

    return YES;
}


- (void)dealloc {
    [window release];
    [viewController release];
    [super dealloc];
}

@end
```

files to create and instance and present our new view controller, which in turn will present our `ARController` object to the user.

Save your changes and click on the Build and Run button in the Xcode toolbar to deploy your application onto your device. Once the application has deployed you should see something much like Figure 9-3 as the `UIImagePickerController` is initialized by the `ARController` object and the user is prompted to allow the application to use their current location.

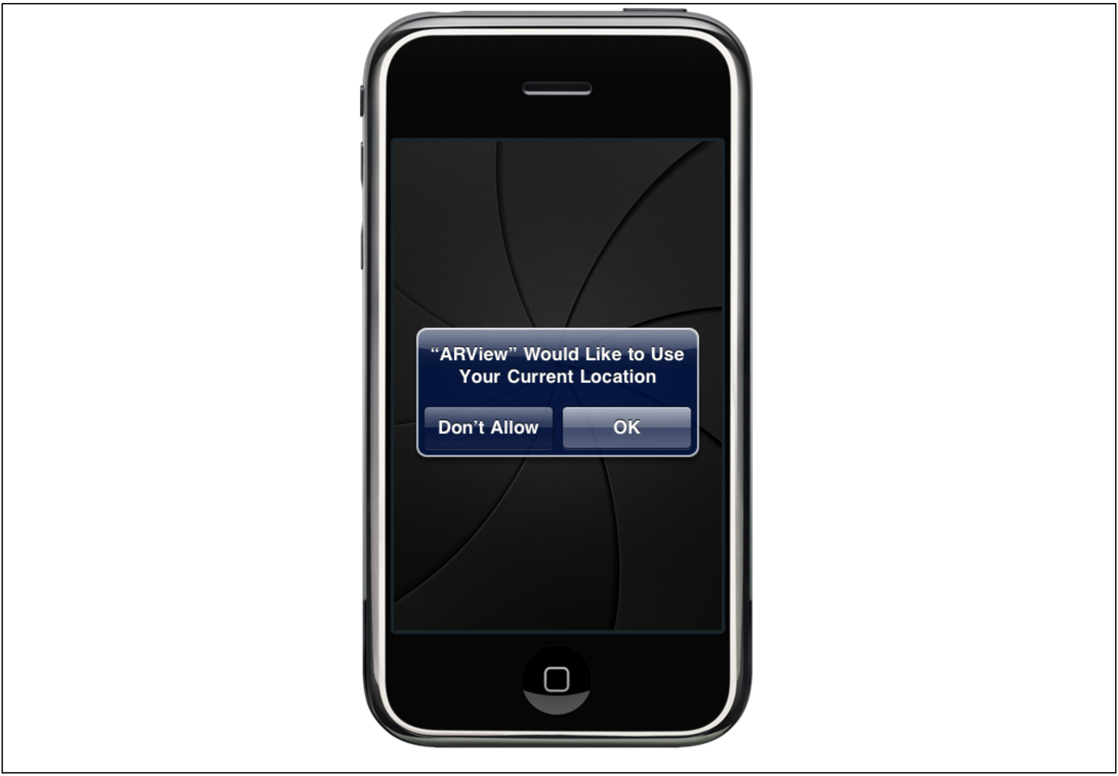

Figure 9-3. The ARController starting up on the iPhone 3GS

Once the `ARController` is initialized the `UIImagePickerController` should open and and you should see something more like Figure 9-4.

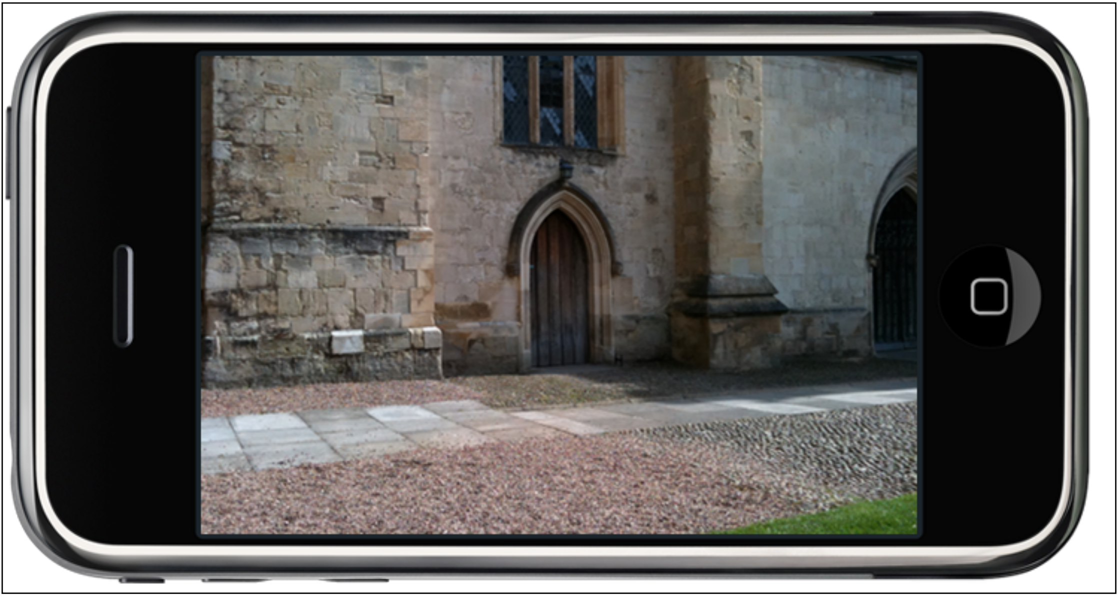

Figure 9-4. The basic camera view on the iPhone 3GS

Now we've started our camera, accelerometer, and location manager we need to start taking account of our sensor readings. Let start off by adding the code to handle device orientation changes to our `ARController` class.

Open the *ARController.h* interface file in the Xcode editor. We'll need an instance variable to store the `currentOrientation`, and at the same time we can definite a `viewRange`, the angular view of the device will change depending on the orientation.

```
        UIDeviceOrientation currentOrientation;
        double viewRange;
```

We need to declare these as properties,

```
    @property (nonatomic) UIDeviceOrientation currentOrientation;
    @property (nonatomic) double viewRange;
```

and then in the corresponding *ARController.m* implementation file synthesize these properties,

```
    @synthesize currentOrientation;
    @synthesize viewRange;
```

We then need to go ahead and modify our `deviceOrienationDidChange:` method to modify the overlay view depending on the current device orientation.

