MOBILE PROGRAMMING SERIES

iOS UICOLLECTIONVIEW

THE COMPLETE GUIDE

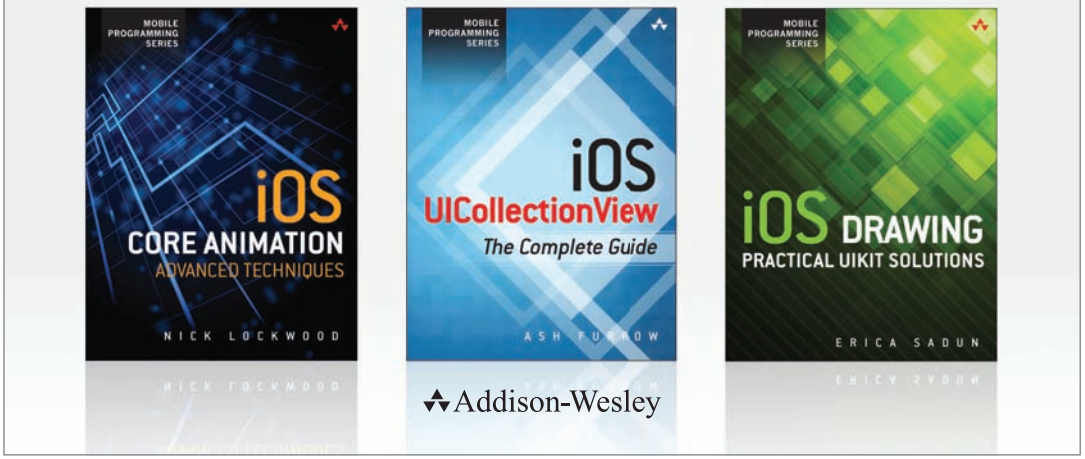
SECOND EDITION

ASH FURROW

iOS UICollectionView: The Complete Guide

Second Edition

Addison-Wesley Mobile Programming Series



Visit **informit.com/mobile** for a complete list of available publications.

The **Addison-Wesley Mobile Programming Series** is a collection of digital-only programming guides that explore key mobile programming features and topics in-depth. The sample code in each title is downloadable and can be used in your own projects. Each topic is covered in as much detail as possible with plenty of visual examples, tips, and step-by-step instructions. When you complete one of these titles, you'll have all the information and code you will need to build that feature into your own mobile application.

Make sure to connect with us!
informit.com/socialconnect

informIT.com
the trusted technology learning source | Addison-Wesley | Safari Books Online

iOS UICollectionView: The Complete Guide

Second Edition

Ash Furrow

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

iOS UICollectionView: The Complete Guide, Second Edition

ISBN-13: 978-0-13-376261-7
ISBN-10: 0-13-376261-0

Acquisitions Editor
Trina MacDonald

Development Editor
Sheri Cain

Managing Editor
Kristy Hart

Project Editor
Andy Beaster

Copy Editor
Keith Cline

Proofreader
Paula Lowell

Technical Editor
Niklas Saers

Publishing Coordinator
Olivia Basegio

Cover Designer
Chuti Prasertsith

❖

For my wife, who inspires me in every way.

❖

Table of Contents

Acknowledgments

I want to thank Angie Doyle and Trina MacDonald at Pearson publishing for contacting me about writing this book. I was planning on writing an ebook about something, but with their guidance and resources, I know this book is way more awesome than anything I could have done on my own.

Rich Wardwell and Niklas Saers have been wonderful technical editors, offering comprehensive advice concerning clarity of both my code and my prose.

I am a strong believer in the open source community, and this book relies on some open source software. Some of it I wrote myself, but some if I didn't. I'd like to thank Mark Pospesel for his contributions to GitHub in "Introducing UICollectionViews." Mark specializes in mathematics, and while writing this book, it's been great to be able to rely on his expertise.

Speaking of the open source community, no book discussing `UICollectionView` would be complete without a tip of the hat to Peter Steinberger's work on PSTCollectionView, a 100% API-compatible replacement for `UICollectionView` that offers backward compatibility with iOS 4.3+. Most of the techniques discussed in this book are directly applicable to PSTCollectionView, and the project is advancing every day. If you need to support older versions of iOS, use PSTCollectionView.

Finally, I could not have completed this book without the support of my wife. Her constant prodding about deadlines made sure I was only a little late most of the time. I am lucky to have such a supportive partner who understands and encourages my compulsion to create and share.

About the Author

Ash Furrow has been developing iOS applications since 2009. He's made several of his own applications available on the store and headed the iOS team at 500px to ship their critically acclaimed app. After spending a year with Teehan+Lax, he's moved on to live abroad and contribute to the open source community.

When he's not busy writing books or blog posts, Ash enjoys digital and analogue photography, often developing his own film.

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: trina.macdonald@pearson.com
Mail: Reader Feedback
Addison-Wesley's Developer's Library
800 East 96th Street
Indianapolis, IN 46240 USA

Preface

At WWDC 2012, Apple unveiled UICollectionView, enabling a new way for apps to render content to users. Collection views are a content- and layout-agnostic tool for developers to display content in apps. User interfaces created with collection views are some of the most immersive, distinctive interfaces in iOS applications.

However, the power afforded to developers by collection views is balanced by the complexity of using them. As the saying goes, Cocoa makes common things easy and uncommon things possible. UICollectionView embodies this sentiment.

I said earlier that collection views are layout-agnostic, and that's true: Developers write their own layouts for collection views to use to organize their content on the screen. Luckily, Apple included a sample layout that displays grids, a common request among developers.

How to Use This Book

This book is meant to tell a story; each chapter builds upon the last one to guide readers through every nook and cranny of UICollectionView. I strongly encourage readers to read each chapter in sequence and follow along with the code samples.

The first chapter makes sure that readers have a common vocabulary when discussing the organization of code in iOS applications. Even if you're a seasoned developer, it's worth a look just to make sure you're on the same page as I am.

The code provided with this book is as valuable as the explanations in the chapters of *why* the code is written the way it is.

All of the code that appears in this book can be downloaded at http://ashfurrow.com/uicollectionview-the-complete-guide/.

Who This Book Is For

This book is for intermediate to advanced iOS developers who want to take full advantage of UICollectionView. If you're trying to write your first-ever iOS application, this book probably isn't for you. I've written this book with the assumption that you understand the concepts of objects and view hierarchies, as well as basic Objective-C syntax.

Organization of This Book

This book is organized into six chapters to guide readers through a comprehensive description of every aspect of collection views:

- **Chapter 1, "Understanding Model-View-Controller on iOS,"** briefly introduces the MVC paradigm of application architecture that's used throughout the remainder of the book.
- **Chapter 2, "Displaying Content Using `UICollectionView`,"** introduces readers to UICollectionView with some basic examples using .xib files and storyboards, as well as view setup using only code. This chapter ends with a case study on application performance tuning.
- **Chapter 3, "Contextualizing Content,"** builds on the basics of cell use from Chapter 2 to explain how to contextualize content for users by using supplementary views. The chapter explores the `UICollectionViewDataSource` and `UICollectionViewDelegate` protocols as well.
- **Chapter 4, "Organizing Content with `UICollectionViewFlowLayout`,"** introduces readers to the idea of creating their own custom layouts while relying on existing logic in `UICollectionViewFlowLayout`. The sample code from Chapter 3 is augmented with decoration views, and custom collection view attributes are used to customize cell layout. The chapter ends with a look at a Cover Flow-esque layout.
- **Chapter 5, "Crafting Custom Layouts Using `UICollectionViewLayout`,"** explains to readers who understand subclassing flow layouts that they can subclass `UICollectionViewLayout` directly for incredibly custom layouts. The chapter also covers changing layouts with animation support, as well as provides some further examples on how to use supplementary views and decoration views with completely custom layouts.
- **Chapter 6, "Adding Interactivity to `UICollectionView`,"** is the crown jewel of this book. It looks back at all the previous chapters' code samples to augment them with interactivity, mostly using gesture recognizers. Additionally, it shows off how to use UIKit Dynamics, a new animation library in iOS 7.

What's New in the Second Edition

The second edition of this book covers what's new in `UICollectionViews` in iOS 7. It removes some gotchas that were present in iOS 6 but were fixed in iOS 7, and it details a few new ones. This book also covers how to use `UICollectionViews` with UIKit Dynamics, an exciting new iOS 7 technology.

Special Thanks

I want to thank Mark Pospesel for his work in the open-source community, specifically his contributions to "Introducing UICollectionViews" available on GitHub: https://github.com/mpospese/IntroducingCollectionViews. A lot of the math in the later chapters is taken from Mark's code. This book would not be as awesome if it weren't for Mark's open source contributions.

1

Understanding Model-View-Controller on iOS

Before you dive into `UICollectionView`, you should get familiar with some of the conventions and terms used in this book. The book starts with the basics of the iOS application lifecycle and then discusses the Model-View-Controller (MVC) paradigm. Even if you're an experienced iOS developer already familiar with these topics, I encourage you to read this chapter to make sure that you're on the same page (or screen, so to speak) that I am while you're reading the rest of this book.

Basics of the Application Lifecycle

The iOS application lifecycle differs a little from typical native applications on other platforms (although recent changes to OS X show Apple is interested in making the iOS lifecycle the norm). Developers no longer have hard-and-fast rules for when their applications are terminated, suspended, and so on. Let's start with a simple scenario to describe a typical application lifecycle.

The user has just turned on his phone, and no applications are running except for those that belong to the operating system. Your application is *not* running. After the user taps your app's icon, Springboard—the part of the OS that operates the Home screen of iOS—launches your app. Your app, and the shared libraries it needs to execute, is loaded into memory while Springboard animates your `Default.png` on the screen. Eventually, your app begins execution, and your application delegate receives the appropriate notification. When your application is running and in the foreground, it is in the **active** state.

On iOS, users tend to only use any given application for a few seconds before returning their phones to their pockets. After the user has put away your app by pressing the Home button on her iPhone or iPad, your application enters the **background** state. Typically, apps have 10 seconds to complete any database saves or other long-running tasks (though applications can request additional time from the OS). When all the background processing

is complete, the application finally becomes **suspended**. While suspended, applications remain in memory but may not execute code. The state of your application is persisted. If the user opens your application while it is suspended, it begins execution exactly where it left off. If memory becomes low, the OS can kill your app while it is in the suspended state. The user can also manually terminate your app from the multitasking tray. Once terminated, applications return to their initial state of not running.

But wait, it gets more complicated! If the user receives a calendar alert, opens the multitasking tray, or gets a phone call, your application can be put into the **inactive** state. Your application is still running, but it is no longer the foremost thing the user interacts with. For example, games pause themselves. As an application developer, you need to be aware of this and use it as an indication that the user might leave your application soon.

The user can open your application without tapping its icon on the Home screen. If your application receives local or push notifications, or if it is registered for custom URL scheme handling, the user can open it in any number of ways.

The application lifecycle is important to understand for all iOS developers who want to make enriched, immersive experiences. These types of applications are exactly what `UICollectionView` is great for, so no comprehensive discussion of `UICollectionView` would be complete without a summary of the application lifecycle.

If your app enters the inactive state, stop updating your interface. It would be disconcerting for a user to see your collection-view contents move about while he's deciding whether to view the details of an appointment that has popped up over your application. Likewise, don't update your app's interface while the application is in the background. The state of the user interface should remain fixed between the switch from active to background and back to active.

How to Use MVC

MVC is not a difficult concept, but there are two main reasons for emphasizing its importance in iOS:

- MVC is used by CocoaTouch (and Cocoa on OS X). If you adhere to the same paradigm as the frameworks used for writing all iOS applications, your code will flow well and not clash with the built-in classes, including `UICollectionView`.
- MVC is generally a good framework, and using it will help you make well-written, maintainable apps.

Now that you know why MVC is important, it's time to look at what MVC is. Figure 1.1 shows the basics of MVC; strong relationships are represented with solid lines, and weak relationships are represented by dashed ones. Strong and weak relationships indicate to the compiler how to manage memory and are important to avoid memory leaks, which would eventually lead to the app being terminated.

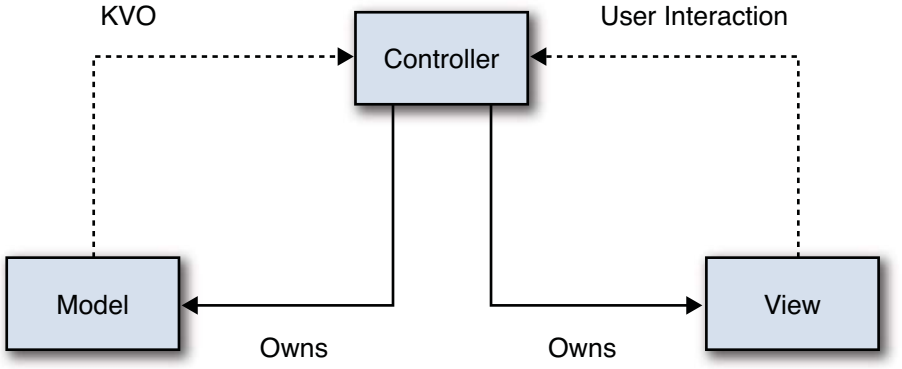


Figure 1.1 Basics of MVC

At the heart of MVC is the controller object. The controller is a view controller—as in `UIViewController`—and it controls the view. It maintains a strong relationship to this view, which is what is presented to the user on the screen. The controller also maintains a strong relationship to the model. The model represents data that is represented in the view.

If your view ever has a reference to your model, or vice versa, you're doing it wrong. This book uses MVC and you should, too.

Most of the code in any given application resides in the controller; controllers mediate the interactions between views and models, which is why the code in controllers is often referred to as *glue code*.

What sort of interactions does a controller mediate? Well, if the view contains a button, the view controller is notified when the user taps that button. User interactions usually trigger actions to modify, create, or delete models belonging to the controller. The controller receives the user interaction from the view, updates the model, and then updates the view to reflect the changes made to the model.

Sometimes, the model changes without user interaction. For example, consider a view that displays a large JPEG, which is being downloaded. When the download completes, the controller should be notified so that it can update the view. On iOS, you have a few different choices for how to notify the controller. My favorite is Key-Value Observation (KVO). Controllers can register themselves as observers on model objects so that they are notified whenever the model's properties are changed. Other ways for models to interact with controllers on iOS include `NSNotificationCenter`, `delegation`, and `NSFetchedResultsController`. I would avoid `NSNotificationCenter` for model-controller interaction in favor of `NSFetchedResultsController` or KVO. Although this book doesn't discuss Core Data, `UICollectionView` works very well with `NSFetchedResultsController` in a similar way to `UITableViewController`.

This last example demonstrates a gaping hole in MVC: Where does the network code go? As a responsible iOS developer, you should keep the view controller to only mediating the interactions between the view and the model. If that's the case, it shouldn't be used to

house the network access code. As discussed in Chapter 6, "Adding Interactivity to `UICollectionView`," the network code should be placed *outside* of the typical MVC pyramid. Network access should not involve the view whatsoever, but it can sometimes involve the model.

Well, that's *mostly* true. In fact, a common paradigm for fetching details about a model from an application programming interface (API) involves Grand Central Dispatch blocks. A block lets developers treat anonymous functions as first-class Objective-C objects. These blocks can be invoked later. Controllers can start a network request and pass the network-fetching object a callback block that updates the view. Technically, the network code has an indirect reference to the view, but you ignore it lest you find yourself falling down a rabbit hole of pedantry.

If you are experienced in iOS development, all of this should sound familiar. `UICollectionView` and `UICollectionViewController` don't exist in silos; they are used within applications with models and with the rest of CocoaTouch. It would be irresponsible to present them in any other context than that of MVC.

MVC and `UICollectionView`

Now that you've read about the MVC paradigm, look at its application in the context of writing `UICollectionView` code.

The view component of MVC with `UICollectionView` is unsurprisingly the `UICollectionView` itself; the controller is either a subclass of `UICollectionViewController` or a subclass of `UIViewController` that conforms to the `UICollectionViewDataSource` and `UICollectionViewDelegate` protocols; the model can be anything.

Like with `UITableView`, your controller can either subclass `UIViewController` and conform to the two protocols for the collection view data source and delegate or it can subclass `UICollectionViewController` itself. If you look in the header file of `UICollectionViewController`, you see that it's very sparse. The controller inherits from `UIViewController`—conforming to `UICollectionViewDataSource` and `UICollectionViewDelegate`—and has a convenience initializer to programmatically create an instance of it using a collection view with a specific layout. It contains a property to access the collection view and another property to specify whether the selection in a collection view becomes cleared when it (re)appears.

When using a `UICollectionViewController` subclass, the `view` property of `UIViewController` points to the same object as the `collectionView` property of `UICollectionViewController`. The view *is* the collection view. If you plan to use only `UICollectionView` to display data to your user, I strongly recommend subclassing this prebuilt controller. In my experience, you run into fewer "gotchas" using these special controllers from Apple.

In some circumstances, subclassing `UIViewController` is preferable. For example, if your view *contains* a collection view, but also contains other views, it's easier to have the collection view as a subview of the controller's view. The distinction is minor, but important.

Figures 1.2 and 1.3 demonstrate the differences in the two approaches to using collection views. `UICollectionViewController` is much simpler; it should be the approach you take first. If you find you can't solve your problem with it, switch to using the second approach. It's usually easy to switch from using the first method to the second.

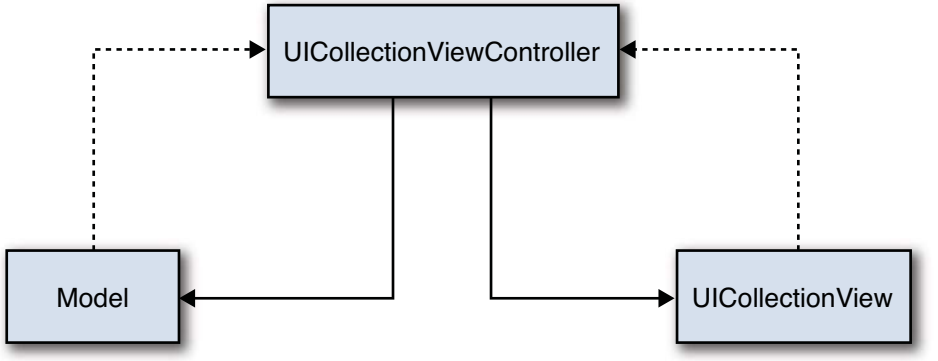


Figure 1.2 Example of MVC using `UICollectionViewController`

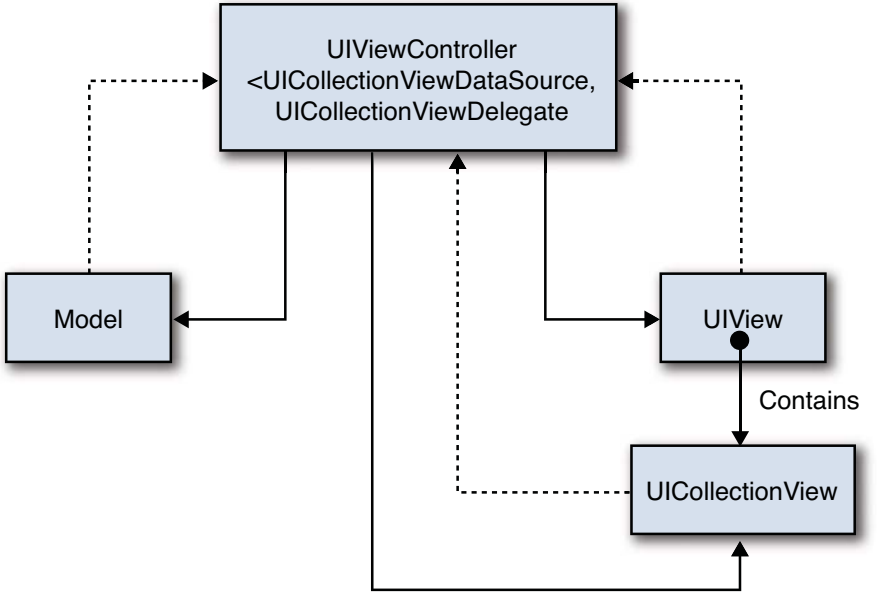


Figure 1.3 Example of MVC using `UICollectionView`'s protocols

This book uses the first approach unless there is a good reason not to. Even though the `view` property of `UICollectionViewController` is the same as its `collectionView` property, the code used in this book carefully distinguishes between the two.

Now that you've seen how collection views fit within the MVC paradigm of iOS apps, look at the following simple example. Don't worry; you experiment a lot with collection views in Chapter 2, "Displaying Content Using `UICollectionView`."

In the following example, you create a simple iPhone app that displays a bunch of cells with random colors. To get started, create a new application with the Single View template. Make sure that Use Storyboards is *unchecked;* this book focuses on collection views, and I don't want to have to diverge to discuss the peculiarities of storyboards. Delete everything in the view controller header file and replace it with the code in Listing 1.1.

Listing 1.1 Basic `UICollectionViewController` Header File

```
@interface AFViewController : UICollectionViewController

@end
```

Replace `AFViewController` with the name of your view controller. My initials are AF, so I prefix my class names with them to avoid namespace collisions.

Next, head over to your .xib file and delete the view. Drag a collection view onto the blank canvas and connect the collection view's `delegate` and `dataSource` outlets to the File's Owner, the view controller. It should look like Figure 1.4 when you're done.

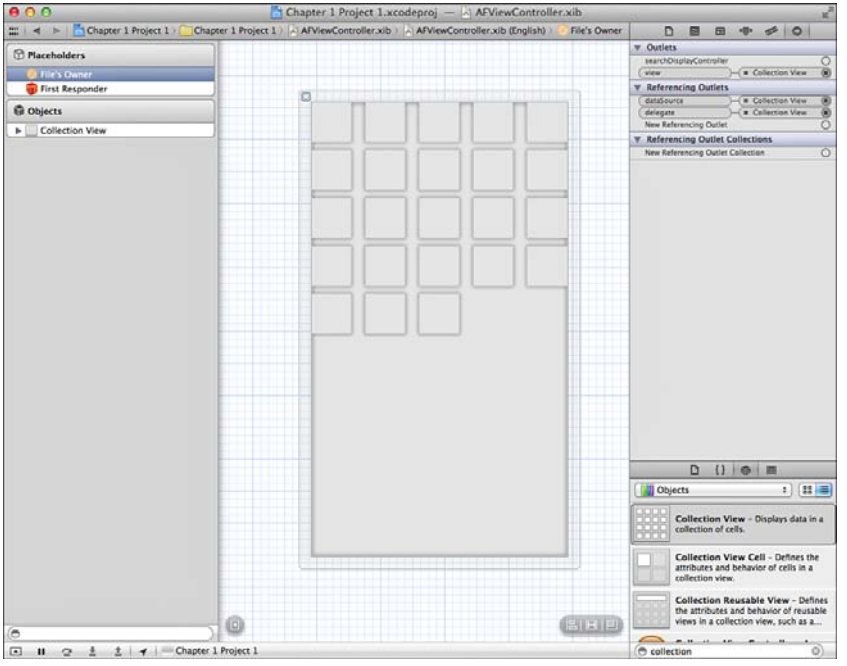


Figure 1.4 Basic `UICollectionView` setup using a .xib

Now comes the fun part: the code! `UICollectionViewDataSource` has two required methods. One returns the number of items in a section, and another configures a cell for a given index path.

If you're not familiar with these terms, don't worry. Chapter 2 explains everything in great detail. This quick example just gets your feet wet.

Following MVC, you need a model. Use a basic array that you'll populate with a bunch of randomly generated colors. The top of your implementation file should look something like Listing 1.2.

Listing 1.2 Setting Up the Model

```
static NSString *kCellIdentifier = @"Cell Identifier";

@implementation AFViewController
{
    NSArray *colorArray;
}

- (void)viewDidLoad
{
```

```
    [super viewDidLoad];

    [self.collectionView registerClass:[UICollectionViewCell class]
forCellWithReuseIdentifier:kCellIdentifier];

    const NSInteger numberOfColors = 100;

    NSMutableArray *tempArray = [NSMutableArray
arrayWithCapacity:numberOfColors];

    for (NSInteger i = 0; i < numberOfColors; i++)
    {
        CGFloat redValue = (arc4random() % 255) / 255.0f;
        CGFloat blueValue = (arc4random() % 255) / 255.0f;
        CGFloat greenValue = (arc4random() % 255) / 255.0f;

        [tempArray addObject:[UIColor colorWithRed:redValue green:greenValue
blue:blueValue alpha:1.0f]];
    }

    colorArray = [NSArray arrayWithArray:tempArray];
}
```

Notice the copy of the array; we're doing so to avoid a mutable instance as our color array, which would be unnecessarily slower.

The `kCellIdentifier` string is used to register a plain `UICollectionViewCell` as the cell for the collection view to use, so don't pay much attention to it. The part that involves the model is the instance variable called `colorArray`. In `viewDidLoad`, you use a `for` loop to populate this array with random colors.

Now that you have the model set up, you need to configure your view to represent it. For this, use the two `UICollectionViewDataSource` methods mentioned earlier (see Listing 1.3).

Listing 1.3 Configuring the View

```
-(NSInteger)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section
{
    return colorArray.count;
}

- (UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
```

```
    UICollectionViewCell *cell = [collectionView
dequeueReusableCellWithReuseIdentifier:kCellIdentifier forIndexPath:indexPath];
//Discussed in Chapter 2 - pay no attention

    cell.backgroundColor = colorArray[indexPath.item];

    return cell;
}
```

The first method—`collectionView:numberOfItemsInSection:`—lets the collection view know how many cells it's going to display. You rely on the model to let the controller know what number to return. Next is `collectionView:cellForItemAtIndexPath:`, which returns a cell that you are responsible for configuring in a way that represents your model. To do this, you grab the model at the given index and use that color as the background color for the cell. If you run the app, you get something like what you see in Figure 1.5. Because the colors are randomly generated, of course, your app will look different.

Figure 1.5 First run of the basic app

Note that we're not using this collection view within a `UINavigationController`, so the status bar is transparent. In production code on iOS 7+, you'll usually encapsulate your collection view within a navigation controller, whose navigation bar is extended behind the status bar.

So, this simple example demonstrates how a model can represent a view and how you can configure a view to represent that model without either being aware of the other. This example demonstrates the platonic ideal of what you should strive for: clear separation between model, view, and controller.

2

Displaying Content Using `UICollectionView`

Now that you understand how collection views fit within an iOS app using the Model-View-Control (MVC) paradigm, it's time to get to the good stuff: code. This chapter starts off easy and shows how you can use storyboards or .xibs to set up collection views, and then it shows you how to set them up in code. Collection views extend their `UIScrollView` superclass, so the chapter takes a brief detour to show how to use that to your advantage with `UIScrollViewDelegate`. You begin customizing actual content to show to your users using cell reuse before finishing off with a case study on performance.

Setting Up Using Code and Storyboards

Traditionally, .xib files were used to lay out interface code for OS X and iOS apps. These files are "freeze-dried" versions of your interface that are thawed at runtime. The benefit of .xibs is that they're easy to use to create basic interfaces; you usually have one instance of `UIViewController` per .xib.

Storyboards, first introduced in iOS 5 in 2011, enable developers to visually lay out the interaction *between view controllers.* Not only can developers visualize the connections between view controllers, but they can also define how their entire application transitions from one view controller to another. The key thing about storyboards is their efficiency; a huge .xib file, which has to be completely loaded into memory, can delay the time it takes for your app to launch. Storyboards efficiently lazy-load only the view controllers necessary.

Of course, anything you can do in a .xib file or storyboard can be done using cold, hard code. If you are integrating collection views into your existing application, which uses .xib files or storyboards, it might be convenient to continue to use them. However, because collection views *require* the use of code for layout, it's often easier to avoid using .xibs and

storyboards altogether. Nevertheless, this chapter explains how to set up the collection view from the last chapter using a storyboard and then set it up again using only code.

Create a new Xcode project with the Single View template. Make sure that Use Storyboards is checked. Open the `MainStorboard.storyboard` file and delete the view controller that's already there. Drag a Collection view controller from the object library in the right pane onto the empty canvas, as shown in Figure 2.1.

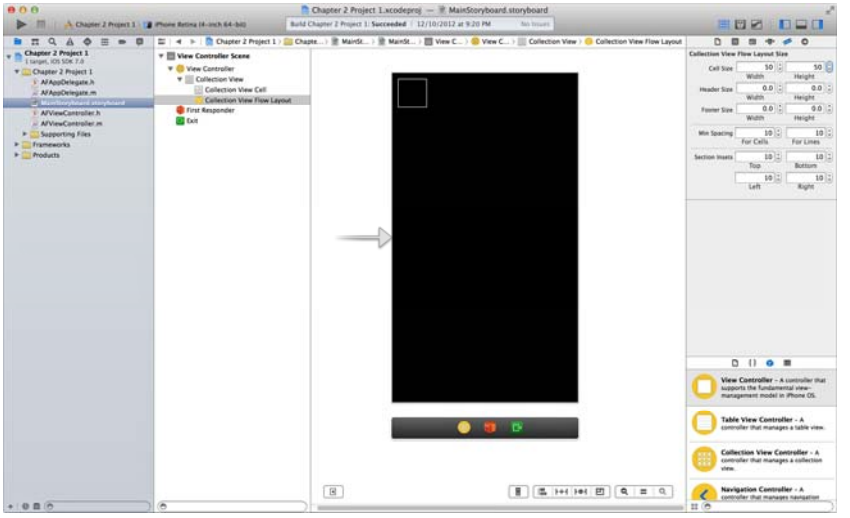


Figure 2.1 Basic collection view using storyboards

You could run the app right now and it would work, but it would be pretty boring. The storyboard has set up the delegate and data source outlets of the collection view to point to your collection view controller. The next step is to customize what that view controller actually does. This part is easy, because you're just going to copy the existing code from Chapter 1, "Understanding Model-View-Controller on iOS."

Open the header for your view controller and change which class it inherits from (by changing `UIViewController` to `UICollectionViewController`). Then copy the implementation file in its entirety from the last chapter. The last, important step is to tell your storyboard which view controller it should use. Click the Collection view controller in the storyboard and open the Identity Inspector. Where it says Class, you see the default placeholder of `UICollectionViewController`. Boring! Replace that with the name of your view controller—in my case, it's `AFViewController`.

This step is crucial; it's how the storyboard knows what code to execute when laying out the collection view. Run your app, and you see the same output as from Chapter 1.

Using storyboards or .xibs, you have an opportunity to change the visual display of the collection view without any code. Select the collection view in the storyboard and open the Attributes Inspector. Here, you can change the scroll direction of the collection view from Vertical, the default, to Horizontal. You can also change properties of the collection view

that belong to its superclass, `UIScrollView`. Change the Style to white, which makes the scroll indicator visible against the black background.

