Develop social iPhone and iPad Games with Game Center, Facebook, Twitter, Airplay and more

Beginning iOS Social Games

Kyle Richter

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Mobile games are social and becoming more integrated into our social lives every day. Game Center and Game Kit are Apple's answers to integrating social aspects into iOS games, making it easier than it has ever been before to add items like leaderboards, achievements, multiplayer, and voice chat. Social network integration exists on everything from cars to refrigerators, and adding Twitter and Facebook support to an iOS game has become just about a requirement. Airplay and game controllers may be the next great leap forward in mobile gaming, if history has taught us anything; those that are a step ahead of the future are positioned for success.

Prerequisites

This book assumes that you have the basic skills and understanding required to create an iOS app. It also assumes that you have the background necessary to work with Xcode 5.0. There will be no primer on how to define methods and variables, install and launch Xcode, or create and work with new classes. There are many excellent books on those topics. When you feel ready to begin working with some of the more advanced Cocoa technologies such as Game Center and Game Kit, be sure that you have the basics mastered to a degree that allows you to move through this book without consulting other texts for help.

In addition to the basic requirements, Game Center and the other frameworks covered also heavily leverage blocks, which are a fairly new programming concept to Objective-C. If you haven't yet worked with blocks, we recommend that you read Apple's guide to them, which you can find by searching for *blocks* at `http://developer.apple.com`. You should also feel comfortable working with all the features that were introduced with the Objective-C 2.0 release.

How This Book Is Organized

As you begin working through this book, you will notice that its chapters are essentially standalone. Every effort has been made so that each chapter can be read independently of the others. If you have no experience with Game Center or Game Kit yet, it is highly recommended that you read the first two chapters before skipping around, as they will provide you with the basic information on

how to get Game Center and Game Kit up and running in your development environment. The final four chapters are built on top of the sample code from Chapter 3 and do not continually build on the existing product.

Each chapter follows along with a simple sample iOS game that is introduced in Chapter 1. Following along with the book from start to finish will walk you through the process of creating a fully functional Game Center and Game Kit–leveraged iOS game. In addition, each chapter will build onto a Game Center Manager class that is designed to be reusable across all of your projects. Chapters 12 and 13 will introduce you to Social Framework and adding Twitter and Facebook support to the sample game. Chapters 14 and 15 cover Airplay and Game Controllers.

If you already have a background in general iOS development or Game Center and are looking for help on a specific technology, each chapter is designed to walk you through its covered technology, as well as provide samples showing how to apply the technology to your software.

Source Code

The source code for the projects found in this book is available at www.apress.com/source-code/. Source code is made available in both ARC (Automatic Reference Counting) and non-ARC formats. The code examples used in the text of this book are non-ARC; this is done because it is easier to add ARC support then to remove it. While ARC is quickly becoming the standard there are developers, managers, and projects, which for one reason or another are not ARC ready yet.

Required Software, Materials, and Equipment

To develop iOS software—and more specifically, Game Center and Game Kit–based iOS software—you will first need an Intel-based Mac computer running OSX 10.8 (Mountain Lion) or newer. While you can develop on 10.6, it will not support the most up-to-date release of Xcode. You will also need a copy of Xcode, which you can download free from the Mac App Store or at http://developer.apple.com. This book has been targeted to work with iOS 7; since it is being released at the time when users will be migrating from iOS 6 to iOS 7, it is also written to support iOS 6. Unless otherwise noted within the text, all code is iOS 6–compatible.

In addition to the software and hardware requirements, you will also need an iOS developer account provided by Apple. This account lets you build and test software on devices, as well as ship your finished product to the App Store. The software developer account is available for $99 USD a year and you can purchase yours at http://developer.apple.com/iPhone.

Chapter 1

Getting Started with Social Gaming

Welcome to *Beginning iOS Social Games*! This book is designed to walk you through the process of adding Game Kit, Game Center, and other social functionality into your iOS apps and games. It is centered on a sample game called UFOs that you will be introduced to later in this chapter. However, if you have an existing app or game to which you want to add social functionality, you may use that project instead. This book is written as a reference and resource tool to aid you in the process of adding social functions into your iOS app. Although I recommend you read it from beginning to end to learn the most about the technologies covered, that is not a requirement. Every chapter stands on its own. You can freely skip ahead to the chapters that are relevant to your project needs and quickly implement that functionality into your app.