```
    - (void)deviceOrientationDidChange:(NSNotification *)notification {

        UIDeviceOrientation orientation = [[UIDevice currentDevice] orientation];

        if ( orientation != UIDeviceOrientationUnknown &&
              orientation != UIDeviceOrientationFaceUp &&
              orientation != UIDeviceOrientationFaceDown) {

             CGAffineTransform transform =
    CGAffineTransformMakeRotation(degreesToRadians(0));
             CGRect bounds = [[UIScreen mainScreen] bounds];

             if (orientation == UIDeviceOrientationLandscapeLeft) {
                  transform         =
    CGAffineTransformMakeRotation(degreesToRadians(90));
                  bounds.size.width  = [[UIScreen mainScreen] bounds].size.height;
                  bounds.size.height = [[UIScreen mainScreen] bounds].size.width;
             } else if (orientation == UIDeviceOrientationLandscapeRight) {
                  transform         =
    CGAffineTransformMakeRotation(degreesToRadians(-90));
                  bounds.size.width  = [[UIScreen mainScreen] bounds].size.height;
                  bounds.size.height = [[UIScreen mainScreen] bounds].size.width;
             } else if (orientation == UIDeviceOrientationPortraitUpsideDown) {
                  transform = CGAffineTransformMakeRotation(degreesToRadians(180));
             }
             self.overlayView.transform = transform;
             self.overlayView.bounds = bounds;
             self.viewRange = self.overlayView.bounds.size.width / 12;!!CO1!!
        }
        self.currentOrientation = orientation;!!CO2!!
    }
```

1. The viewRange is the current angular view subtended by the screen, this will change dependant on whether the device is in portrait or landscape mode.
2. Here we store the current orientation of the device for use elsewhere in the code.

We also need to go ahead and add the code to handle the `CLLocationManager` delegate call back methods. We'll need to add `currentHeading` and `currentLocation` instance variables to the `ARController` interface file,

```
        CLLocation *currentLocation;
        CLHeading *currentHeading;
```

and declare them as properties.

```
    @property (nonatomic, retain) CLLocation *currentLocation;
    @property (nonatomic, retain) CLHeading *currentHeading;
```

We're also going to make use of a couple of new methods, so while we're editing the interface file we should declare them now,

```
    - (void)updateCurrentCoordinate;
    - (void)updateCurrentLocation:(CLLocation *)newLocation;
```

Save your changes to the interface file and switch to the corresponding implementation and go ahead and synthesize our new variables,

```
@synthesize currentLocation;
@synthesize currentHeading;
```

and add the Core location delegate methods, `locationManager:didUpdateHeading:` and `locationManager:didUpdateLocation:fromLocation:`.

```
- (void)locationManager:(CLLocationManager *)manager didUpdateHeading:(CLHeading
*)newHeading {
    if (newHeading.headingAccuracy > 0) {!!CO1!!
         self.currentHeading = newHeading;
         [self updateCurrentCoordinate];
    }
}

- (void)locationManager:(CLLocationManager *)manager didUpdateToLocation:
(CLLocation *)newLocation fromLocation:(CLLocation *)oldLocation {
    if (newLocation != oldLocation) {
         [self updateCurrentLocation:newLocation]; !!CO2!!
    }

}

- (BOOL)locationManagerShouldDisplayHeadingCalibration:(CLLocationManager
*)manager {!!CO3!!
    return YES;
}
```

1. Here we update the current heading of the device and call our method to update the current ARGeoCoordinate of the device itself based on this change in heading.
2. Here we call our method to handle the updating the device’s current position.
3. This method will allow Core Location to present the heading calibration panel if needed.

Once you’ve added the delegate methods add the following placeholder methods to the code,

```
- (void)updateCurrentLocation:(CLLocation *)newLocation {
    self.currentLocation = newLocation;

    // Code for update the current coordiante of the device,
    // and positions of the displayed ARGeoCoordinates relative
    // to the current coordinate of the device goes here...
}

- (void)updateCurrentCoordinate {

    // Code to update the current coordinate of the device based
    // on the change in orientation of the device go here...

    // Code to update display the current ARGeoGoordinates to the
    // AR overlayView of the device goes here...
}
```

and then in our `initWithViewController:` method we need to add some default initialization of our variables,

```
    self.currentLocation = [[CLLocation alloc] initWithLatitude:37.33231
longitude:-122.03118];
    self.currentOrientation = UIDeviceOrientationPortrait;
    self.viewRange = self.overlayView.bounds.size.width / 12;
```

Finally ensure that our new variables are released in the `dealloc:` method.

```
    [currentLocation release];
    [currentHeading release];
```

Make sure you've saved your changes and build and test your code, it should compile cleanly at this point and when run on your device you should get something resembling Figure 9-4 once again.

> Since there is very little onscreen feedback for the ARView project you might want to add some debugging statements into your code using the NSLog function to report location and heading update, and device orientation, messages back to you to test your new methods.

Okay, once you've convinced yourself that the ARController class is working to this point we need to pause our work on the controller and look at the ARCoordinate and ARGeoCoordinate classes.

Right-click on the ARToolkit group and choose Add→New File... from the menu. Choose to create an Objective-C class, which is a subclass of NSObject. Name the new class ARCoordinate when prompted.

Click on the *ARCoordinate.h* file to open it in the Xcode editor and add the following code to the class,

```
#import <Foundation/Foundation.h>

@interface ARCoordinate : NSObject {
    double coordinateDistance;
    double coordinateInclination;
    double coordinateAzimuth;
    NSString *coordinateTitle;
}

- (id)initWithRadialDistance:(double)distance andInclination:(double)inclination
andAzimuth:(double)azimuth;

@property (nonatomic, retain) NSString *coordinateTitle;
@property (nonatomic) double coordinateDistance;
@property (nonatomic) double coordinateInclination;
@property (nonatomic) double coordinateAzimuth;

@end
```

and then in the corresponding implementation update it as follows.