Open the Size Inspector, and you can change the attributes of the collection view layout, shown in Figure 2.2. (Collection views abstract these properties to their layout objects; read more on that in Chapter 3, "Contextualizing Content.") Here, you can change the cell size, which is 50 by 50 points by default. Bump the width down to 20 and keep the height set to 50. The header and footer sizes don't work just yet because you haven't used headers or footers.

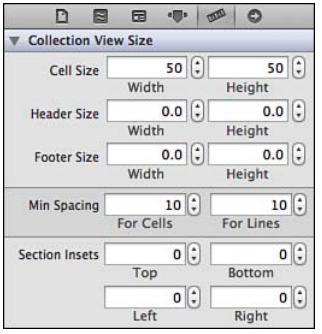


Figure 2.2 Size Inspector of a collection view layout

You can change the distance between cells in the collection view using the Min Spacing section in the Size Inspector. This is only the *minimum* distance; the default layout, called Flow, makes sure that cells are a minimum distance from one another. The Section Insets area of the Size Inspector enables you to specify the distance surrounding an entire section. (Remember that you only have one section so far.) You take a closer look at section insets in Chapter 3, so don't worry about the specifics for now. It's a personal pet peeve of mine to have too small a margin around content, so bump up the section insets to 10 points each. Run the app to see the visual differences in the collection view. It should resemble Figure 2.3.

Figure 2.3 Changes made with storyboards

Not bad at all. Don't worry that the status bar is visible in front of our content; that is the default on iOS 7. We'll solve this problem later by placing our Collection view controller inside of a navigation controller. The problem with Figure 2.3 is that only some of the properties of a collection view layout are accessible with storyboards or .xib files. In addition, if you override the properties you've set in a storyboard in code, or you forget that you've set something in the storyboard, it can lead to a debugging headache. For this reason, I strongly prefer to use a code-only approach with collection views.

Now you can re-create your interface using only code. Create a new Xcode project with the Empty Application template. (For anyone who has never created an app from an empty template, this can be a big step.) Create a new file using File, New, File or ⌘N. Select Objective-C Class and call it something like `AFViewController`. In the field for Subclass, enter `UICollectionViewController`. Make sure not to select With XIB for User Interface.

Open the application delegate implementation file and add an `#import` statement to import the new view controller's header file. Change the implementation to look like the code in Listing 2.1.

Listing 2.1 Setting Up the Application

```
#import "AFAppDelegate.h"
#import "AFViewController.h"
```

```
@implementation AFAppDelegate

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];

    UICollectionViewFlowLayout *collectionViewLayout =
        [[UICollectionViewFlowLayout alloc] init];

    collectionViewLayout.scrollDirection =
UICollectionViewScrollDirectionHorizontal;
    collectionViewLayout.sectionInset = UIEdgeInsetsMake(10, 10, 10, 10);
    collectionViewLayout.itemSize = CGSizeMake(20, 50);
    self.window.rootViewController = [[AFViewController alloc]
        initWithCollectionViewLayout:collectionViewLayout];

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}
```

Next, open the view controller's implementation file and add the following line to the `viewDidLoad` method (see Listing 2.2).

Listing 2.2 Setting the Scroll Indicator Color

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    //All that other stuff

    self.collectionView.indicatorStyle = UIScrollViewIndicatorStyleWhite;
}
```

Build and run the app, and you see that everything you customized using storyboards has been replicated using just code. High five!

Before you dive deeper into collection views and laying out content, the following section takes you on a quick diversion to discuss `UIScrollView`.

`UIScrollView`: A Brief Overview

`UICollectionView` is a direct subclass of `UIScrollView`, much like `UITableView`. Similarly to the `UICollectionView` inheritance, the `UICollectionViewDelegate` protocol conforms to the `UIScrollViewDelegate` protocol. In practical terms, this means that if an object is the delegate of a collection view, it receives callbacks notifying it of `UICollectionViewDelegate` events as well as `UIScrollViewDelegate` events.

`UIScrollView` is a versatile class in UIKit and has been around since iOS was iPhone OS 2.0. It provides a friendly way for developers to scroll content, whether it be a list of emails, a grid of apps, or a single photo. If you can scroll something in any given app, chances are that the app uses a scroll view.

Scroll views give a familiar feel to the user and make any application that uses them seem more like it belongs in iOS and less like its developer wrote his own scroll view. Scroll views offer a lot of power to developers for very little work; all that developers need to do is set up the scroll view and add subviews to it. In addition, you get to rely on the work that Apple has already done for you, like emulating physics and deceleration. Take a look at an example in which the user can scroll to see more content than can fit on the screen simultaneously.

Create a new Xcode project with the Single View template. Copy a large image into the project and open the main view controller's implementation file. Replace the `viewDidLoad` implementation with the one in Listing 2.3.

Listing 2.3 A Simple Scroll View Example

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    //First we create an image to display to the user.
    //Replace "cat.jpg" with whatever your image is named
    UIImage *image = [UIImage imageNamed:@"cat.jpg"];

    //Next we create an image view to display the image.
    //It should be the same size as the image with its origin
    //in the top-left corner
    UIImageView *imageView = [[UIImageView alloc] initWithImage:image];
    imageView.frame = CGRectMake(0, 0,
        image.size.width, image.size.height);

    //Finally we create our scroll view. We give it a frame
    //corresponding to our view's bounds so it fills the entire view
    UIScrollView *scrollView = [[UIScrollView alloc]
        initWithFrame:self.view.bounds];
```

```
    //This line is very important - it makes the scroll view scroll
    scrollView.contentSize = image.size;
    //This is just to get rotation to work correctly
    scrollView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;

    //Finally, set up the view hierarchy
    [scrollView addSubview:imageView];
    [self.view addSubview:scrollView];
}
```

Run the application, and you see output similar to Figure 2.4; the image is too large to fit on the screen at one time, but the user can scroll around the image to see it all. (Notice the scroll indicators.) The magic that makes this all work is the `contentSize` property. This is a `CGSize` value that represents the size (in points) of the scrollable area. Its default value is zero, and it must be set to use any scroll view, even if the content size is smaller than the scroll view's own size.

Figure 2.4 A simple scroll view example

When the scroll view knows the size of the content it's displaying, it scrolls. The `contentSize` property can change at any time.

Figure 2.5 demonstrates the idea of content size. The light region of the photo, in the upper left, defines the visible part of the image when the application first launches. This is the size of the scroll view and is represented by dashed lines. The solid lines represent the content size of the scroll view.

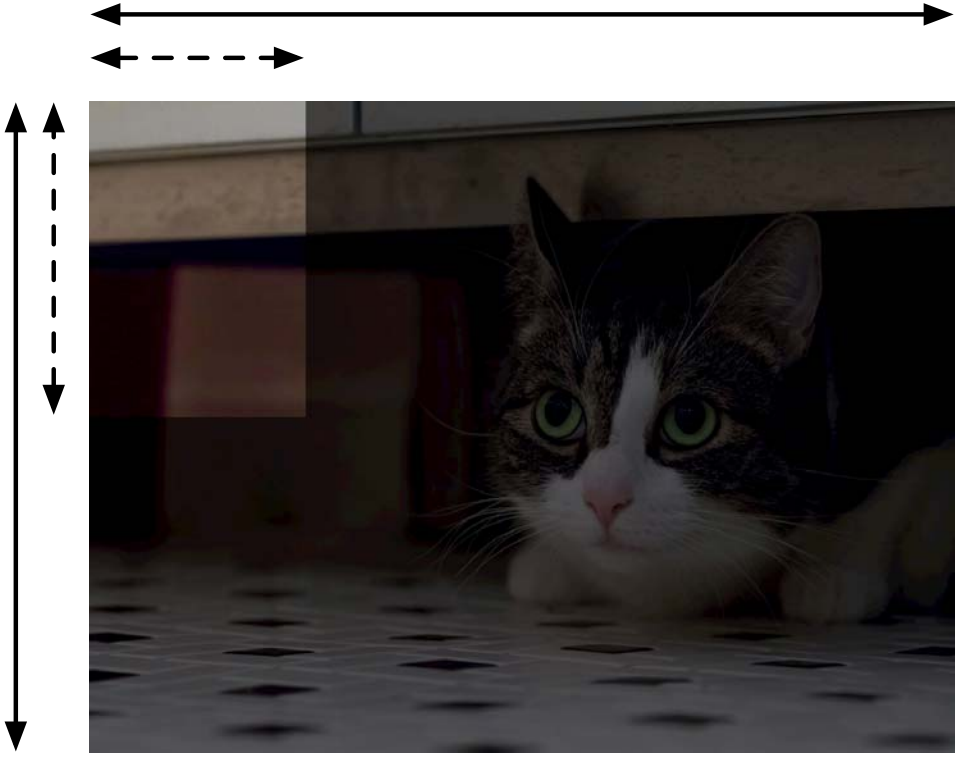


Figure 2.5 Content size example

When the user scrolls the scroll view, the content area visible to the user changes. The position of the content view within the scroll view is called the *content offset* and is represented by the `contentOffset` property, a `CGPoint` value. This property is defined by the distance from the visible region's origin (top-left corner) to the origin of the content. Figure 2.6 demonstrates content offset with white dashed lines. The content size remains the same, but the content offset changes to respond to user interaction.

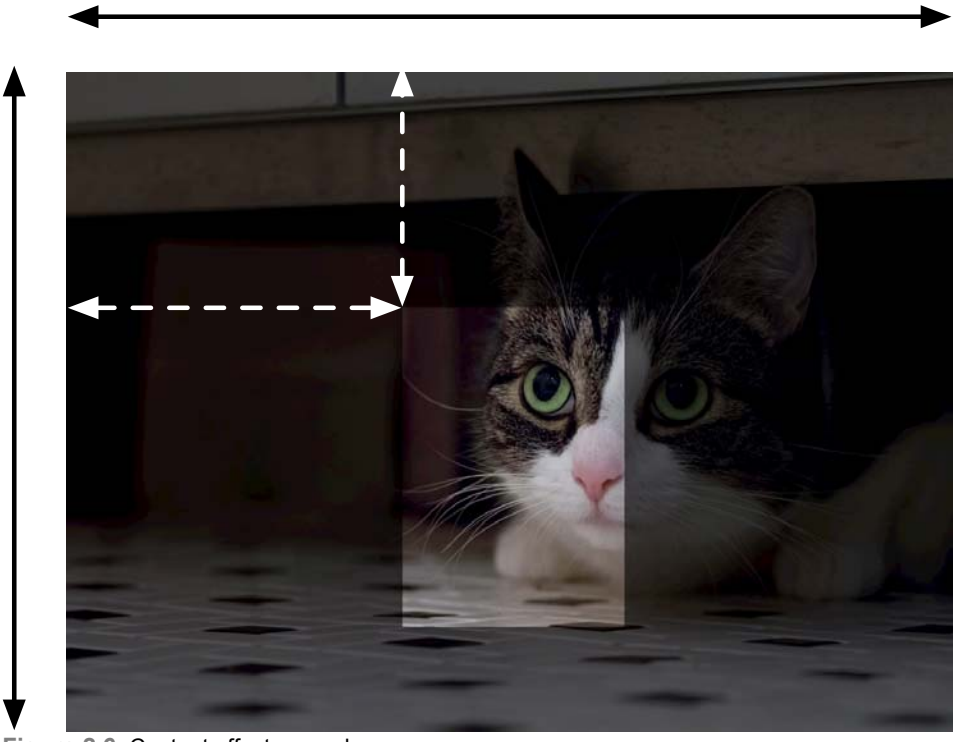


Figure 2.6 Content offset example

Content offset can be changed programmatically; the `contentOffset` property is `readwrite`. More interestingly, you can use the `setContentOffset:animated:` method to animate the change in content offset. This "moves" the scroll view, just as it would if the user moved it herself. The content offset can also be changed with `scrollRectToVisible:animated:`, but this is more often used with zooming than simple scrolling.

The last thing I want to mention about scroll views is the `contentInset` property. This is a `UIEdgeInset` value that represents the area around the scroll view's content that it should "pad." Setting the `contentInset` property to `UIEdgeInsetsMake(10, 10, 10, 10)` would create a 10-point margin surrounding the scroll view's content. The edge inset values can also be negative; this would represent area around the scroll view content that can't be seen by the user (unless she scrolls past the edge of the scroll view). Try playing around with `contentInset` to see how it works.

This `contentInset` is a widely used property and is often employed using `UITableView` and custom pull-to-refresh controls. It's also useful if you have a navigation bar over top of a view controller with `wantsFullScreenLayout` set to `YES`. The inset's top value would be equal to negative the height of the status bar and the navigation bar.

Those are the three main components to `UIScrollView`: `contentSize`, `contentOffset`, and `contentInset`. Now it's time for a quick discussion about the scroll view delegate before the chapter moves on to some more collection view material.

There are three groups of methods in `UIScrollViewDelegate`: those responding to dragging and scrolling, those responding to zooming, and those responding to scrolling animations initiated explicitly by code (see Table 2.1). You're going to be dealing only with the first and last groups because collection views don't use the zoom functionality of `UIScrollView`.

Table 2.1 Useful `UIScrollViewDelegate` Methods

Method Name	Description
`scrollViewDidScroll:`	Called whenever the content offset of the scroll view changes, either programmatically or in response to user interaction. Possible use could be in a custom pull-to-refresh control.
`scrollViewWillBeginDragging:`	Called whenever the scroll view is about to be dragged by the user. Possible use could be disabling updates to the scroll view that might interrupt smooth-scrolling performance.
`scrollViewWillEndDragging:withVelocity:targetContentOffset:`	Called whenever the user has lifted his finger from the scroll view after dragging. The second parameter specifies a speed, in points/second, that the scroll view has at the moment the user lifts his finger. The third parameter is a pointer to a `CGPoint`, representing where the scroll view will scroll. Modifying the `CGPoint` at that pointer changes where the scroll view scrolls to. Possible use could be calculating what content is *going to be* visible when the scrolling animation ends and prefetching it from an application programming interface (API).
`scrollViewDidEndDragging:willDecelerate:`	Called whenever the user has lifted his finger from the scroll view after dragging. The second parameter specifies whether the scroll view animates its deceleration to come to a stop or if it was already stopped when the user lifted his finger. Possible use includes restarting any paused computations halted in `scrollViewWillBeginDragging:`, as long as the second parameter is `NO`.

`scrollViewShouldScrollToTop:`	Called whenever the operating system needs to determine whether tapping on the status bar should animate the scroll view to the top. Only one visible scroll view should return `YES` from this method at a time.
`scrollViewDidScrollToTop:`	Called after the scroll view scrolled to the top in response to the user tapping the status bar.
`scrollViewWillBeginDecelerating:`	Called whenever the scroll view is about to begin a decelerating animation.
`scrollViewDidEndDecelerating:`	Called after the scroll view's deceleration animation completes. Possible use includes restarting any paused computations halted in `scrollViewWillBeginDragging:`.
`scrollViewDidEndScrollingAnimation:`	Called after the scroll view's content offset change animation has completed. This method is only invoked on the delegate if the content offset was changed programmatically and with explicit animation enabled.

You use some scroll view delegate methods later on in more advanced chapters in this book and in some case studies. They are useful tools to solving many problems, and you should be aware of them.

`UICollectionViewCell` Reuse: How and Why

`UICollectionView` uses a memory-efficient scheme to configure individual cells for display. As one software engineer at Apple phrased it, "malloc is expensive."
What he meant was that allocating new portions of memory is actually an expensive operation if you do it a lot. What `UICollectionView` does is very clever: It reuses cells it's no longer displaying.

Note

This should sound familiar to anyone familiar with `UITableView`. With iOS 6, Apple took the best parts of `UITableView` to make `UICollectionView`. Many things will seem familiar, but you might be surprised at how much is new.

`UICollectionView` relies on its `dataSource` to tell it how many cells to display and to configure each individual cell before it is presented to the user. When scrolling, this needs to be incredibly fast, which is why cells have reuse. The following explains exactly what happens.

For every type of cell that's going to be displayed, you should use a cell reuse identifier. This is an `NSString` that you typically store as a static variable. Before any cell with that reuse identifier can be displayed, it needs to be registered with the collection view. This is a big departure from `UITableView`. You usually register cells in `viewDidLoad` and don't reregister them later on.

When registering a cell, you provide either a `UINib` instance or a `Class`. I prefer a class instead of a nib because it gives me more control over the layout and performance.

Use either the `registerClass:forCellWithReuseIdentifier:` or `registerNib:forCellWithReuseIdentifier:` to register cells. From that point on, whenever `dequeueReusableCellWithReuseIdentifier:forIndexPath:` is called, you are guaranteed to have an allocated, initialized cell corresponding to your reuse identifier (see Figure 2.7).

This differs from `UITableView`, which historically required developers to check for a `nil` return value from an attempt to dequeue a cell (though it now supports the new method). With collection views, you are *guaranteed* to be returned a valid cell.

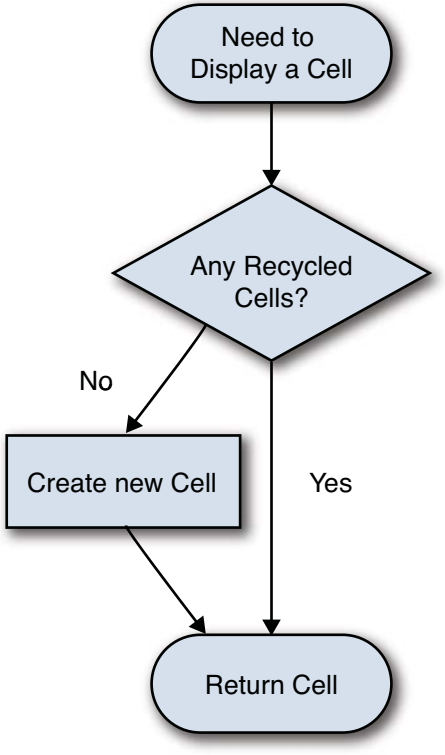


Figure 2.7 Collection view cell reuse

If your collection view only ever has 20 cells visible onscreen simultaneously, your collection view is only ever allocated 20 cells; when a cell scrolls offscreen, it's added to a reuse queue to be reused again. This technique lets applications maintain an insanely low memory footprint and an insanely high frame rate while scrolling through a collection view with hundreds or thousands of cells.

Most examples in this book, and most of the real-world uses for collection views, only display one type of cell and therefore have just one reuse identifier. It's completely reasonable to have more than one type of identifier if you're displaying more than one type of cell.

Displaying Content to Users

Alright! You've made it through a chapter on MVC and half a chapter on the basics of `UICollectionView`. It's high time to see some code.

You're going to build a basic application that displays some custom content to the user. It's going to be an iPad app, so you can use really big cells. What you're going to do at first is build a basic collection view that enables the user to add new cells with a plus button, and the cells display the time that they were added. This is just a warm-up for what comes later.

Create a new Xcode project using the Empty Application template. Create a new file, an Objective-C class that extends `UICollectionViewController`, and give it a suitable name. In your application delegate's implementation file, `#import` the view controller's header and create an instance of the view controller to be the root view controller of a navigation controller, the window's root view controller (see Listing 2.4).

Listing 2.4 Setting Up the Application

```
#import "AFAppDelegate.h"
@implementation AFAppDelegate

- (BOOL)application:(UIApplication *)application
    didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]];
    self.window.backgroundColor = [UIColor whiteColor];

    UICollectionViewFlowLayout *flowLayout =
        [[UICollectionViewFlowLayout alloc] init];
    AFViewController *viewController =
         [[AFViewController alloc] initWithCollectionViewLayout:flowLayout];

    UINavigationController *navigationController =
```

```
        [[UINavigationController alloc]
initWithRootViewController:viewController];
    navigationController.navigationBar.barStyle = UIBarStyleBlack;
    self.window.rootViewController = navigationController;

    [self.window makeKeyAndVisible];
    return YES;
}
```

You're relying on a `UINavigationController` because it provides a lot of nice things for free. In this case, you get a cool navigation bar on which you can include buttons. This implementation of `applicationDidFinishLaunchingWithOptions:` is a little more lightweight than the example earlier in this chapter; you're going to be following some "best practices" a little closer this time. The app delegate creates just the basics for the view controller, and it further customizes itself.

Create a new Objective-C class that extends `UICollectionViewCell`. You're not going to add any code to it yet. You just need to `#import` it in the view controller's implementation file.

Open the view controller's implementation file and create a static `NSString` instance with some indicative value; you'll use this as your reuse identifier. Add two instance variables: One is an `NSMutableArray` representing the model, and the other is an `NSDateFormatter` that you'll use to format content to the user (see Listing 2.5).

Listing 2.5 Instance Variables and Static Identifier Setup

```
#import "AFCollectionViewCell.h"

static NSString *CellIdentifier = @"Cell Identifier";

@implementation AFViewController
{
    //This is our model
    NSMutableArray *datesArray;
    NSDateFormatter *dateFormatter;
}
```

Next, create a `viewDidLoad` implementation that sets up an empty model (your `datesArray`) and a date formatter instance. Also configure your layout and collection view to look pretty, register your `UICollectionViewCell` subclass for this reuse identifier, and add a button to your navigation bar (see Listing 2.6).

Listing 2.6 Configuring a `UICollectionView` in `viewDidLoad`

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //instantiate our model
    datesArray = [NSMutableArray array];
    dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setDateFormat:
         [NSDateFormatter dateFormatFromTemplate:@"h:mm:ss a" options:0
locale:[NSLocale currentLocale]]];

    //configure our collection view layout
    UICollectionViewFlowLayout *flowLayout =
         (UICollectionViewFlowLayout *)self.collectionView.collectionViewLayout;
    flowLayout.minimumInteritemSpacing = 40.0f;
    flowLayout.minimumLineSpacing = 40.0f;
    flowLayout.sectionInset = UIEdgeInsetsMake(10, 10, 10, 10);
    flowLayout.itemSize = CGSizeMake(200, 200);

    //configure our collection view
    [self.collectionView registerClass:[AFCollectionViewCell class]
        forCellWithReuseIdentifier:CellIdentifier];
    self.collectionView.indicatorStyle = UIScrollViewIndicatorStyleWhite;

    //configure our navigation item
    UIBarButtonItem *addButton = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
        target:self action:@selector(userTappedAddButton:)];

    self.navigationItem.rightBarButtonItem = addButton;
    self.navigationItem.title = @"Our Time Machine";
}
```

Awesome. You could run the application right now, but all you would see is an empty screen with a plus button and a title. So, finish with the view controller code before writing your collection view cell subclass (see Listing 2.7). You need to implement your `UICollectionViewDataSource` methods.

Listing 2.7 `UICollectionViewDataSource` Methods

```
-(NSInteger)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section
{
```

```
    return datesArray.count;
}

-(UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
    cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    AFCollectionViewCell *cell = (AFCollectionViewCell *)[collectionView
        dequeueReusableCellWithReuseIdentifier:CellIdentifier
        forIndexPath:indexPath];

    cell.text = [dateFormatter stringFromDate:datesArray[indexPath.item]];

    return cell;
}
```

Right now, this throws a compiler error. Don't worry, though. After you write the rest of your code, it will work. You need a method to respond to your Add button. Create two methods: one with the selector name you gave the `addButton` in `viewDidLoad`, and one that you can call from anywhere in your code to add a new date to `datesArray` (see Listing 2.8).

Listing 2.8 Configuring a `UICollectionView` in `viewDidLoad`

```
-(void)userTappedAddButton:(id)sender
{
    [self addNewDate];
}

-(void)addNewDate
{
    [self.collectionView performBatchUpdates:^{
        //create a new date object and update our model
        NSDate *newDate = [NSDate date];
        [datesArray insertObject:newDate atIndex:0];

        //update our collection view
        [self.collectionView insertItemsAtIndexPaths:
            @[[NSIndexPath indexPathForItem:0 inSection:0]]];
    } completion:nil];
}
```

You're calling `performBatchUpdates:completion:` on the `UICollectionView`. This gets you animation (defined by your layout class; more on that in Chapter 3) for free.

Amazing! Now all you have to do is write your `UICollectionViewCell` subclass. Go to the header file you created earlier (see Listing 2.9). You're going to give it a single, `NSString` property.

Listing 2.9 `UICollectionViewCell` Subclass Header

```
@interface AFCollectionViewCell : UICollectionViewCell

@property (nonatomic, copy) NSString *text;

@end
```

Now your compiler would stop complaining, but nothing really interesting would happen if you ran the app. Open the implementation file for the cell and add a `UILabel` instance variable. Override the `initWithFrame:` method with the implementation in Listing 2.10.

Listing 2.10 `UICollectionViewCell` Subclass Initialization

```
@implementation AFCollectionViewCell
{
    //subview of our contentView
    UILabel *textLabel;
}

#pragma mark - Initialization

- (id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    self.backgroundColor = [UIColor whiteColor];

    textLabel = [[UILabel alloc] initWithFrame:self.bounds];
    textLabel.textAlignment = NSTextAlignmentCenter;
    textLabel.font = [UIFont boldSystemFontOfSize:20];
    [self.contentView addSubview:textLabel];

    return self;
}
```

Next, you're going to override the `text` property to update the label. You're also going to override an important method of `UICollectionViewCell` called `prepareForReuse` (see Listing 2.11).

Listing 2.11 `UICollectionViewCell` Reuse

```
-(void)prepareForReuse
{
    [super prepareForReuse];

    self.text = @"";
}

-(void)setText:(NSString *)text
{
    _text = [text copy];

    textLabel.text = self.text;
}
```

This updates your cell's label with the string that's being set as your `text` property. In `prepareForReuse`, you call `super` (very important!) and then set your text to the empty string. This is really important; you need to reset your cell to its starting, neutral state as much as possible. Otherwise, the data source for the collection view might forget to reset parts of it, and you can end up with an inconsistent and confusing user interface.

Run the application, and you see an empty screen. Tap the plus button to add a new cell to the collection view. Notice the animation you get as a new cell is added to the top of the collection view (see Figure 2.8). Nice! You also have rotation support included, for free.

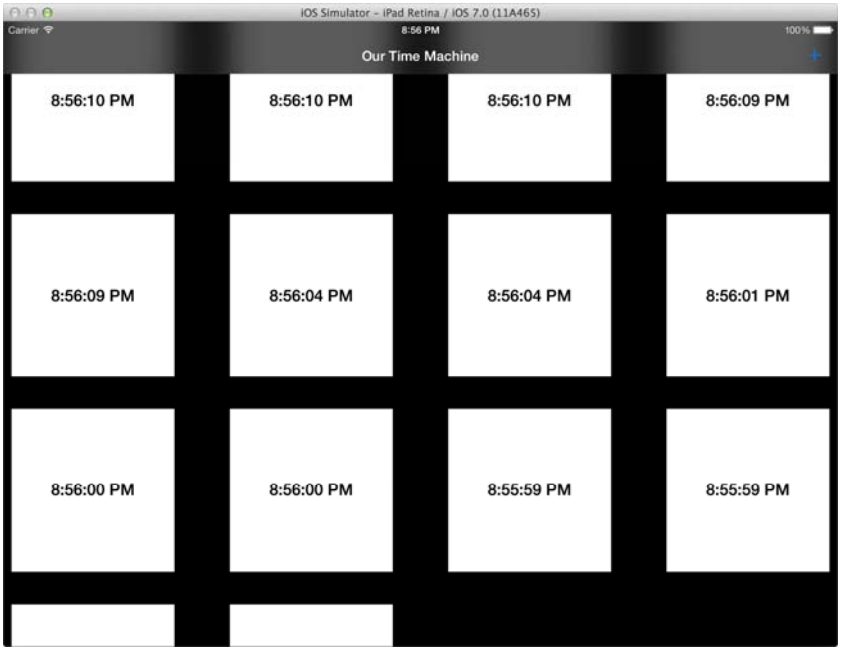


Figure 2.8 Basic cell display example

I don't want to sound like a broken record about MVC, but it's important to note that the cell doesn't have any idea *what* it is displaying; it's passed a string that happens to contain a date that corresponds to the model. What's important is that you're not passing it the `NSDate` object itself.

Now that you have a basic collection view example, take a closer look at the `UICollectionView` class itself. The cell has two Boolean properties of importance: `selected` and `highlighted`. The highlighted state depends completely on the user interaction; when the user has her finger pressed down on a cell, it becomes highlighted automatically. Cell selection is less transient; cells become selected (if their collection view supports selection) when the user lifts her finger. Cells stay selected either until some code you've written unselects them or until the user taps them again. When being tapped to become either selected or unselected, cells become highlighted temporarily. The setters for these properties can (and often are) called from within animation blocks. Be aware when overriding their implementations that changes you make will likely be implicitly animated.

Selection and highlighting can be confusing. Don't worry, though, because the next example explores it a little more. In the meantime, Figure 2.9 should help.

Figure 2.9 `UICollectionViewCell` view hierarchy

In the custom subclass from the last exercise, you added the `UILabel` subview to `self.contentView` and not `self`. In general, you should not add subviews directly to a collection view cell; always add them to its `contentView`. Here's why.

`UICollectionViewCell` has three subviews, denoted in Figure 2.10. The black rectangle, at the back, is the collection view cell itself. The green view at the front is the `contentView`, and it's there that you add your subviews. The two intervening views are the `selectedBackgroundView` and the `backgroundView`. These are both optional and can be set at any time. The `backgroundView`, if set, is permanently present.

When the cell becomes selected, the `selectedBackgroundView` is added to the view hierarchy; when the cell becomes unselected, it is removed. Note that these two events can be called from within animation blocks and the `selectedBackgroundView` is animated in (with a crossfade by default).

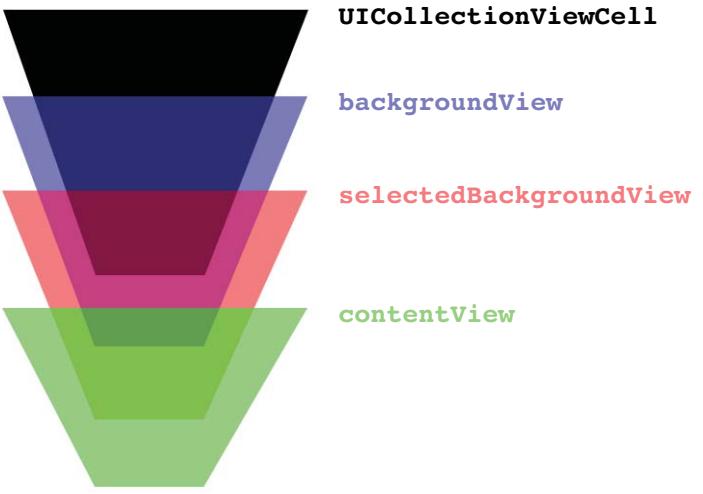


Figure 2.10 `UICollectionViewCell` view hierarchy

Now that you have a better understanding of the view hierarchy in `UICollectionViewCell`, you can continue to another example that helps illustrate the uses of these properties, `contentView`, and images.

You're going to create an app that displays 12 images repeated in 10 different sections. Each section is going to have its own background color, unless it is selected, demonstrating how to use the `selectedBackgroundView`. You're using 12 images because they fit a single section in one screen full of content.

Create a new Xcode project based on the Empty template. Create a subclass of `UICollectionViewController` and a subclass of `UICollectionViewCell`, just like last time. Set up an instance of the view controller as the window's root view controller in the app delegate—no need to use a navigation controller this time.

Use two arrays to store your models: one for the images and one for the colors you're using for the background color. You're going to tweak the cell size, inter-item spacing, and line spacing from the last example. In addition, you're going to enable multiple selection on the collection view; this is going to enable users to select more than one cell at one time and also enables users to deselect cells by tapping them. Everything in the `viewDidLoad` method in Listing 2.12 should look familiar. I created a series of JPEG images, named `0.jpg` to `11.jpg`, for 12 total.

Listing 2.12 Configuring a `UICollectionView` in `viewDidLoad`

