SAMPLE CHAPTER

iOS 7 IN ACTION

Brendan G. Lim
Martin Conte Mac Donell

MANNING

iOS 7 in Action

by Brendan G. Lim
Martin Conte Mac Donell

Chapter 14

brief contents

14 Applying motion effects and dynamics

This chapter covers

- Motion effects using `UIMotionEffects`
- Adding the parallax effect
- Realistic animations with UIKit Dynamics
- Simulating gravity, collisions, and elasticity
- Creating custom behaviors

With iOS 7 came flat textures devoid of gradients and out went skeuomorphic design that mimicked real-life physical objects. There was also the addition of parallax, which made interface objects appear to be three-dimensional by altering their position ever so slightly depending on the angle at which you're holding your device. This parallax effect and many others can be achieved by using the new motion APIs in UIKit. Also, before iOS 7 you needed to dive into complex math and physics if you wanted to create realistic physics effects in your views. Now there's also a whole new slew of APIs in UIKit Dynamics that you can use to create these realistic effects without having to be a mathematician. You'll learn about both motion and dynamics in this chapter. Together we'll build a fun little app that will serve as a catalog to showcase a few of the great things you can now do.

14.1 Creating your application

The app you'll prepare for this chapter will be rather quick and simple. It will involve two images, a basketball court floor for the background and an image of a basketball, which we'll use to demonstrate motion and dynamics. Open Xcode and create a new single-view application called Motion Ball.

Next, you'll need to download an archive that contains four images for the background and the basketball you'll use in the app. There are two versions of each image, a retina and non-retina version. Open your web browser and go to http://blim.co/HEbROX to download the image archive. Once it's downloaded, open the archive and you'll see the following files:

- basketball.png
- basketball@2x.png
- background.png
- background@2x.png

Jump back into Xcode and select your image assets in the project navigator. Next, drag all four files into the window that displays all of your image assets. You should see them added to your project, as shown in figure 14.1.

Next, open your storyboard and add a `UIImageView` that fills the entire view, and set the image to background, as shown in figure 14.2.

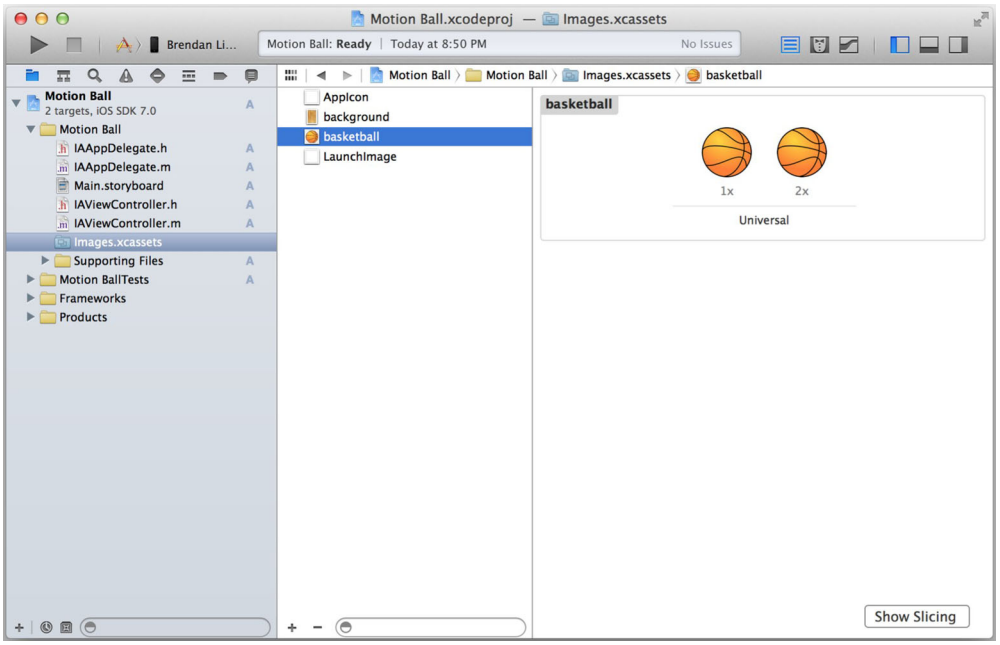

Figure 14.1 Add the images to your image assets in your Xcode project.