```
#import "ARCoordinate.h"

@implementation ARCoordinate

@synthesize coordinateTitle;
@synthesize coordinateDistance;
@synthesize coordinateInclination;
@synthesize coordinateAzimuth;

- (id)initWithRadialDistance:(double)distance andInclination:(double)inclination
andAzimuth:(double)azimuth {
    if ( self = [super init] ) {
         self.coordinateDistance = distance;
         self.coordinateInclination = inclination;
         self.coordinateAzimuth = azimuth;
    }
    return self;
}

- (void)dealloc {
    [coordinateTitle release];
    [super dealloc];
}

@end
```

Save your changes and go ahead and similarly create a new ARGeoCoordinate class, again a subclass of NSObject. Once you've done so click on the *ARGeoCoordinate.h* file to open it in the Xcode editor and modify it as follows,

```
#import <Foundation/Foundation.h>
#import <CoreLocation/CoreLocation.h>

#import "ARCoordinate.h"

@interface ARGeoCoordinate : ARCoordinate {!!CO1!!
    CLLocation *coordinateGeoLocation;
}

@property (nonatomic, retain) CLLocation *coordinateGeoLocation;

- (id)initWithCoordiante:(CLLocation *)location andTitle:(NSString*)title;
- (id)initWithCoordiante:(CLLocation *)location andTitle:(NSString*)title
andOrigin:(CLLocation *)origin;
- (float)angleFromCoordinate:(CLLocationCoordinate2D)first toCoordinate:
(CLLocationCoordinate2D)second;
- (void)calibrateUsingOrigin:(CLLocation *)origin;

@end
```

1. Here we've changed the ARGeoCoordinate class from being a subclass of NSObject to being a subclass of ARCoordinate.

and then in the corresponding implementation modify is as below.

```
#import "ARGeoCoordinate.h"

@implementation ARGeoCoordinate

@synthesize coordinateGeoLocation;

- (id)initWithCoordiante:(CLLocation *)location andTitle:(NSString*)title {

    if ( self = [super init] ) {
         self.coordinateGeoLocation = location;
         self.coordinateTitle = title;
    }
    return self;
}

- (id)initWithCoordiante:(CLLocation *)location andOrigin:(CLLocation *)origin {

    if ( self = [super init] ) {
         self.coordinateGeoLocation = location;
         self.coordinateTitle = @"";
         [self calibrateUsingOrigin:origin];
    }
    return self;
}

- (void)calibrateUsingOrigin:(CLLocation *)origin {

    double baseDistance = [origin getDistanceFrom:self.coordinateGeoLocation];
    self.coordinateDistance = sqrt( pow(   [origin altitude]
                                                  -
[self.coordinateGeoLocation altitude], 2)
                                                 + pow(baseDistance, 2) );

    float angle = sin(ABS([origin altitude] - [self.coordinateGeoLocation
altitude]) / self.coordinateDistance);

    if ([origin altitude] > [self.coordinateGeoLocation altitude]) {
         angle = -angle;
    }
    self.coordinateInclination = angle;
    self.coordinateAzimuth = [self angleFromCoordinate:[origin coordinate]
toCoordinate:[self.coordinateGeoLocation coordinate]];

}

- (float)angleFromCoordinate:(CLLocationCoordinate2D)first toCoordinate:
(CLLocationCoordinate2D)second {

    float longitudinalDifference = second.longitude - first.longitude;
    float latitudinalDifference  = second.latitude  - first.latitude;
    float possibleAzimuth = (M_PI * .5f) - atan(latitudinalDifference /
longitudinalDifference);

    if (longitudinalDifference > 0)
         return possibleAzimuth;
    else if (longitudinalDifference < 0)
         return possibleAzimuth + M_PI;
    else if (latitudinalDifference < 0)
         return M_PI;

    return 0.0f;
}

@end
```

The most important part of our `ARGeoCoordinate` class is the `calibrateUsingOrigin:` method. Here we set the super-class properties (the distance, inclination, azimuth) of the geo-located point based on an origin geo-coordiante, the current position of the device.

Now we have out `ARCoordiante` and `ARGeoCoordinate` we can return to the `ARController` class. We can use our `ARCoordinate` class to represent the current orientation of our device in azimuth. Open the *ARController.h* interface file in the Xcode editor and add the following class definitions,

```
@class ARCoordinate;
@class ARGeoCoordinate;
```

and then add an `ARCoordinate` instance variable inside the class declaration,

```
    ARCoordinate *currentCoordinate;
```

and declare it as a property.

```
@property (nonatomic, retain) ARCoordinate *currentCoordinate;
```

Next click on the corresponding *ARController.m* implementation file to open it in the Xcode editor, here we need to import the relevant class interface files,

```
#import "ARCoordinate.h"
#import "ARGeoCoordinate.h"
```

and synthesize our new property.

```
@synthesize currentCoordinate;
```

Then in the class' `initWithViewController:` method we need to create an instance of the `ARCoordinate` class and set an initial value for our `currentCoordinate` instance variable.

```
    self.currentCoordinate = [[ARCoordinate alloc] initWithRadialDistance:1.0
andInclination:0 andAzimuth:0];
```

Then returning to our `updateCurrentCoordinate:` method,

```
- (void)updateCurrentCoordinate {

    double adjustment = 0;
    if (self.currentOrientation == UIDeviceOrientationLandscapeLeft)
         adjustment = degreesToRadians(270);
    else if (currentOrientation == UIDeviceOrientationLandscapeRight)
```

```
            adjustment = degreesToRadians(90);
        else if (currentOrientation == UIDeviceOrientationPortraitUpsideDown)
            adjustment = degreesToRadians(180);

        self.currentCoordinate.coordinateAzimuth =
            degreesToRadians(self.currentHeading.magneticHeading) - adjustment;

    }
```

we can update this coordinate based on the current device orientation and heading. We should also take into account the accelerometer readings to describe the current viewing angle of the device. Go ahead and declare and instance variable in the ARController interface file to hold this information,

```
        double viewAngle;
```

and declare it as a property.

```
    @property (nonatomic) double viewAngle;
```

Then in the implementation synthesize this property.

```
    @synthesize viewAngle;
```

Finally we need to add the accelerometer:didAccelerate: method to handle updating that viewAngle property and the current coordinate of the device.