```
static NSString *CellIdentifier = @"Cell Identifier";

@implementation AFViewController
{
    //models
    NSArray *imageArray;
    NSArray *colorArray;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    //Set up our models
    NSMutableArray *mutableImageArray = [NSMutableArray arrayWithCapacity:12];
    for (NSInteger i = 0; i < 12; i++)
    {
        NSString *imageName = [NSString stringWithFormat:@"%d.jpg", i];
        [mutableImageArray addObject:[UIImage imageNamed:imageName]];
    }
    imageArray = [NSArray arrayWithArray:mutableImageArray];

    NSMutableArray *mutableColorArray = [NSMutableArray arrayWithCapacity:10];
    for (NSInteger i = 0; i < 10; i++)
    {
        CGFloat redValue = (arc4random() % 255) / 255.0f;
        CGFloat blueValue = (arc4random() % 255) / 255.0f;
        CGFloat greenValue = (arc4random() % 255) / 255.0f;

        [mutableColorArray addObject:[UIColor colorWithRed:redValue
            green:greenValue blue:blueValue alpha:1.0f]];
    }
    colorArray = [NSArray arrayWithArray:mutableColorArray];

    //configure our collection view layout
    UICollectionViewFlowLayout *flowLayout =
        (UICollectionViewFlowLayout *)
        self.collectionView.collectionViewLayout;
    flowLayout.minimumInteritemSpacing = 20.0f;
    flowLayout.minimumLineSpacing = 20.0f;
    flowLayout.sectionInset = UIEdgeInsetsMake(10, 10, 10, 10);
    flowLayout.itemSize = CGSizeMake(220, 220);

    //configure our collection view
```

```
    [self.collectionView registerClass:[AFCollectionViewCell class]
        forCellWithReuseIdentifier:CellIdentifier];
    self.collectionView.indicatorStyle = UIScrollViewIndicatorStyleWhite;
    self.collectionView.allowsMultipleSelection = YES;
}
```

Because you're displaying multiple sections, you need to implement a new, optional `UICollectionViewDelegate` method called `numberOfSectionsIn-CollectionView:`. The `collectionView:cellForItemAtIndexPath:` implementation shown in Listing 2.13 is also going to look familiar.

Listing 2.13 `UICollectionViewDataSource` Methods

```
-(NSInteger)numberOfSectionsInCollectionView:(UICollectionView *)collectionView
{
    return colorArray.count;
}

-(NSInteger)collectionView:(UICollectionView *)collectionView
        numberOfItemsInSection:(NSInteger)section
{
    return imageArray.count;
}

-(UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
    cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    AFCollectionViewCell *cell = (AFCollectionViewCell *)
        [collectionView d
            equeueReusableCellWithReuseIdentifier:CellIdentifier
            forIndexPath:indexPath];

    cell.image = imageArray[indexPath.item];
    cell.backgroundColor = colorArray[indexPath.section];

    return cell;
}
```

Open the collection view cell subclass and add an `UIImageView` instance variable to the class. Also add a strong `UIImage` property named `image`. Write a new initializer to instantiate the instance variable (see Listing 2.14).

Listing 2.14 `UICollectionViewCell` Subclass Initialization

```
@implementation AFCollectionViewCell
{
    UIImageView *imageView;
}

- (id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    self.backgroundColor = [UIColor whiteColor];

    imageView = [[UIImageView alloc] initWithFrame:
        CGRectInset(self.bounds, 10, 10)];
    [self.contentView addSubview:imageView];

    UIView *selectedBackgroundView = [[UIView alloc] initWithFrame:CGRectZero];
    selectedBackgroundView.backgroundColor = [UIColor whiteColor];
    self.selectedBackgroundView = selectedBackgroundView;

    return self;
}
```

You're moving in the frame of the image view by 10 points to create a border around the image. Override the `setImage:` method to set the image of the `imageView`. You're also going to fill in a `prepareForReuse` implementation and include an implementation for `setHighlighted:` (see Listing 2.15). Also notice that you've set your `selectedBackgroundView` to a plain white view. This white view will be placed in front of the cell (and in front of any background view, which you don't have in this case) while the cell is selected.

Listing 2.15 `UICollectionViewCell` Overridden Methods

```
-(void)prepareForReuse
{
    [super prepareForReuse];

    self.backgroundColor = [UIColor whiteColor];
    self.image = nil; //also resets imageView's image
}

-(void)setHighlighted:(BOOL)highlighted
{
    [super setHighlighted:highlighted];
```

```
    if (highlighted)
    {
        imageView.alpha = 0.8f;
    }
    else
    {
        imageView.alpha = 1.0f;
    }
}

-(void)setImage:(UIImage *)image
{
    _image = image;

    imageView.image = image;
}
```

Remember to always call super's implementation of overridden properties (unless you purposefully don't want to and have a *really* good reason). In the implementation, you decrease the alpha of your image view to 80% when it is highlighted. Run the application.

Play around with the application. Tap cells to make them selected, and then tap them again. Notice that if you tap and drag, the collection view cancels your tap and scrolls instead. This is because the `UIScrollView` property `canCancelContentTouches` is set to `YES`. Also notice how the collection view delays highlighting the cell until you hold down the touch for a few tenths of a second. This is because the `UIScrollView` property `delaysContentTouches` is set to `YES`. In the `viewDidLoad` implementation, play with these two methods to experiment with how they affect the user experience of the collection view (and, in fact, all scroll views, as these are the default values).

Note a couple of things about the `selectedBackgroundView` and `backgroundView` properties of `UICollectionViewCell`. First, they will be stretched to fit to whatever cell they're assigned. This is why you were able to initialize the selected background view in this example with a frame of `CGRectZero`. Next, some attributes, like `alpha`, will be reset to their defaults (in `alpha`'s case, `1.0f`) by the collection view. Be aware of these issues when troubleshooting display problems with backgrounds of cells.

If you want proof that the `selectedBackgroundView` is placed within the view hierarchy, you can set it to have a slightly transparent color. Change the background color of the `selectedBackgroundView` to something like `[UIColor colorWithWhite:1.0f alpha:0.8f]`. Now you'll be able to see through the `selectedBackground` view to its superview, the collection view itself.

Before the chapter wraps up with a case study on performance, I want to make a quick diversion to revisit storyboards and .xib files. Now that you understand how collection view

cells work and you can create subclasses to customize their appearance, take a look at how to approach the previous exercise using storyboards and .xibs.

Using .xibs is the most similar to code, so start with that. Open the Xcode project from the previous exercise (copy it first if you're not using source control) and add a new file.

Under the left pane of the new file dialog, select User Interface, and then double-tap the Empty file to create a new, empty .xib. Give it the same name as your collection view cell subclass. In the object library, find Collection View Cell and drag it onto the empty canvas.

Select the new cell and open the Size Inspector and set the sizes to 220 wide and 220 tall. These are going to be reconfigured by the collection view anyway, so it's only to help us get a visual sense of what the cell will look like. Open the Attributes Inspector and make the background color white. Open the Identity Inspector and set the type of the collection view cell to your subclass.

In the subclass, you need to remove a lot of code. The `initWithFrame:` initializer will no longer be called. Create a new method called `awakeFromNib`. This method is called when an instance of the class is "thawed" from the nib. In this method, you place your custom `selectedBackgroundView` initialization. See how some things need to be done with code, anyway?

Drag an image view onto the cell. Set the springs and struts (or Autolayout constraints) so that the image view is inset by 10 points on all sides. Move the instance variable to the header file and prefix it with the keyword `IBOutlet` so that the .xib can see it. Command-click and drag from the collection view cell to the image view inside it; select the `imageView` outlet from the menu that appears.

Finally, you need to tell the collection view to use this nib instead of initializing its own copies of the collection view cell subclass itself. In `viewDidLoad`, change the setup for the collection view, as shown in Listing 2.16.

Listing 2.16 `UICollectionViewCell` Registration Using `UINib`

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    //all that other stuff

    [self.collectionView registerNib:
        [UINib nibWithNibName:@"AFCollectionViewCell" bundle:nil]
        forCellWithReuseIdentifier:CellIdentifier];
}
```

Run the application and see that it behaves exactly as it did with code. Notice that even though you're using `UINib`, you are still forced to use a subclass implementation file.

Finally, you're going to use storyboards to produce the same effect. Empty the `applicationDidFinishLaunchingWithOptions:` implementation to just return `YES`. Delete the .xib file. Pare down the `viewDidLoad` implementation to look like Listing 2.17.

Listing 2.17 `UICollectionViewCell` Registration Using Storyboards

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    //Set up our models
    NSMutableArray *mutableImageArray = [NSMutableArray arrayWithCapacity:12];
    for (NSInteger i = 0; i < 12; i++)
    {
        NSString *imageName = [NSString stringWithFormat:@"%d.jpg", i];
        [mutableImageArray addObject:[UIImage imageNamed:imageName]];
    }
    imageArray = [NSArray arrayWithArray:mutableImageArray];

    NSMutableArray *mutableColorArray = [NSMutableArray arrayWithCapacity:10];
    for (NSInteger i = 0; i < 10; i++)
    {
        CGFloat redValue = (arc4random() % 255) / 255.0f;
        CGFloat blueValue = (arc4random() % 255) / 255.0f;
        CGFloat greenValue = (arc4random() % 255) / 255.0f;

        [mutableColorArray addObject:
            [UIColor colorWithRed:redValue
                green:greenValue blue:blueValue alpha:1.0f]];
    }
    colorArray = [NSArray arrayWithArray:mutableColorArray];

    //configure our collection view
    self.collectionView.allowsMultipleSelection = YES;
}
```

Your `viewDidLoad` now only sets up the models and sets one property on the collection view that can't be set with the Attributes Inspector of a storyboard.

Create a new storyboard file called MainStoryboard. Open the Xcode project settings and set MainStoryboard as the Main Storyboard. (You read that correctly, folks.) Drag a `UICollectionViewController` onto the empty storyboard and set its custom class to be

the one your code lives in. Expand the view hierarchy of the collection view until you get to the collection view cell. Set its custom class to be your `UICollectionViewCell` subclass and set its Reuse Identifier to Cell Identifier in the Attributes Inspector. Open the Size inspector and set it to 220 wide by 220 tall.

Add an image view as a subview to the cell; command-click and drag from the cell to the image view to set the `imageView` outlet of the cell. Set the image view to 200 wide by 200 tall and set its springs and struts (or Autolayout constraints) so that it grows with the cell.

Finally, click the Collection View Flow Layout object in the view hierarchy. Set the Min Spacing and Section Insets to those you previously used in code (see Figure 2.11). Run the application.

The advantage to using storyboards or .xib files is that you get to lay out your interface visually. This can be a lot of help when working with a designer or when first learning about view hierarchies in CocoaTouch. However, storyboards and .xib files don't offer a lot of compelling advantages, other than their visual nature. There are two problems with storyboards and collection views: the tight coupling between the collection view cell (its reuse identifier) and the code, and some tricky debugging when you modify the settings of the storyboard in code at runtime. You can't rely on what you see visually at compile time because it's likely going to be changed by code at runtime, anyway.

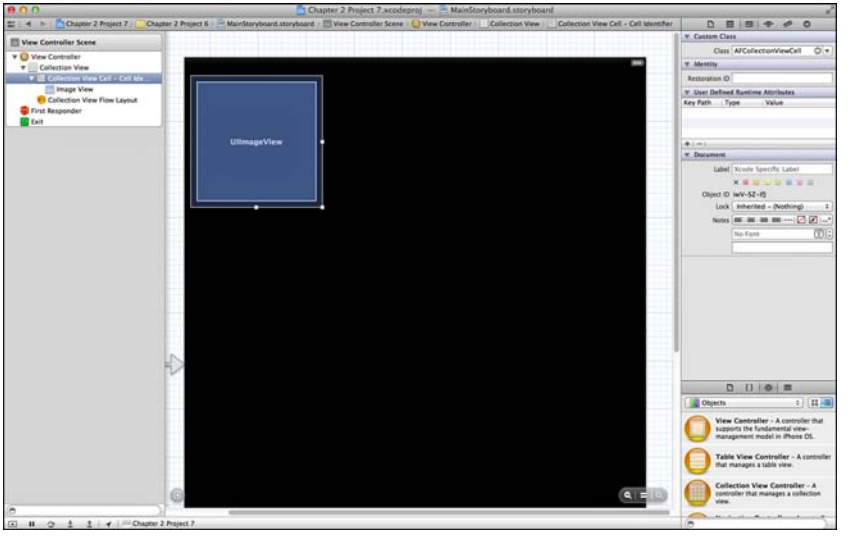


Figure 2.11 The same example, using storyboards

From this point forward, I don't devote any more attention to .xib files or storyboards. You've seen how they work, so if you're incorporating them into an existing project that uses them, you'll be able to apply the techniques in this book. Even if you're still getting used to laying out interface in code instead of visually, I encourage you to use .xib files rather than storyboards. Remember to use custom `UICollectionViewCell` subclasses to

keep your code loosely coupled; your view controller shouldn't know about the internals of the cell's view hierarchy.

Now that you have a good understanding of `UICollectionViewCell` and how to display content to users, let's take a look at performance.

Case Study: Evaluating Performance of `UICollectionView`

When you evaluate the performance of any iOS app, you *must* measure it on an actual device. The specific device matters a little bit, but it's most important that you *don't rely on the simulator*. Although it is useful for a lot of things, like NSZombies, the simulator has an entire personal computer powering it, which isn't the case for most users' iPhones.

Choosing a test device can be a little tricky. Obviously, something really new like an iPhone 5 would not be an ideal choice for testing performance of your app when strained. However, don't rely on using the oldest or slowest hardware, either. Although the iPhone 3GS only has one core, it can perform much better than the iPhone 4 in practice because the iPhone 4 has more random access memory (RAM) and a multicore central processing unit (CPU), but has to push out four times as many pixels to its Retina screen.

Beyond the iPhone, you also have to consider the iPhone touch. If you're writing an app that pushes the limits of the device, you should be testing on all hardware/software combinations. However, many iOS developers are just single-person operations whipping up some cool apps who don't have thousands of dollars to spend on testing hardware (or understanding significant others who are willing to indulge our addiction to Apple products). If you don't have an old iPhone lying around, an iPod touch works well and is inexpensive.

With regard to collection views and other scroll views, the most important aspect of performance is perceived scrolling responsiveness. Notice that I said *perceived* responsiveness. You measure this by measuring the screen refresh rate. Ideally, this rate would be 60 frames per second (fps), the native refresh rate. That means that during each invocation of the main run loop, your application has only has 16 milliseconds to complete. That's not a lot of time. This case study is going to highlight the places that inefficient code severely affects performance and shows you how to restructure your code to keep it lean. Open the Performance Problems Example project in the same code. (The solutions are also there, with the prefix "Solved.")

Here's one more tip before getting to the actual profiling: While you are measuring the performance of your app, the performance of the CPU will be hampered by Instruments (kind of like an observer effect). To avoid this, open Preferences in Instruments and check the Always Use Deferred Mode check box. This collects the data locally on the device and doesn't send it to the computer until the run has completed.

After you have your device and have jumped through Apple's hoops to run your app on it, connect it to your computer. Make sure that your device is selected from the Scheme drop-

down menu. In Xcode, open the Product menu and select Profile (Command-I). This builds your application with Release build settings (like compiler optimizations) and opens the Instruments template chooser (see Figure 2.12).

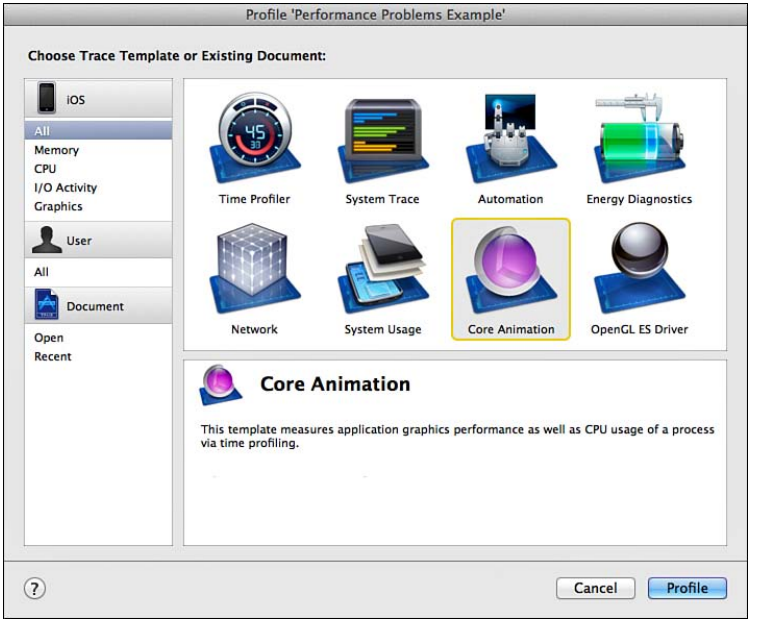


Figure 2.12 Instruments template chooser

Remember that you get different templates depending on if you're using the simulator or an actual device. Choose the Core Animation template. This gives you the screen refresh rate as well as the CPU usage, which tells you where the CPU is spending the majority of its time executing code. Click Profile and scroll the app. Use the scrolling and notice how awful the responsiveness is. When you're satisfied that this is really, really bad code, click the Stop button to see the results in Instruments (see Figure 2.13).

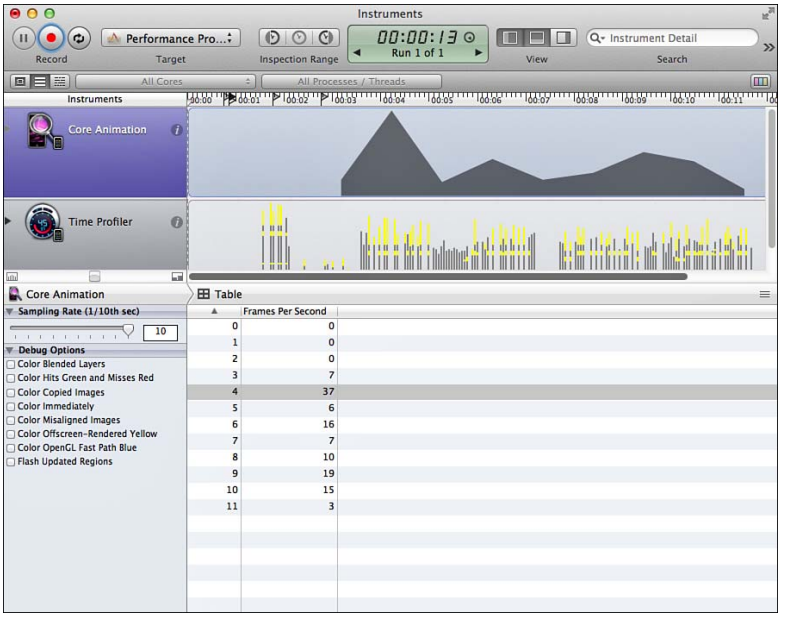


Figure 2.13 Core Animation profiler template results

Holy screen refresh rates, Batman! A peak of only 37fps is terrible. Select the Time Profiler and open the Extended Detail pane (see Figure 2.14).

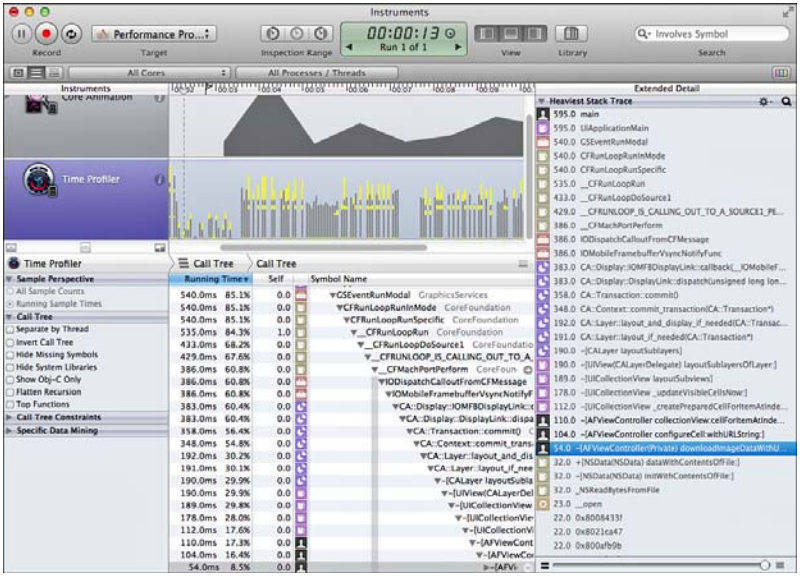


Figure 2.14 Core Animation profiler template results Extended Detail pane

You can see that the most amount of time is being spent downloading the images from the Internet on the main thread. Never a good idea! Furthermore, you're not caching the downloaded data anywhere. Add an `NSCache` instance to your view controller to hold your cached data results. This class is a handy little key/value store that automatically evicts items when memory becomes low. Initialize it in `loadView` (see Listing 2.18).

Listing 2.18 Downloading Images in a Background Thread

```
-(void)configureCell:(AFCollectionViewCell *)cell atIndexPath:(NSIndexPath
*)indexPath withURLString:(NSString *)urlString
{
    //Try to pull out a cached NSData instance from our cache
    id data = [photoDataCache objectForKey:urlString];

    if (data)
    {
        //This branch executes if the objectForKey: is non-nil,
        //meaning we have downloaded the image before.


        if ([data isKindOfClass:[NSNull class]])
        {
            //This indicates that the instance is NSNull, so we
            //should not use it.

            //nop
        }
        else
        {
            //We can successfully decompress our JPEG data
            UIImage *image = [UIImage imageWithData:data];
            [cell setImage:image];
        }
    }
    else
    {
        //Download the image in a background queue
        dispatch_async(
            dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

            NSData *data = [self downloadImageDataWithURLString:urlString];

            //Now that we have the data, dispatch back to the main queue
            //to use it. UIImage is part of UIKit and can *only* be
            //accessed on the main thread
            dispatch_async(dispatch_get_main_queue(), ^{
```

```
                UIImage *image = [UIImage imageWithData:data];

                if (image)
                {
                    //This cell instance passed in as a parameter might
                    //have been reused by now. Call
                    //reloadItemsAtIndexPaths: instead.

                    [photoDataCache setObject:data forKey:urlString];
                    [photoCollectionView reloadItemsAtIndexPaths:@[indexPath]];
                }
                else
                {
                    //This indicates the JPEG decompression failed.
                    //Set NSNull in our cache
                    [photoDataCache setObject:[NSNull null] forKey:urlString];
                }
            });
        });
    }
}
```

This code is mostly straightforward. Notice that you added a new parameter, an index path, to the method signature. This is used to reload the item at that index path; referencing the cell directly is unsafe because it might have been reused. You can run into problems reloading the item directly if the cell has been deleted, for instance. This simple example fits the needs of this chapter. If you're doing anything more complicated, I suggest relying on a fetched results controller to update the collection view.

Note

There are better ways to cache photos, like Core Data. There are also better ways to download data from the Internet, but the point of this case study is to examine problems with collection view performance, not general software architecture.

Rerun the profiler with the modified code. You can see that the performance has improved significantly. That's good, but take a closer look to see whether you can improve things even more. If you take a look at the Extended Detail pane, the method that is taking up the most time is still `configureCell:atIndexPath:withURLString:`. It looks like `imageWithData:` is taking up a lot of CPU time.

You could take a few approaches to deal with this. You could cache the decompressed JPEGs, which is a good idea, but it has its drawbacks. The biggest problem is that it consumes a lot of memory. Your images are 145 by 145 pixels and have 3 channels at 8 bits

a channel. That means that after each image is decompressed, it takes up 145 * 145 * 3 = 63KB. That doesn't sound like a lot, but the app is running on memory-constrained devices, and the OS will kill the app if it uses too much memory.

Instead, decompress the JPEG data on the background queue. "Ha!" you say, "`UIImage` is part of UIKit, and telling me to use it on the background queue is a fool's errand!" You're not wrong, but an alternative exists. `UIImage` is a handy class, but is quite opaque in terms of telling us if it has decompressed the image already. For instance, use the Core Graphics to decompress the image (see Listing 2.19).

Listing 2.19 Category on `NSData` to Decompress JPEG Data

```
//Place this line in an external header file
typedef void (^JPEGWasDecompressedCallback)(UIImage *decompressedImage);

-(void)af_decompressedImageFromJPEGDataWithCallback:
    (JPEGWasDecompressedCallback)callback
{
    uint8_t character;
    [self getBytes:&character length:1];

    if (character != 0xFF)
    {
        //This is not a valid JPEG.

        callback(nil);

        return;
    }

    // get a data provider referencing the relevant file
    CGDataProviderRef dataProvider =
        CGDataProviderCreateWithCFData((__bridge CFDataRef)self);

    // use the data provider to get a CGImage; release the data provider
    CGImageRef image =
        CGImageCreateWithJPEGDataProvider(dataProvider, NULL, NO,
            kCGRenderingIntentDefault);
    CGDataProviderRelease(dataProvider);

    // make a bitmap context of a suitable size to draw to, forcing decode
    size_t width = CGImageGetWidth(image);
    size_t height = CGImageGetHeight(image);
       size_t bytesPerRow = roundUp(width * 4, 16);
       size_t byteCount = roundUp(height * bytesPerRow, 16);
```

```
        void *imageBuffer = malloc(byteCount);

    if (width == 0 || height == 0)
    {
        dispatch_async(dispatch_get_main_queue(), ^{
            callback(nil);
        });
    }

    CGColorSpaceRef colourSpace = CGColorSpaceCreateDeviceRGB();

    CGContextRef imageContext =
    CGBitmapContextCreate(imageBuffer, width, height, 8, bytesPerRow,
        colour kCGImageAlphaNone | kCGImageAlphaNoneSkipLast);
        //Despite what the docs say these are not the same thing

    CGColorSpaceRelease(colourSpace);

    // draw the image to the context, release it
    CGContextDrawImage(imageContext, CGRectMake(0, 0, width, height), image);
    CGImageRelease(image);

    // now get an image ref from the context
    CGImageRef outputImage = CGBitmapContextCreateImage(imageContext);

    CGContextRelease(imageContext);
    free(imageBuffer);

    dispatch_async(dispatch_get_main_queue(), ^{
        UIImage *decompressedImage = [UIImage imageWithCGImage:outputImage];
        callback(decompressedImage);
        CGImageRelease(outputImage);
    });
}
```

This category is *very* useful. Call the decompression method on a background queue and everything is taken care of for you: The `NSData` instance is decompressed, if it is in fact a JPEG, on the queue that the method is invoked from. When the decompression is complete, it invokes a callback block and takes care of cleaning up the `CGImageRef` memory.

Now that you can safely decompress JPEGs on the background queue, incorporate that into your code, as in Listing 2.20.

Listing 2.20 Decompressing Images in a Background Thread

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    [data af_decompressedImageFromJPEGDataWithCallback:
        ^(UIImage *decompressedImage) {
        [cell setImage:decompressedImage];
    }];
});
```

Because JPEG decompression takes only a few milliseconds, I'm updating the cell directly in Listing 2.20. If you're decompressing JPEGs that are megabytes large, this isn't going to work for you, but displaying images that large in a collection view is a bad idea, generally.

If you rerun Instruments, you see that the most expensive operation, overall, is allocating space from `UICollectionViewCell`'s `initWithFrame:`. This is really good. Anecdotally, the app runs way smoother.

High five! But take a look at two other parts of the codebase that could be improved.

The images are 145 by 145 pixels, but your cells are 145 by 100 logical pixels. `UIImageView` is scaling the images down to fit. You could change the content mode to center them, instead, so that they aren't scaled.

Ideally, your image size and cell size should be the same so that the OS doesn't have to resize anything, which improves performance. However, if you're using a third-party API, you won't have control over the image size.

The only other thing I could recommend to improve performance of this example is to turn on the `masksToBounds` property of the cell's layer; you used this property in conjunction with the `cornerRadius`. This causes a strain on the CPU because it requires offscreen rendering passes and can cause a lot of problems. It's something to check if you can't figure out why your collection view is slow.

If the collection view background is opaque, you can use a PNG in a `UIImageView` subview of the `contentView` to mask out the corners. This is a good approach, too, but don't use resizable images if you can avoid doing so. If all your cells are the same size, use a nonresizable `UIImage` to mask the corners because it renders faster.

That's all for the first performance example. Take a look at the next example, called "Performance Problems Example II," in the same code. Build and run the app to get a feel for how it works.

The use case for this app is the following: You've been hired by an up-and-coming startup that just got some angel funding and who are building a social network for cats (see Figure 2.15). You are to prototype the "Facebook Wall" equivalent of their future mobile app so that they can grab millions in venture capital funding.

Figure 2.15 A social network for cats

The app displays comments in cells that have different background colors. The model is set up in `setupModel`. Just ignore this method; it is not relevant to the case study. Also, notice how Xcode lets you use emoji in Objective-C source code. How cool is that?

Profile the app and use the same Core Animation template as the last example. The peak frame rate is 48fps, which isn't terrible, but not ideal. When you open the Extended Detail pane, what you see should cause some alarm (see Figure 2.16).