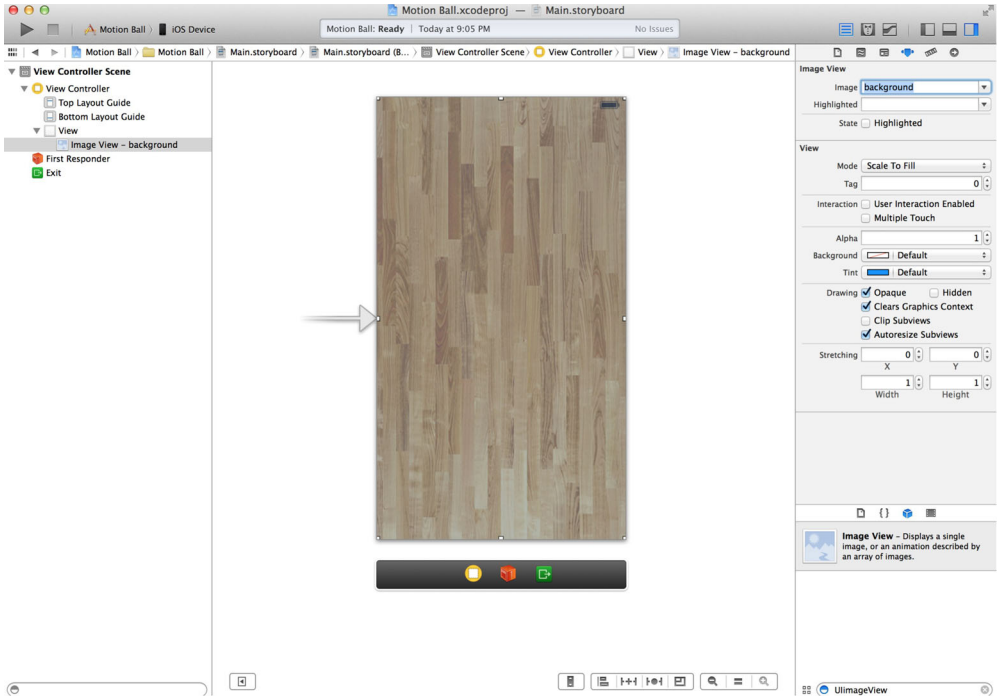

Figure 14.2 Add a `UIImageView` to show the background image you added to the image assets.

The last view you'll need to add is another `UIImageView` that's positioned in the center of the screen. Set its size to 150 x 150 and the image to basketball, as shown in figure 14.3.

Once the basketball's been added, create an outlet for it in IAViewController.h called basketball, as shown in figure 14.4. That's all the setup you'll need to do for your app. Let's now take a look at motion effects.

14.2 Using motion effects

Motion effects are used to add effects to views in your application based on the motion of the device running the application. This is accomplished by applying a motion effect to a specific view. These effects can be applied for horizontal and vertical movement when an iOS device is tilted. This also means that you can only see these results on a real device because you can't simulate this in the iOS Simulator. You'll see how to add a parallax effect to your application.

14.2.1 Adding the parallax effect

The parallax effect utilizes the accelerometer and gyroscope data to determine how a single axis on a view should be adjusted when a device is tilted horizontally or vertically. It'd be easy to show you what the parallax effect looks like if we were able to show you an animated GIF within this book. Instead, look at figure 14.5, which shows you what the parallax effect looks like when you tilt a device horizontally or vertically.