```
    - (void)accelerometer:(UIAccelerometer *)meter didAccelerate:(UIAcceleration
    *)acceleration {

        switch (currentOrientation) {
            case UIDeviceOrientationLandscapeLeft:
                self.viewAngle = atan2(acceleration.x, acceleration.z);
                break;
            case UIDeviceOrientationLandscapeRight:
                self.viewAngle = atan2(-acceleration.x, acceleration.z);
                break;
            case UIDeviceOrientationPortrait:
                self.viewAngle = atan2(acceleration.y, acceleration.z);
                break;
            case UIDeviceOrientationPortraitUpsideDown:
                self.viewAngle = atan2(-acceleration.y, acceleration.z);
                break;
            default:
                break;
        }

        [self updateCurrentCoordinate];!!CO1!!
    }
```

1. As well as calling updateCurrentCoordinate: from the locationManager:didUpdateHeading: method we need to call it here from the accelerometer:didAccelerate: method. This means that the current ARCoordinate of the device, describing the current azimuth, is updated when the yaw, pitch or roll of the device changes.

Build and test your code, there still isn't any visible changes in our interface so you may want to make use of the NSLog function to add some debug output to your code.

Now we have a more or less working framework. The ARController knows everything it needs to know to plot a ARGeoCoordinate into the overlay view. So lets go ahead and add an NSMutableArray to hold a number of ARGeoCoordinate objects that we want to display in the overlay view.

In the *ARController.h* interface file add the following instance variable,

```
        NSMutableArray *coordinates;
```

and some methods to allow us to update the contents of this array.

```
    - (void)addCoordinate:(ARCoordinate *)coordinate animated:(BOOL)animated;
```

```
- (void)removeCoordinate:(ARCoordinate *)coordinate animated:(BOOL)animated;
```

There is no need to expose this instance variable as a property.

Then in the corresponding implementation file we need to initialize the array in the `initWithViewController:` method

```
    coordinates = [[NSMutableArray alloc] init];
```

and add some skeleton implementation to our code for our two accessor methods.

```
- (void)addCoordinate:(ARCoordinate *)coordinate animated:(BOOL)animated {
    [coordinates addObject:coordinate];
}

- (void)removeCoordinate:(ARCoordinate *)coordinate animated:(BOOL)animated {
    [coordinates removeObject:coordinate];
}
```

Now we have a number of coordinates in our `ARController` object we need to modify our `updateCurrentLocation:` method to calibrate these `ARGeoCoordiante` objects when the device location changes.

```
- (void)updateCurrentLocation:(CLLocation *)newLocation {
    self.currentLocation = newLocation;

    for (ARGeoCoordinate *geoLocation in coordinates ) {
         if ( [geoLocation isKindOfClass:[ARGeoCoordinate class]]) {
              [geoLocation calibrateUsingOrigin:self.currentLocation];
         }
    }

}
```

Build and test your code. It should compile cleanly.

At this stage we have an `ARController` that knows the current location of the device, it's azimuth and orientation. It has an array of `ARGeoCoordinate` objects to display in the camera overlay view, and as the device moves these are recalibrated so that the distance, inclination and azimuth of the point is updated relative to the device.

We also need to update these coordinates as the orientation and azimuth of the device changes, so we will need to call a method every time the azimuth of the device changes, for instance as it is panned around. So at the bottom of the code in the `updateCurrentCoordinate:` method, add the line,

```
    [self updateLocations];
```

and then go ahead and add a stub implementation of the method to the `ARController` class.

```
- (void)updateLocations {

    // If there are no locations to draw, return
    if ( !coordinates || [coordinates count] == 0 ) return;

    for (ARCoordinate *item in coordinates) {

        // Code to update overlay view goes here...
    }
}
```

We also have to declare this method in our interface file, but things are getting a bit untidy, lets re-factor our `ARController` interface adding a private interface to hide the `updateCurrentCoordinate:`, `updateCurrentLocation:` and our new `updateLocations:` methods away from the public interface of the class by adding a private interface to our `ARController` class.

Your *ARController.h* interface file should look like this after you do that,

```
#import <Foundation/Foundation.h>
```

```
#import <CoreLocation/CoreLocation.h>
#import <UIKit/UIKit.h>

@class ARCoordinate;
@class ARGeoCoordinate;

@interface ARController : NSObject <UIAccelerometerDelegate,
CLLocationManagerDelegate>  {

    UIDeviceOrientation currentOrientation;
    CLLocation *currentLocation;
    CLHeading *currentHeading;
    ARCoordinate *currentCoordinate;

    double viewRange;
    double viewAngle;

    UIViewController *rootController;
    UIImagePickerController  *pickerController;
    UIView *overlayView;

    CLLocationManager *locationManager;
    UIAccelerometer*accelerometer;

    NSMutableArray *coordinates;

}

@property (nonatomic) UIDeviceOrientation currentOrientation;
@property (nonatomic, retain) CLLocation *currentLocation;
@property (nonatomic, retain) CLHeading *currentHeading;
@property (nonatomic, retain) ARCoordinate *currentCoordinate;

@property (nonatomic) double viewRange;
@property (nonatomic) double viewAngle;

@property (nonatomic, retain) UIViewController *rootController;
@property (nonatomic, retain) UIImagePickerController *pickerController;
@property (nonatomic, retain) UIView *overlayView;

@property (nonatomic, retain) CLLocationManager *locationManager;
@property (nonatomic, retain) UIAccelerometer *accelerometer;

- (id)initWithViewController:(UIViewController *)theView;
- (void)presentModalARControllerAnimated:(BOOL)animated;

- (void)addCoordinate:(ARCoordinate *)coordinate animated:(BOOL)animated;
- (void)removeCoordinate:(ARCoordinate *)coordinate animated:(BOOL)animated;

@end

@interface ARController (Private)

- (void)updateCurrentCoordinate;
- (void)updateCurrentLocation:(CLLocation *)newLocation;
- (void)updateLocations;

@end
```

Make sure you've saved all your changes to the `ARController` class. Because before we can write the `updateLocations:` method, which will display the `ARGeoCoordinate` objects in the overlay view, we need something to display. We'll need a `UIView`-based class, a marker, which we can associate with a geo-coordinate.

Right-click on the ARToolkit group and choose Add→New File... from the menu. Choose to create an Objective-C class, which is a subclass of `UIView`. Name the new class `ARMarker` when prompted, modify the interface file as follows;

```
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>

@class ARCoordinate;

@interface ARMarker : UIView {

}

- (id)initForCoordinate:(ARCoordinate *)coordinate;

@end
```

and the implementation file as below.