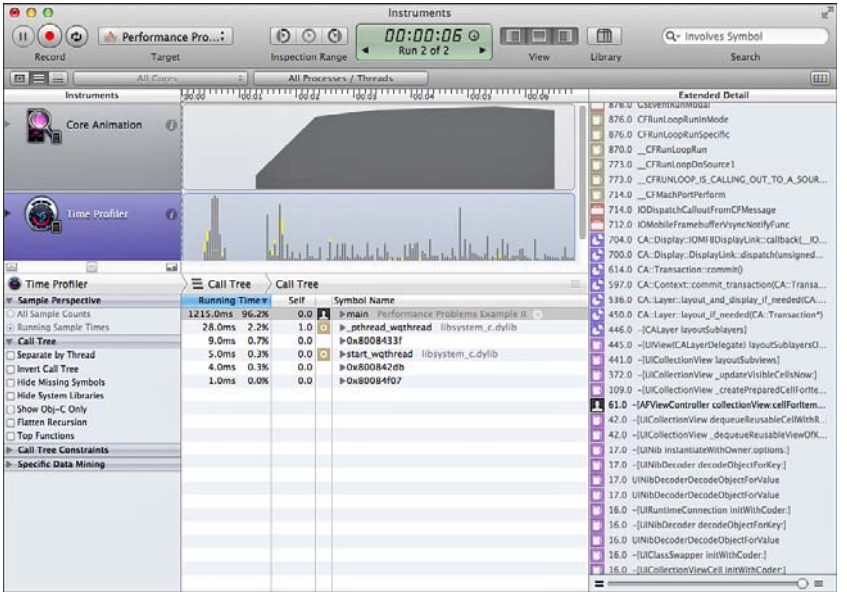


Figure 2.16 First profiling run in Instruments

The most expensive operation performance is unloading the .xib file. What's with that? Open `AFCollectionViewCell.xib` and bask in the sheer existential horror of the view hierarchy.

Obviously, Figure 2.17 represents a pedagogical example. You would never have a view hierarchy in a nib *quite* this bad. Even though you have a quite complex view hierarchy, all you're really doing is displaying some text on a colored background. You can draw this, and the equivalent of all those useless views, a lot faster in `drawRect:`. You can apply the same logic in your own cell subclasses; if you have a complex view hierarchy that takes too long to draw, implement `drawRect:` and ditch the view hierarchy. `drawRect:` can also be a slow performer, however, and using it in a simple example like this is only to illustrate how it's done. You should use it only when drawing components of your view manually is faster than rendering a necessarily complex view hierarchy.

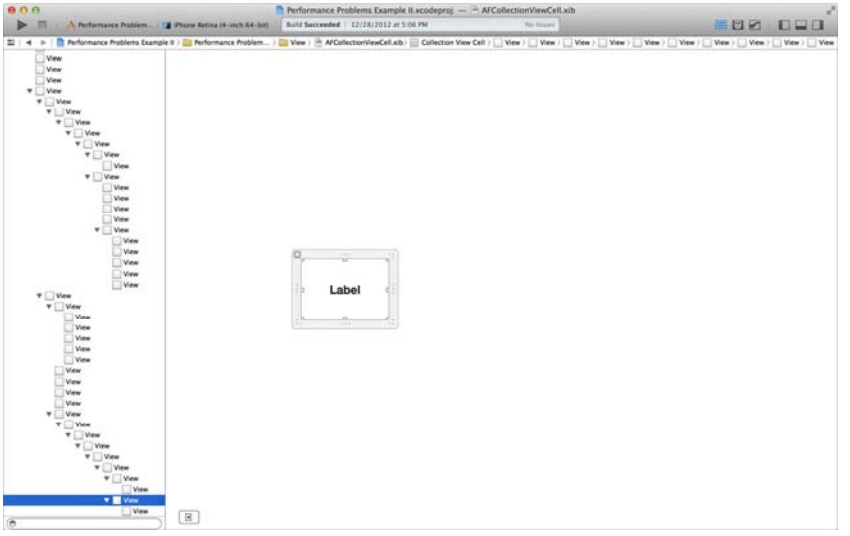


Figure 2.17 Crazy view hierarchy

Delete the .xib and the two properties from the cell's header file. Instead of registering a `UINib` in the view controller's `viewDidLoad`, register a `Class`. Instead of using a separate background view for the color, just draw it in `drawRect:`. Create a new string property for the cell's text. You're going to override the getter and setter for `backgroundColor` to do some clever drawing (see Listing 2.21).

Listing 2.21 New Cell Subclass

```
static inline void addRoundedRectToPath(CGContextRef context, CGRect rect, float
ovalWidth, float ovalHeight)
{
    float fw, fh;
    if (ovalWidth == 0 || ovalHeight == 0) {
        CGContextAddRect(context, rect);
        return;
    }
    CGContextSaveGState(context);
    CGContextTranslateCTM (context, CGRectGetMinX(rect), CGRectGetMinY(rect));
    CGContextScaleCTM (context, ovalWidth, ovalHeight);
    fw = CGRectGetWidth (rect) / ovalWidth;
    fh = CGRectGetHeight (rect) / ovalHeight;
    CGContextMoveToPoint(context, fw, fh/2);
    CGContextAddArcToPoint(context, fw, fh, fw/2, fh, 1);
    CGContextAddArcToPoint(context, 0, fh, 0, fh/2, 1);
    CGContextAddArcToPoint(context, 0, 0, fw/2, 0, 1);
    CGContextAddArcToPoint(context, fw, 0, fw, fh/2, 1);
```

```
    CGContextClosePath(context);
    CGContextRestoreGState(context);
}

@implementation AFCollectionViewCell
{
    UIColor *realBackgroundColor;
}

-(id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    self.opaque = NO;
    self.backgroundColor = [UIColor clearColor];

    return self;
}

-(void)prepareForReuse
{
    ...
    //Not relevant to this part of the case study
}

-(void)drawRect:(CGRect)rect
{
    CGContextRef context = UIGraphicsGetCurrentContext();

    CGContextSaveGState(context);

    [realBackgroundColor set];

    addRoundedRectToPath(context, self.bounds, 10, 10);
    CGContextClip(context);

    CGContextFillRect(context, self.bounds);

    CGContextRestoreGState(context);

    [[UIColor whiteColor] set];

    [self.text
        drawInRect:CGRectInset(self.bounds, 10, 10)
        withFont:[UIFont boldSystemFontOfSize:20]
        lineBreakMode:NSLineBreakByWordWrapping
```

```
        alignment:NSTextAlignmentCenter];
}

#pragma mark - Overridden Properties

-(void)setBackgroundColor:(UIColor *)backgroundColor
{
    [super setBackgroundColor:[UIColor clearColor]];

    realBackgroundColor = backgroundColor;

    [self setNeedsDisplay];
}

-(UIColor *)backgroundColor
{
    return realBackgroundColor;
}

-(void)setText:(NSString *)text
{
    _text = [text copy];

    [self setNeedsDisplay];
}

@end
```

The `addRoundedRectToPath` C method is handy and can be easily modified to only round certain corners. These methods should usually be placed in a separate source file so that they can be reused. Take a look at Listing 2.22.

Listing 2.22 Decompressing Images in a Background Thread

```
-(void)configureCell:(AFCollectionViewCell *)cell withModel:(AFModel *)model
{
    cell.backgroundColor = [model.color colorWithAlphaComponent:0.6f];
    cell.text = model.comment;
}
```

Listing 2.22 is an efficient implementation. The drawing code is straightforward and the code using the cell works well within the MVC architecture. Reprofile the application.

Huh. Figure 2.18 shows there's some method called `performLongRunningTask` being called when you dequeue a cell. It's severely hampering the frame refresh rate. You shouldn't be performing long-running tasks on the main thread, and if you look at the code, it's being called from `prepareForReuse`. The developer has conflated preparation for reuse with having shown the user the cell. This is really not acceptable. Refactor this logic into the view controller instead (see Listing 2.23).

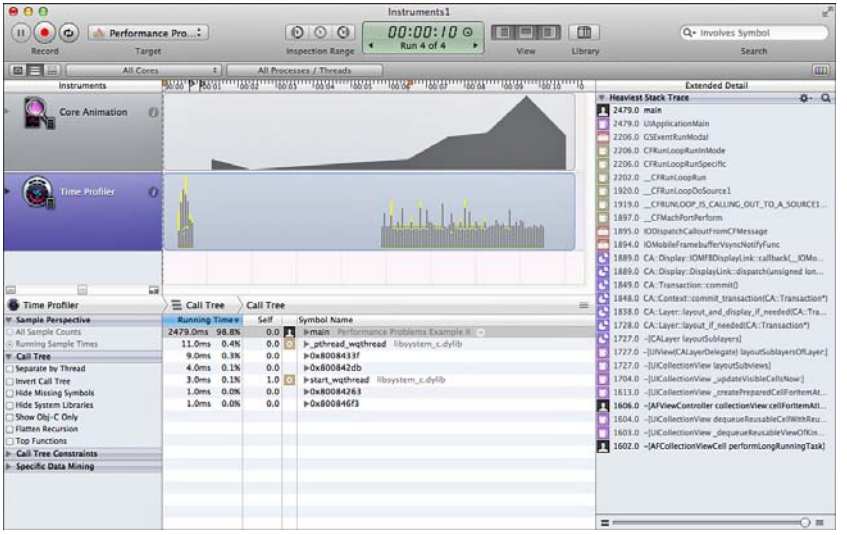


Figure 2.18 First profiling run in Instruments

Note

A little trick is going on. I have an empty `for` loop in `performLongRunningTask` to cause performance problems on purpose, but to get this example to work, I've had to disable compiler optimizations. LLVM is too smart, and it strips out the empty loop if compiler optimizations are enabled.

Listing 2.23 Refactoring Code out of the Cell Subclass

```
-(void)collectionView:(UICollectionView *)collectionView
didEndDisplayingCell:(UICollectionViewCell *)cell forItemAtIndexPath:(NSIndexPath
*)indexPath
{
    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0),
^{
        [self performLongRunningTask];
    });
}
```

```
-(void)performLongRunningTask
{
    /*
     Let's run some long-running task. Maybe
     it's some complicated view hierarchy math that
     could be simplified with Autolayout.
     */
    for (int i = 0; i < 5000000; i++);
}
```

Now the code to invoke the long-running task is in the appropriate place and the task is performed on a background queue. Great. Reprofile the app. The frame rate is something around 55fps, which is pretty good. The slowest part of the code is `drawRect:`, which can causes some performance problems. As stated earlier, `drawRect:` is only a good route to take when you have a necessarily complex view hierarchy.

3

Contextualizing Content

Displaying content to users with basic cells lets developers show off some spartan content. We can use supplementary views to spice things up. First, we look at some new delegate methods to customize the layout of individual cells within the collection view. Then, we apply the techniques and practices we learned in the preceding chapter to contextualize our content for our users. We do this with a chapter-long case study, which is called Survey in the sample code.

Supplementary Views

Supplementary views are views that scroll with the cells of a collection view and display some kind of information about the data of that collection. These are "data-driven" views, which stand in contrast to the decoration views we explore in the next chapter. For now, you just need to know that supplementary views provide supplemental information to the user; they *must* show some data, or else they're decoration views.

The data source provides the collection view with the information it needs to configure the supplementary views, but the supplementary views are laid out by the `UICollection-ViewLayout` object. iOS comes with the basic flow layout as a subclass of the general layout. For most cases, using flow layout gives you all the power and flexibility you need.

Built in to the `UICollectionViewFlowLayout` class are two supplementary views: headers and footers. Although these are the two built-in supplements to collection view cells, supplementary views can be used for *much* more than just headers and footers; headers and footers are just specific cases of supplementary views. You'll see more examples of supplementary views that aren't headers or footers in the next two chapters.

Remember: Any view that isn't a cell and displays data or metadata about your collection should be a supplementary view.

Supplementary views behave similarly to collection view cells; you register a `Class` or `UINib` with the collection view, and the supplementary views are reused over and over again. However, you also must provide a size for the supplementary views.

Suppose, for example, that we have an idea for a start-up, and we need to prototype the interface for our app. We're going to build an application to present users with a selection of photographs. They'll select one, and then based on their selections, we'll present new ones to them that we think they might like. This recommendation engine is completely fake, but we just need to code the user interface to get that sweet, sweet VC cash.

Our faked-out recommendation engine will use two types of models:

- A simple wrapper for a name and an image and we'll call it `AFPhotoModel`
- A list of `AFPhotoModel`s we're asking the user to select from

Each list represents one section in our collection view. Let's call this second model `AFSelectionModel`. Their interfaces are shown in Listing 3.1, but they are straightforward.

Listing 3.1 Interfaces for Models Used in the Survey App

```
//AFPhotoModel.h
@interface AFPhotoModel : NSObject

+(instancetype)photoModelWithName:(NSString *)name image:(UIImage *)image;

@property (nonatomic, copy) NSString *name;
@property (nonatomic, strong) UIImage *image;

@end


//AFSectionModel.h
extern const NSUInteger AFSelectionModelNoSelectionIndex;

@interface AFSelectionModel : NSObject

+(instancetype)selectionModelWithPhotoModels:(NSArray *)photoModels;

@property (nonatomic, strong, readonly) NSArray *photoModels;
@property (nonatomic, assign) NSUInteger selectedPhotoModelIndex;
@property (nonatomic, readonly) BOOL hasBeenSelected;

@end
```

The view controller will hold on to an array of section model objects, which are set up for you in the private `setupModel` method. All the photos come with the sample code. The view controller also knows which section we're currently prompting the user for, called `currentModelArrayIndex`.

Let's say that we want to add a plain supplementary view as a header to a collection view. How would we do that? Well, just like with cells, we'll create a new class called `AFCollectionHeaderView`, except this header will subclass `UICollectionReusableView`. This superclass provides some of the same functionality as `UICollectionViewCell`, but is much more lightweight. Supplementary views, out of the box, do not support the advanced features, such as selection and highlighting, that cells do. Listing 3.2 shows the implementation I used for my supplementary view.

Listing 3.2 Supplementary View Implementation

```
//Header file
@interface AFCollectionHeaderView : UICollectionReusableView

@property (nonatomic, copy) NSString *text;

@end

// Implementation File
@implementation AFCollectionHeaderView
{
    UILabel *textLabel;
}

- (id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    textLabel = [[UILabel alloc] initWithFrame:CGRectInset(
        CGRectMake(0, 0, CGRectGetWidth(frame), CGRectGetHeight(frame)), 30,
10)];
    textLabel.backgroundColor = [UIColor clearColor];
    textLabel.textColor = [UIColor whiteColor];
    textLabel.font = [UIFont boldSystemFontOfSize:20];
    textLabel.autoresizingMask = UIViewAutoresizingFlexibleHeight |
        UIViewAutoresizingFlexibleWidth;
    [self addSubview:textLabel];

    return self;
}

-(void)prepareForReuse
```

```
{
    [super prepareForReuse];

    [self setText:@""];
}

-(void)setText:(NSString *)text
{
    _text = [text copy];

    [textLabel setText:text];
}

@end
```

Everything is pretty similar to cells, even `prepareForReuse`. The label's frame is a little wonky, but we're just taking our frame and insetting it by 30 points from the left and 10 points from the top. This gives it a nice margin.

Providing Supplementary Views

Now, we need to register the `Class` object with the collection view so it can create supplementary views for itself (see Listing 3.3). We'll put this code in `viewDidLoad`.

Listing 3.3 Registering Supplementary Views

```
// After we have set up the flow layout
surveyFlowLayout.headerReferenceSize = CGSizeMake(60, 50);

// After we have set up the collection view
[surveyCollectionView registerClass:[AFCollectionHeaderView class]
         forSupplementaryViewOfKind:UICollectionElementKindSectionHeader
                withReuseIdentifier:HeaderIdentifier];
```

The `HeaderIdentifier` is a static `NSString`, similar to the one that we've used before for cells. The second parameter is a string to specify what kind of supplementary view you're registering. We're using the built-in *header* kind, `UICollectionElement-KindSectionHeader`. There is also `UICollectionElementKindSectionFooter`. These two supplementary views are provided by the flow layout, but you can specify your own (as you'll see in later chapters).

The `headerReferenceSize` tells the collection view layout how large to make the headers. If you forget to set this, the default is zero, so your headers won't be displayed. This is a common mistake; so if your headers aren't appearing, check to make sure that you're specifying a size.

How this size value is interpreted is actually pretty interesting. When scrolling horizontally, only the `width` of the `CGSize` you specify is used; the header is stretched vertically to fill its space. When scrolling vertically, only the `height` of the `CGSize` is used; the header is stretched horizontally to fill its space. The layout of the headers (and footers) is shown in Figure 3.1. (The green arrows indicate the scroll direction.)

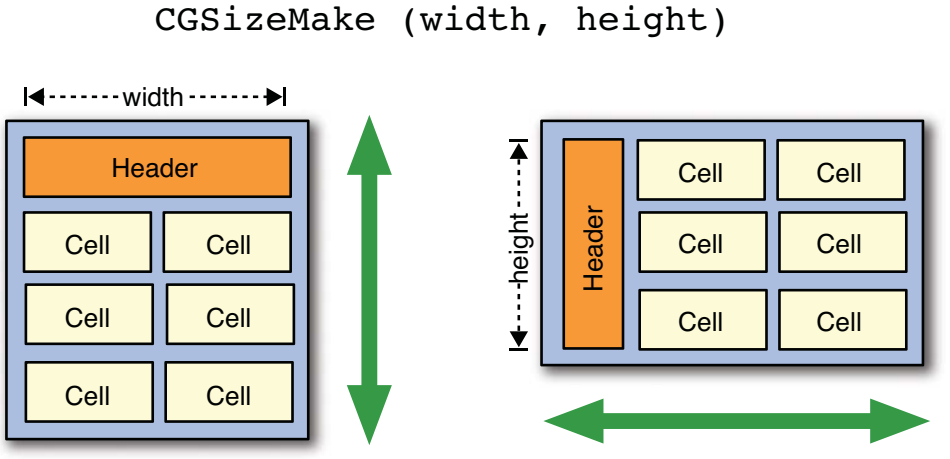


Figure 3.1 Basic collection view using storyboards

Now that we have registered the `Class` and specified a size, it's time to actually return the supplementary view. We do this by implementing a new method in `UICollectionViewDelegate` called `collectionView:viewForSupplementaryElementOfKind:atIndexPath:`. The second parameter is the same "kind" of string we used when registering our header. We only have one type of supplementary view, so we'll ignore the parameter in this example. However, if you have a collection view with more than one supplementary view type, you'll have to be careful to only return the correct kind (see Listing 3.4).

Listing 3.4 Providing Supplementary Views

```
-(UICollectionReusableView *)collectionView:(UICollectionView
*)collectionView viewForSupplementaryElementOfKind:(NSString *)kind
atIndexPath:(NSIndexPath *)indexPath
{
    //Provides a view for the headers in the collection view
```

```
    AFCollectionHeaderView *headerView = (AFCollectionHeaderView
*)[collectionView dequeueReusableSupplementaryViewOfKind:kind
withReuseIdentifier:HeaderIdentifier forIndexPath:indexPath];

    if (indexPath.section == 0)
    {
        //If this is the first header, display a prompt to the user
        [headerView setText:@"Tap on a photo to start the recommendation
engine."];
    }
    else if (indexPath.section <= currentModelArrayIndex)
    {
        //Otherwise, display a prompt using the selected photo from the previous
section
        AFSelectionModel *selectionModel = selectionModelArray[indexPath.section
- 1];

        AFPhotoModel *selectedPhotoModel = [self
photoModelForIndexPath:[NSIndexPath
indexPathForItem:selectionModel.selectedPhotoModelIndex
inSection:indexPath.section - 1]];

        [headerView setText:[NSString stringWithFormat:@"Because you liked
%@...", selectedPhotoModel.name]];
    }

    return headerView;
}
```

If we're prompting users for the first section, we'll give them some instructions on what to do. Otherwise, we know what photo they selected last time (it's stored in our array or `AFSelectionModels`), so we'll tell them that, based on their last selection, we think they'll like these next photos.

Let's hook up our data source methods to start showing the user some content other than header views (see Listing 3.5).

Listing 3.5 `UICollectionViewDataSource` Methods for Survey

```
-(NSInteger)numberOfSectionsInCollectionView:(UICollectionView
*)collectionView
{
    // Return the smallest of either our current model index plus one,
    // or our total number of sections. This will show 1 section when we
    // only want to display section zero, etc.
    // It will prevent us from returning 11 when we only have 10 sections.
    return MIN(currentModelArrayIndex + 1, selectionModelArray.count);
```

```
}

-(NSInteger)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section
{
    //Return the number of photos in the section model
    return [[selectionModelArray[currentModelArrayIndex] photoModels] count];
}

-(UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    AFCollectionViewCell *cell = (AFCollectionViewCell *)[collectionView
        dequeueReusableCellWithReuseIdentifier:CellIdentifier
        forIndexPath:indexPath];

    //Configure the cell
    [self configureCell:cell forIndexPath:indexPath];

    return cell;
}
```

This should be familiar to you by now. The only vexing line is `configureCell:forIndexPath:`, because we have not yet implemented it or a custom cell subclass. Let's do both now.

I want the photos to be presented with a white matte border.

After users make a selection, we want the section to look grayed out with the photo they selected staying selected. We'll make our background color white and place the image view inset by 10 points on all sides. The code for the cell looks like Listing 3.6.

Listing 3.6 `UICollectionViewCell` Subclass for Survey

```
//Header File
@interface AFCollectionViewCell : UICollectionViewCell

@property (nonatomic, strong) UIImage *image;

-(void)setDisabled:(BOOL)disabled;

@end

//Implementation File
@implementation AFCollectionViewCell
{
```

```
    UIImageView *imageView;
}

- (id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    imageView = [[UIImageView alloc] initWithFrame:CGRectZero];
    imageView.backgroundColor = [UIColor blackColor];
    [self.contentView addSubview:imageView];

    UIView *selectedBackgroundView = [[UIView alloc] initWithFrame:CGRectZero];
    selectedBackgroundView.backgroundColor = [UIColor orangeColor];
    self.selectedBackgroundView = selectedBackgroundView;

    self.backgroundColor = [UIColor whiteColor];

    return self;
}

-(void)prepareForReuse
{
    [super prepareForReuse];

    [self setImage:nil];
    [self setSelected:NO];
}

-(void)layoutSubviews
{
    imageView.frame = CGRectInset(self.bounds, 10, 10);
}

-(void)setImage:(UIImage *)image
{
    _image = image;

    imageView.image = image;
}

-(void)setDisabled:(BOOL)disabled
{
    self.contentView.alpha = disabled ? 0.5f : 1.0f;
    self.backgroundColor = disabled ? [UIColor grayColor] : [UIColor whiteColor];
}
@end
```

The implementation is fairly straightforward. We define a `setDisabled:` method that we'll use to gray out the cell once the user has make a selection for that section. Everything else should look familiar from Chapter 2, "Displaying Content Using `UICollectionView`."

As you can see, in the header implementation, we used autoresizing masks to keep the label stretched to the width of the header. Here, we're using `layoutSubviews` to reposition the image view. These are both valid approaches and sometimes autoresizing masks or Autolayout are either too complicated or too cumbersome to use.

Now let's look at that `configureCell:forIndexPath:` method (see Listing 3.7).

Listing 3.7 Configuring Cells for Survey

```
-(void)configureCell:(AFCollectionViewCell *)cell
        forIndexPath:(NSIndexPath *)indexPath
{
    //Set the image for the cell
    [cell setImage:[[self photoModelForIndexPath:indexPath] image]];

    //By default, assume the cell is not disabled
    //and not selected
    [cell setDisabled:NO];
    [cell setSelected:NO];

    //If the cell is not in our current last index, disable it
    if (indexPath.section < currentModelArrayIndex)
    {
        [cell setDisabled:YES];

        //If the cell was selected by the user previously,
        //select it now
        if (indexPath.row == [selectionModelArray[indexPath.section]
            selectedPhotoModelIndex])
        {
            [cell setSelected:YES];
        }
    }
}
```

The code grabs the specific photo model and sets the image from it to the cell. It then checks whether the cell is in a section that's already had a selection made (to gray it out), and then checks whether the cell was the one that was selected. Notice this isn't a property; we don't want the cell to contain information about the model. We just need a convenient method to change its appearance.

Responding to User Interactions

Now comes the fun part! We're going to write code to respond to user selections. There is a `UICollectionViewDelegate` method that's called whenever the user makes a selection (but not when the developer makes a selection programmatically). Take a look at Listing 3.8.

Listing 3.8 Responding to Cell Selection

```
-(void)collectionView:(UICollectionView *)collectionView
didSelectItemAtIndexPath:(NSIndexPath *)indexPath
{
    //The user has selected a cell

    //No matter what, deselect that cell
    [collectionView deselectItemAtIndexPath:indexPath animated:YES];

    if (currentModelArrayIndex >= selectionModelArray.count - 1)
    {
        //Let's just present some dialogue to indicate things are done.
        [[[UIAlertView alloc] initWithTitle:@"Recommendation Engine"
            message:@"Based on your selections, we have concluded you have
excellent taste in photography!"
            delegate:nil cancelButtonTitle:nil
            otherButtonTitles:@"Awesome!", nil] show];

        return;
    }

    //Set the selected photo index
    [selectionModelArray[currentModelArrayIndex]
        setSelectedPhotoModelIndex:indexPath.item];

    [collectionView performBatchUpdates:^{
        currentModelArrayIndex++;
        [collectionView insertSections:[NSIndexSet
            indexSetWithIndex:currentModelArrayIndex]];
        [collectionView reloadSections:[NSIndexSet
            indexSetWithIndex:currentModelArrayIndex-1]];
    } completion:^(BOOL finished) {
        [collectionView scrollToItemAtIndexPath:[NSIndexPath indexPathForItem:0
            inSection:currentModelArrayIndex]
             atScrollPosition:UICollectionViewScrollPositionTop animated:YES];
    }];
}
```

This method is called when a cell has been selected. First, we want to deselect the cell. Then, we'll check to see if we have completed the survey. If so, we'll set an ivar to `YES` and throw up a dialog to the users. If they have not finished, we need to update the model (setting the selected index) and add the next section. This is done in `performBatchUpdates:completion:`. This is a block-based approach to updating the collection view. Updates we make to the model and the collection view in the first block are executed with simultaneous animations.

Apple's documentation is clear that the model should be updated first, and then the appropriate changes made to the collection view. We want to reload the section that the user just made the selection in and add the next section. The same approach can be used to remove items or sections, as you'll see in later chapters. Table 3.1 shows the different methods for modifying collection view content.

Table 3.1 Methods for Modifying Collection View Content

Method Name
`insertSections:`
`deleteSections:`
`reloadSections:`
`moveSection:toSection:`
`insertItemsAtIndexPaths:`
`deleteItemsAtIndexPaths:`
`reloadItemsAtIndexPaths:`
`moveItemAtIndexPath:toIndexPath:`

If we ran the app as it exists right now, we'd get something like Figure 3.2. It's not a bad start, but if we show this to investors, we won't get the VC funding we deserve. There are a few things wrong with the app so far.

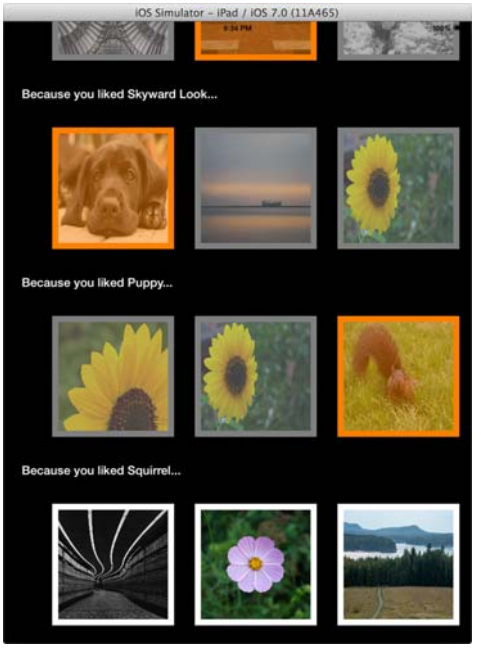


Figure 3.2 Running the Survey application

First thing, the photos are stretched. Unlike in Chapter 2, the photos don't all have a square aspect ratio. That means that when we put them into square cells, they get stretched. That's not ideal. What *would* be ideal is if we could have different cells be different sizes. Now there's an idea! Your cell size is usually set on the flow layout object. However, the flow object has its own delegate protocol. We can *override* the settings on the flow layout object by responding to its delegate methods (see Listing 3.9).

Listing 3.9 Providing Different Cell Sizes

```
-(CGSize)collectionView:(UICollectionView *)collectionView
                 layout:(UICollectionViewLayout *)collectionViewLayout
 sizeForItemAtIndexPath:(NSIndexPath *)indexPath
{
    //Provides a different size for each individual cell

    //Grab the photo model for the cell
    AFPhotoModel *photoModel = [self photoModelForIndexPath:indexPath];

    //Determine the size and aspect ratio for the model's image
    CGSize photoSize = photoModel.image.size;
    CGFloat aspectRatio = photoSize.width / photoSize.height;
```

```
    //start out with the detail image size of the maximum size
    CGSize itemSize = kMaxItemSize;

    if (aspectRatio < 1)
    {
        //The photo is taller than it is wide, so constrain the width
        itemSize = CGSizeMake(kMaxItemSize.width * aspectRatio,
            kMaxItemSize.height);
    }
    else if (aspectRatio > 1)
    {
        //The photo is wider than it is tall, so constrain the height
        itemSize = CGSizeMake(kMaxItemSize.width,
            kMaxItemSize.height / aspectRatio);
    }

    return itemSize;
}
```

This code calculates the aspect ratio of the photo and returns an item size that's equivalent to an aspect stretch fit content mode. The photos will be scaled to fit within a maximum size. I've included landscape, portrait, and square-crop photos in the example to show you that it works (see Figure 3.3). Let's look at what the app looks like now.

Figure 3.3 Multiple sizes of cells

That's much better! It would look even more polished if the cells were always evenly spaced, but that requires subclassing the flow layout, so we'll save it for Chapter 4, "Organizing Content with `UICollectionViewFlowLayout`."

Another problem with the app is that you can make selections on sections that have already had selections made. That's a serious flaw. Luckily, there are a pair of methods that the collection view interrogates before it highlights or selects cells. We'll only implement the first one to prevent highlighting. Because highlighting is the first step of becoming selected, we won't need to implement the method to disable selection.

Note

Preventing highlighting and selection applies only to user interaction. Highlighting and selecting programmatically will continue to work.

Listing 3.10 will only allow highlighting if the section is the one we're currently prompting the user for and so long as the user hasn't already finished the survey.

Listing 3.10 Disabling Cell Highlighting

```
-(BOOL)collectionView:(UICollectionView *)collectionView
shouldHighlightItemAtIndexPath:(NSIndexPath *)indexPath
```

```
{
    return indexPath.section == currentModelArrayIndex && !isFinished;
}
```

Table 3.2 provides a full list of the ways you can customize the user's interaction with your collection view.