While this book covers a number of aspects of social gaming, its focus is Apple's social gaming platform Game Kit, and by extension Game Center. When Apple announced Game Kit on March 17, 2009, it was presented as an answer to the difficulty of real-time networking on iOS devices, which until that point had been challenging. Game Kit added support for Bluetooth and LAN as well as voice chat services. Shortly after, Apple announced the Game Center addition to Game Kit as part of iOS 4.0. With the newly announced SDK version, Apple brought a wealth of new features—the Game Center being the most important to the scope of this book. With iOS 5, Game Center once again saw additions, most notably the addition of turn-based gaming to the framework. Apple continued the tradition of supporting Game Center in iOS 6 with the addition of Game Center Challenges. In addition to the Game Center changes, Apple added OS-level support for Twitter in iOS 5 and for Facebook in iOS 6. With the introduction of iOS 7 at WWDC 2013 Game Center saw a complete redesign from the ground up to match the flatter look and feel introduced with UIKit.

Commonly developers in the iOS community have a tendency to think of Game Center as a separate set of Application Programming Interfaces (APIs). This is a fallacy. Game Center is an integral part of Game Kit. The two complement one another and work hand in hand. You will see much evidence of this in the following pages. For the purpose of this book, we are going to address both of these technologies together as Game Kit; however, we may still refer to Game Center–specific functionality by its proper name.

The first ten chapters of this book are dedicated to Game Kit functionality; the remaining chapters cover additional social elements such as Store Kit and In App Purchasing in Chapter 11, Twitter and Facebook sharing in Chapters 12 and 13, Airplay mirroring in Chapter 14, and iOS 7 Game Controllers in chapter 15.

Please note that despite their names, Game Kit and Game Center are not designed for just games. Although recently Apple has begun cracking down on Game Center technology being used in non-games. Some developers have received the following type of rejection email from Apple.

"The intended use of Game Center is to complement game apps or game functionality within an app. However, we noticed that your app does not contain any game play or game features."

These rejections seem to apply mainly to the use of leaderboards and achievements in non-gaming apps. The argument can easily be made that adding a leaderboard or achievement system to your app adds a gameplay element. If you happen to receive this rejection you still have the option of appealing it, I have yet to hear from a developer who has not successfully appealed on these grounds. There haven't been any instances of rejection for using Game Kit networking in any app that I have observed.

Game Kit: An Overview

Game Kit can be broken up into three individual sections: networking, Game Center, and voice chat. Although all of these services work together to create one seamless environment, it can be helpful to look at each individually. While there might be overlap, each section can be considered as a primary category. While the API itself does not differentiate these sections, it can be useful to keep them separate while learning and thinking about Game Kit development.

Networking

Networking in Game Kit allows you to send and receive data between one or more peers. Game Kit networking also provides a connection protocol to connect to local clients that are found on your Wi-Fi network, or locally using Bluetooth. Game Center also extends this functionality with WAN matchmaking.

Game Kit supports creating an ad-hoc Bluetooth or local wireless network between two iOS devices. With the introduction of iOS 4.0, Game Kit began supporting networking on the world area network supporting up to 16 players at once. Game Kit networking is covered in Chapters 6, 7, and 8. Game Center matchmaking is covered in Chapter 5.

Game Center

Game Center handles authentication, friends, leaderboards, achievements, challenges, and invitations. In a sense, Game Center is providing the developer with the server services that are related to social interaction. It can also be argued that Game Center contains its own networking system. While this is true, we will be grouping that topic in the preceding section on networking, which is covered extensively in Chapter 5. Game Center technologies, such as leaderboards and achievements, are covered in Chapters 3 and 4.

> **Note** The term *Game Center*, as used in various print and reference documentation, sometimes refers to the collective set of Game Center APIs as well as to the Game Center app itself.

Voice Chat

"Game Voice," as Apple refers to it, allows any app (not just games) to provide voice communication over a network connection. The APIs handle the capturing and playback of audio feeds for the user and provide services to handle connections, communications, errors, and disconnections. This technology is discussed in Chapter 10.

Sample Game: UFOs

In my experience, most developers are "experience-type" learners. This means that they learn best by doing, not by watching or listening. When I first started to learn how to program, I would copy source code out of code magazines line by line into a Commodore 64. The experience of physically typing in each line of code is what made the information stick. Listening to a lecture or watching someone else write code made it difficult to retain a good deal of the information. I can't imagine I would have stayed with this career path if lectures and demonstrations were my only ways of learning. This book is designed in the spirit of experience-type learners.