```
#import "ARMarker.h"
#import "ARCoordinate.h"

#define BOX_WIDTH 150
#define BOX_HEIGHT 100

@implementation ARMarker

- (id)initForCoordinate:(ARCoordinate *)coordinate {

    CGRect theFrame = CGRectMake(0, 0, BOX_WIDTH, BOX_HEIGHT);

    if (self = [super initWithFrame:theFrame]) {

         UILabel *titleLabel = [[UILabel alloc] initWithFrame:CGRectMake(0, 0,
BOX_WIDTH, 20.0)];

         titleLabel.backgroundColor = [UIColor colorWithWhite:.3 alpha:.8];
         titleLabel.textColor = [UIColor whiteColor];
         titleLabel.textAlignment = UITextAlignmentCenter;
         titleLabel.text = coordinate.coordinateTitle;
         [titleLabel sizeToFit];
         [titleLabel setFrame: CGRectMake(BOX_WIDTH / 2.0 -
titleLabel.bounds.size.width / 2.0 - 4.0, 0, titleLabel.bounds.size.width + 8.0,
titleLabel.bounds.size.height + 8.0)];

         UIImageView *pointView   = [[UIImageView alloc]
initWithFrame:CGRectZero];
         pointView.image = [UIImage imageNamed:@"point.png"];
         pointView.frame = CGRectMake((int)(BOX_WIDTH / 2.0 -
pointView.image.size.width / 2.0), (int)(BOX_HEIGHT / 2.0 -
pointView.image.size.height / 2.0), pointView.image.size.width,
pointView.image.size.height);

         [self addSubview:titleLabel];
         [self addSubview:pointView];
         [self setBackgroundColor:[UIColor clearColor]];
         [titleLabel release];
         [pointView release];
    }

    return self;
}

- (void)dealloc {
    [super dealloc];
}

@end
```

You'll notice that the `ARMarker` class loads a *point.png* image using the `imageNamed:` method. This is a simple image we're going to plot on top our `overlayView` to represent the `ARGeoCoordinate`. You can either use a small image from the web, or create a simple marker inside your favorite paint program. I use a simple 25×25 cross. Once youv're created your image, drag and drop it into you're the Resources folder in your project, remembering to check the Copy items into destination group's folder (if needed) check box.

Each ARCoordiante will need a corresponding marker, so the simplest thing to do here is to store the marker inside our coordinate class, returning to the *ARCoordinate.h* interface file declare the class,

```
@class ARMarker;
```

and add the following instance variable,

```
    ARMarker *coordinateMarker;
```

and declare it as a property.

```
@property (nonatomic, retain) ARMarker *coordinateMarker;
```

Then in the corresponding implementation file go ahead and synthesize the property.

```
@synthesize coordinateMarker;
```

remembering release it in the dealloc: method

```
- (void)dealloc {
    [coordinateMarker release];
    [coordinateTitle release];
    [super dealloc];
}
```

The easiest place to generate a marker for the coordinate is in the ARController class when we add it to our coordinates array. Open up *ARController.m* and import the *ARMarker.h* interface file,

```
#import "ARMarker.h"
```

and then modify the addCoordainte:animated: methods as highlighted below.

```
- (void)addCoordinate:(ARCoordinate *)coordinate animated:(BOOL)animated {
    ARMarker *marker = [[ARMarker alloc] initForCoordinate:coordinate];
    coordinate.coordinateMarker = marker;
    [marker release];
    [coordinates addObject:coordinate];
}
```

Build and test, your code should compile cleanly

Now we have a view for each ARGeoCoordinate held by the ARController, we need to return to the updateLocations: method. This method should check to see if the ARCoordinate underlying out geo-coordinate is currently visible depending on the azimuth of the device and plot the ARMarker for that ARCoordinate onto the overlayView if it is visible. If not, and if the marker is already in the overlayView, it needs to remove it from the view.

Modify your updateLocations: method as below.

```
- (void)updateLocations {

    // If there are no locations to draw, return
    if ( !coordinates || [coordinates count] == 0 ) return;

    for (ARCoordinate *item in coordinates) {

         UIView *viewToDraw = item.coordinateMarker;

         if ([self viewportContainsCoordinate:item]) {

              // Code to add viewToDraw to overlayView

         } else {

              // Code to remove viewToDraw from overlayView
         }
    }
}
```

You'll notice we've added another method `viewportContainsCoordinate:` to encapsulate the logic about whether the coordinate is currently visible. Lets go ahead and implement that method along with its supporting method `deltaAzimuthForCoordinate:`.

```
- (BOOL)viewportContainsCoordinate:(ARCoordinate *)coordinate {

    double deltaAzimuth = [self deltaAzimuthForCoordinate:coordinate];
    BOOL result = NO;
    if (deltaAzimuth <= degreesToRadians(self.viewRange)) {
         result = YES;
    }
    return result;
}

- (double)deltaAzimuthForCoordinate:(ARCoordinate *)coordinate {

    double currentAzimuth = self.currentCoordinate.coordinateAzimuth;
    double pointAzimuth   = coordinate.coordinateAzimuth;

    double deltaAzimith = ABS( pointAzimuth - currentAzimuth);

    // If values are on either side of the Azimuth of North we need to
    // adjust it. Only check the degree range.
    if (currentAzimuth < degreesToRadians(self.viewRange) &&
         pointAzimuth > degreesToRadians(360-self.viewRange)) {
         deltaAzimith   = (currentAzimuth + ((M_PI * 2.0) - pointAzimuth));
    } else if (pointAzimuth < degreesToRadians(self.viewRange) &&
                currentAzimuth > degreesToRadians(360-self.viewRange)) {
         deltaAzimith   = (pointAzimuth + ((M_PI * 2.0) - currentAzimuth));
    }
    return deltaAzimith;
}
```

Add these two new methods to the private interface of the `ARController` class.

```
@interface ARController (Private)

- (void)updateCurrentCoordinate;
- (void)updateCurrentLocation:(CLLocation *)newLocation;
- (void)updateLocations;

- (BOOL)viewportContainsCoordinate:(ARCoordinate *)coordinate;
- (double)deltaAzimuthForCoordinate:(ARCoordinate *)coordinate;

@end
```

If you're falling along closely you'll probably notice that there above method has a flaw. We'll run into problems as the user pans the device around in Azimuth. If you haven't spotted the problem, ask yourself what happens if the azimuth of the current coordinate is negative or greater than 2π? That a situation that could easily happen with our current code.

We could fix this here in the `ARController`, but the easiest place to fix this is actually in the `ARCoordiante` class itself. Let's go ahead and override the `coordinateAzimuth` accessor method in our `ARCoordinate` class to fix this problem.