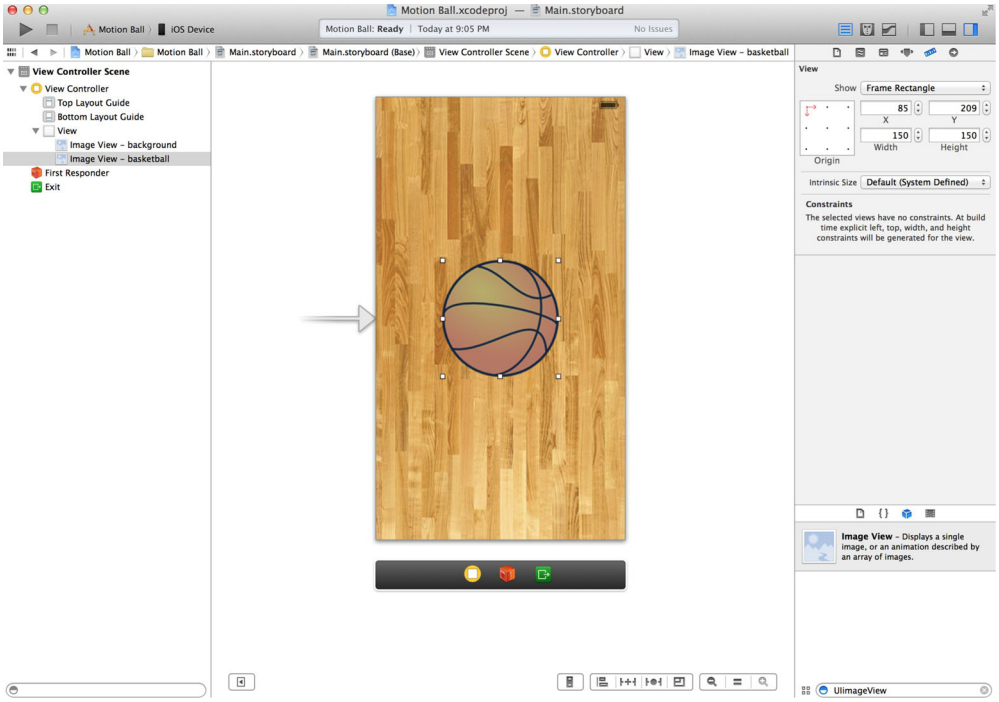

Figure 14.3 Add another image view to the center of the screen to represent the basketball.

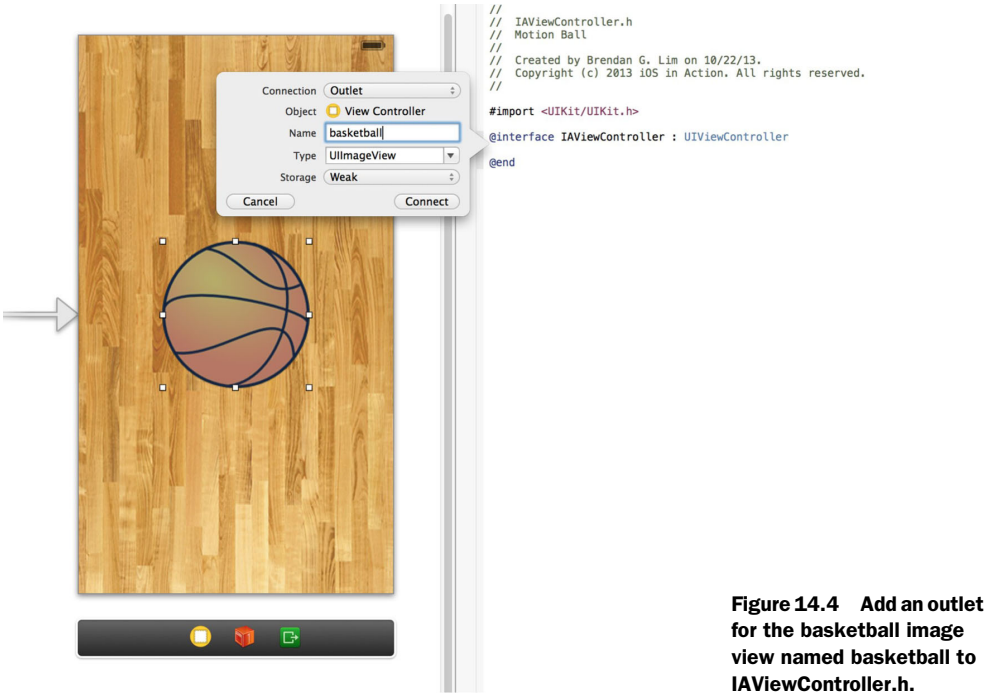

Figure 14.4 Add an outlet for the basketball image view named basketball to IAViewController.h.