The first thing we need to cover, before moving into Game Kit itself, is working with the supplied sample game. The game, which we call "UFOs," is designed not to be an award-winning, addictive game, but rather to be simple enough that it can be thought of as any generic project, and allow you to focus on the social gaming aspects. I have made every effort to reduce the amount of code to less than 300 lines. Although the game itself is simple, it is vital that all readers understand the code as if they wrote it themselves. Keeping the example simple will allow you, as the reader, to detach yourself from the project itself and focus on the Game Kit–specific information. We start by playing the game, then looking at the source code.

> **Note** The source code for all the chapters, as well as the sample project, is available at `www.apress.com`. The sample code is provided in two formats: one that supports Automatic Reference Counting (ARC) environments, a feature added in iOS 5, and one using the manual memory management system. Other than support for ARC, these projects are identical.
>
> It is important to note that the sample code that appears in print is the non-ARC version. If you are using ARC, you will need to make adjustments. For more information, see the Apple article on transitioning to ARC at `http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/RN-TransitioningToARC/Introduction/Introduction.html`.

UFOs: Understanding the Game

The first thing you need to do is open the base project that you downloaded from apress.com. Figure 1-1 shows the file structure for the project. We'll quickly run the game to see what it's like.

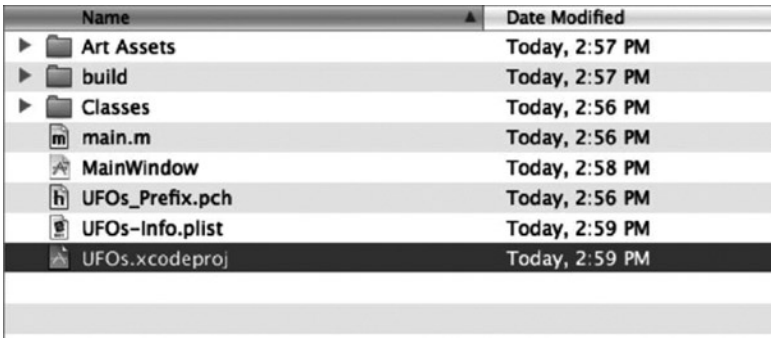

Figure 1-1. *The file structure for the UFO sample project, as seen by the Finder*

To play the game, select Build and Run from Build menu bar. The game will launch to a generic screen with one button labeled "Play." Go ahead and select the Play button. You will be taken to the game screen, as seen in Figure 1-2.

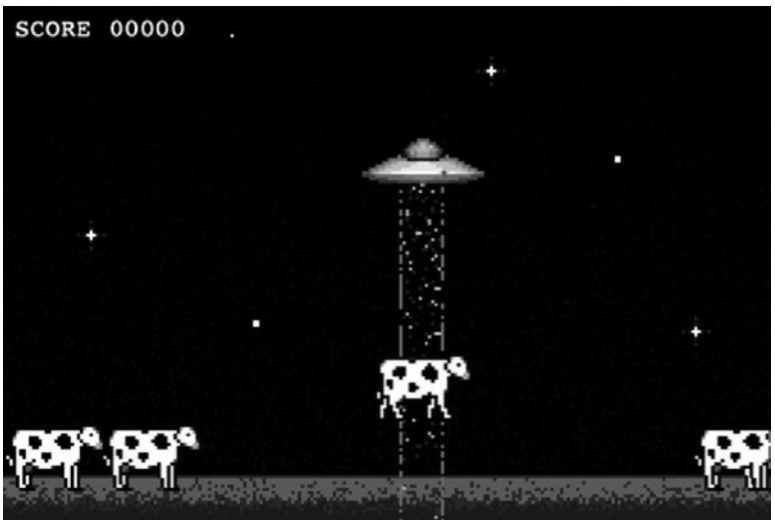

Figure 1-2. *A look at the gameplay view from the UFOs sample project*

The objective of the game is typical; tilt the device up/down or left/right to move your ship around the screen. Once you are positioned over a cow, tap anywhere on the screen and hold until the cow has been abducted. You are awarded one point for every cow you abduct. There is no ending to the game. Every time you abduct a cow, a new one will be created and placed on the grass.

Now that you understand how the game is played, you can take a look at the source code that powers the game engine.