```
- (double)coordinateAzimuth {
    if ( coordinateAzimuth < 0.0 ) {
         coordinateAzimuth = (M_PI * 2.0) + coordinateAzimuth;
    } else if ( coordinateAzimuth > (M_PI * 2.0) ) {
         coordinateAzimuth = coordinateAzimuth - (M_PI * 2.0);
    }
    return coordinateAzimuth;
}
```

Build and test your code. The code should compile cleanly, except for a warning `unused variable 'viewToDraw'`. Don't worry about that, we're just about to fix it. Go back to the `updateLocations:` method, and update it as highlighted below.

```
- (void)updateLocations {

    if ( !coordinates || [coordinates count] == 0 ) return;

    for (ARCoordinate *item in coordinates) {

         UIView *viewToDraw = item.coordinateMarker;

         if ([self viewportContainsCoordinate:item]) {

              CGPoint point = [self pointForCoordinate:item];
              float width     = viewToDraw.bounds.size.width;
              float height = viewToDraw.bounds.size.height;

              viewToDraw.frame =
              CGRectMake(point.x - width / 2.0, point.y - (height / 2.0), width,
height);

              if ( !([viewToDraw superview]) ) {
                   [self.overlayView addSubview:viewToDraw];
                   [self.overlayView sendSubviewToBack:viewToDraw];
              }

         } else {

              [viewToDraw removeFromSuperview];
         }
    }
}
```

If the ARCoordinate is present in our view we use another new method pointForCoordiante: to grab the actual (x,y) coordinate for the ARGeoCoordinate in our overlayView, otherwise we remove the view from the superview, the overlay.

Add the following two methods to your ARController implementation.

```
- (CGPoint)pointForCoordinate:(ARCoordinate *)coordinate {

    CGPoint point;
    CGRect viewBounds = self.overlayView.bounds;

    double currentAzimuth = self.currentCoordinate.coordinateAzimuth;
    double pointAzimuth = coordinate.coordinateAzimuth;
    double pointInclination  = coordinate.coordinateInclination;

    double deltaAzimuth = [self deltaAzimuthForCoordinate:coordinate];
    BOOL isBetweenNorth = [self isNorthForCoordinate:coordinate];

    if ((pointAzimuth > currentAzimuth && !isBetweenNorth) ||
         (currentAzimuth > degreesToRadians(360-self.viewRange) &&
          pointAzimuth < degreesToRadians(self.viewRange))) {

         // Right side of Azimuth
         point.x = (viewBounds.size.width / 2) + ((deltaAzimuth /
degreesToRadians(1)) * 12);

         } else {

         // Left side of Azimuth
         point.x = (viewBounds.size.width / 2) - ((deltaAzimuth /
degreesToRadians(1)) * 12);
         }
    point.y = (viewBounds.size.height / 2)
    + (radiansToDegrees(M_PI_2 + self.viewAngle)  * 2.0)
    + ((pointInclination / degreesToRadians(1)) * 12);

    return point;

}
```

```
-(BOOL)isNorthForCoordinate:(ARCoordinate *)coordinate {

    BOOL isBetweenNorth = NO;
    double currentAzimuth = self.currentCoordinate.coordinateAzimuth;
    double pointAzimuth = coordinate.coordinateAzimuth;

    if ( currentAzimuth < degreesToRadians(self.viewRange) &&
         pointAzimuth > degreesToRadians(360-self.viewRange) ) {
         isBetweenNorth = YES;
    } else if ( pointAzimuth < degreesToRadians(self.viewRange) &&
                 currentAzimuth > degreesToRadians(360-self.viewRange)) {
         isBetweenNorth = YES;
    }
    return isBetweenNorth;

}
```

Remember you should also to declare them in the private interface of our `ARController`.

```
@interface ARController (Private)

- (void)updateCurrentCoordinate;
- (void)updateCurrentLocation:(CLLocation *)newLocation;
- (void)updateLocations;

- (BOOL)viewportContainsCoordinate:(ARCoordinate *)coordinate;
- (double)deltaAzimuthForCoordinate:(ARCoordinate *)coordinate;
- (CGPoint)pointForCoordinate:(ARCoordinate *)coordinate;
- (BOOL)isNorthForCoordinate:(ARCoordinate *)coordinate;

@end
```

Build and test your code, it should all compile cleanly. We're pretty much there, the only thing left to do is to go to our *RootController.m* and add some test points.

In the `RootController` implementation you should import the `ARGeoCoordinate` interface file,

```
#import "ARGeoCoordinate.h"
```

and in the `loadView:` method you should go ahead and add some test data points. I used cities and countries to test my code.

```
- (void)loadView {
    self.arController = [[ARController alloc] initWithViewController:self];

    ARGeoCoordinate *tempCoordinate;
    CLLocation     *tempLocation;

    tempLocation = [[CLLocation alloc] initWithLatitude:51.500152
longitude:-0.126236];
    tempCoordinate = [[ARGeoCoordinate alloc] initWithCoordiante:tempLocation
andTitle:@"London"];
    tempCoordinate.coordinateInclination = M_PI/30;
    [self.arController addCoordinate:tempCoordinate animated:NO];
    [tempLocation release];

    // Add more test points here if desired...