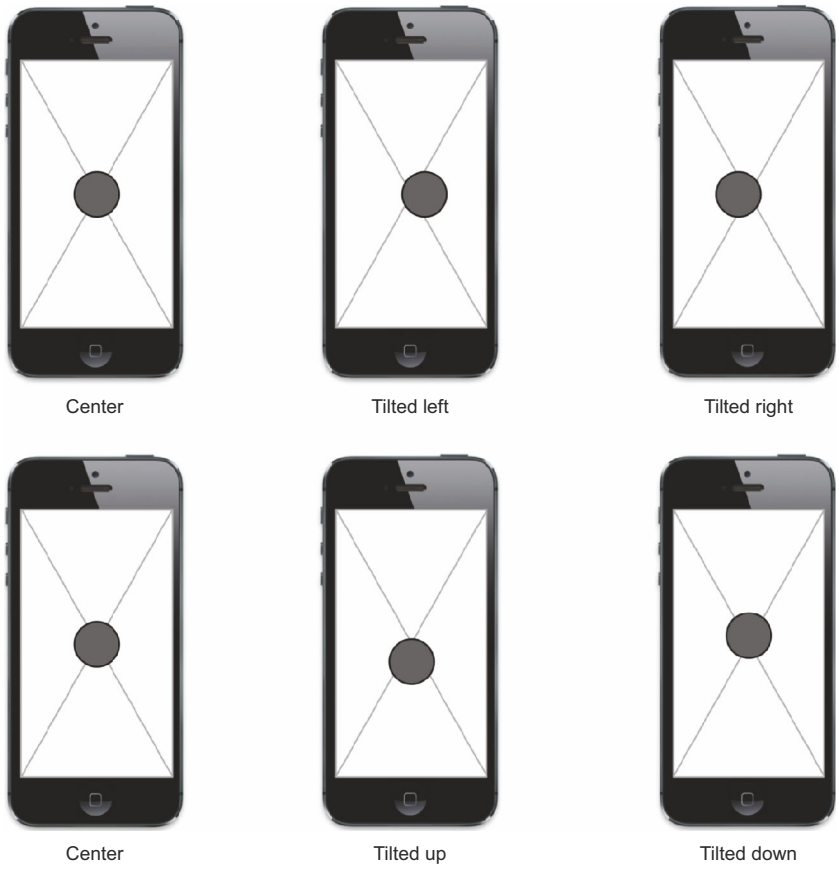

Figure 14.5 Demonstrating the parallax effect on a circle, which changes location when the device is tilted horizontally or vertically

In this figure the motion effect is applied to the dark gray circle. The background is used as a reference point to demonstrate the way the ball is positioned as you move the device to the left/right or up/down.

To create these effects there's a new class in the UIKit framework called `UIMotionEffect`, which serves as an abstract class. This means you can't use this class directly to create your own motion effects, but you can subclass it. Luckily there's an out-of-the-box class called `UIInterpolatingMotionEffect` that you can use to create the parallax effect.

To create a `UIInterpolatingMotionEffect` you can use the `initWithKeyPath:type:` method. The first parameter, the key path, is used to specify on which axis on the view you'd like to apply the effect. Normally this is done on the center point of the view. For instance, if you were to apply a horizontal motion effect, you'd apply it to the center.x key path. For a vertical effect you'd use center.y. The second parameter in the `initWithKeyPath:type:` method specifies the type of motion to track,

which is represented by two constants, `UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis` for horizontal tracking and `UIInterpolatingMotionEffectTypeTiltVerticalAxis` for vertical.

Once a `UIInterpolatingMotionEffect` instance is created, you can add a maximum value and a minimum value that are used to specify the amount a view should move when a device is tilted horizontally all the way to the left or to the right. This is accomplished by specifying an `NSNumber` for the `maximumRelativeValue` and `minimumRelativeValue` properties.

Add the parallax effect to your app so that the basketball's position changes when the device is tilted horizontally or vertically. Jump into IAViewController.m and add the following method.

Listing 14.1 Add parallax effect to basketball

```
- (void) addParallaxEffect
{
    UIInterpolatingMotionEffect *horizontalEffect =
➥ [[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.x"
➥ type:UIInterpolatingMotionEffectTypeTiltAlongHorizontalAxis];

    UIInterpolatingMotionEffect *verticalEffect =
➥ [[UIInterpolatingMotionEffect alloc] initWithKeyPath:@"center.y"
➥ type:UIInterpolatingMotionEffectTypeTiltAlongVerticalAxis];

    verticalEffect.maximumRelativeValue = @(20);
    verticalEffect.minimumRelativeValue = @(-20);

    horizontalEffect.maximumRelativeValue = @(20);
    horizontalEffect.minimumRelativeValue = @(-20);

    [self.basketball addMotionEffect:verticalEffect];
    [self.basketball addMotionEffect:horizontalEffect];
}
```

1 Create vertical motion effect.
2 Create horizontal motion effect.
3 Set vertical maximum relative value.
4 Set vertical minimum relative value.
5 Set horizontal maximum relative value.
6 Set horizontal minimum relative value.
7 Add vertical motion effect to basketball.
8 Add horizontal motion effect to basketball.

Here you first create a vertical effect using the key path center.y so that the vertical effect is applied to the y-axis ❶. You then create a horizontal effect on the x-axis ❷. Next, you set the maximum ❸ and minimum ❹ values to 20 points for the vertical effect. Then you set the same for the maximum ❺ and minimum ❻ on the horizontal effect. Lastly you add the motion effect for vertical ❼ and horizontal ❽ to the `UIImageView` that represents your basketball.

The last thing to do is to make a call to this method within the `viewDidLoad` method:

```
- (void) viewDidLoad
{
    [super viewDidLoad];
    [self addParallaxEffect];
}
```

You can try this out only if you have a paid developer account because you'll need to run this application on your device to see it in action. When you do run this, you

should see the basketball showcasing the parallax effect when the device is moved vertically or horizontally. Next, you'll be learning about UIKit Dynamics so that you can apply realistic animations to your basketball.