Table 3.2 Methods for Customizing Selection Behavior

Method Name
`collectionView:shouldHighlightItemAtIndexPath:`
`collectionView:shouldSelectItemAtIndexPath:`
`collectionView:shouldDeselectItemAtIndexPath:`

Providing Cut/Copy/Paste Support

We're getting closer to that VC round, I can feel it! However, this chapter is called "Contextualizing Content," and I think we could make this app more contextually aware. What if users had the ability to copy the name of the photo when they used a tap-and-hold gesture? I know what you're thinking: "Adding interactivity isn't until Chapter 6, 'Adding Interactivity to `UICollectionView`.'" Well, yes, that's true. However, copy, cut, and paste are built in to collection view! Let's take a look at how easy it is to get that functionality.

First, we need to let the collection view know that we support menus. There's a simple delegate method for this, as shown in Listing 3.11.

Listing 3.11 Enabling Menus for Collection Views

```
-(BOOL)collectionView:(UICollectionView *)collectionView
 shouldShowMenuForItemAtIndexPath:(NSIndexPath *)indexPath
{
    return YES;
}
```

Notice that you can selectively enable the copy/cut/paste menu for individual cells instead of for an entire collection view. Very awesome. Because all of our cells represent photos with names, we'll just return `YES` for everything.

Next, the collection view will interrogate its delegate for the different types of actions it can perform. It will call the method in Listing 3.12 over and again for each action it supports.

Currently, the only actions are `cut:`, `copy:`, and `paste:`, but Apple may add more later, so write code to accommodate for that.

Listing 3.12 Selectively Enabling Menu Functions for Collection Views

```
-(BOOL)collectionView:(UICollectionView *)collectionView
     canPerformAction:(SEL)action
   forItemAtIndexPath:(NSIndexPath *)indexPath
           withSender:(id)sender
{
    if ([NSStringFromSelector(action) isEqualToString:@"copy:"])
    {
        return YES;
    }

    return NO;
}
```

The method is passed a `SEL` selector and asked if the delegate can perform that action on the model represented by that index path. We need to turn the selector into its string equivalent and compare it to `"copy:"`, the only selector we want to support.

Next is the easy part—actually performing the copy operation:

```
-(void)collectionView:(UICollectionView *)collectionView
        performAction:(SEL)action
   forItemAtIndexPath:(NSIndexPath *)indexPath
           withSender:(id)sender
{
    if ([NSStringFromSelector(action) isEqualToString:@"copy:"])
    {
        UIPasteboard *pasteboard = [UIPasteboard generalPasteboard];
        [pasteboard setString:[[self photoModelForIndexPath:indexPath] name]];
    }
}
```

We just check again to determine that we want to perform the copy operation, grab the general pasteboard from the system, and set the string.

Now that you understand better the collection view and the data source and delegate protocols, let's move on to Chapter 4, where we closely examine `UICollection-ViewFlowLayout`.

4

Organizing Content with `UICollectionViewFlowLayout`

You now have the skills to use `UICollectionView` to display custom content to your users and can display cells as well as supplementary views. Up until now, we have focused on the actual content, not how it's organized on the screen. This chapter explores how `UICollectionView` is architected to use `UICollectionViewLayout` to organize its content. We take a close look at `UICollectionViewFlowLayout` and how subclassing it can get you a lot of customizability without a lot of extra work. We finish with a short history lesson as we explore `UITableView` and how it's related to `UICollectionView`.

What Is a Layout?

`UICollectionViewLayout` is an abstract class that should not be created itself; its only purpose is to be subclassed. Each collection view has a layout associated with it whose job it is to lay content out. Layouts are *not* concerned with the data contained in the views they lay out; they are only interested in their layout to the user.

`UICollectionViewFlow` is a direct subclass lays out content in line-based, line-breaking style. We've already seen `UICollectionViewFlowLayout` in its most basic form, a grid. We spend the rest of this chapter exploring the power that a simple flow layout subclass gives you as a developer. You can create astounding layouts with very little code, if you know where to put it.

A layout subclass has a few responsibilities. The collection view relies on its layout to tell it how to display its cells. This is a key concept: Layout content is *not* done by subclassing `UICollectionView`. Although this is a common pattern for layout subviews when subclassing `UIScrollView`, we're going to avoid subclassing `UICollectionView` unless absolutely necessary.

So, a collection view asks its layout for clues about how to lay out its content. What's the actual sequence of events that happens when a collection view displays content to a user?

First, the collection view interrogates its data source for information about the contents to be displayed to the user. This includes the number of sections and the number of items and supplementary views in each individual section.

Next, the collection view gathers information from its layout object about how to display the cells, supplementary views, and decoration views. This information is stored in instances of a class called `UICollectionViewLayoutAttributes`.

Finally, the collection view forwards information about the layout to the cells, supplementary views, and decoration views. Each of these classes is responsible for using the information it has been given to apply those layout attributes to itself. Deferring to the superclass's implementation, or omitting an implementation entirely, will ensure that the layout attributes already handled by the collection view, like `frame`, are applied. Your implementations should concentrate on any custom attributes that you've added (but more on that later).

These steps occur whenever the existing layout is invalidated, which you can force by calling `invalidateLayout` on the layout object.

Now you are aware of the different classes used in laying content out: `UICollectionView`, which is the view that presents content to the user; `UICollectionViewCell`, which is responsible for displaying one unit of content to a user at a time; `UICollectionViewLayout`, which determines the attributes of items and returns that information to the collection view; and `UICollectionViewLayoutAttributes`, which is a class in which the layout stores information to be marshaled to the cells, supplementary views, and decoration views.

If you step back and look at these classes, a clear division exists between which ones are involved in data and their own layouts and those that are *solely* responsible for layout. Figure 4.1 shows the division: `UICollectionView` collects information about the data from the classes in the orange box and combines it with information about the layout from the classes in the blue box.

Notice that the layout has an indirect reference to the delegate. This connection can be used by the layout to interrogate the delegate about information concerning the layout of specific items. For example, the `UICollectionViewDelegateFlowLayout` protocol extends the `UICollectionViewDelegate` and is used by `UICollectionViewFlowLayout` to ask the delegate about item-specific layout information. This topic is complicated, but you've already seen an example of this in the preceding chapter, when the delegate specified individual dimensions for different items. You'll see an example later of extending this further.

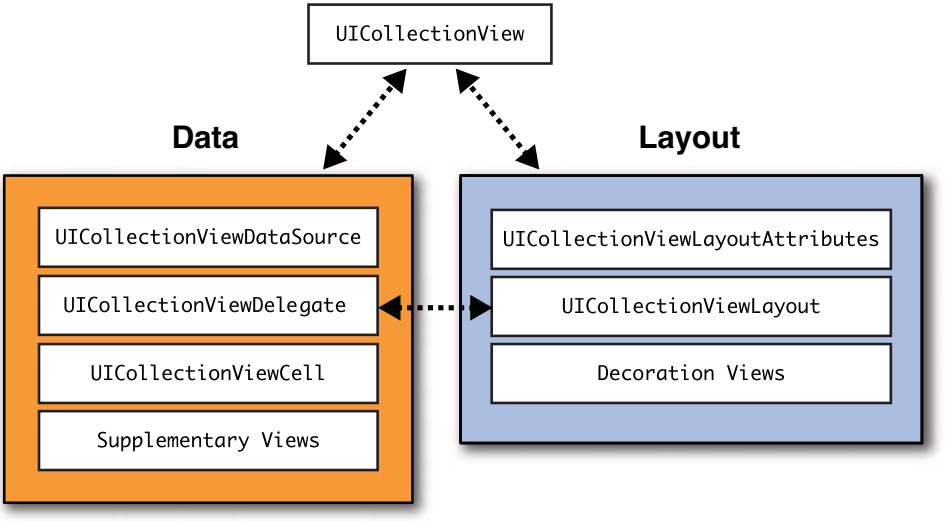


Figure 4.1 Distinction between data and layout classes

We've covered the basics: what a layout is and what it does and how it interacts with the rest of the collection view architecture. This has been, so far, very academic. Let's get to some code.

Subclassing `UICollectionViewFlowLayout`

We've already seen a lot of complex behavior and layouts be generated using the built-in `UICollectionViewFlowLayout`, so why would one choose to subclass it? There are a number of reasons:

- To modify the attributes of the layout you're subclassing beyond what is possible with delegate methods
- To incorporate decoration views in your layout
- To add new kinds of supplementary views
- To extend `UICollectionViewLayoutAttributes` to add new attributes of items for your layout class to manage
- To add gesture support
- To customize the animation of insertion, update, and deletion updates to the collection view

With the exception of the gesture support, covered in Chapter 6, "Adding Interactivity to `UICollectionView`," covers, we will look at code examples for each of the reasons to subclass flow layout.

Let's look back at our Survey example—the code for which is in Better Survey. There are a few ways that we can make this better, and the first one is shown in Figure 4.2. Because not all of our cells have the same size, cells won't be aligned vertically anymore. Out of the box, `UICollectionViewFlowLayout` does not provide support for the kind of "evenly spaced-out" feel that I think would look better here. Luckily, what we want falls under the "line-based, breaking layout" category of flow layouts, so I think we'll be able to subclass `UICollectionViewFlowLayout` to accomplish the visual style we're going for.

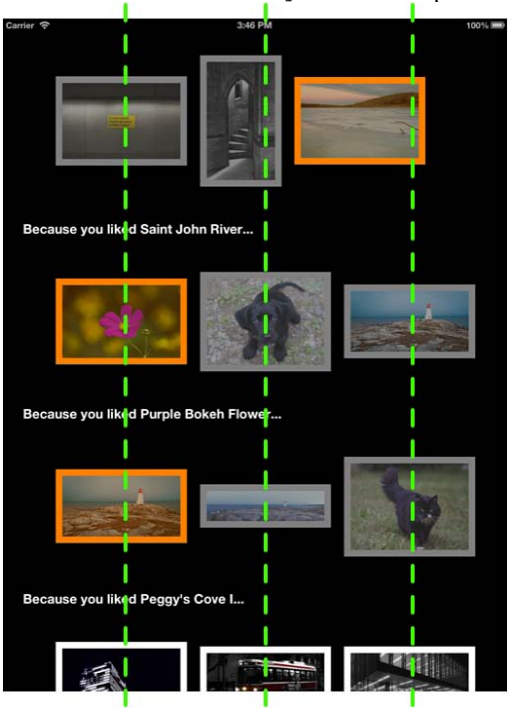


Figure 4.2 Flow layout not aligning cells in a grid

I'm going to create a new file in Xcode and call it `AFCollectionViewFlowLayout`; it's going to subclass `UICollectionViewFlowLayout` (see Listing 4.1). Next, we can move a lot of the layout logic out of the view controller into our layout.

Listing 4.1 `AFCollectionViewFlowLayout` **Header File**

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

#define kMaxItemDimension   200.0f
```

```
#define kMaxItemSize        CGSizeMake(kMaxItemDimension, kMaxItemDimension)

@interface AFCollectionViewFlowLayout : UICollectionViewFlowLayout
@end
```

You can see that we've moved the maximum cell size into the header file for the layout. This is a more appropriate place for it than in the view controller.

Next, we'll implement our own `init` method so we can set up our properties there (see Listing 4.2).

Listing 4.2 `AFCollectionViewFlowLayout` **Initializer**

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

    self.sectionInset = UIEdgeInsetsMake(30.0f, 80.0f, 30.0f, 20.0f);
    self.minimumInteritemSpacing = 20.0f;
    self.minimumLineSpacing = 20.0f;
    self.itemSize = kMaxItemSize;
    self.headerReferenceSize = CGSizeMake(60, 70);

    return self;
}
```

Finally, we need to change the creation of the layout object in the view controller. Use `#import` to import the `AFCollectionViewFlowLayout` header and change the creation of the layout and collection view to the code shown in Listing 4.3.

Listing 4.3 Simplified Layout and Collection View Creation

```
AFCollectionViewFlowLayout *surveyFlowLayout =
    [[AFCollectionViewFlowLayout alloc] init];

UICollectionView *surveyCollectionView =


    [[UICollectionView alloc]


        initWithFrame:CGRectZero collectionViewLayout:surveyFlowLayout];
```

By moving the setup of the layout into its initializer, we've written a lot less code in the view controller. In addition, if we ever reuse the layout, we don't have repeated code in two places. A view controller reusing this layout could always further customize the layout properties, but they don't *have* to. This is a good pattern you should adhere to when writing your own custom layouts.

Next, we need to override two methods in our `UICollectionViewFlowLayout` subclass, which will be called when the collection view is laying out its cells, supplementary views, and decoration views. The two methods are `layoutAttributesForElementsInRect:` and `layoutAttributesForItemAtIndexPath:`. We're also going to have create a third, private method called `applyLayoutAttributes:`, which we discuss later. Both of the overridden methods will call this custom one (see Listing 4.4).

Listing 4.4 Applying Customized Attributes

```
-(NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    NSArray *attributesArray = [super layoutAttributesForElementsInRect:rect];

    for (UICollectionViewLayoutAttributes *attributes in attributesArray)
    {
        [self applyLayoutAttributes:attributes];
    }

    return attributesArray;
}

-(UICollectionViewLayoutAttributes *)
    layoutAttributesForItemAtIndexPath:(NSIndexPath *)indexPath
{
    UICollectionViewLayoutAttributes *attributes = [super
layoutAttributesForItemAtIndexPath:indexPath];

    [self applyLayoutAttributes:attributes];

    return attributes;
}
```

The first thing both of our methods do is call their superclass's implementation. By doing this, we get all of the `UICollectionViewFlowLayout` behavior for free. After we retrieve the default attributes, we'll tweak them ourselves.

Now let's look at `applyLayoutAttributes:`. We first check the layout attributes' `representedElementKind` property. For normal `UICollectionViewCells`, this will be nil. Otherwise, it will be the supplementary view type registered with the collection view; in our case, it would be `UICollectionElementKindSectionHeader`.

One other point worth keeping in mind is that center and size define the position and size, respectively, of an item. When calculating these, you can end up rendering views on half-pixels, making them blurry. The `frame` property is a convenience method for accessing the size and center of the layout attributes. By setting the frame to be the `CGRectIntegral` of itself (see Listing 4.5), we ensure that views are not rendered on pixel boundaries.

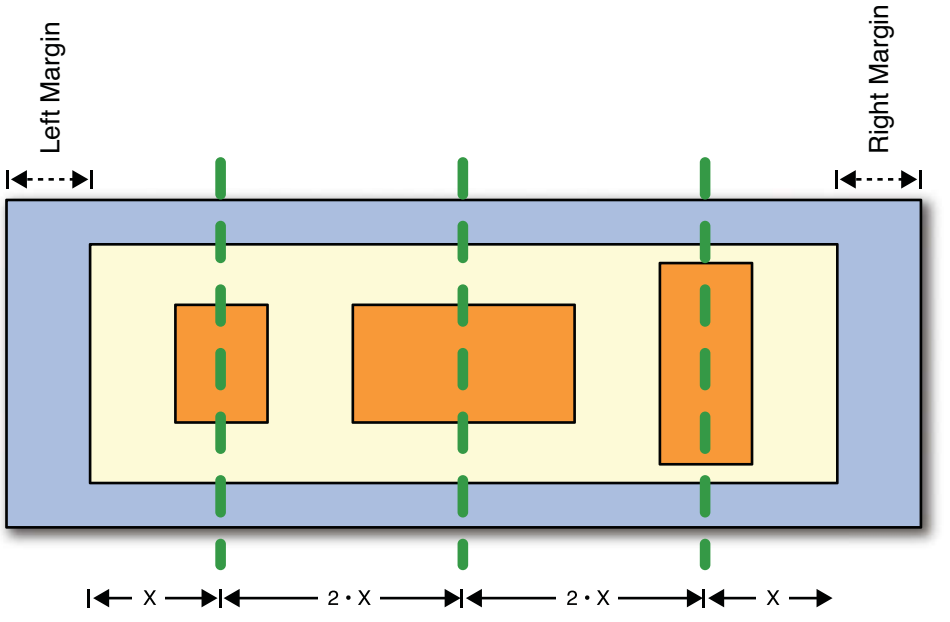
Listing 4.5 Applying Customized Attributes

```
-(void)applyLayoutAttributes:(UICollectionViewLayoutAttributes *)
    attributes
{
    // Check for representedElementKind being nil, indicating this is
    // a cell and not a header or decoration view
    if (attributes.representedElementKind == nil)
    {
        CGFloat width = [self collectionViewContentSize].width;
        CGFloat leftMargin = [self sectionInset].left;
        CGFloat rightMargin = [self sectionInset].right;

        NSUInteger itemsInSection = [[self collectionView]
            numberOfItemsInSection:attributes.indexPath.section];
        CGFloat firstXPosition =
             (width - (leftMargin + rightMargin)) / (2 * itemsInSection);
        CGFloat xPosition = firstXPosition +
             (2*firstXPosition*attributes.indexPath.item);

        attributes.center = CGPointMake(leftMargin + xPosition,
            attributes.center.y);
        attributes.frame = CGRectIntegral(attributes.frame);
    }
}
```

Listing 4.5 is only a codified version of the formula laid out in Figure 4.3. It has been generalized to allow an arbitrary number of items in each row, instead of just three.



$$\text{Total Width} = \text{Left Margin} + \text{Right Margin} + 6 \cdot X$$

$$X = \frac{\text{Total Width} - (\text{Left Margin} + \text{Right Margin})}{6}$$

Figure 4.3 Math to distribute items evenly

Ah, you knew there would be some math in this ebook eventually! But, it's actually not that complicated.

If we were to run the application again, we would see that the cells are spread out evenly, as shown in Figure 4.4.

Figure 4.4 Items evenly horizontally distributed

Now that we've got our cells laid out in a nice grid pattern, let's add a decoration view. Decoration views are visual supplements to `UICollectionView`'s data-driven content. They don't display information about the cells; instead, they accompany the cells for visual effect: a designer's best friend.

I'm no designer, but I've managed to come up with the idea of a binder. Our app is going to flaunt this "flat design" craze and lay our photos on top of a three-ring binder. I've taken a photo of a binder and stretched it out. We want to have this decoration view lay behind each row of photos.

Because decoration views are not data driven, there will be no code added to the view controller. Instead, all the code for the decoration view will live inside our `AFCollectionViewFlowLayout` and a subclass of `UICollectionReusableView`. This class, `UICollectionReusableView`, is the parent class of `AFCollectionHeaderView` and even `UICollectionViewCell`. It provides common logic for reusing any particular view in a collection view, which includes cells, supplementary views, and decoration views. Because these classes can be reused, we can take what we've already learned about reuse and apply it to decoration views. Let's do so now.

Create a new class that extends `UICollectionReusableView`. I call mine `AFDecorationView`. It has no properties, and its implementation looks rather boring (see Listing 4.6).

Listing 4.6 Decoration View Implementation

```
@implementation AFDecorationView
{
    UIImageView *binderImageView;
}

- (id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    binderImageView = [[UIImageView alloc]
        initWithImage:[UIImage imageNamed:@"binder"]];
    binderImageView.frame = CGRectMake(10, 0,
        CGRectGetWidth(frame), CGRectGetHeight(frame));
    binderImageView.contentMode = UIViewContentModeLeft;
    binderImageView.autoresizingMask = UIViewAutoresizingFlexibleHeight |
UIViewAutoresizingFlexibleWidth;
    [self addSubview:binderImageView];

    return self;
}

@end
```

All this class does is, when initialized, adds a `UIImageView` to its view hierarchy with our "binder" image in it. There is no need to override `prepareForReuse` because there is no data-specific content in our decoration view.

Now that we have created our decoration view subclass, let's add it to our collection view. This is a little trickier than the header views because nothing is built in to `UICollectionView` for us; we need to build everything ourselves.

`#import` the decoration view's header file into the layout subclass. Modify the implementation of `layoutAttributesForElementsInRect:` to look like Listing 4.7.

Listing 4.7 Decoration View Implementation

```
-(NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    NSArray *attributesArray = [super
        layoutAttributesForElementsInRect:rect];
```

```
    NSMutableArray *newAttributesArray = [NSMutableArray array];

    for (UICollectionViewLayoutAttributes *attributes in attributesArray)
    {
        [self applyLayoutAttributes:attributes];

     // THIS IF STATEMENT WAS ADDED
        if (attributes.representedElementCategory ==
            UICollectionElementCategorySupplementaryView)
        {
            UICollectionViewLayoutAttributes *newAttributes =

                [self layoutAttributesForDecorationViewOfKind:
                AFCollectionViewFlowLayoutBackgroundDecoration
                atIndexPath:attributes.indexPath];

            [newAttributesArray addObject:newAttributes];
        }
    }

    attributesArray = [attributesArray
        arrayByAddingObjectsFromArray:newAttributesArray];

    return attributesArray;
}
```

The `if` statement checking the element category of the layout attributes was added. We want to add one decoration view per section, and each section has only one header, so we'll piggy-back on that logic to add our supplementary view.

The code itself may look a little strange. Remember that `layoutAttributesForElementsInRect:` is called for *all* types of elements, not just cells. So, when it's called for our header view, our `if` statement evaluates to `YES` and we create a new layout attribute. The array we return will include this new attribute.

Next, we need an implementation for `layoutAttributesForDecorationViewOfKind:atIndexPath:` because the default implementation returns `nil`, and when we try to add it to our mutable dictionary, our app would crash.

We need to implement a method that will create a new `UICollectionViewLayoutAttributes` object and customize its properties so that the decoration view would fit behind our cell contents (see Listing 4.8).

Listing 4.8 Creating Decoration View Layout Attributes

```
-(UICollectionViewLayoutAttributes
*)layoutAttributesForDecorationViewOfKind:(NSString *)decorationViewKind
atIndexPath:(NSIndexPath *)indexPath
```

```
{
    UICollectionViewLayoutAttributes *layoutAttributes =
        [UICollectionViewLayoutAttributes
            layoutAttributesForDecorationViewOfKind:decorationViewKind
                                        withIndexPath:indexPath];

    if ([decorationViewKind
        isEqualToString:AFCollectionViewFlowLayoutBackgroundDecoration])
    {
        UICollectionViewLayoutAttributes *tallestCellAttributes;
        NSInteger numberOfCellsInSection = [self.collectionView
            numberOfItemsInSection:indexPath.section];

        for (NSInteger i = 0; i < numberOfCellsInSection; i++)
        {
            NSIndexPath *cellIndexPath = [NSIndexPath
                indexPathForItem:i
                       inSection:indexPath.section];

            UICollectionViewLayoutAttributes *cellAttribtes = [self
                layoutAttributesForItemAtIndexPath:cellIndexPath];

            if (CGRectGetHeight(cellAttribtes.frame) >
                CGRectGetHeight(tallestCellAttributes.frame))
            {
                tallestCellAttributes = cellAttribtes;
            }
        }

        CGFloat decorationViewHeight =
            CGRectGetHeight(tallestCellAttributes.frame) +
            self.headerReferenceSize.height;

        layoutAttributes.size = CGSizeMake(
            [self collectionViewContentSize].width, decorationViewHeight);
        layoutAttributes.center = CGPointMake(
            [self collectionViewContentSize].width / 2.0f,
            tallestCellAttributes.center.y);

        // Place the decoration view behind all the cells
        layoutAttributes.zIndex = -1;
    }

    return layoutAttributes;
}
```

This implementation creates a new `UICollectionViewLayoutAttributes` object using the class method `layoutAttributesForDecorationViewOfKind:withIndexPath:`. Then, it customizes the properties in the attributes depending on what we're looking for. We want our decoration view to be vertically centered with the tallest item in its section, so we need to loop over each of those. Luckily, the logic to retrieve these attributes has already been implemented in `layoutAttributesForItemAtIndexPath:`. When we ask our super class for the attributes for a given cell, it will query the collection view delegate for the size (code we've already written).

We can leverage this existing functionality to handle the heavy lifting. We're not calculating the center of the decoration view, really, we're just relying on the vertical center of the tallest item, which has already been calculated for us. Hooray!

So, after we define the size and height of our decoration view, we need to set its z-index. This tells the collection view which order to render its items in. Items that overlap but have the same z-index have an undefined rendering order. We want the decoration view to render *behind* all the cells, which have the default z-index of 0, so we set our decoration view's z-index to -1.

The only other thing we need to do is register our decoration view class with the layout (see Listing 4.9). We'll add the highlighted following line to the `AFCollectionViewFlowLayout`'s `init` method.

Listing 4.9 Registering a Decoration View with a Layout

```
NSString * const AFCollectionViewFlowLayoutBackgroundDecoration =
@"DecorationIdentifier";

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

    self.sectionInset = UIEdgeInsetsMake(30.0f, 80.0f, 30.0f, 20.0f);
    self.minimumInteritemSpacing = 20.0f;
    self.minimumLineSpacing = 20.0f;
    self.itemSize = kMaxItemSize;
    self.headerReferenceSize = CGSizeMake(60, 70);
    // THIS LINE WAS ADDED!
    [self registerClass:[AFDecorationView class]
forDecorationViewOfKind:AFCollectionViewFlowLayoutBackgroundDecoration];

    return self;
}
```

Amazing! Figure 4.5 shows we're nearly there. The final thing I think this demo app could use is some nice animations. Support for animations is already built into `UICollectionViewLayout`; we just need to implement a few methods.

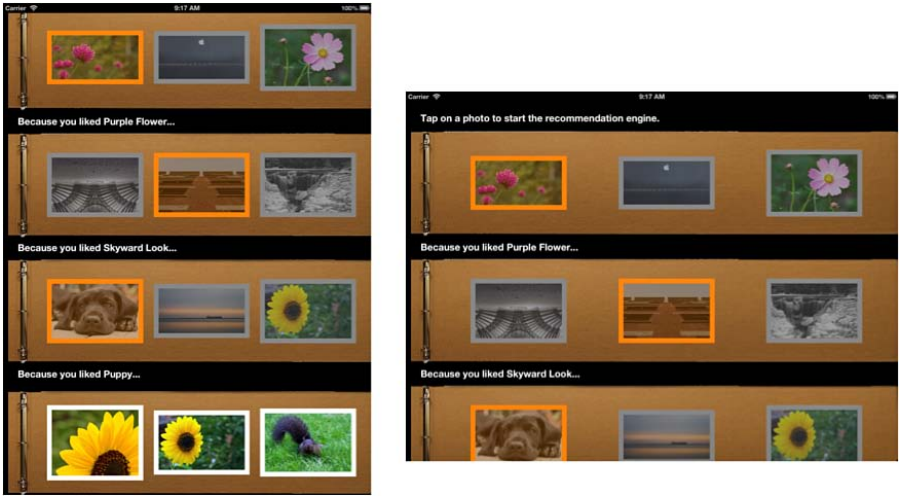


Figure 4.5 Decoration views implemented

`initialLayoutAttributesForAppearingItemAtIndexPath:` is called whenever a new item is added or updated to the collection view. We can use it to supply initial layout attributes for the item at the beginning of the animation and the collection view will interpolate animatable properties, like `frame` and `alpha`, to their normal position. There is also a corresponding method called `finalLayoutAttributesForDisappearingItemAtIndexPath:` for animating removal of items from the collection view.

We can animate more than just items, though. There are corresponding appear/disappear methods for supplementary views and decoration views. The default implementation of `UICollectionViewLayout` returns `nil`, indicating a simple crossfade. We can also return nil to use a crossfade.

The remaining problem is that we reload other sections when we insert a new one. This will cause more than just the appearing sections to animate. Let's limit which sections we animate.

Before any updates are performed to the collection view, `prepareForCollectionViewUpdates:` is called with an array of `UICollectionViewUpdateItem` objects as a parameter. These are the updates that are about to happen. After they are completed, `finalizeCollectionViewUpdates` is called. These come in pairs. We'll create an instance variable `NSMutableSet` to hold on to the sections that are being inserted. We use a set because it has constant-time lookup (see Listing 4.10).

Listing 4.10 Updated `init` Method to Create Mutable Set

```
@implementation AFCollectionViewFlowLayout
```

```
{
    NSMutableSet *insertedSectionSet;
}

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

    self.sectionInset = UIEdgeInsetsMake(30.0f, 80.0f, 30.0f, 20.0f);
    self.minimumInteritemSpacing = 20.0f;
    self.minimumLineSpacing = 20.0f;
    self.itemSize = kMaxItemSize;
    self.headerReferenceSize = CGSizeMake(60, 70);
    [self registerClass:[AFDecorationView class]
forDecorationViewOfKind:AFCollectionViewFlowLayoutBackgroundDecoration];

    // NOTICE THIS NEW SHINY LINE?
    insertedSectionSet = [NSMutableSet set];

    return self;
}
```

Now we just need implementations of `prepareForCollectionViewUpdates:` and `finalizeCollectionViewUpdates` to update the set. It is *very important* to always call your super implementation for these methods (see Listing 4.11).

Listing 4.11 Updating the Mutable Set Contents

```
-(void)prepareForCollectionViewUpdates:(NSArray *)updateItems
{
    [super prepareForCollectionViewUpdates:updateItems];

    [updateItems enumerateObjectsUsingBlock:^(UICollectionViewUpdateItem
        *updateItem, NSUInteger idx, BOOL *stop) {
        if (updateItem.updateAction == UICollectionUpdateActionInsert)
        {
            [insertedSectionSet
                addObject:@(updateItem.indexPathAfterUpdate.section)];
        }
    }];
}

-(void)finalizeCollectionViewUpdates
{
    [super finalizeCollectionViewUpdates];
```

```
    [insertedSectionSet removeAllObjects];
}
```

You can see that, when we are preparing for updates, our layout checks the update action to see if it's an item that's being inserted. If so, it adds an `NSNumber` instance representing the item's index path's section to the set. Duplicates are ignored in sets, so we don't have to check whether it already exists.

When the updates have been finalized, we remove all the items from the mutable set, resetting it to an empty state for the next batch of updates.

Now that we have that out of the way, let's look at the code for animating in the items and the decoration view, which is shown in Listing 4.12.

Listing 4.12 Animating in Cells and Decoration Views

```
-(UICollectionViewLayoutAttributes *)
    initialLayoutAttributesForAppearingDecorationElementOfKind:(NSString*)
    elementKind atIndexPath:(NSIndexPath *)decorationIndexPath
{
    //returning nil will cause a crossfade

    UICollectionViewLayoutAttributes *layoutAttributes;

    if ([elementKind
        isEqualToString:AFCollectionViewFlowLayoutBackgroundDecoration])
    {
        if ([insertedSectionSet
            containsObject:@(decorationIndexPath.section)])
        {
            layoutAttributes = [self
                layoutAttributesForDecorationViewOfKind:elementKind
                atIndexPath:decorationIndexPath];
            layoutAttributes.alpha = 0.0f;
            layoutAttributes.transform3D = CATransform3DMakeTranslation(
                -CGRectGetWidth(layoutAttributes.frame), 0, 0);
        }
    }

    return layoutAttributes;
}



-(UICollectionViewLayoutAttributes *)
    initialLayoutAttributesForAppearingItemAtIndexPath:(NSIndexPath *)
```

```
    itemIndexPath
{
    //returning nil will cause a crossfade

    UICollectionViewLayoutAttributes *layoutAttributes;

    if ([insertedSectionSet containsObject:@(itemIndexPath.section)])
    {
        layoutAttributes = [self
            layoutAttributesForItemAtIndexPath:itemIndexPath];
        layoutAttributes.transform3D = CATransform3DMakeTranslation(
            [self collectionViewContentSize].width, 0, 0);
    }

    return layoutAttributes;
}
```

Because the default implementations return nil, we don't have to worry about calling super.