}
```

Save all your changes. We're done. If all everything has gone to plan you should now have a working AR application. Click on the Build and Run button in the Xcode toolbar and deploy your application onto your device. If you pan your view around you should see your test points in the camera view, see Figure 9-5.

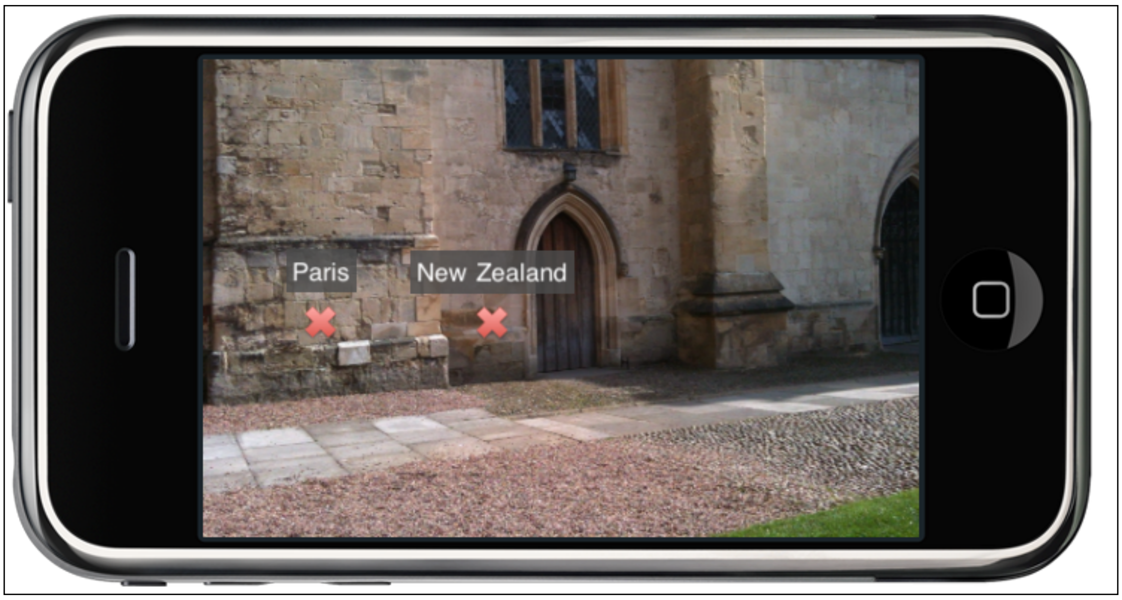

Figure 9-5. The ARView application running on my iPhone 3GS

Going Further

The code we've built in this chapter is a good generic base for building your own AR applications. There are some obvious features you can add; the ability to add more text to an `ARMarker` rather than just a title, the ability to customize the image used for each `ARCoordinate` rather than using the dame default image for each, the ability to scale and rotate the `ARMarker` objects depending on distance and azimuth to the current device, the ability to tap on an `ARMarker` object and open further views, or the ability to create an `ARMarker` with an arbitrary `UIView`, and of course the ability to dismiss the `ARController` entirely and return to the `RootController` view.

Adding Interactivity to ARController

Lets address one of those points, adding (very simple) interactivity to our `ARMarker` object. This should give you a good idea how to go about extending the ARView project in other ways.

We're going to add a simple `UIButton` to each `ARMarker`, see the changes in the implementation highlighted below.

```
@implementation ARMarker

- (id)initForCoordinate:(ARCoordinate *)coordinate {

    CGRect theFrame = CGRectMake(0, 0, BOX_WIDTH, BOX_HEIGHT);
    if (self = [super initWithFrame:theFrame]) {

         UILabel *titleLabel =
          [[UILabel alloc] initWithFrame:CGRectMake(0, 0, BOX_WIDTH, 20.0)];

         titleLabel.backgroundColor = [UIColor colorWithWhite:.3 alpha:.8];
         titleLabel.textColor = [UIColor whiteColor];
         titleLabel.textAlignment = UITextAlignmentCenter;
         titleLabel.text = coordinate.coordinateTitle;
         [titleLabel sizeToFit];
         [titleLabel setFrame: CGRectMake(BOX_WIDTH / 2.0 -
titleLabel.bounds.size.width / 2.0 - 4.0, 0, titleLabel.bounds.size.width + 8.0,
titleLabel.bounds.size.height + 8.0)];

         UIImageView *pointView   = [[UIImageView alloc]
initWithFrame:CGRectZero];
```

```
        pointView.image = [UIImage imageNamed:@"point.png"];
        pointView.frame = CGRectMake((int)(BOX_WIDTH / 2.0 -
pointView.image.size.width / 2.0), (int)(BOX_HEIGHT / 2.0 -
pointView.image.size.height / 2.0), pointView.image.size.width,
pointView.image.size.height);

        UIButton *buttonView = [UIButton buttonWithType:UIButtonTypeInfoLight];
        buttonView.frame = CGRectMake(
        (int)(BOX_WIDTH / 2.0 + 20.0 - buttonView.bounds.size.width / 2.0),
        (int)(BOX_HEIGHT / 2.0 - buttonView.bounds.size.height/2.0),
        buttonView.bounds.size.width, buttonView.bounds.size.height );
        [buttonView addTarget:self action:@selector(infoButtonPushed:)
forControlEvents:UIControlEventTouchUpInside];

        [self addSubview:titleLabel];
        [self addSubview:pointView];
        [self addSubview:buttonView];
        [self setBackgroundColor:[UIColor clearColor]];
        [titleLabel release];
        [pointView release];
    }
    return self;
}

-(void)infoButtonPushed:(id)notification {
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Button Pushed"
message:@"Pushed the button." delegate:nil cancelButtonTitle:@"OK"
otherButtonTitles:nil];
    [alert show];
    [alert release];
}