14.3 Using UIKit Dynamics

UIKit Dynamics gives you a way to animate views to produce realistic effects. Previously in iOS 6, in order to create these types of animations you'd need a deep understanding of math, physics, and Core Animation. UIKit Dynamics now allows you to add these types of effects to your apps.

14.3.1 Introduction to UIKit Dynamics

As is probably apparent from its name, UIKit Dynamics is part of the UIKit framework. The top-level class with the physics engine that's responsible for generating the effects you'll be using is the `UIDynamicAnimator` class. This class will accept *behaviors* that are applied to a specific view. Behaviors provide instructions to the animator's physics engine, which in turn applies the effects. These behaviors are represented by the `UIDynamicBehavior` class. Each `UIDynamicBehavior` can be applied to multiple `UIDynamicItems`. What's a `UIDynamicItem`? The `UIDynamicItem` is a protocol that defines a center, bounds, and a two-dimensional transform. Luckily the `UIView` class conforms to the `UIDynamicItem` protocol.

Take a look at figure 14.6 to see how these all relate to each other.

Also, the `UICollectionViewLayoutAttributes` class (you learned about this when reading about collection views) can be used with dynamic behaviors. In this chapter we'll be sticking to applying effects to `UIViews`.

When creating a new `UIDynamicAnimator` instance you'll have to pass in a reference view using the `initWithReferenceView:` method. This reference view will be the top-level view in your view controller, as shown here:

```
UIDynamicAnimator *animator = [[UIDynamicAnimator alloc]
➥ initWithReferenceView:self.view];
```

When using a `UIDynamicBehavior` you can use one of the out-of-the-box subclasses or create your own. The behaviors already created for us will solve almost all of our

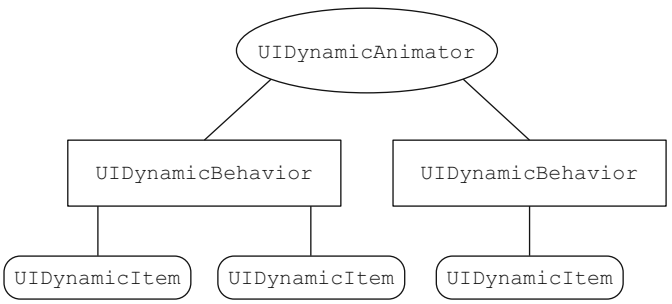

Figure 14.6 The `UIDynamicAnimator` takes in `UIDynamicBehaviors`, which can be applied to multiple objects that conform to the `UIDynamicItem` protocol.

needs, which is one of the reasons why we don't need to be rocket scientists to work with them. Each of them has its own specific behavior, as shown in table 14.1.

Table 14.1 Different types of `UIDynamicBehaviors`

Behavior	Description
`UIAttachmentBehavior`	Connection between two dynamic items
`UICollisionBehavior`	Collision between dynamic items
`UIGravityBehavior`	Gravity effect applied to dynamic items
`UIDynamicItemBehavior`	Effect to match ending velocity of a user gesture
`UIPushBehavior`	Apply force to a dynamic item from a push
`UISnapBehavior`	Snap a dynamic item to a specific point

How about you get started by adding some dynamic behaviors to your basketball? First, open Xcode and add a property to IAViewController.h called `animator`, as shown here:

```
@property (strong, nonatomic) UIDynamicAnimator *animator;
```

Now go to IAViewController.m and set the `animator` property in the `viewDidLoad` method using the `view` property as the reference view:

```
self.animator = [[UIDynamicAnimator alloc]
➥ initWithReferenceView:self.view];
```

Next, you should set the basketball to be able to interact with touch actions, because that's how you're going to be triggering these effects. Add the following to the bottom of `viewDidLoad` as well:

```
[self.basketball setUserInteractionEnabled:YES];
```

Great—you're ready to start adding some awesome behaviors.

14.3.2 Applying the gravity behavior

The gravity behavior is one of the simplest behaviors to use. The goal is to have the dynamic item, the basketball, fall with simulated gravity after it's tapped. To do this, you'll create two new methods, `setupGravity` and `dropBall:` in IAViewController.m. The `setupGravity` method is only to add a gesture recognizer to execute `dropBall:` when the ball is tapped. Add the following code:

```
- (void) setupGravity
{
    UITapGestureRecognizer *tapGesture = [[UITapGestureRecognizer alloc]
➥ initWithTarget:self
➥ action:@selector(dropBall:)];
    [self.basketball addGestureRecognizer:tapGesture];
}
```

Now add the following to the bottom of `viewDidLoad`:

```
[self setupGravity];
```

Finally, you can add the gravity behavior, which is executed when the basketball is tapped. This code is shown in the following listing.