The two implementations are similar because they construct very similar animations. For decoration views, we check to make sure that the decoration view is the one we've set up; although there are no other decoration views, this is good practice in case we add more later.

In either case, we check to ensure that the index path's section of the item is included in our set of sections that were inserted. If it is, we grab an instance of `UICollectionViewLayoutAttributes` from our earlier implementations of `layoutAttributesForItemAtIndexPath:` or `layoutAttributesForDecorationViewOfKind:atIndexPath:` — we're leveraging the code we've already written.

Then, we set up a transform to move the decoration view left and the cells right so that they are completely out of the visible collection view when the animation starts. We also set the decoration view's `alpha` to zero so that it fades in.

Now, whenever a new section is inserted, the user sees the binder move in from the left and the photos move in from the right. This is a really nice touch.

One of the key architectural takeaways you should have from this section is that writing `UICollectionViewFlowLayout` subclasses is all about relying on existing code wherever possible. If you find yourself doing complex math to calculate something that's already laid out, check to see whether there is some way you can access that information.

Laying Out Items with Custom Attributes

`UICollectionViewLayoutAttributes` is a class, which means we can subclass it. Why would we want to do that? To add support for more attributes, of course! Let's look at what I mean.

The class contains the following properties, which are applied to items at runtime:

- Frame (convenience property for center and size)
- Center
- Size
- 3D Transform
- Alpha (opacity)
- Z-index
- Hidden
- Element category (cell, supplementary view, or decoration view)
- Element kind (nil for cells)

These are all really great, and you can accomplish a lot with them as they are. However, sometimes, you might want to add your own.

That's what we're going to do now.

This project is called Dimensions in the sample code. It has some images and model setup done, which I do not cover here. The problem it's trying to solve is that photos sometimes look best when stretched to aspect fill, clipping off the extra bits of the image to fit in its container. Other times, you want to use aspect fit, which will scale down the image so that the entire thing is visible within a container. We're going to write a layout that will handle this for us as a layout attribute.

I've created a new Xcode project with the Single View application template. After removing the .xib, I changed the main window setup in the application delegate to look like Listing 4.13.

Listing 4.13 Setting Up the View Controller

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]];
    UINavigationController *navigationController =
        [[UINavigationController alloc]
        initWithRootViewController:[[AFViewController alloc] init]];
    navigationController.navigationBar.barStyle = UIBarStyleBlack;

    self.viewController = navigationController;
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];

    return YES;
```

```
}
```

All we've done is set up a navigation controller with a root custom view controller that Xcode created for us and that we'll implement in a moment. Note that I had to change the type of the `viewController` property to be a generic `UIViewController`.

Now that we have our view controller onscreen, we can set up the collection view and layout (see Listing 4.14).

Listing 4.14 Setting Up the Collection View

```
@implementation AFViewController
{
    //Array of model objects
    NSArray *photoModelArray;

    UISegmentedControl *aspectChangeSegmentedControl;

    AFCollectionViewFlowLayout *photoCollectionViewLayout;
}

//Static identifier for cells
static NSString *CellIdentifier = @"CellIdentifier";

-(void)loadView
{
    // Create our view

    // Create an instance of our custom flow layout.
    photoCollectionViewLayout = [[AFCollectionViewFlowLayout alloc] init];

    // Create a new collection view with our flow layout and set
    // ourself as delegate and data source.
    UICollectionView *photoCollectionView = [[UICollectionView alloc]
        initWithFrame:CGRectZero
        collectionViewLayout:photoCollectionViewLayout];
    photoCollectionView.dataSource = self;
    photoCollectionView.delegate = self;

    // Register our classes so we can use our custom
    // subclassed cell and header
    [photoCollectionView registerClass:[AFCollectionViewCell class]
        forCellWithReuseIdentifier:CellIdentifier];

    // Set up the collection view geometry to cover the whole
    // screen in any orientation and other view properties.
```

```
    photoCollectionView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
    photoCollectionView.allowsSelection = NO;
    photoCollectionView.indicatorStyle = UIScrollViewIndicatorStyleWhite;

    // Finally, set our collectionView (since we are a collection
    // view controller, this also sets self.view)
    self.collectionView = photoCollectionView;

    // Set up our model
    [self setupModel];
}
```

This should be familiar code to you by now. Note that we've disabled selection for all cells in the collection view. We also have a segmented control as an instance variable. This is going to go in the navigation bar so the user can select between aspect fit and aspect fill.

We're going to implement the `AFCollectionViewFlowLayout` class referenced in `loadView` in a moment, but let's look at the rest of the view controller code first. It sets up the segmented control in our navigation bar (see Listing 4.15).

Listing 4.15 Setting Up the Segmented Control

```
-(void)viewDidLoad
{
    [super viewDidLoad];

    aspectChangeSegmentedControl = [[UISegmentedControl alloc]
        initWithItems:@[@"Aspect Fit", @"Square"]];
    aspectChangeSegmentedControl.selectedSegmentIndex = 0;
    aspectChangeSegmentedControl.segmentedControlStyle =
UISegmentedControlStyleBar;
    [aspectChangeSegmentedControl addTarget:self
        action:@selector(aspectChangeSegmentedControlDidChangeValue:)
        forControlEvents:UIControlEventValueChanged];

    self.navigationItem.titleView = aspectChangeSegmentedControl;
}
```

The rest of the view controller implementation is pretty standard (see Listing 4.16).

Listing 4.16 Boilerplate `UICollectionViewController`

```
-(AFPhotoModel *)photoModelForIndexPath:(NSIndexPath *)indexPath
{
    if (indexPath.item >= [photoModelArray count]) return nil;
```

```
    return photoModelArray[indexPath.item];
}

-(void)configureCell:(AFCollectionViewCell *)cell forIndexPath:(NSIndexPath
*)indexPath
{
    //Set the image for the cell
    [cell setImage:[[self photoModelForIndexPath:indexPath] image]];
}

-(NSInteger)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section
{
    return [photoModelArray count];
}

-(UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    AFCollectionViewCell *cell = (AFCollectionViewCell *)[collectionView
dequeueReusableCellWithReuseIdentifier:CellIdentifier forIndexPath:indexPath];

    //Configure the cell
    [self configureCell:cell forIndexPath:indexPath];

    return cell;
}
```

The last remaining method in our view controller is going to be the method to respond to the user interacting with the segmented control (see Listing 4.17).

Listing 4.17 Responding to User Interaction with Segmented Control

```
-(void)aspectChangeSegmentedControlDidChangeValue:(id)sender
{
    // We need to explicitly tell the collection view layout
    // that we want the change animated.
    [UIView animateWithDuration:0.5f animations:^{
        // This just swaps the two values

        if (photoCollectionViewLayout.layoutMode ==
            AFCollectionViewFlowLayoutModeAspectFill)
        {
            photoCollectionViewLayout.layoutMode =
                AFCollectionViewFlowLayoutModeAspectFit;
```

```
        }
        else
        {
            photoCollectionViewLayout.layoutMode =
                AFCollectionViewFlowLayoutModeAspectFill;
        }
    }];
}
```

We haven't defined the `layoutMode` property yet, so let's do that now. This is where the custom layout attributes subclass comes in. We want to add a new layout attribute to specify the scaling mode of photos. Create a new class that subclasses `UICollectionViewLayout-Attributes` (see Listing 4.18).

Listing 4.18 `UICollectionViewLayoutAttributes` **Subclass Header**

```
typedef enum : NSUInteger{
    AFCollectionViewFlowLayoutModeAspectFit,    //Default
    AFCollectionViewFlowLayoutModeAspectFill
}AFCollectionViewFlowLayoutMode;

@interface AFCollectionViewLayoutAttributes : UICollectionViewLayoutAttributes

@property (nonatomic, assign) AFCollectionViewFlowLayoutMode layoutMode;

@end
```

That's all we really need—a definition of layout modes and a property to hold them. However, look at the definition of `UICollectionViewLayoutAttributes`; notice that it conforms to the `NSCopying` protocol. It is *very important* that we also conform to this protocol and implement `copyWithZone:` (see Listing 4.19). Otherwise, our property will always be zero (as guaranteed by the compiler). *New in iOS 7:* You now *must* override `isEqual:` when subclassing layout attributes.

Listing 4.19 `UICollectionViewLayoutAttributes` **Subclass Implementation**

```
@implementation AFCollectionViewLayoutAttributes

-(id)copyWithZone:(NSZone *)zone
{
    AFCollectionViewLayoutAttributes *attributes = [super copyWithZone:zone];

    attributes.layoutMode = self.layoutMode;
```

```
    return attributes;
}

-(BOOL)isEqual:(id)object {
    return [super isEqual:object] &&
        (self.layoutMode == [object layoutMode]);
}

@end
```

Now we can implement our flow layout subclass. I've created a new class called `AFCollectionViewFlowLayout` that subclasses `UICollectionViewFlowLayout`. It's shown in Listing 4.20 and should look familiar from the improved Survey app shown earlier in this chapter.

Listing 4.20 Custom Flow Layout Header

```
#import "AFCollectionViewLayoutAttributes.h"

#define kMaxItemDimension   140
#define kMaxItemSize        CGSizeMake(kMaxItemDimension, kMaxItemDimension)

@protocol AFCollectionViewDelegateFlowLayout <UICollectionViewDelegateFlowLayout>

@optional
-(AFCollectionViewFlowLayoutMode)collectionView:(UICollectionView
*)collectionView layout:(UICollectionViewLayout*)collectionViewLayout
layoutModeForItemAtIndexPath:(NSIndexPath *)indexPath;

@end

@interface AFCollectionViewFlowLayout : UICollectionViewFlowLayout

@property (nonatomic, assign) AFCollectionViewFlowLayoutMode layoutMode;

@end
```

What we've done is extend the `UICollectionViewDelegateFlowLayout` protocol to create our own. Just like we customized the size of individual cells for the Survey app, we want to provide an interface where developers using our layout can specify individual aspect ratios for the photos in their cells.

Now that we have our custom layout attributes class, let's take a brief look at the parts of our custom layout that you should already be familiar with (see Listing 4.21).

Listing 4.21 Custom Flow Layout Implementation

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

    // Some basic setup. 140+140 + 3*13 ~= 320, so we can get a
    //  two-column grid in portrait orientation.
    self.itemSize = kMaxItemSize;
    self.sectionInset = UIEdgeInsetsMake(13.0f, 13.0f, 13.0f, 13.0f);
    self.minimumInteritemSpacing = 13.0f;
    self.minimumLineSpacing = 13.0f;

    return self;
}

-(void)applyLayoutAttributes:(AFCollectionViewLayoutAttributes *)attributes
{
    // Check for representedElementKind being nil, indicating this
    // is a cell and not a header or decoration view
    if (attributes.representedElementKind == nil)
    {
        // Pass our layout mode onto the layout attributes
        attributes.layoutMode = self.layoutMode;

        if ([self.collectionView.delegate respondsToSelector:
            @selector(collectionView:layout:
            layoutModeForItemAtIndexPath:)])
        {
            attributes.layoutMode =
[(id<AFCollectionViewDelegateFlowLayout>)self.collectionView.delegate
collectionView:self.collectionView layout:self
layoutModeForItemAtIndexPath:attributes.indexPath];
        }
    }
}

-(NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    NSArray *attributesArray = [super layoutAttributesForElementsInRect:rect];

    for (AFCollectionViewLayoutAttributes *attributes in attributesArray)
    {
        [self applyLayoutAttributes:attributes];
    }

    return attributesArray;
```

```
}

-(UICollectionViewLayoutAttributes *)layoutAttributesForItemAtIndexPath:
    (NSIndexPath *)indexPath
{
    AFCollectionViewLayoutAttributes *attributes =
(AFCollectionViewLayoutAttributes *)[super
layoutAttributesForItemAtIndexPath:indexPath];

    [self applyLayoutAttributes:attributes];

    return attributes;
}
```

This is the same kind of code we saw in our first flow layout subclass earlier in the chapter. The difference is that we're using `AFCollectionViewLayoutAttributes` instead of `UICollectionViewLayoutAttributes` and we're passing on our `layoutMode`.

In `applyLayoutAttributes:`, we check the collection view's delegate to see whether it responds to the selector we defined in the `AFCollectionViewDelegateFlowLayout` protocol. If it does, we cast it to an `id` conforming to the protocol so we can grab the layout mode from it.

Observant readers might be asking themselves how the collection view knows to use our custom subclass of `UICollectionViewLayoutAttributes`. The answer is pretty easy. There is a class method our layout needs to implement that tells the collection view which custom class to use (see Listing 4.22). Obviously, the default implementation returns `UICollectionViewLayoutAttributes`.

Listing 4.22 Using a Custom Layout Attributes Class

```
+(Class)layoutAttributesClass
{
    return [AFCollectionViewLayoutAttributes class];
}
```

The only other component missing is that our layout can end up in an invalid state. If we change our layout mode without updating the cells that are already laid out on the screen, cells that are already visible will have the old layout, whereas ones that become visible due to scrolling or insertion will have the new layout. What we need is to call `invalidateLayout` whenever our layout mode changes (see Listing 4.23).

Listing 4.23 Invalidating Layout in Overridden Setter

```
-(void)setLayoutMode:(AFCollectionViewFlowLayoutMode)layoutMode
{
```

```
    // Update our backing ivar...
    _layoutMode = layoutMode;

    // then invalidate our layout.
    [self invalidateLayout];
}
```

I know this has been a lot of code with no payoff, but bear with me a little longer. Even though we have our custom layout and are setting the custom property, we still don't have any code that applies that property to the cell. I've created a `UICollectionViewCell` subclass called `AFCollectionViewCell`. It displays the image set by its `setImage:` method. The implementation, shown in Listing 4.24, is almost identical to the one used in the Survey app from Chapter 3. However, two key differences exist.

First, we're declaring an instance variable for the layout mode, and second, we're using that instance variable in a new method that sets the image view's frame. The issue is related to changes made under the hood in iOS 7; methods are now called in a different order, so it's important to set the image's frame whenever a new image is set (which makes sense because the image's frame depends on the image's aspect ratio, which we don't know until we have the `UIImage` instance).

Listing 4.24 Standard Collection View Cell Displaying an Image

```
@implementation AFCollectionViewCell
{
    UIImageView *imageView;
    AFCollectionViewFlowLayoutMode layoutMode;
}


- (id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    // Set up our image view
    imageView = [[UIImageView alloc] initWithFrame:CGRectMake(0, 0,
        CGRectGetWidth(frame), CGRectGetHeight(frame))];
    imageView.contentMode = UIViewContentModeScaleAspectFill;
    imageView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
        UIViewAutoresizingFlexibleHeight;
    imageView.clipsToBounds = YES;
    [self.contentView addSubview:imageView];

    // This will make the rest of our cell, outside the image view, appear
transparent against a black background.
    self.backgroundColor = [UIColor blackColor];
```

```
    return self;
}

-(void)prepareForReuse
{
    [super prepareForReuse];

    [self setImage:nil];
}

#pragma mark - Public Methods

-(void)setImage:(UIImage *)image
{
    [imageView setImage:image];
    [self setImageViewFrame];
}

-(void)setImageViewFrame {
    //start out with the detail image size of the maximum size
    CGSize imageViewSize = self.bounds.size;

    if (layoutMode == AFCollectionViewFlowLayoutModeAspectFit)
    {
        //Determine the size and aspect ratio for the model's image
        CGSize photoSize = imageView.image.size;
        CGFloat aspectRatio = photoSize.width / photoSize.height;

        if (aspectRatio < 1)
        {
            //The photo is taller than it is wide, so constrain the width
            imageViewSize = CGSizeMake(CGRectGetWidth(self.bounds) *
                aspectRatio, CGRectGetHeight(self.bounds));
        }
        else if (aspectRatio > 1)
        {
            //The photo is wider than it is tall, so constrain the height
            imageViewSize = CGSizeMake(CGRectGetWidth(self.bounds),
                CGRectGetHeight(self.bounds) / aspectRatio);
        }
    }

    // Set the size of the imageView ...
    imageView.bounds = CGRectMake(0, 0,
        imageViewSize.width, imageViewSize.height);
```

```
    // And the center, too.
    imageView.center = CGPointMake(CGRectGetMidX(self.bounds),
        CGRectGetMidY(self.bounds));
}


@end
```

Importantly, the image view's `clipsToBounds` property is set to `YES`. This makes sure that when the photo is being scaled to fit within the image view and clips part of itself, the clipped regions won't be visible.

Next, we have the code to actually apply the layout mode to the cell (see Listing 4.25).

Listing 4.25 Applying Custom Layout Attributes

```
-(void)applyLayoutAttributes:(UICollectionViewLayoutAttributes *)layoutAttributes
{
    [super applyLayoutAttributes:layoutAttributes];

    // Important! Check to make sure we're actually this special subclass.
    // Failing to do so could cause the app to crash!
    if (![layoutAttributes isKindOfClass:[AFCollectionViewLayoutAttributes
class]])
    {
        return;
    }

    AFCollectionViewLayoutAttributes *castedLayoutAttributes =
        (AFCollectionViewLayoutAttributes *)layoutAttributes;


    layoutMode = castedLayoutAttributes.layoutMode;

    [self setImageViewFrame];
}
```

This method belongs to `UICollectionReusableView` because layout attributes are applicable to cells, supplementary views, and decoration views. First, you *must* call super's implementation. Next, it checks to ensure that the layout attributes are an instance of our custom subclass before casting the pointer.

We use the layout mode to determine if we should leave the image view's `size` set to our `bounds` size, or if we should adjust it. If the mode is aspect fit, we adjust it using similar logic to the survey view controller in Chapter 3, "Contextualizing Content." Finally, we set the

`bounds` and the `center` of the image view. We use the size and position instead of the `contentMode` so that we can easily animate the transition from one mode to another. (`bounds` and `center` are implicitly animatable properties.)

Finally, after all that code, you can run the app and transition between aspect fit and aspect fill photos (see Figure 4.6). It will animate the transition, even if animating a scroll or rotation.

Figure 4.6 Aspect fit and aspect fill layout modes

Going Beyond Grids

So far, all we've seen flow layout do is some variation on a grid. While a grid is, indeed, a line-based, breaking layout, it is just one specific case of such a layout. Let's take things further and do something really fun.

We're going to build a Cover Flow layout. Before we do, I want to especially thank Mark Pospesel for building his Introducing Collection Views project on GitHub. The code in this section of my book draws heavily upon his examples, used with his permission. The sample code for this section is available under the name of Cover Flow.

The first thing we'll do, after our standard "create a single-view Xcode project and remove the .xib file" is to open the project settings in the project navigator pane. Under Build Phases, expand Link Binary with Libraries and click the plus sign. Select QuartzCore and

open up the Prefix file under the Supporting Files group. Mine is called CoverFlow-Prefix.pch; it's a header file that's imported into all header files. Add `#import <QuartzCore/QuartzCore.h>` to the PCH. Now we have access to all of QuartzCore all throughout the project. We'll need this later to use `CALayer`. This is such a common step for me in creating Xcode projects; it's a wonder that Apple doesn't include it by default.

The view controller is going to be very similar to the Dimensions, except this time we'll have two layouts. We're going to use a segmented control in the navigation bar, like last time, to switch between these two layouts (see Listing 4.26).

Listing 4.26 Creating Two Layouts

```
@implementation AFViewController
{
    //Array of selection objects
    NSArray *photoModelArray;

    UISegmentedControl *layoutChangeSegmentedControl;

    AFCoverFlowFlowLayout *coverFlowCollectionViewLayout;
    UICollectionViewFlowLayout *boringCollectionViewLayout;
}

//Static identifiers for cells and supplementary views
static NSString *CellIdentifier = @"CellIdentifier";

-(void)loadView
{
    //Create our view

    // Create our awesome cover flow layout
    coverFlowCollectionViewLayout = [[AFCoverFlowFlowLayout alloc] init];

    boringCollectionViewLayout = [[UICollectionViewFlowLayout alloc] init];
    boringCollectionViewLayout.itemSize = CGSizeMake(140, 140);
    boringCollectionViewLayout.minimumLineSpacing = 10.0f;
    boringCollectionViewLayout.minimumInteritemSpacing = 10.0f;

    // Create a new collection view with our flow layout and
    // set ourself as delegate and data source
    UICollectionView *photoCollectionView = [[UICollectionView alloc]
        initWithFrame:CGRectZero
        collectionViewLayout:boringCollectionViewLayout];
    photoCollectionView.dataSource = self;
    photoCollectionView.delegate = self;

    // Register our classes so we can use our custom
```

```
    // subclassed cell and header
    [photoCollectionView registerClass:[AFCollectionViewCell class]
forCellWithReuseIdentifier:CellIdentifier];

    // Set up the collection view geometry to cover the whole
    // screen in any orientation and other view properties
    photoCollectionView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
    photoCollectionView.allowsSelection = NO;
    photoCollectionView.indicatorStyle = UIScrollViewIndicatorStyleWhite;

    // Finally, set our collectionView (since we are a collection
    // view controller, this also sets self.view)
    self.collectionView = photoCollectionView;

    // Set up our model
    [self setupModel];
}

-(void)viewDidLoad
{
    [super viewDidLoad];

    // Crate a segmented control to sit in our navigation bar
    layoutChangeSegmentedControl = [[UISegmentedControl alloc]
initWithItems:@[@"Boring", @"Cover Flow"]];
    layoutChangeSegmentedControl.selectedSegmentIndex = 0;
    layoutChangeSegmentedControl.segmentedControlStyle =
        UISegmentedControlStyleBar;
    [layoutChangeSegmentedControl
        addTarget:self
           action:@selector(layoutChangeSegmentedControlDidChangeValue:)
 forControlEvents:UIControlEventValueChanged];

    self.navigationItem.titleView = layoutChangeSegmentedControl;
}
```

The data source methods for configuring the collection view are identical to those used in the preceding section, so I do not include them here. However, we are going to implement a new `UICollectionViewDelegateFlowLayout` method that will be responsible for returning the edge insets for our layouts (see Listing 4.27). We use this approach because the Cover Flow layout requires different section edge insets, depending on the orientation of the interface and the specific device its running on. I like to keep this kind of logic out of the `UICollectionViewLayout` subclass, if possible.

Listing 4.27 Custom Section Insets

```
-(UIEdgeInsets)collectionView:(UICollectionView *)collectionView
    layout:(UICollectionViewLayout *)collectionViewLayout
    insetForSectionAtIndex:(NSInteger)section
{
    if (collectionViewLayout == boringCollectionViewLayout)
    {
        // A basic flow layout that will accommodate three
        // columns in portrait
        return UIEdgeInsetsMake(10, 20, 10, 20);
    }
    else
    {
        if (UIInterfaceOrientationIsPortrait(self.interfaceOrientation))
        {
            // Portrait is the same in either orientation
            return UIEdgeInsetsMake(0, 70, 0, 70);
        }
        else
        {
            // We need to get the height of the main screen to see
            // if we're running on a 4" screen. If so, we need
            // extra side padding.
            if (CGRectGetHeight([[UIScreen mainScreen] bounds]) > 480)
            {
                return UIEdgeInsetsMake(0, 190, 0, 190);
            }
            else
            {
                return UIEdgeInsetsMake(0, 150, 0, 150);
            }
        }
    }
}
```

These values were determined mainly by experimentation to see what looked right. I would encourage you to take this approach, instead of divining them mathematically, for the simple reason that it doesn't matter if something is mathematically correct if it doesn't look correct to your users.

Finally, we need to implement our user interaction code. Shown in Listing 4.28, you'll notice it is similar to the last example.

Listing 4.28 Changing Layouts

```
-(void)layoutChangeSegmentedControlDidChangeValue:(id)sender
```

```
{
    // Change to the alternate layout

    if (layoutChangeSegmentedControl.selectedSegmentIndex == 0)
    {
        [self.collectionView
          setCollectionViewLayout:boringCollectionViewLayout
                         animated:NO];
    }
    else
    {
        [self.collectionView
            setCollectionViewLayout:coverFlowCollectionViewLayout
                           animated:NO];
    }

    // Invalidate the new layout
    [self.collectionView.collectionViewLayout invalidateLayout];
}
```

We explicitly do not animate the change in layout because they are so different that the animation between them looks jarring to the user. As you'll see in the next chapter, changing between layouts with animation is actually pretty easy to do.

After changing the layout, we need to invalidate the new layout. Although this is not included in the documentation, I've noticed some strange behavior on some layouts if you omit it. Experiment to see what works for your custom layouts.

We're going to create a new custom `UICollectionViewLayoutAttributes` subclass to hold two values: one to indicate whether we should rasterize the layer, and the other to indicate how "masked out" the cell should appear. We can't use `alpha` because cells behind the semitransparent ones would "bleed through." The new subclass is shown in Listing 4.29. For our cover view layout, cells will always be rasterized because otherwise they get some jagged edges due to their 3D transform.

As for the masking layer, we want items that are not at the center of the collection view to not be as prominent, so we'll place a semitransparent mask view over top of each cell.

Listing 4.29 Custom Layout Attributes Class for Cover Flow

```
// .h file

@interface AFCollectionViewLayoutAttributes : UICollectionViewLayoutAttributes

@property (nonatomic, assign) BOOL shouldRasterize;
@property (nonatomic, assign) CGFloat maskingValue;
```

```
@end

// .m file

@implementation AFCollectionViewLayoutAttributes

-(id)copyWithZone:(NSZone *)zone
{
    AFCollectionViewLayoutAttributes *attributes = [super copyWithZone:zone];

    attributes.shouldRasterize = self.shouldRasterize;
    attributes.maskingValue = self.maskingValue;

    return attributes;
}

-(BOOL)isEqual:(AFCollectionViewLayoutAttributes *)other {
    return [super isEqual:other] && (self.shouldRasterize ==
other.shouldRasterize
        && self.maskingValue == other.maskingValue);
}

@end
```

Next, let's look at the custom `UICollectionViewFlowLayout` subclass itself (see Listing 4.30). I omitted the `#define`s at the top of the file that are used later. I'll include them there.

Listing 4.30 Custom Layout Attributes Class for Cover Flow

```
@implementation AFCoverFlowFlowLayout

#pragma mark - Overridden Methods

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

    // Set up our basic properties
    self.scrollDirection = UICollectionViewScrollDirectionHorizontal;
    self.itemSize = CGSizeMake(180, 180);

    // Gets items up close to one another
    self.minimumLineSpacing = -60;

    // Makes sure we only have 1 row of items in portrait mode
```

```
    self.minimumInteritemSpacing = 200;

    return self;
}

+(Class)layoutAttributesClass
{
    return [AFCollectionViewLayoutAttributes class];
}

-(BOOL)shouldInvalidateLayoutForBoundsChange:(CGRect)oldBounds
{
    // Very important — needed to re-layout the cells when scrolling.
    return YES;
}

-(NSArray*)layoutAttributesForElementsInRect:(CGRect)rect
{
    NSArray* layoutAttributesArray = [super
        layoutAttributesForElementsInRect:rect];

    // We're going to calculate the rect of the collection view visisble to the
user.
    CGRect visibleRect = CGRectMake(
        self.collectionView.contentOffset.x,
        self.collectionView.contentOffset.y,
        CGRectGetWidth(self.collectionView.bounds),
        CGRectGetHeight(self.collectionView.bounds));

    for (UICollectionViewLayoutAttributes* attributes in layoutAttributesArray)
    {
        // We're going to calculate the rect of the collection
        // view visible to the user.
        // That way, we can avoid laying out cells that are not visible.
        if (CGRectIntersectsRect(attributes.frame, rect))
        {
            [self applyLayoutAttributes:attributes forVisibleRect:visibleRect];
        }
    }

    return layoutAttributesArray;
}

- (UICollectionViewLayoutAttributes
*)layoutAttributesForItemAtIndexPath:(NSIndexPath *)indexPath
{
```

```
    UICollectionViewLayoutAttributes *attributes = [super
layoutAttributesForItemAtIndexPath:indexPath];

    // We're going to calculate the rect of the collection view visible
    // to the user.
    CGRect visibleRect = CGRectMake(
        self.collectionView.contentOffset.x,
        self.collectionView.contentOffset.y,
        CGRectGetWidth(self.collectionView.bounds),
        CGRectGetHeight(self.collectionView.bounds));

    [self applyLayoutAttributes:attributes forVisibleRect:visibleRect];

    return attributes;
}
```

Most of this is standard-looking flow layout code. However, notice that we are calculating the visible rectangle in the collection view. This rectangle is going to be used to determine how much 3D transform and translation to apply to each cell. We'll calculate it easily by getting the content offset and bounds size of the collection view.

We also return `YES` in `shouldInvalidateLayoutForBoundsChange` so that when the user scrolls, the transforms of the cells are recalculated (at every frame refresh).

The `minimumLineSpacing` is negative because we want our cells to be "bunched up" close together, and in horizontally scrolling collection views, the line spacing is the distance between each vertical column of cells. As you can see in Figure 4.7, the line space is calculated as the space between the lines and the inter-item spacing is the space in between the cells *along the line.*

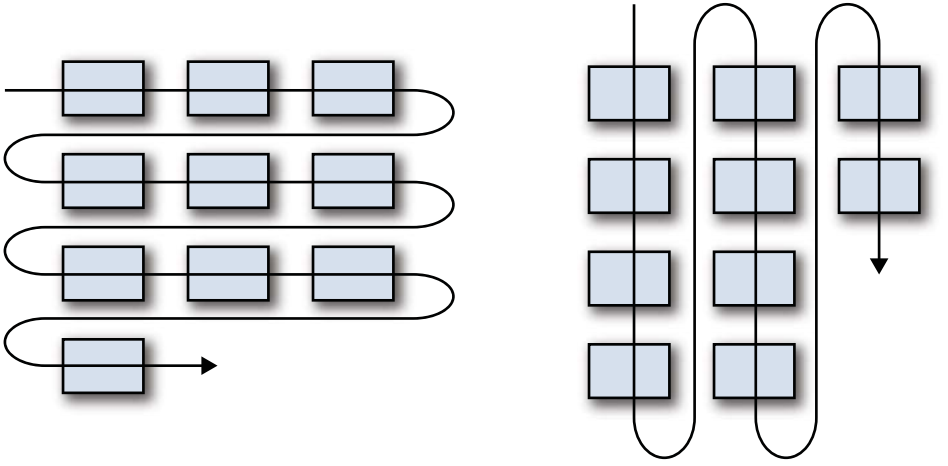


Figure 4.7 Difference between line space and inter-item spacing depending on scroll direction

It can be tricky to wrap your head around, so remember that in vertically scrolling collection views, line spacing and inter-item spacing are analogous to line height and kerning in writing, respectively. In horizontally scrolling collection views, they are flipped.