UFOs: Examining the Source Code

In your group tree, you will see the three class files we will be working with, `UFOAppDelegate`, `UFOViewController`, and `UFOGameViewController`. These files all have an associated header (.h file) and implementation file (.m file). The group tree is shown in Figure 1-3.

***Figure 1-3.** A look at the group tree structure for the sample project from within Xcode*

First, take a look at the `UFOAppDelegate.h` and `UFOAppDelegate.m` files. These should look familiar to you from other iOS development work. They are nothing more than a base `UINavgiationController` subclass. If you need to familiarize yourself with the code found here, take a look at Apple's Sample Code for new projects (https://developer.apple.com/library/ios/navigation/#section=Resource%20Types&topic=Sample%20Code).

The next group of files is also relatively simple; take a look at `UFOViewController.h` and `UFOViewController.m`. These are the associated classes for the landing or home screen. All that we have here right now is a play button, but we will be adding leaderboards, achievement, and multiplayer controls to this view as we progress through this book.

Finally, we will be working with `UFOGameViewController.m`. This is the main class that will be powering all gameplay, and it is where the majority of the Game Kit functionality will be added.

Setting Up the Accelerometer Delegate

We will start at the top of the source file you downloaded and work our way through it; open the `UFOGameViewController` file in Xcode. We have modified the `init` method of the UFOGameController to register for accelerometer feedback. Take a look at the following code snippet, which is discussed in detail next.

```
- (id)init
{
    if (self != [super init]) {
        return nil;
    }
```

```
    self.motionManager = [[CMMotionManager alloc] init];
    self.motionManager.accelerometerUpdateInterval = 0.05;

    [self.motionManager startAccelerometerUpdatesToQueue:[NSOperationQueue currentQueue]
    withHandler:^(CMAccelerometerData  *accelerometerData, NSError *error)
    {
        [self motionOccurred:accelerometerData];
        if(error)
       {
             NSLog(@"%@", error);
       }
}];

    return self;
}
```

We will be using the core motion framework for capturing accelerometer data and moving the character. The input frequency is set to 0.05 seconds, which will provide a very fluid reactive control system.

Next, take a look at the -viewDidLoad method. Let's break this down into sections to understand exactly what is going on here.

```
accelerometerDamp = 0.3f;
accelerometerOAngle = 0.6f;
movementSpeed = 15;
```

Here we set some class variables to hold onto some data that we will need when we begin to process the accelerometer input. We will be working with these variables again when we start to deal with ship movement. For now, you don't need to understand exactly what they are doing, just that they have been set. A new method is created called motionOccurred which will be invoked by the core motion framework to handle tilt input. The previously defined damping methods are applied and the information is passed to the movePlayer: method, which will be discussed in the following sections.

```
-(void)motionOccurred:(CMAccelerometerData *)accelerometerData;
{
    //Use a basic low-pass filter to only keep the gravity in the accelerometer values
        accel[0] = accelerometerData.acceleration.x * accelerometerDamp + accel[0] *
(1.0 - accelerometerDamp);
        accel[1] = accelerometerData.acceleration.y * accelerometerDamp + accel[1] *
(1.0 - accelerometerDamp);
        accel[2] = accelerometerData.acceleration.z * accelerometerDamp + accel[2] *
(1.0 - accelerometerDamp);

        if(!tractorBeamOn)
                [self movePlayer:accel[0] :accel[1]];
}
```

Drawing the Player to the View

Next we need to create our "player:"

```
CGRect playerFrame = CGRectMake(100, 70, 80, 34);

myPlayerImageView = [[UIImageView alloc] initWithFrame: playerFrame];
[myPlayerImageView setAnimationDuration:0.75];
[myPlayerImageView setAnimationRepeatCount:99999];
NSArray *imageArray = [NSArray arrayWithObjects: [UIImage imageNamed: @"Saucer1.png"],↩
 [UIImage imageNamed: @"Saucer2.png"], nil];
[myPlayerImageView setAnimationImages:imageArray];
[myPlayerImageView startAnimating];
[self.view addSubview: myPlayerImageView];
```

To do this, we create a new `UIImageView` and initialize it with a predefined frame. The next four lines of code are a little-known, but very useful, part of `UIImageView`. We are setting an array of images that the `UIImageView` will cycle through. In this example, we set two images to be rotated through. We also specify how long we want the full animation to take (3/4 second for our purposes), and the number of times we want the animation to repeat. Once we have set up the animation details, we call `startAnimating` on the `UIImageView`. Then, all that is left for us to do is add the `UIImageView` to the main view. Now we have a player on the screen that is animating!