Listing 14.2 Applying gravity to a view on tap

```
- (void) dropBall:(UITapGestureRecognizer *)recognizer
{
    UIGravityBehavior *gravity = [[UIGravityBehavior alloc]
➥ initWithItems:@[self.basketball]];
    [self.animator addBehavior:gravity];
}
```

❶ Create gravity behavior for basketball.

❷ Apply behavior to animator.

All of the magic is done in the two lines found in this method. You first create a new instance of `UIGravityBehavior` as the dynamic item ❶. Then you add the behavior to the animator ❷.

If you run the application now, you'll see the ball fall off the screen as soon as it's tapped. The start and end results are shown in figure 14.7.

It's amazing how this was accomplished with such a small amount of code. It actually took you more lines to set up the tap gesture than it did to simulate the gravity

Figure 14.7 After tapping, the ball will animate and fall off the screen as if gravity was pulling it down. After the animation, the ball will be off the screen.

effect. What if you wanted the ball to fall and come to a stop at the bottom of your view? You can do this by adding a collision behavior.

14.3.3 Applying a collision behavior

Collision behaviors allow you to define how dynamic items should react when they come in contact with one another. This is accomplished by using the `UICollisionBehavior` class. In the case of your basketball, when the gravity behavior is applied, it falls but never stops falling. You can get it to stop at the bottom of the screen by adding a collision behavior. Because the animator was created with knowledge of a reference view—the main view of your view controller—it knows where the imposed boundaries are. You can use this to your advantage in this situation.

The `UICollisionBehavior` class's `translatesReferenceBoundsIntoBoundary` property allows you to tell it to set the bounds of the reference view as the boundary of the behavior by setting it to `YES`. You'll add the collision behavior within the `dropBall:` method, as shown in the next listing.

Listing 14.3 Adding the collision behavior

```
- (void) dropBall:(UITapGestureRecognizer *)recognizer
{
    UIGravityBehavior *gravity = [[UIGravityBehavior alloc]
➥ initWithItems:@[self.basketball]];
    [self.animator addBehavior:gravity];

    UICollisionBehavior *collision = [[UICollisionBehavior alloc]
➥ initWithItems:@[self.basketball]];
    collision.translatesReferenceBoundsIntoBoundary = YES;
    [self.animator addBehavior:collision];
}
```

❶ Create the collision behavior.

❷ Set reference view bounds as boundary.

❸ Apply behavior to animator.

Here you're adding three lines to add the collision behavior. First, you create a new `UICollisionBehavior` with the basketball view as the dynamic item ❶. Next, you set the bounds of the reference view as the boundary for the behavior ❷. Lastly, you add the behavior to the animator ❸.

Run the application and tap the basketball. You should see that the ball lightly bounces as soon as it hits the bottom boundary of the reference view. The start and finish positions are shown in figure 14.8.

One thing you also see in this example is that you can add multiple behaviors to an animator to make a unique effect. Next, you're going to learn how to make this image come to life by making it bounce like a real basketball.

14.3.4 Adding dynamic behavior

You're going to use the `UIDynamicBehavior` class to help with the bounce effect. The `UIDynamicBehavior` has a property that specifies the elasticity of an object that helps makes this possible. This property accepts a `CGFloat` that ranges from 0.0 (no bounce) to 1.0 (continuous elasticity).

Figure 14.8 After tapping it, the ball will fall with the same gravity behavior but will come to a light stop at the bottom of the reference view due to the collision behavior you added.

Also, a basketball never falls straight down without rotating. The `UIDynamicBehavior` class lets you specify angular and linear velocities for a specific dynamic item. You'll be adding an angular velocity by using the `addAngularVelocity:forItem:` method. This will be used to give the ball a little rotation when it collides with the boundaries of the reference view.

To simulate friction there's a `friction` property that takes values from 0.0 (no friction) to 1.0 (strong friction). Also, to give the object realistic density to simulate mass, there's even a `density` property that you can specify. The density specified is directly related to the pixel size of the dynamic item that it's applied to. For example, an object that is 100 x 100 with a 1.0 density value will accelerate to 100 points per second^2.

Update the `dropBall:` method as shown in the following listing.