Next up is the intensive math used to apply the perspective 3D transform to our cells (see Listing 4.31). (Again, I need to thank Mark Pospesel for his help.)

Listing 4.31 Cover Flow Layout Math

```
#define ACTIVE_DISTANCE          100
#define TRANSLATE_DISTANCE       100
#define ZOOM_FACTOR              0.2f
#define FLOW_OFFSET              40
#define INACTIVE_GREY_VALUE      0.6f

-(void)applyLayoutAttributes:(UICollectionViewLayoutAttributes *)attributes
forVisibleRect:(CGRect)visibleRect
{
    // Applies the cover flow effect to the given layout attributes.

    // We want to skip supplementary views.
    if (attributes.representedElementKind) return;

    // Calculate the distance from the center of the visible rect to the
    // center of the attributes. Then normalize it so we can compare them
    // all. This way, all items further away than the active get the same
```

```
// transform.
CGFloat distanceFromVisibleRectToItem =
    CGRectGetMidX(visibleRect) - attributes.center.x;

CGFloat normalizedDistance =
    distanceFromVisibleRectToItem / ACTIVE_DISTANCE;

// Handy for use in making a number negative selectively
BOOL isLeft = distanceFromVisibleRectToItem > 0;

// Default values
CATransform3D transform = CATransform3DIdentity;
CGFloat maskAlpha = 0.0f;

if (fabsf(distanceFromVisibleRectToItem) < ACTIVE_DISTANCE)
{
    // We're close enough to apply the transform in relation to
    // how far away from the center we are.

    transform = CATransform3DTranslate(
        CATransform3DIdentity,
        (isLeft? - FLOW_OFFSET : FLOW_OFFSET)*
            ABS(distanceFromVisibleRectToItem/TRANSLATE_DISTANCE),
        0,
        (1 - fabsf(normalizedDistance)) * 40000 + (isLeft? 200 : 0));

    // Set the perspective of the transform.
    transform.m34 = -1/(4.6777f * self.itemSize.width);

    // Set the zoom factor.
    CGFloat zoom = 1 + ZOOM_FACTOR*(1 - ABS(normalizedDistance));
    transform = CATransform3DRotate(transform,
        (isLeft? 1 : -1) * fabsf(normalizedDistance) *
        45 * M_PI / 180,
        0,
        1,
        0);
    transform = CATransform3DScale(transform, zoom, zoom, 1);
    attributes.zIndex = 1;

    CGFloat ratioToCenter = (ACTIVE_DISTANCE -
        fabsf(distanceFromVisibleRectToItem)) / ACTIVE_DISTANCE;
    // Interpolate between 0.0f and INACTIVE_GREY_VALUE
    maskAlpha = INACTIVE_GREY_VALUE + ratioToCenter *
        (-INACTIVE_GREY_VALUE);
}
```

```
    else
    {
        // We're too far away - just apply a standard
        // perspective transform.

        transform.m34 = -1/(4.6777 * self.itemSize.width);
        transform = CATransform3DTranslate(transform,
            isLeft? -FLOW_OFFSET : FLOW_OFFSET, 0, 0);
        transform = CATransform3DRotate(transform, (
            isLeft? 1 : -1) * 45 * M_PI / 180, 0, 1, 0);
        attributes.zIndex = 0;

        maskAlpha = INACTIVE_GREY_VALUE;
    }

    attributes.transform3D = transform;

    // Rasterize the cells for smoother edges.
    [(AFCollectionViewLayoutAttributes *)attributes
        setShouldRasterize:YES];
    [(AFCollectionViewLayoutAttributes *)attributes
        setMaskingValue:maskAlpha];
}
```

Phew! Don't worry if it seems like a lot. I'll go through the high-level details, and you can experiment around with the specifics later; this isn't a book about `CATransform3D`, after all. The important thing to know is that you can apply a transform in *three dimensions* with collection views. Cool!

The first `if` branch executes if the attribute's item is close enough to the center of the visible area. It will give it a zoom, translation, and a 3D perspective transform depending on how close it is to the center. If an item is exactly at the center, the transform does nothing.

The `else` branch executes if the item is far enough away from the center to make sure items don't become *too* transformed. Imagine Cover Flow where the items extending to the edges kept having more and more transform applied; they would eventually become so transformed that they would flip around to their other sides!

We also want to set up a default mask value of zero and always set the rasterization to `YES`. Let's run the app now to see what's happening. Notice that you can switch between the plain flow layout and the Cover Flow layout really easily (see Figure 4.8).

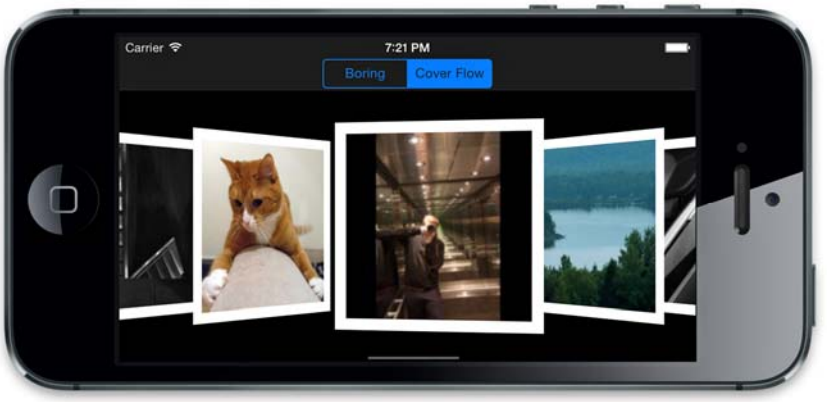


Figure 4.8 In-progress Cover Flow implementation

It looks great. However, there are a few things wrong with this. First, notice that the collection view is stopped halfway between cells; in the real Cover Flow, the scroll view comes to rest with an item perfectly centered. Next, you can clearly see that our layout attributes for masking and rasterization are not being applied. Hmm. Oh, that's because we haven't written the code to do that, yet. Let's deal with the first problem, first.

`targetContentOffsetForProposedContentOffset:withScrollingVelocity:` is a method defined in `UICollectionViewLayout` and is available to be overridden by subclasses, including ours. It provides an opportunity for subclasses to define where the collection view will "snap" to. We're going to implement it and use our existing code in `layoutAttributesForElementsInRect:` to get the attributes for the elements in the proposed rect (see Listing 4.32). Then, we'll find the attribute whose item will be closest to the center of the proposed visible rect. Then, we'll find out how far away that item will be and return an adjusted content offset that centers that view.

Listing 4.32 Stopping on Cells

```
-(CGPoint)
targetContentOffsetForProposedContentOffset:(CGPoint)proposedContentOffset
                       withScrollingVelocity:(CGPoint)velocity
{
    // Returns a point where we want the collection view to stop
    // scrolling at.  First, calculate the proposed center of the
    // collection view once the collection view has stopped
    CGFloat offsetAdjustment = MAXFLOAT;
    CGFloat horizontalCenter = proposedContentOffset.x +
        (CGRectGetWidth(self.collectionView.bounds) / 2.0);

    // Use the center to find the proposed visible rect.
    CGRect proposedRect = CGRectMake(
```

```
        proposedContentOffset.x,
        0.0,
        self.collectionView.bounds.size.width,
        self.collectionView.bounds.size.height);

    // Get the attributes for the cells in that rect.
    NSArray* array = [self
        layoutAttributesForElementsInRect:proposedRect];

    // This loop will find the closest cell to proposed center
    // of the collection view.
    for (UICollectionViewLayoutAttributes* layoutAttributes in array)
    {
        // We want to skip supplementary views
        if (layoutAttributes.representedElementCategory !=
            UICollectionElementCategoryCell)
            continue;

        // Determine if this layout attribute's cell is closer than
        // the closest we have so far
        CGFloat itemHorizontalCenter = layoutAttributes.center.x;
        if (fabsf(itemHorizontalCenter - horizontalCenter) <
            fabsf(offsetAdjustment))
        {
            offsetAdjustment = itemHorizontalCenter - horizontalCenter;
        }
    }

    return CGPointMake(proposedContentOffset.x + offsetAdjustment,
        proposedContentOffset.y);
}
```

Now, our app will snap to the nearest item. Let's implement our `UICollectionViewCell` subclass next. Listing 4.33 has the complete implementation, but the important method is `applyLayoutAttributes:`.

Listing 4.33 Cover Flow Cell Implementation

```
@implementation AFCollectionViewCell
{
    UIImageView *imageView;
    UIView *maskView;
}

- (id)initWithFrame:(CGRect)frame
```

```
{
    if (!(self = [super initWithFrame:frame])) return nil;

    // Set up our image view
    imageView = [[UIImageView alloc] initWithFrame:
        CGRectInset(CGRectMake(0,
            0,
            CGRectGetWidth(frame),
            CGRectGetHeight(frame)),
        10, 10)];
    imageView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
    imageView.clipsToBounds = YES;
    [self.contentView addSubview:imageView];

    maskView = [[UIView alloc] initWithFrame:CGRectMake(
        0,
        0,
        CGRectGetWidth(frame),
        CGRectGetHeight(frame))];
    maskView.backgroundColor = [UIColor blackColor];
    maskView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
    maskView.alpha = 0.0f;
    [self.contentView insertSubview:maskView aboveSubview:imageView];

    // This will make the rest of our cell, outside the image view, appear
transparent against a black background.
    self.backgroundColor = [UIColor whiteColor];

    return self;
}

#pragma mark - Overridden Methods

-(void)prepareForReuse
{
    [super prepareForReuse];

    [self setImage:nil];
}

-(void)applyLayoutAttributes:(UICollectionViewLayoutAttributes *)layoutAttributes
{
    [super applyLayoutAttributes:layoutAttributes];
    maskView.alpha = 0.0f;
    self.layer.shouldRasterize = NO;
```

```
    // Important! Check to make sure we're actually this special subclass.
    // Failing to do so could cause the app to crash!
    if (![layoutAttributes isKindOfClass:[AFCollectionViewLayoutAttributes
class]])
    {
        return;
    }

    AFCollectionViewLayoutAttributes *castedLayoutAttributes =
(AFCollectionViewLayoutAttributes *)layoutAttributes;

    self.layer.shouldRasterize = castedLayoutAttributes.shouldRasterize;
    maskView.alpha = castedLayoutAttributes.maskingValue;
}

#pragma mark - Public Methods

-(void)setImage:(UIImage *)image
{
    [imageView setImage:image];
}

@end
```

Now we can run the application and see the effect of "fading out" the other cells and the snap-to effect (see Figure 4.9).

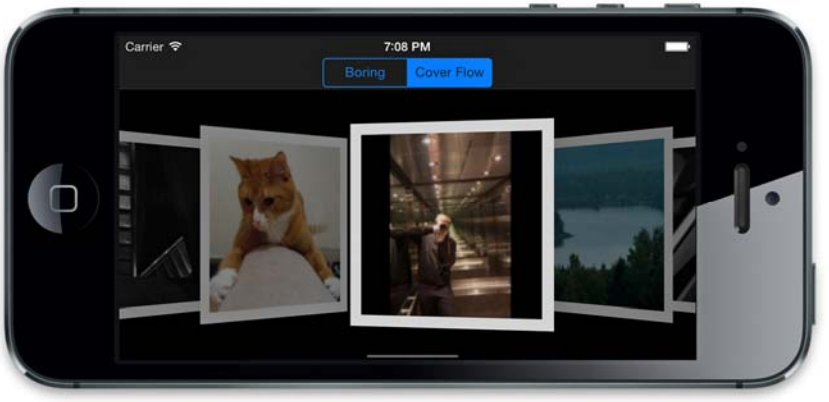


Figure 4.9 Final Cover Flow implementation

Nice! Play around with it. Experiment with rotation and changing layouts while the collection view is decelerating. Find its capabilities and limitations.

Now that implementation is complete, I want to talk about a few things that I found problematic with collection views.

First, rotation animation on the Cover Flow view isn't perfect. I can't seem to get it seamless; I think it might have something to do with changing `contentSize` during rotation.

I originally tried changing the layout to Cover Flow *during rotation* so that the normal flow layout would be used in portrait and the Cover Flow layout would be used in landscape. Changing layouts during rotation was very problematic because the `contentSize` is not reliable in the layout subclass during rotation and even more unreliable when changing layouts.

I researched these problems and found the precise order of events when a layout is used:

1. `prepareLayout` is called on the layout so it has an opportunity to perform any upfront computations.
2. `collectionViewContentSize` is called on the layout to determine the collection view's content size.
3. `layoutAttributesForElementsInRect:` is called.

Then, the layout becomes live and continues to call `layoutAttributesForElementsInRect:` and `layoutAttributesForItemAtIndexPath:` until the layout becomes invalidated. Then, the process is repeated again.

Using the content size in a layout is probably not a good idea; `UICollectionView` is still very new, and the community is still determining the best practices for working with it.

Depending on your idea for a layout, it might be best to turn to `UICollectionViewLayout`, as we do the next chapter. However, always consider whether `UICollectionViewFlowLayout` can accomplish your goals first. It does a lot of heavy lifting for you.

We've now covered decoration views, collection view layouts, layout attributes, and custom animations. You've solidified your knowledge from the first three chapters and dipped your toes into the water for the upcoming chapter. We're on the cusp of doing some really interesting stuff, but first, let's take a look back at `UITableView`.

`UITableView`: `UICollectionView`'s Daddy

`UICollectionView` was only introduced in iOS 6, but `UITableView` has been around since the original iPhone SDK was released in 2008. Many of the same principles used with `UITableView` apply to `UICollectionView`, but some have been modified.

`UITableView` has only recently started to use the class registration method to create its cells. This is the *only* way to do so with collection views.

Table view "batch updates" are done by calling a method to start the updates, performing them, and then calling another method to indicate that the updates are over. Collection views, however, *only* offer the (better, in my opinion) block-based `performBatchUpdates:` method.

Those are some minute differences in the way developers accomplish their goals with the classes. A much bigger philosophical difference between the two classes is that table view

cells handle *a lot* of their internal layout. This starkly contrasts to collection view cells, which handle *none at all*. This forces developers to implement their own `UITableViewCell` subclasses from the ground up, every time. Meanwhile, `UITableViewCell` has four different "styles" that define how its two text labels, image view, "accessory" view, and editing style are laid out. Quite the difference!

I believe that if Apple were to introduce `UITableView` today, knowing what they've learned about framework design in the past 6 years, `UITableViewCell` would not have styles at all. Instead, they would have a few direct subclasses that developers could use, or they could implement their own subclasses.

Even though `UITableView` appears bloated by the standards of a modern Objective-C framework, `UICollectionViews` owes a lot of its sleekness to the lessons Apple has learned since originally crafting `UITableView`.

5

Crafting Custom Layouts Using `UICollectionViewLayout`

In the preceding chapter, I wrote that `UICollectionViewFlowLayout` is great for line-based, breaking layouts and that you should always resort to using it first. Sometimes, however, our layouts are sufficiently complicated to warrant the use of something more powerful. `UICollectionViewLayout` is the superclass of `UICollectionViewFlowLayout` and it is hands on. You are responsible for everything—the layout of cells, the size of the collection view—everything. We'll take a look at an example where you'd want to use it, revisit decoration views, and explore a little bit of changing between layouts programmatically with animation. At the end of this chapter, we build a really cool photos application using a web service and a few cool custom layouts.

Subclassing `UICollectionViewLayout`

I don't want to scare you away from subclassing `UICollectionViewLayout`, but let me reiterate that this is a last resort to be used only when the option of subclassing `UICollectionViewFlowLayout` instead has been explored. Treat that as your warning, lest you find yourself writing a lot of code you don't have to.

If your layout is not based on a line that breaks when it hits the edge of the screen, subclassing `UICollectionViewLayout` directly is probably for you. If you find yourself writing code to reproduce the logic in `UICollectionViewFlowLayout`, reconsider subclassing it directly.

`UICollectionView` does no heavy lifting for you; you have to do everything yourself. Let's look at a relatively simple example to see what I mean.

When Apple introduced `UICollectionView` at WWDC 2012, they had a few sessions that talked about the class and its layouts. Unfortunately, the sample code they provided was sparse and riddled with inaccuracies or simplifications. We're going to take a look at one of the layouts they produced—the circle layout—with our own twist.

Each one of our cells is going to be arranged in a circle around some point on the screen. (We're going to "future proof" this for adding interactivity in the next chapter; I only discuss the layout aspects for now.) Each cell is the same distance from that point. We'll also adjust the `transform3D` of each cell so that it "points" to the center of the circle. Finally, we'll revisit decoration views; it's been a while since we dealt with them, and it'll be fun to reapply some of our new techniques to them.

To make it fun, we'll add two buttons in a navigation bar: one for adding new cells and one for deleting them (with animations, of course). We'll also have a basic `UICollectionViewFlow` layout to show you how to animate in between layouts. Although this is supposed to be really easy, it can often require some ingenuity to get working correctly.

Start by creating an empty app. In the application delegate, create a `UINavigationController` property to be our window's root view controller. Instantiate it with an instance of our own view controller. You should all be familiar with this process by now. Just don't forget to add QuartzCore to the libraries you link against. I place `#import <QuartzCore/QuartzCore.h>` in my precompiled header so that I don't have to import it in every file. See Listing 5.1 for the basic app setup.

Listing 5.1 Basic App Setup

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    self.viewController = [[UINavigationController alloc]
        initWithRootViewController:[[AFViewController alloc] init]];
    self.viewController.navigationBar.barStyle = UIBarStyleBlack;
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

Our model is going to be simple; it's just an integer of the number of cells to display that we'll increment and decrement as we add and remove cells. Really easy. We'll create a property for this number and ones for our two layouts and our segmented control (see Listing 5.2).

Listing 5.2 Private Properties

```
@interface AFViewController ()

@property (nonatomic, assign) NSInteger cellCount;
```

```
@property (nonatomic, strong) AFCollectionViewCircleLayout *circleLayout;
@property (nonatomic, strong) AFCollectionViewFlowLayout *flowLayout;

@property (nonatomic, strong)
    UISegmentedControl *layoutChangeSegmentedControl;

@end
```

Our `loadView` and `viewDidLoad` methods are also straightforward; they instantiate our properties and set up our navigation item, as shown in Listing 5.3.

Listing 5.3 Setting Up the View Controller

```
static NSString *CellIdentifier = @"CellIdentifier";

-(void)loadView
{
    // Create our view

    // Create instances of our layouts
    self.circleLayout = [[AFCollectionViewCircleLayout alloc] init];
    self.flowLayout = [[AFCollectionViewFlowLayout alloc] init];

    // Create a new collection view with our flow layout and set
    // ourself as delegate and data source.
    UICollectionView *collectionView = [[UICollectionView alloc]
        initWithFrame:CGRectZero
        collectionViewLayout:self.circleLayout];
    collectionView.dataSource = self;
    collectionView.delegate = self;

    // Register our classes so we can use our custom subclassed
    // cell and header
    [collectionView registerClass:[AFCollectionViewCell class]
        forCellWithReuseIdentifier:CellIdentifier];

    // Set up the collection view geometry to cover the whole screen
    // in any orientation and other view properties.
    collectionView.autoresizingMask = UIViewAutoresizingFlexibleWidth |
        UIViewAutoresizingFlexibleHeight;

    // Finally, set our collectionView (since we are a collection
    // view controller, this also sets self.view)
    self.collectionView = collectionView;
```

```
    // Setup our model
    self.cellCount = 12;
}

- (void)viewDidLoad
{
    [super viewDidLoad];

    self.navigationItem.leftBarButtonItem = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
        target:self
        action:@selector(addItem)];
    self.navigationItem.rightBarButtonItem = [[UIBarButtonItem alloc]
        initWithBarButtonSystemItem:UIBarButtonSystemItemTrash
        target:self
        action:@selector(deleteItem)];

    self.layoutChangeSegmentedControl = [[UISegmentedControl alloc]
        initWithItems:@[@"Circle", @"Flow"]];
    self.layoutChangeSegmentedControl.selectedSegmentIndex = 0;
    self.layoutChangeSegmentedControl.segmentedControlStyle =
        UISegmentedControlStyleBar;
    [self.layoutChangeSegmentedControl addTarget:self
        action:@selector(layoutChangeSegmentedControlDidChangeValue:)
        forControlEvents:UIControlEventValueChanged];
    self.navigationItem.titleView = self.layoutChangeSegmentedControl;
}

-(void)layoutChangeSegmentedControlDidChangeValue:(id)sender
{
    // This just swaps the two values
    if (self.collectionView.collectionViewLayout == self.circleLayout)
    {
        [self.flowLayout invalidateLayout];
        self.collectionView.collectionViewLayout = self.flowLayout;
    }
    else
    {
        [self.circleLayout invalidateLayout];
        self.collectionView.collectionViewLayout = self.circleLayout;
    }
}
```

The `layoutChangeSegmentedControlDidChangeValue:` implementation is very basic. We'll add more to it later to spice things up with animations a bit. Notice that it

invalidates layouts before giving them to the collection view to use. This is *really* important. If we don't do this, the collection view might be in landscape orientation but be laid out with portrait calculations. I know this sounds like the kind of thing that should have been taken care of for you, but you have to do it yourself. We also have to explicitly enable rotations for iOS 6.

Listing 5.4 Enabling Rotations

```
-(BOOL)shouldAutorotate
{
    return YES;
}

-(NSUInteger)supportedInterfaceOrientations
{
    return UIInterfaceOrientationMaskAll;
}
```

The flow layout implementation is straightforward, as Listing 5.5 shows.

Listing 5.5 Simple Flow Layout

```
@implementation AFCollectionViewFlowLayout

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

    self.itemSize = CGSizeMake(200, 200);
    self.sectionInset = UIEdgeInsetsMake(13.0f, 13.0f, 13.0f, 13.0f);
    self.minimumInteritemSpacing = 13.0f;
    self.minimumLineSpacing = 13.0f;

    self.insertedRowSet = [NSMutableSet set];
    self.deletedRowSet = [NSMutableSet set];

    return self;
}

@end
```

Now that we have our basic flow layout, let's get the guts of this example: the circle layout (see Listing 5.6). Create a new class that extends `UICollectionViewLayout`. We're going to override `collectionViewContentSize` to return simply the size of

the collection view itself, preventing it from ever scrolling. We'll also override `prepareLayout` to set up the center of our circle and its radius; we will grab the number of cells in the collection view here.

This could represent a conflict in the separation of concerns in our app's architecture. After all, aren't layouts supposed to be unaware of the data that they're helping display? That is true. However, in *this* case, the number of cells being displayed affects the layout, so it's appropriate to access this information.

Listing 5.6 Circle Layout

```
-(void)prepareLayout
{
    [super prepareLayout];

    CGSize size = self.collectionView.bounds.size;

    self.cellCount = [[self collectionView] numberOfItemsInSection:0];
    self.center = CGPointMake(size.width / 2.0, size.height / 2.0);
    self.radius = MIN(size.width, size.height) / 2.5;
}

-(CGSize)collectionViewContentSize
{
    CGRect bounds = [[self collectionView] bounds];
    return bounds.size;
}

- (UICollectionViewLayoutAttributes
*)layoutAttributesForItemAtIndexPath:(NSIndexPath *)path
{
    UICollectionViewLayoutAttributes* attributes =
        [UICollectionViewLayoutAttributes
                layoutAttributesForCellWithIndexPath:path];

    attributes.size = CGSizeMake(kItemDimension, kItemDimension);
    attributes.center =
        CGPointMake(self.center.x + self.radius * cosf(2 * path.item * M_PI /
            self.cellCount - M_PI_2), self.center.y + self.radius *
               sinf(2 * path.item * M_PI / self.cellCount - M_PI_2));

    attributes.transform3D = CATransform3DMakeRotation(
        (2 * M_PI * path.item / self.cellCount), 0, 0, 1);

    return attributes;
}
```

```
-(NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    NSMutableArray* attributes = [NSMutableArray array];

    for (NSInteger i = 0 ; i < self.cellCount; i++)
    {
        NSIndexPath* indexPath = [NSIndexPath
            indexPathForItem:i inSection:0];
        [attributes addObject:[self
            layoutAttributesForItemAtIndexPath:indexPath]];
    }

    return attributes;
}
```

The `layoutAttributesForItemAtIndexPath:` might look a little confusing, but it's just a simple formula for the points along a circle. We also rotate each cell to make its bottom edge parallel to a tangent of the circle.

Finally, we need to create our cell subclass and our `UICollectionViewDataSource` methods (see Listing 5.7).

Listing 5.7 Simple Cell Subclass

```
@interface AFCollectionViewCell ()

@property (nonatomic, strong) UILabel *label;

@end

@implementation AFCollectionViewCell

- (id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    self.backgroundColor = [UIColor orangeColor];

    self.label = [[UILabel alloc] initWithFrame:
        CGRectMake(0, 0,
            CGRectGetWidth(frame),
            CGRectGetHeight(frame))];
    self.label.backgroundColor = [UIColor clearColor];
    self.label.textAlignment = NSTextAlignmentCenter;
    self.label.textColor = [UIColor whiteColor];
    self.label.font = [UIFont boldSystemFontOfSize:24];
```

```
    [self.contentView addSubview:self.label];

    return self;
}

-(void)prepareForReuse
{
    [super prepareForReuse];

    [self setLabelString:@""];
}

-(void)setLabelString:(NSString *)labelString
{
    self.label.text = labelString;
}


-(void)applyLayoutAttributes:(UICollectionViewLayoutAttributes *)layoutAttributes
{
    [super applyLayoutAttributes:layoutAttributes];

    self.label.center = CGPointMake(
        CGRectGetWidth(self.contentView.bounds) / 2.0f,
        CGRectGetHeight(self.contentView.bounds) / 2.0f);
}

@end
```

The cell simply displays some text; it will be used by both of our layouts to display the cell item number. The implementation of `applyLayoutAttributes:` sets the `center` point of the label so that it will be interpolated during the layout change animation, later. We can't use `frame` here because that will change the bounds of the label immediately instead of with animation. Listing 5.8 shows a basic collection view data source implementation.

Listing 5.8 Simple `UICollectionViewDataSource` Implementation

```
- (NSInteger)collectionView:(UICollectionView *)view
numberOfItemsInSection:(NSInteger)section;
{
    return self.cellCount;
}

(UICollectionViewCell *)collectionView:
```

```
    (UICollectionView *)collectionView
    cellForItemAtIndexPath:(NSIndexPath *)indexPath;
{
    AFCollectionViewCell *cell = (AFCollectionViewCell *)
        [collectionView dequeueReusableCellWithReuseIdentifier:CellIdentifier
        forIndexPath:indexPath];

    [cell setLabelString:[NSString
        stringWithFormat:@"%d", indexPath.row]];

    return cell;
}
```

Let's run the app and see what it looks like. Figure 5.1 shows our app running.

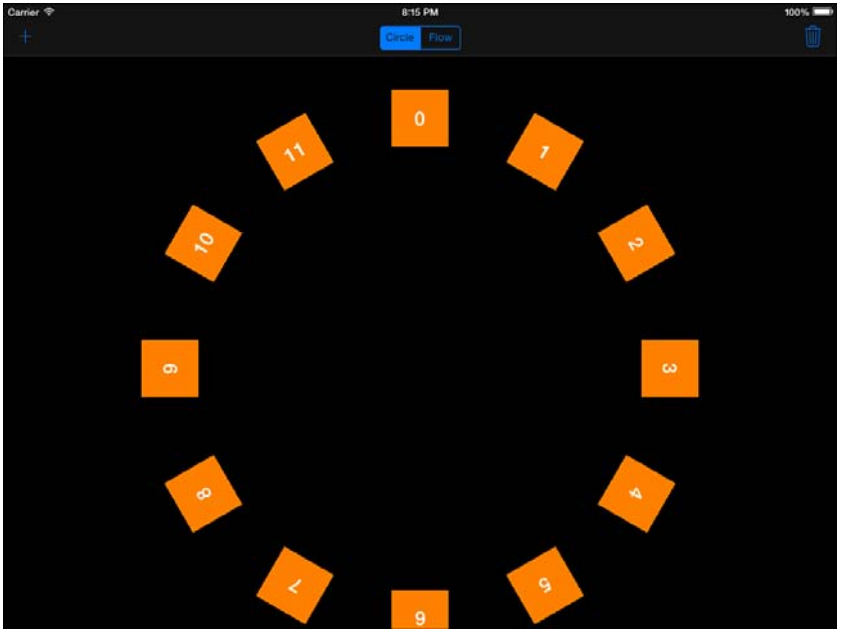


Figure 5.1 Basic circle layout

Not bad! Remember, this is a pretty simple layout. It doesn't do anything fancy at all; it doesn't even scroll. Let's take a look at the flow layout. Remember, we didn't do anything special in the flow layout ; it's all baked in (see Figure 5.2).

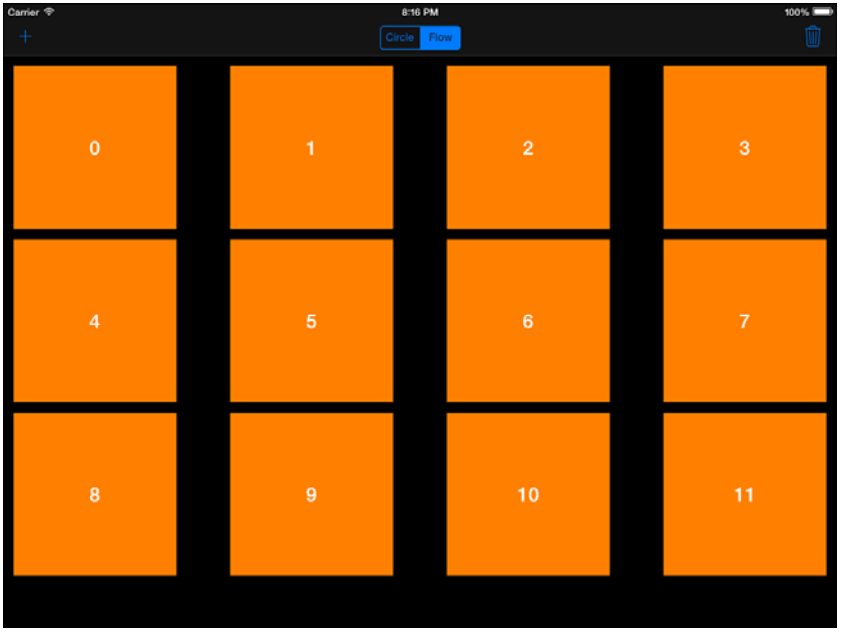


Figure 5.2 Basic flow layout

Let's make this a little more interesting. Rotate the device and notice that the animation for the circle layout is not great. You actually have to switch layouts to get the collection view to realize that its orientation has changed. If you recall from the Cover Flow layout we did in Chapter 4, "Organizing Content with `UICollectionViewFlowLayout`," we need to let the collection view layout know that it should invalidate itself when the `bounds` of the collection view change. Implement the following method in the circle layout; it will let the collection view know that the layout becomes invalid on any change to its `bounds` (such as the change on rotation) (see Listing 5.9).