Setting Up Cows, Beams, and Scores

There are a number of game play elements that we need to create and display. We will begin by creating the score label that will inform the user of their progress while playing.

```
cowArray = [[NSMutableArray alloc] init];
tractorBeamImageView = [[UIImageView alloc] initWithFrame: CGRectZero];
score = 0;
[scoreLabel setText:[NSString stringWithFormat: @"SCORE %05.0f", score]];
```

The score label itself has already been placed on the view using Interface Builder.

```
For (int x = 0; x < 5; x++) {
        [self spawnCow];
}

[self updateCowPaths];
```

The last thing we need to do in the `viewDidLoad` method is create some cows for placement on the screen. There is a helper method to spawn these cows. Every time it is called, it will create a new cow and place it on the screen. We will take a look at this a little later in this section. We also call another helper method to update the walking path for the cows, and we will look at this method in depth later in this section.

Handling Rotation Events

The next method that appears in our code is `shouldAutorotateToInterfaceOrientation`, listed here:

```
-

(BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)↩
interfaceOrientation
{
        if (UIInterfaceOrientationIsLandscape(interfaceOrientation)) {
                return YES;
       }

        return NO;
}
```

While this may appear to be a minor piece of code, it is important to make sure the game doesn't allow the user to rotate into portrait mode, while allowing the user to play in either landscape orientation.

Adding Player Movements

That takes care of all the initialization and setup code. Now we can move into the more exciting parts of the game. First we need to look at user input and actions, and then the gameplay functionality.

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

        accel[0] = acceleration.x * accelerometerDamp + accel[0] * (1.0 -↩
 accelerometerDamp);
        accel[1] = acceleration.y * accelerometerDamp + accel[1] * (1.0 -↩
 accelerometerDamp);
        accel[2] = acceleration.z * accelerometerDamp + accel[2] * (1.0 -↩
 accelerometerDamp);

        if (!tractorBeamOn) {
                [self movePlayer:accel[0] :accel[1]];
        }
}
```

The first method to look at is the accelerometer delegate method. We are taking the values of the accelerometer and applying a dampener to them to give a more realistic feel. We then perform a test to make sure that the tractor beam is off (we don't want to be able to move the UFO if it is on), and then pass the values to our `movePlayer` method, which is shown next.

```
- (void)movePlayer:(float)vertical:(float)horizontal;
{
        vertical += accelerometerOAngle;
```

```
        if (vertical > .50)
        {
                vertical = .50;
        }
        else if (vertical < -.50)
        {
                vertical = -.50;
        }

        if (horizontal > .50)
        {
                horizontal = .50;
        }
        else if (horizontal < -.50)
        {
                horizontal = -.50;
        }
        CGRect playerFrame = myPlayerImageView.frame;

        if ((vertical < 0 && playerFrame.origin.y < 120) || (vertical > 0 &&↩
 playerFrame.origin.y > 20))
        {
                playerFrame.origin.y -= vertical*movementSpeed;
        }

        if ((horizontal < 0 && playerFrame.origin.x < 440) || (horizontal > 0 &&↩
 playerFrame.origin.x > 0))
        {
                playerFrame.origin.x -= horizontal*movementSpeed;
        }

        myPlayerImageView.frame = playerFrame;
}
```

This method is much simpler than you might think at first glance. The first chunk of code sets our maximum speed. The next section ensures that the user cannot move their UFO off the screen. Once we have passed both of these safety checks, we update the player's frame, which moves the UFO.

Watching for Touch Events

The next aspect of the game that we need to worry about is touch events. We will be using a touch to initiate and control the tractor beam. The first step is overriding the `touchesBegan` event.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
        currentAbductee = nil;

        tractorBeamOn = YES;
```

```
tractorBeamImageView.frame = CGRectMake(myPlayerImageView.frame.origin.x+25,↩
 myPlayerImageView.frame.origin.y+10, 28, 318);
        tractorBeamImageView.animationDuration = 0.5;
        tractorBeamImageView.animationRepeatCount = 99999;
NSArray *imageArray = [NSArray arrayWithObjects: [UIImage imageNamed:↩
 @"Tractor1.png"], [UIImage imageNamed: @"Tractor2.png"], nil];

        tractorBeamImageView.animationImages = imageArray;
        [tractorBeamImageView startAnimating];

        [self.view insertSubview:tractorBeamImageView atIndex:4];

        UIImageView *cowImageView = [self hitTest];

        If (cowImageView)
        {
                currentAbductee = cowImageView;
                [self abductCow: cowImageView];
        }

}
```