Listing 14.4 Adding dynamic behavior

```
- (void) dropBall:(UITapGestureRecognizer *)recognizer
{
    UIGravityBehavior *gravity = [[UIGravityBehavior alloc]
➥ initWithItems:@[self.basketball]];
    [self.animator addBehavior:gravity];
```

```
    UICollisionBehavior *collision = [[UICollisionBehavior alloc]
➥ initWithItems:@[self.basketball]];
    collision.translatesReferenceBoundsIntoBoundary = YES;
    [self.animator addBehavior:collision];

    UIDynamicItemBehavior *bounce = [[UIDynamicItemBehavior alloc]
➥ initWithItems:@[self.basketball]];
    [bounce addAngularVelocity:0.2f forItem:self.basketball];
    bounce.elasticity = 0.8f;
    bounce.friction = 0.2f;
    bounce.density = 0.4f;
    [self.animator addBehavior:bounce];
}
```

❶ Create dynamic item behavior.

❷ Add angular velocity to basketball.

❸ Specify elasticity amount.

❹ Specify friction amount.

❺ Specify density of basketball.

❻ Add behavior to animator.

Here you're creating a new dynamic item behavior for the basketball as the dynamic item ❶. Then you're adding an angular velocity of 0.2 ❷, elasticity of 0.8 ❸, friction of 0.2 ❹, and density of 0.4 ❺. Lastly you're adding the behavior to the animator ❻.

When you run the app and tap the basketball, you'll notice that it rotates as it bounces and returns to its normal position as it comes to a stop at the bottom right of the screen. Figure 14.9 shows you how the ball ends up after the animator has finished.

You're encouraged to play with the values for angular velocity, elasticity, friction, and density to see just how they affect the animation.

Figure 14.9 After you tap the ball, the dynamic item behavior will cause the ball to rotate and fall based on the angular velocity, elasticity, friction, and density that you added.

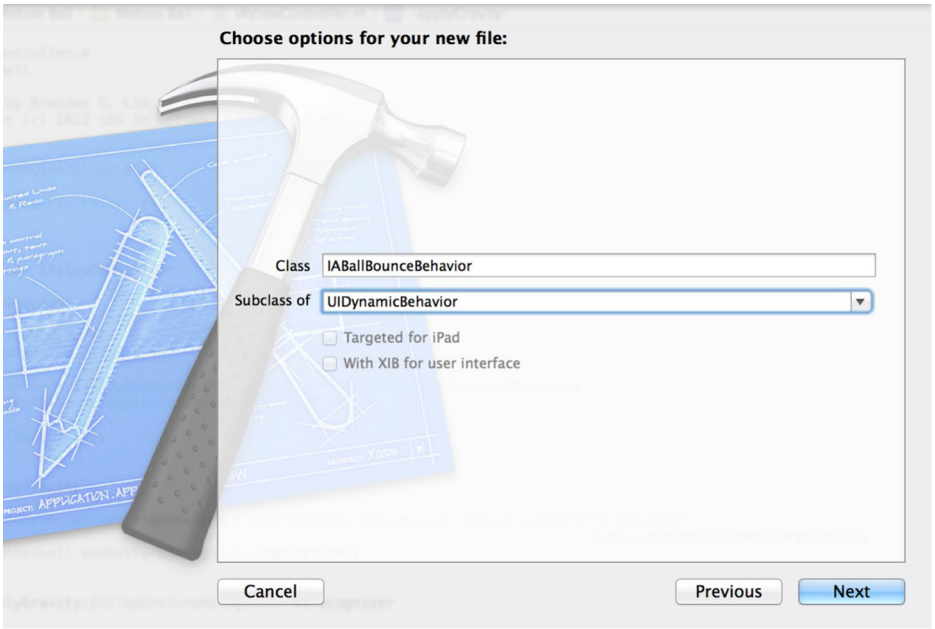

Figure 14.10 Create a subclass of `UIDynamicBehavior` called `IABallBounceBehavior`.

14.3.5 Creating a custom UIDynamicBehavior subclass

If you want to package all of these behaviors into one class, it's easy to do by creating a subclass of `UIDynamicBehavior`. First, go to the project navigator and create a new file in the Motion Ball group called IABallBounceBehavior that's a subclass of `UIDynamicBehavior`, as shown in figure 14.10.

After the new file's been created, open IABallBounceBehavior.h and add the following method declaration:

```
-(id) initWithItems:(NSArray *)items;
```

Next, open IABallBounceBehavior.m. Here you'll be adding all of the behaviors that you've just created, but instead of adding them to an animator, you'll be adding them as child behaviors. Add the code shown in the following listing.