- (void)dealloc {
    [super dealloc];
}

@end
```

Pushing the button will popup a simple `UIAlertView` with a message. Make the changes and click on the Build and Run button you should see something like Figure 9-6.

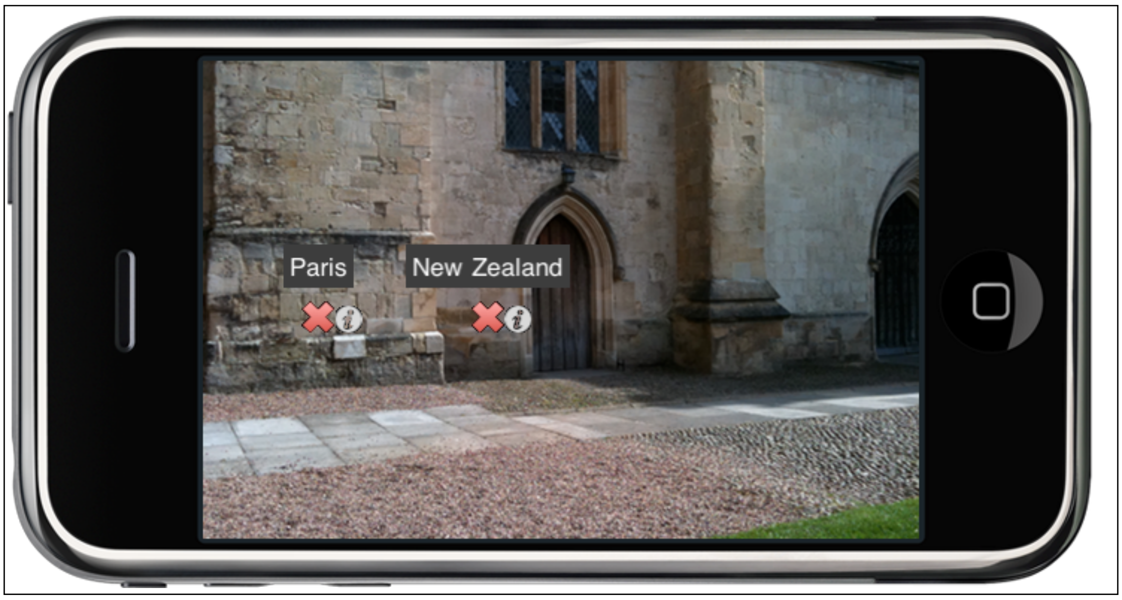

Figure 9-6. The newly interactive ARView application

Tap on one of the information buttons (shown beside the cross shaped marker). If you're in portrait mode everything will work, however if you've rotated the device (as I've done above in Figure 9-6) then the `UIAlertView` will have the wrong orientation. There is an easy fix for this problem, open up the ARController implementation and modify the deviceOrientationDidChange: method as highlighted below,

```
- (void)deviceOrientationDidChange:(NSNotification *)notification {
    UIDeviceOrientation orientation = [[UIDevice currentDevice] orientation];
    UIApplication *app = [UIApplication sharedApplication];

    if ( orientation != UIDeviceOrientationUnknown &&
          orientation != UIDeviceOrientationFaceUp &&
          orientation != UIDeviceOrientationFaceDown) {

         CGAffineTransform transform =
CGAffineTransformMakeRotation(degreesToRadians(0));
         CGRect bounds = [[UIScreen mainScreen] bounds];
         [app setStatusBarHidden:YES];
         [app setStatusBarOrientation:UIInterfaceOrientationPortrait animated:
NO];

         if (orientation == UIDeviceOrientationLandscapeLeft) {
              transform          =
CGAffineTransformMakeRotation(degreesToRadians(90));
              bounds.size.width  = [[UIScreen mainScreen] bounds].size.height;
              bounds.size.height = [[UIScreen mainScreen] bounds].size.width;
              [app setStatusBarOrientation:UIInterfaceOrientationLandscapeRight
animated: NO];!!CO1!!

         } else if (orientation == UIDeviceOrientationLandscapeRight) {
              transform          =
CGAffineTransformMakeRotation(degreesToRadians(-90));
              bounds.size.width  = [[UIScreen mainScreen] bounds].size.height;
              bounds.size.height = [[UIScreen mainScreen] bounds].size.width;
              [app setStatusBarOrientation:UIInterfaceOrientationLandscapeLeft
animated: NO];

         } else if (orientation == UIDeviceOrientationPortraitUpsideDown) {
              transform = CGAffineTransformMakeRotation(degreesToRadians(180));
              [app
setStatusBarOrientation:UIInterfaceOrientationPortraitUpsideDown animated: NO];
         }
         self.overlayView.transform = transform;
         self.overlayView.bounds = bounds;
         self.viewRange = self.overlayView.bounds.size.width / 12;
    }
    self.currentOrientation = orientation;
}
```

1. Notice that UIDeviceOrientationLandscapeRight is assigned to UIInterfaceOrientationLandscapeLeft and UIDeviceOrientationLandscapeLeft is assigned to UIInterfaceOrientationLandscapeRight; the reason for this is that rotating the device requires rotating the content in the opposite direction.

Save your changes and rebuild. This time you should see something like Figure 9-7 when you tap on one of the information buttons in landscape mode.

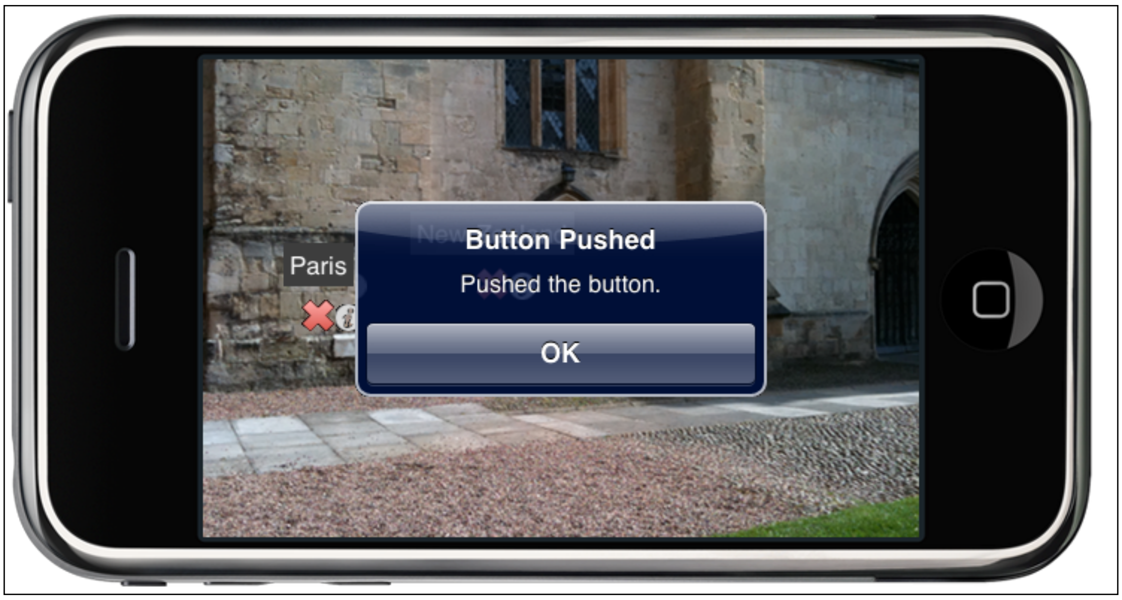

Figure 9-7. Pushing the button in ARView

Congratulations, you now have a fully functional, interactive augmented reality application.