We first clear out the pointer to the current abducted cow. This value should be `nil` already, but it is best to be diligent. We then set a `BOOL` for whether the tractor beam is on to YES. At this point, we need to draw the tractor beam. To do this, we set the frame for our `tractorBeamImageView` to where the player's UFO is currently located. We will be using the same animation shortcut that was discussed earlier in this section to animate the tractor beam. We then add the tractor beam `imageView` to the main view; we use an `insertSubview` method here to make sure the tractor beam is below the cows but above the background. Then we call our `hitTest` method, which we will look at a little later in this chapter. If we get a result back from the `hitTest`, we call the `abductCow` method.

Before we can move on to the `hitTest` and `abductCow` methods, we must first finish handling the touch events. The only other touch event that we are concerned with at this point is the `touchesEnded` delegate call. When the user removes their finger from the screen, we want to remove the tractor beam from the view and let the user resume movement.

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
        tractorBeamOn = NO;

        [tractorBeamImageView removeFromSuperview];

        if (currentAbductee) {
[UIView beginAnimations: @"dropCow" context:nil];
                [UIView setAnimationDuration: 1.0];
                [UIView setAnimationCurve:UIViewAnimationCurveEaseIn];
                [UIView setAnimationBeginsFromCurrentState: YES];

                CGRect frame = currentAbductee.frame;
```

```
                frame.origin.y = 260;
                frame.origin.x = myPlayerImageView.frame.origin.x +15;

                currentAbductee.frame = frame;

                [UIView commitAnimations];
        }

        currentAbductee = nil;
}
```

Set your state variable for the `tractorBeamOn` to `NO`. Then we can remove the tractor beam image from the view. The next section of code drops the cow back to the ground (if there was one midway in the air). To do this, we just begin a simple animation where we return the cow to ground level. The last thing we need to do is reset the `currentAbductee` pointer to `nil`.

Spawning and Moving Cows

We also have a convenient method to spawn a new cow. This is the method we call from `viewDidLoad` to give the player a base number of cows to try to abduct; we also call it whenever we are finished abducting a cow.

```
- (void)spawnCow;
{
        UIImageView *cowImageView = [[UIImageView alloc] initWithFrame:CGRectMake ↩
(arc4random()%480, 260, 64, 42)];

        cowImageView.image = [UIImage imageNamed: @"Cow1.png"];
        [[self view] addSubview: cowImageView];
        [cowArray addObject: cowImageView];

        [cowImageView release];
}
```

> **Tip** The `arc4Random()` function will return a random number the same way that `rand()` or `random()` will, but it will automatically seed itself when called for the first time.

We create a new `imageView` that will represent the cow. We then use an `arc4Random()` function to produce a random `x` position. We set the image that the cow will be using and add it to the main view. The last thing we need to do here is add the `imageView` to our `cowArray`. We will be using this for a hit test as well as updating the movement paths.

While UFOs is not designed to be an extremely challenging game, we do want to add at least some aspects of difficulty to the gameplay. The following method will cause our cows to wander randomly around the screen.

```
- (void)updateCowPaths
{
        for (int x = 0; x < [cowArray count]; x++) {
UIImageView *tempCow = [cowArray objectAtIndex: x];

                If (tempCow != currentAbductee) {
continue;
                }

                [UIView beginAnimations:@"cowWalk" context:nil];
                [UIView setAnimationDuration: 3.0];

 [UIView setAnimationCurve:UIViewAnimationCurveLinear];

                float currentX = tempCow.frame.origin.x;
                float newX = currentX + arc4random()%100-50;

                if (newX > 480) {
                        newX = 480;
                }
                if (newX < 0) {
                        newX = 0;
                }

                if (tempCow != currentAbductee) {
                        tempCow.frame = CGRectMake(newX, 260, 64, 42);
                }

                [UIView commitAnimations];

                tempCow.animationDuration = 0.75;
                tempCow.animationRepeatCount = 99999;