Listing 14.5 Custom ball bounce behavior

```
-(id) initWithItems:(NSArray *)items
{
    if (self = [super init]) {
        UIGravityBehavior *gravity = [[UIGravityBehavior alloc]
➥ initWithItems:items];
        [self addChildBehavior:gravity];

        UICollisionBehavior *collision = [[UICollisionBehavior alloc]
➥ initWithItems:items];
```

1 Create gravity behavior.

2 Add gravity as child behavior.

3 Create collision behavior.

```
        collision.translatesReferenceBoundsIntoBoundary = YES;
        [self addChildBehavior:collision];

        UIDynamicItemBehavior *dynamic = [[UIDynamicItemBehavior alloc]
➥ initWithItems:items];
        for (UIView *item in items)
            [dynamic addAngularVelocity:0.2f forItem:item];

        dynamic.elasticity = 0.8f;
        dynamic.friction = 0.2f;
        dynamic.density = 0.4f;
        [self addChildBehavior:dynamic];
    }

    return self;
}
```

Add collision as child behavior. ❹

❺ **Create dynamic item behavior.**

❻ **Apply angular velocity to all items passed in.**

❼ **Add dynamic item behavior as child behavior.**

This should look extremely familiar. All you're doing is adding the custom behaviors you've already created into an initializer for the `IABallBounceBehavior` class. You first create the gravity behavior ❶ and add it as a child behavior ❷. Then you create the collision behavior ❸ and add that as a child behavior ❹. Lastly you create the dynamic item behavior ❺, apply velocity to all items in the items array ❻, and then add that as a child behavior ❼.

Next, open IAViewController.m. You can replace all of the dynamic behaviors with your new `IABallBounceBehavior`. First, you'll need to import the new class by adding the following to the top of the `IAViewController` class:

```
#import "IABallBounceBehavior.h"
```

Finally, you can replace all of the code within `dropBall:` with the two lines shown here:

```
IABallBounceBehavior *ballBounce = [[IABallBounceBehavior alloc]
➥ initWithItems:@[self.basketball]];
[self.animator addBehavior:ballBounce];
```

This shows how simple it is to wrap different dynamic behaviors to achieve a desired effect in one single behavior.

14.4 Summary

There are so many things you can do with motion effects and dynamics, and you've only just skimmed the surface. There literally are endless possibilities, even with dynamics alone. On top of using both of these, you can also combine Core Animation and gestures to make something truly unique. All of these together, if used wisely, can simulate realistic and fun animations that can help provide a level of delight for your users as they use your applications.

- You can use motion effects to provide a parallax effect to your views.
- The parallax effect works by using data from the accelerometer and gyroscope when a device is tilted horizontally or vertically.
- To be able to test motion effects, you need to run the applications on a physical device.

- UIKit Dynamics provides realistic animations without the need of a complex understanding of math and physics.
- Many dynamic behaviors can be used out of the box, such as gravity, collisions, and more.
- You can add multiple dynamic behaviors as child behaviors into a single `UIKitDynamicBehavior` subclass.

iOS DEVELOPMENT

iOS 7 IN ACTION

Lim • Mac Donell

To develop great apps you need a deep knowledge of iOS. You also need a finely tuned sense of what motivates 500 million loyal iPhone and iPad users. iOS 7 introduces many new visual changes, as well as better multitasking, dynamic motion effects, and much more. This book helps you use those features in apps that will delight your users.

iOS 7 in Action is a hands-on guide that teaches you to create amazing native iOS apps. In it, you'll explore thoroughly explained examples that you can expand and reuse. If this is your first foray into mobile development, you'll get the skills you need to go from idea to app store. If you're already creating iOS apps, you'll pick up new techniques to hone your craft, and learn how to capitalize on new iOS 7 features.

What's Inside

- Native iOS 7 design and development
- Learn Core Data, AirPlay, Motion Effects, and more
- Create real-world apps using each core topic
- Use and create your own custom views
- Introduction and overview of Objective-C

This book assumes you're familiar with a language like C, C++, or Java. Prior experience with Objective-C and iOS is helpful.

Brendan Lim is a Y Combinator alum, the cofounder of Kicksend, and the author of *MacRuby in Action*.
Martin Conte Mac Donell, aka fz, is a veteran of several startups and an avid open source contributor.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/iOS7inAction

"A practical journey through the iOS 7 SDK."
—Stephen Wakely
Thomson Reuters

"A kickstart for newbs and a deft guide for experts."
—Mayur S. Patil
Clearlogy Solutions

"Mobile developer: don't you dare not read this book!"
—Ecil Teodoro, IBM

"The code examples are excellent and the methodology used is clear and concise."
—Gavin Whyte
Verify Data Pty Ltd

"Everything you need to know to ship an app, and more."
—Daniel Zajork
API Healthcare Corporation

ISBN 13: 978-1-617291-42-5
ISBN 10: 1-617291-42-0
54499
9 781617 291425

MANNING $44.99 / Can $47.99 [INCLUDING eBOOK]