                //flip cow
                if (newX < currentX) {
NSArray *flippedCowImageArray = [NSArray arrayWithObjects:➥
 [UIImage imageNamed: @"Cow1Reversed.png"], [UIImage imageNamed: @"Cow2Reversed.png"],➥
 [UIImage imageNamed: @"Cow3Reversed.png"], nil];

        [tempCow setAnimationImages:flippedCowImageArray];
} else {
NSArray *cowImageArray = [NSArray arrayWithObjects: [UIImage imageNamed:➥
 @"Cow1.png"], [UIImage imageNamed: @"Cow2.png"], [UIImage imageNamed: @"Cow3.png"], nil];
        [tempCow setAnimationImages:cowImageArray];
}

                [tempCow startAnimating];
                }
        }

        //change the paths for the cows every 3 seconds
        [self performSelector:@selector(updateCowPaths) withObject:nil afterDelay:3.0];
}
```

In the first line of the updateCowPaths method, we cycle through our array of cow objects. We then randomize a new x position for the cow. A quick check ensures that we are not instructing the cow to walk off the screen. Then we commit the animation. We also need to handle the direction change for the cow.

> **Note** The code that we use to handle that event is not the most efficient way of flipping an image, but it is the easiest to learn if you are new to this type of game.

As we did previously with the tractor beam and the UFO images, we will add some animation frames so the cows appear more interesting. The last thing we do is call performSelector with a delay of three seconds. This will update the cow's path every three seconds, adding a slightly more challenging movement system.

Performing a Hit Test with a UIImage

Before we worry about how to set up the cow abduction, there are preliminary steps for abducting the cow itself. For starters, we must implement a hitTest method that is being called from the touchesBegan event discussed earlier in this section.

```
- (UIImageView*)hitTest
{
        if (!tractorBeamOn) {
                return nil;
        }

        for (int x = 0; x < [cowArray count]; x++) {
UIImageView *tempCow = [cowArray objectAtIndex: x];
                CALayer *cowLayer= [[tempCow layer] presentationLayer];
                CGRect cowFrame = [cowLayer frame];

                if (CGRectIntersectsRect(cowFrame, tractorBeamImageView.frame)) {
tempCow.frame = cowLayer.frame;
                        [tempCow.layer removeAllAnimations];
                        return tempCow;
                }
        }

        return nil;
}
```

The first line is another sanity check, this time to ensure that we are not calling the hitTest method when the tractorBeam is not on. Once we know we are supposed to be checking for the hit, we iterate through our array of cow objects. Since the cows are in the middle of an animation, we cannot rely on the data from the frame, as it will show where the cow will end up and not where the cow currently is.

To determine where the cow currently is, we ask for the `presentationLayer`. Core Graphics provides a useful method for testing whether two `GCRects` intersect, and that is what we will be using here. If we hit a cow, we return the object. If we get to the end of our loop without passing a hit test, we return `nil`.

> **Tip** The `presentationLayer` method can be called on any `CALayer` to provide a best guess for the current values of a layer that is in the process of being animated.

Abducting a Cow

In our `touchesBegan` method, we tested to see if `hitTest` returned a cow. If it did, we call `abductCow` with the object that was returned. Here's the code:

```
- (void)abductCow:(UIImageView *)cowImageView;
{
        [UIView beginAnimations: @"abduct" context:nil];
        [UIView setAnimationDuration: 4.0];
        [UIView setAnimationCurve:UIViewAnimationCurveEaseIn];
        [UIView setAnimationDelegate: self];
        [UIView setAnimationDidStopSelector: @selector(finishAbducting)];
        [UIView setAnimationBeginsFromCurrentState: YES];

        CGRect frame = cowImageView.frame;
        frame.origin.y = myPlayerImageView.frame.origin.y;
        cowImageView.frame = frame;

        [UIView commitAnimations];
}
```

We begin an animation event on our cow object (which is an `imageView`). We also set a `didStopSelector`, which will be called once the animation has finished. We set the new y axis coordinate for the cow to our UFO's current y coordinate and begin the animation.

Once the animation has stopped, we get a callback to `finishAbducting`. This allows us to increase the score, clean up the abducting code, and spawn a new cow.

```
- (void)finishAbducting;
{
        if (!currentAbductee || !tractorBeamOn) {
return;
        }

        [cowArray removeObjectIdenticalTo: currentAbductee];
        [tractorBeamImageView removeFromSuperview];

        tractorBeamOn = NO;
```