Listing 5.9 Invalidating Layout on `bounds` Change

```
-(BOOL)shouldInvalidateLayoutForBoundsChange:(CGRect)newBounds
{
    return YES;
}
```

That's better. But I know that we can do even better.

Animating `UICollectionViewLayout` Changes

Let's change our implementation of `layoutChangeSegmentedControlDidChange-Value:` to explicitly animate the change in collection view layout (see Listing 5.10).

Listing 5.10 Changing Layouts with Animation

```
-(void)layoutChangeSegmentedControlDidChangeValue:(id)sender
{
    // We need to explicitly tell the collection view layout
     // that we want the change animated.
    if (self.collectionView.collectionViewLayout == self.circleLayout)
    {
        [self.flowLayout invalidateLayout];
        [self.collectionView
             setCollectionViewLayout:self.flowLayout animated:YES];
    }
    else
    {
        [self.circleLayout invalidateLayout];
        [self.collectionView
             setCollectionViewLayout:self.circleLayout
             animated:YES];
    }
}
```

You *must* use the `setCollectionViewLayout:animated:` method to get animations; setting the `collectionViewLayout` property in an animation block is not enough. Although this method will still animate the change, some cells will be duplicated during the animation. It's a shame that Apple can't decide if `collectionViewLayout` is an implicitly animatable property or not.

Now that we animate the change in layouts, `UICollectionView` layout will interpolate the changes in the layout attributes for each cell.

Figure 5.3 shows two intermediate stages of the collection view layout change animation. As a developer, you get this animation for free from `UICollectionView`. Not bad!

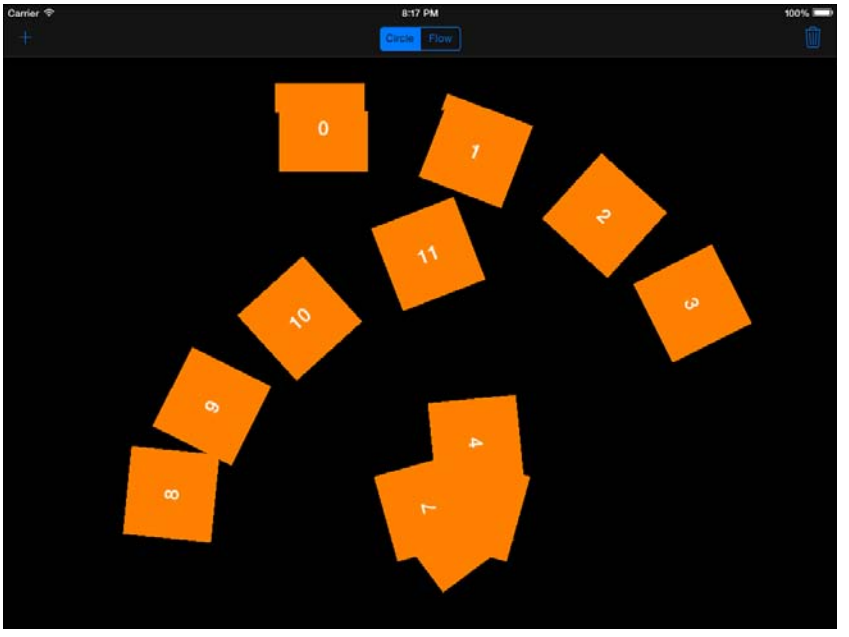


Figure 5.3 Layout change animation

Now that we have our layout change animations finished, let's add some fancy insertion and deletion animations to match. We'll do the flow layout first because it's easier.

Remember that we need to implement `prepareForCollectionViewUpdates:` and `finalizeCollectionViewUpdates` so that we only animate the inserted or deleted items. We'll create two mutable sets to hang onto the items that are being inserted or deleted.

We'll add a fade animation to the cells and make them spin. I want them to spin clockwise, so our initial rotation before insertion will be -90º and our final rotation after deletion will be 90º. These will have to be specified in radians (see Listing 5.11).

Listing 5.11 Animating Insertions and Deletions

```
@interface AFCollectionViewFlowLayout ()

@property (nonatomic, strong) NSMutableSet *insertedRowSet;
@property (nonatomic, strong) NSMutableSet *deletedRowSet;

@end

@implementation AFCollectionViewFlowLayout

-(id)init
```

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

    self.itemSize = CGSizeMake(200, 200);
    self.sectionInset =
        UIEdgeInsetsMake(13.0f, 13.0f, 13.0f, 13.0f);
    self.minimumInteritemSpacing = 13.0f;
    self.minimumLineSpacing = 13.0f;

    // Must instantiate these in init or else they'll
    // always be empty
    self.insertedRowSet = [NSMutableSet set];
    self.deletedRowSet = [NSMutableSet set];

    return self;
}

-(void)prepareForCollectionViewUpdates:(NSArray *)updateItems
{
    [super prepareForCollectionViewUpdates:updateItems];

    [updateItems
        enumerateObjectsUsingBlock:^(UICollectionViewUpdateItem *updateItem,
            NSUInteger idx, BOOL *stop) {
        if (updateItem.updateAction ==
            UICollectionUpdateActionInsert)
        {
            [self.insertedRowSet
               addObject:@(updateItem.indexPathAfterUpdate.item)];
        }
        else if (updateItem.updateAction ==
            UICollectionUpdateActionDelete)
        {
            [self.deletedRowSet
              addObject:@(updateItem.indexPathBeforeUpdate.item)];
        }
    }];
}

-(void)finalizeCollectionViewUpdates
{
    [super finalizeCollectionViewUpdates];

    [self.insertedRowSet removeAllObjects];
    [self.deletedRowSet removeAllObjects];
}
```

```
- (UICollectionViewLayoutAttributes *)
        initialLayoutAttributesForAppearingItemAtIndexPath:
        (NSIndexPath *)itemIndexPath
{
    if ([self.insertedRowSet containsObject:@(itemIndexPath.item)])
    {
        UICollectionViewLayoutAttributes *attributes = [self
layoutAttributesForItemAtIndexPath:itemIndexPath];
        attributes.alpha = 0.0;
        attributes.center = self.center;
        return attributes;
    }

    return nil;
}

- (UICollectionViewLayoutAttributes
*)finalLayoutAttributesForDisappearingItemAtIndexPath:(NSIndexPath
*)itemIndexPath
{
    if ([self.deletedRowSet containsObject:@(itemIndexPath.item)])
    {
        UICollectionViewLayoutAttributes *attributes =
            [self layoutAttributesForItemAtIndexPath:
                 itemIndexPath];
        attributes.alpha = 0.0;
        attributes.center = self.center;
        attributes.transform3D = CATransform3DConcat(CATransform3DMakeRotation((2
* M_PI * itemIndexPath.item / (self.cellCount + 1)), 0, 0, 1),
CATransform3DMakeScale(0.1, 0.1, 1.0));

        return attributes;
    }

    return nil;
}

@end
```

Let's add code to insert and delete items, as shown in Listing 5.12.

Listing 5.12 Inserting and Deleting Items

```
-(void)addItem
{
    [self.collectionView performBatchUpdates:^{
```

```
        self.cellCount = self.cellCount + 1;
        [self.collectionView
            insertItemsAtIndexPaths:@[[NSIndexPath
                indexPathForItem:self.cellCount-1
                inSection:0]]];
    } completion:nil];
}

-(void)deleteItem
{
    // Always have at least once cell in our collection view
    if (self.cellCount == 1) return;

    [self.collectionView performBatchUpdates:^{
        self.cellCount = self.cellCount - 1;
        [self.collectionView
            deleteItemsAtIndexPaths:@[[NSIndexPath
                indexPathForItem:self.cellCount
                inSection:0]]];
    } completion:nil];
}
```

That's the complete implementation for animations for insertion and deletion. Notice that we didn't have to do any custom animation work ourselves; we only had to override the existing methods and set existing properties.

The attributes have an alpha value of zero applied in both insertion and deletion animations. The `transform3D` property is used to rotate the cell by a quarter radians (90º) clockwise for each animation. In addition, we scale down the cell to 10% of its usual size. The order which we do these typically matters, but not in this case.

The order we concatenate transforms is usually important because `CATrasform3D` is not communicative. Concatenating transforms uses a post-order multiplication; so if you want a scale, followed by a translation, you need to concatenate the scale transform to the translation transform. Always apply the transforms in the opposite order you want them applied. See Figure 5.4 for our running app with insertion/deletion animations.

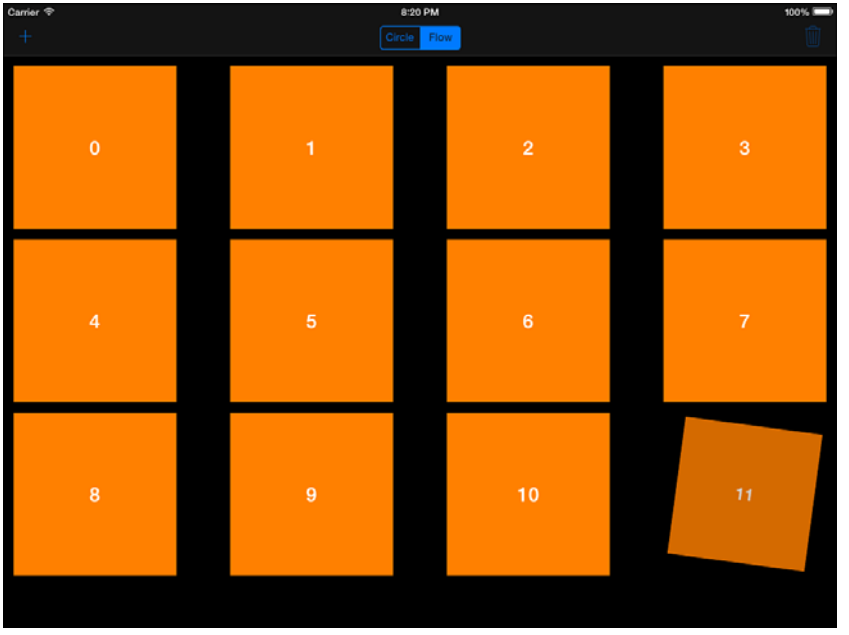


Figure 5.4 Flow layout deletion animation

In the circle layout class, add the same `insertedRowSet` and `deletedRowSet` private properties and instantiate them in `init`. Also write identical implementations for `prepareForCollectionViewUpdates:` and `finalizeCollectionViewUpdates`, which I won't include in Listing 5.13.

Listing 5.13 Animating Insertions and Deletions in the Circle Layout

```
(UICollectionViewLayoutAttributes *)
    initialLayoutAttributesForAppearingItemAtIndexPath:
    (NSIndexPath *)itemIndexPath
{
    UICollectionViewLayoutAttributes *attributes = [super
        initialLayoutAttributesForAppearingItemAtIndexPath:
        itemIndexPath];

    if ([self.insertedRowSet
            containsObject:@(itemIndexPath.item)])
    {
        attributes = [self
            layoutAttributesForItemAtIndexPath:itemIndexPath];
        attributes.alpha = 0.0;
        attributes.center = self.center;
```

```
        return attributes;
    }

    return attributes;
}

(UICollectionViewLayoutAttributes *)
    finalLayoutAttributesForDisappearingItemAtIndexPath:
    (NSIndexPath *)itemIndexPath
{
    // The documentation says that this returns nil. It is lying.
    UICollectionViewLayoutAttributes *attributes = [super
        finalLayoutAttributesForDisappearingItemAtIndexPath:
        itemIndexPath];

    if ([self.deletedRowSet containsObject:@(itemIndexPath.item)])
    {
        attributes = [self
            layoutAttributesForItemAtIndexPath:itemIndexPath];
        attributes.alpha = 0.0;
        attributes.center = self.center;
        attributes.transform3D =
            CATransform3DConcat(
                CATransform3DMakeScale(0.1, 0.1, 1.0),
                CATransform3DMakeRotation(
                    (2 * M_PI * itemIndexPath.item /
                        (self.cellCount + 1)),
                    0, 0, 1));

        return attributes;
    }

    return attributes;
}
```

You can see that we're applying nearly the same animations for insertion and deletion.

One difference between the insertion and deletion animations is the rotation. For insertions, we don't specify one beyond what we already calculate when calling `layoutAttributesForItemAtIndexPath:`. This won't work for deletion, and the reason is very subtle. The `cellCount` property is already updated by the time either `initialLayoutAttributesForAppearingItemAtIndexPath:` or `finalLayoutAttributesForDisappearingItemAtIndexPath:` are called. When inserting, this means that the rotation angle calculated in `layoutAttributesForItemAtIndexPath:`

reflects the correct angle for the new number of cells, which is what we want. However, when deleting, we don't want the cell to have the updated angle reflecting the new `cellCount`; we want it to have its old angle. This means that we need to recalculate the rotation angle.

We concatenate two 3D transforms: a scale down to 10% of the item's size, and a rotation calculated with the *old* `cellCount`: `cellCount + 1`. Again, the order of the transforms is not important in this case. See Figure 5.5 for our new animation.

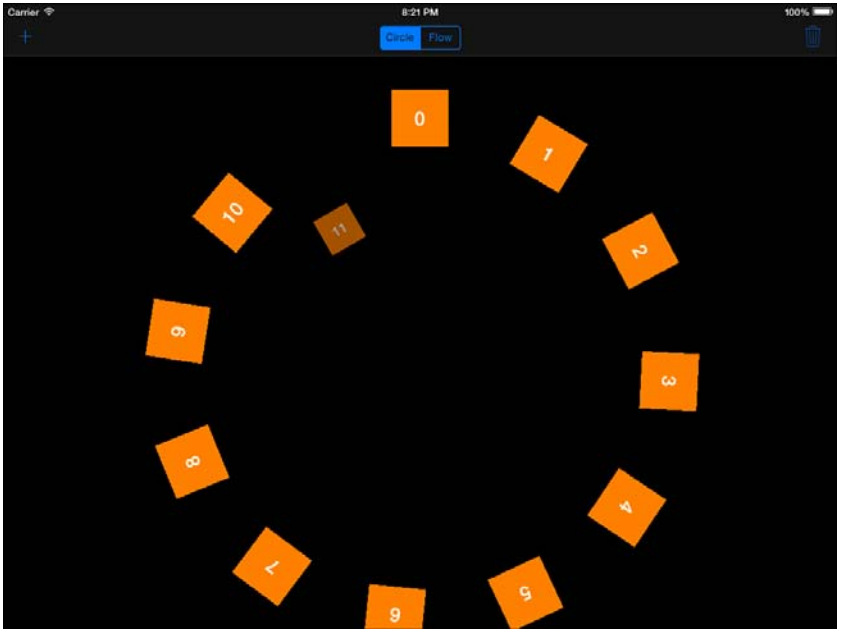


Figure 5.5 Circle layout deletion animation

What we've got so far is pretty good; we're animating all the things we can animate.

Let's add a decoration view to the center of the circle layout that will point to the same location that a minute hand would point to, given the current time. This will remind us how to implement decorations views and show you that you use the same method you do with `UICollectionViewFlowLayout`.

First, let's implement the decoration view class (see Listing 5.14). Recall that any decoration view must subclass `UICollectionReusableView`.

Listing 5.14 Implementing the Decoration View

```
@implementation AFDecorationView

- (id)initWithFrame:(CGRect)frame
```

```
{
    if (!(self = [super initWithFrame:frame])) return nil;

    self.backgroundColor = [UIColor whiteColor];

    CAGradientLayer *gradientLayer = [CAGradientLayer layer];
    gradientLayer.colors = @[(id)[[UIColor blackColor] CGColor],
        (id)[[UIColor clearColor] CGColor]];
    gradientLayer.backgroundColor = [[UIColor clearColor]
        CGColor];
    gradientLayer.frame = self.bounds;

    self.layer.mask = gradientLayer;

    return self;
}

@end
```

We create a gradient mask that spans the length of the decoration view so that we can tell which side is which. It also looks pretty cool, but you don't want to use `CALayer`'s `mask` property too heavily because it slows down view rendering.

Next, we need to register the decoration view class in our circle layout's `init` method (see Listing 5.15).

Listing 5.15 Registering the Decoration View Class

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

    self.insertedRowSet = [NSMutableSet set];
    self.deletedRowSet = [NSMutableSet set];

    [self registerClass:[AFDecorationView class]
        forDecorationViewOfKind:AFCollectionViewFlowDecoration];

    return self;
}
```

To display our decoration view, we need to add a decoration view `UICollection-ViewLayoutAttributes` object to our `layoutAttributesForElementsInRect:` implementation (see Listing 5.16).

Listing 5.16 Adding Decoration Views to the Collection View

```
-(NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    NSMutableArray* attributes = [NSMutableArray array];

    for (NSInteger i = 0 ; i < self.cellCount; i++)
    {
        NSIndexPath* indexPath = [NSIndexPath
            indexPathForItem:i inSection:0];
        [attributes addObject:[self
            layoutAttributesForItemAtIndexPath:indexPath]];
    }

    if (CGRectContainsPoint(rect, self.center))
    {
        [attributes addObject:[self
            layoutAttributesForDecorationViewOfKind:
                AFCollectionViewFlowDecoration
            atIndexPath:
                [NSIndexPath indexPathForItem:0
                    inSection:0]]];
    }

    return attributes;
}
```

The check to make sure the `rect` contains the `center` point is probably superfluous, but good practice nonetheless. Now that we have added the decoration view to the collection view, we need to give it the appropriate transform, as shown in Listing 5.17.

Listing 5.17 Decoration View Layout Attributes

```
-(UICollectionViewLayoutAttributes *)
    layoutAttributesForDecorationViewOfKind:
    (NSString *)decorationViewKind
    atIndexPath:(NSIndexPath *)indexPath
{
    UICollectionViewLayoutAttributes *layoutAttributes =
        [UICollectionViewLayoutAttributes
            layoutAttributesForDecorationViewOfKind:
            decorationViewKind withIndexPath:indexPath];

    if ([decorationViewKind
            isEqualToString:AFCollectionViewFlowDecoration])
    {
```

```
        CGFloat rotationAngle = 0.0f;

        if ([self.collectionView.delegate
            conformsToProtocol:
            @protocol(AFCollectionViewDelegateCircleLayout)])
        {
            rotationAngle =
                [(id<AFCollectionViewDelegateCircleLayout>)
                    self.collectionView.delegate
            rotationAngleForSupplmentaryViewInCircleLayout:self];
        }

        layoutAttributes.size = CGSizeMake(20, 200);
        layoutAttributes.center = self.center;
        layoutAttributes.transform3D =
            CATransform3DMakeRotation(rotationAngle, 0, 0, 1);

        // Place the decoration view behind all the cells
        layoutAttributes.zIndex = -1;
    }

    return layoutAttributes;
}
```

I've created an `AFCollectionViewDelegateCircleLayout` protocol that we use to query the collection view's delegate to determine the rotation we should use (see Listing 5.18).

Listing 5.18 `AFCollectionViewDelegateCircleLayout` **Implementation**

```
-
(CGFloat)rotationAngleForSupplmentaryViewInCircleLayout:(AFCollectionViewCircleLa
yout *)circleLayout
{
    CGFloat timeRatio = 0.0f;

    NSDate *date = [NSDate date];
    NSDateComponents *components = [[NSCalendar currentCalendar]
components:NSMinuteCalendarUnit fromDate:date];
    timeRatio = (CGFloat)(components.minute) / 60.0f;

    return (2 * M_PI * timeRatio);
}
```

It's a simple implementation that grabs the current minute, makes the assumption that each hour has only 60 minutes (bad form, I know), and calculates the current angle of the minute hand of an analogue clock, shown in Figure 5.6.

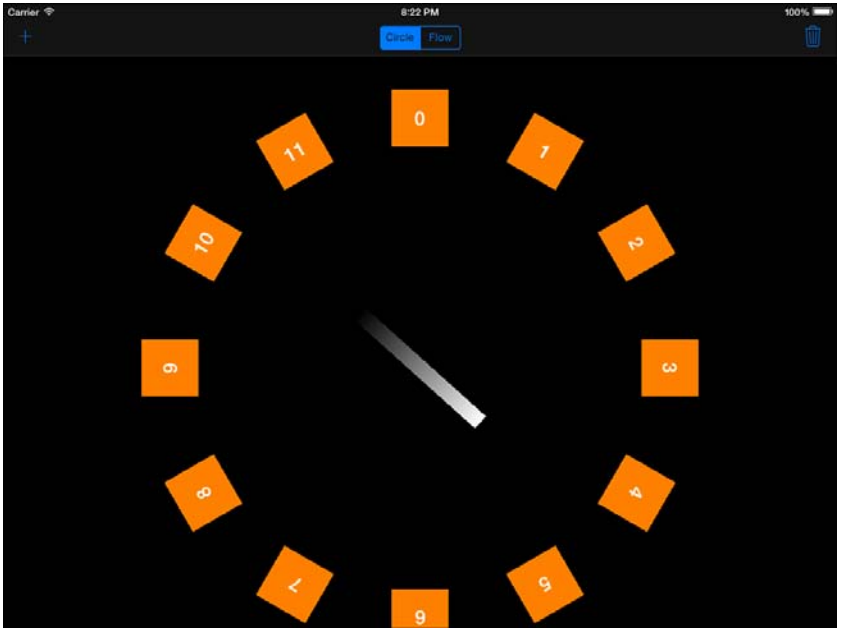


Figure 5.6 Circle layout decoration view

In iOS 6, decoration views were somewhat notorious for being unreliable, particularly in rotation animations. I spoke with some Apple engineers at WWDC 2013 and was able to bring a few edge cases to their attention.

Stacking Layouts

Let's tie things together and make something like a real app. When I worked for 500px, I wrote their open source iOS SDK, which we'll now use to make a basic app to display pictures from their website. We'll also use the image downloader I wrote to download the images once we retrieve the URLs from the 500px API. The sample code for this project is called One Hundred Pixels, since this is about one-fifth of any real 500px app.

First, you need to register an application with 500px. This will get you a consumer key and consumer secret pair, which you need to sign API requests. Create a new application and include the 500px iOS SDK and the `AFImageDownloader` classes in the Xcode project. `#import` these into your precompiled header and set up the `PXRequest`

class in the `applicationDidFinishLaucningWithOptions:` method (see Listing 5.19).

Listing 5.19 Setting Up Your Consumer Key

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:
        [[UIScreen mainScreen] bounds]];
    self.viewController = [[UINavigationController alloc]
        initWithRootViewController:
            [[AFViewController alloc] init]];
    self.viewController.navigationBar.barStyle = UIBarStyleBlack;
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];

    [PXRequest setConsumerKey:@"YOUR_CONSUMER_KEY"
        consumerSecret:@"doesn't matter for this app"];

    return YES;
}
```

Now we're ready to make API calls. We're going to get the 20 most recent images in the Popular, Editors Choice, and Upcoming streams. Let's treat each stream as its own section; create an `enum` to keep track of them. We'll also create three private properties in our implementation file (see Listing 5.20).

Listing 5.20 Using `enum` to Differentiate Sections

```
enum {
    AFViewControllerPopularSection = 0,
    AFViewControllerEditorsSection,
    AFViewControllerUpcomingSection,
    AFViewControllerNumberSections
};

@interface AFViewController ()

@property (nonatomic, strong) AFCollectionViewFlowLayout *
    flowLayout;
@property (nonatomic, strong) AFCollectionViewStackedLayout *
    stackLayout;
@property (nonatomic, strong) AFCoverFlowFlowLayout *
    coverFlowLayout;
```

```
@property (nonatomic, strong) NSMutableArray *popularPhotos;
@property (nonatomic, strong) NSMutableArray *editorsPhotos;
@property (nonatomic, strong) NSMutableArray *upcomingPhotos;

@end
```

I've also included three properties for layouts we haven't defined yet. We'll create those classes soon.

Let's create our API requests to fetch the images, shown in Listing 5.21. First, we need to set up our mutable arrays and our collection view. Although we haven't implemented the code for the following layouts, you can use a standard `UICollectionViewFlowLayout` for now to see the intermediate steps.

Listing 5.21 Setting Up Our View and Fetching from the API

```
-(void)loadView
{
    // Create our view

    // Create instances of our layouts
    self.stackLayout = [[AFCollectionViewStackedLayout alloc]
        init];
    self.flowLayout = [[AFCollectionViewFlowLayout alloc] init];
    self.coverFlowLayout = [[AFCoverFlowFlowLayout alloc] init];

    // Create a new collection view with our flow layout and
    // set ourself as delegate and data source.
    UICollectionView *collectionView = [[UICollectionView alloc]
        initWithFrame:CGRectZero
        collectionViewLayout:self.stackLayout];
    collectionView.dataSource = self;
    collectionView.delegate = self;

    // Register our classes so we can use our custom
    // subclassed cell and header
    [collectionView registerClass:[AFCollectionViewCell class]
        forCellWithReuseIdentifier:CellIdentifier];
    [collectionView registerClass:[AFCollectionViewHeaderView
            class]
        forCellWithReuseIdentifier:HeaderIdentifier];

    // Set up the collection view geometry to cover the whole
    // screen in any orientation and other view properties.
    collectionView.autoresizingMask =
        UIViewAutoresizingFlexibleWidth |
```

```
        UIViewAutoresizingFlexibleHeight;

    // Finally, set our collectionView (since we are a
    // collection view controller, this also sets self.view)
    self.collectionView = collectionView;

    // Setup our model
    self.popularPhotos = [NSMutableArray arrayWithCapacity:20];
    self.editorsPhotos = [NSMutableArray arrayWithCapacity:20];
    self.upcomingPhotos = [NSMutableArray arrayWithCapacity:20];
}

-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];

    void (^block)(NSDictionary *, NSError *) =
      ^(NSDictionary *results, NSError *error) {

        NSMutableArray *array;
        NSInteger section;

        if ([[results valueForKey:@"feature"] isEqualToString:@"popular"])
        {
            array = self.popularPhotos;
            section = AFViewControllerPopularSection;
        }
        else if ([[results valueForKey:@"feature"] isEqualToString:@"editors"])
        {
            array = self.editorsPhotos;
            section = AFViewControllerEditorsSection;
        }
        else if ([[results valueForKey:@"feature"] isEqualToString:@"upcoming"])
        {
            array = self.upcomingPhotos;
            section = AFViewControllerUpcomingSection;
        }
        else
        {
            NSLog(@"%@", [results valueForKey:@"feature"]);
        }

        NSInteger item = 0;
        for (NSDictionary *photo in [results valueForKey:@"photos"])
        {
            NSString *url = [[[photo valueForKey:@"images"]
```

```
                lastObject] valueForKey:@"url"];

            [AFImageDownloader
    imageDownloaderWithURLString:url
    autoStart:YES
    completion:^(UIImage *decompressedImage) {
                [array addObject:decompressedImage];
                [self.collectionView
    reloadItemsAtIndexPaths:@[[NSIndexPath
        indexPathForItem:item inSection:section]]];
            }];

            item++;
        }
    };

    [PXRequest
        requestForPhotoFeature:PXAPIHelperPhotoFeaturePopular
        resultsPerPage:20
        completion:block];
    [PXRequest
        requestForPhotoFeature:PXAPIHelperPhotoFeatureEditors
        resultsPerPage:20
        completion:block];
    [PXRequest
        requestForPhotoFeature:PXAPIHelperPhotoFeatureUpcoming
        resultsPerPage:20
        completion:block];
}
```

We're reusing the same callback block for all three requests, saving on code duplication. Each request will return 20 photo objects, and we'll use `AFImageDownloader` to fetch and decompress the JPEG images.

Our controller is going to *always* have 20 cells per section; we're not going to add cells to our collection view as we download images because they might not be downloaded in the proper order, and it's a lot more work than is in the scope of this chapter. For this reason, our data source methods are straightforward (see Listing 5.22).

Listing 5.22 Setting Up Our View and Fetching from the API

```
-(NSInteger)numberOfSectionsInCollectionView:
    (UICollectionView *)collectionView
{
    return AFViewControllerNumberSections;
}
```

```
(NSInteger)collectionView:(UICollectionView *)view
    numberOfItemsInSection:(NSInteger)section;
{
    return 20;
}

(UICollectionViewCell *)collectionView:
    (UICollectionView *)collectionView
    cellForItemAtIndexPath:(NSIndexPath *)indexPath;
{
    AFCollectionViewCell *cell = (AFCollectionViewCell *)
    [collectionView dequeueReusableCellWithReuseIdentifier:
        CellIdentifier
        forIndexPath:indexPath];

    NSArray *array;

    switch (indexPath.section) {
        case AFViewControllerPopularSection:
            array = self.popularPhotos;
            break;
        case AFViewControllerEditorsSection:
            array = self.editorsPhotos;
            break;
        case AFViewControllerUpcomingSection:
            array = self.upcomingPhotos;
            break;
    }

    if (indexPath.row < array.count)
    {
        [cell setImage:array[indexPath.item]];
    }

    return cell;
}
```

The convenient thing about using an `enum` to record sections is that `AFView-ControllerNumberSections` will change automatically if we add or remove sections.

Now that we have our controller finished, let's define our collection view cell. It's similar to the image cells used in Chapter 4's Cover Flow example. See Listing 5.23 for our cell subclass.

Listing 5.23 `UICollectionViewCell` **Subclass**

```
@interface AFCollectionViewCell ()

@property (nonatomic, strong) UIImageView *imageView;
@property (nonatomic, strong) UIView *maskView;

@end

@implementation AFCollectionViewCell

- (id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    self.backgroundColor = [UIColor whiteColor];

    self.imageView = [[UIImageView alloc]
        initWithFrame:CGRectInset(CGRectMake(0, 0,
            CGRectGetWidth(frame),
            CGRectGetHeight(frame)), 10, 10)];

    self.imageView.autoresizingMask =
        UIViewAutoresizingFlexibleWidth |
        UIViewAutoresizingFlexibleHeight;
    self.imageView.clipsToBounds = YES;
    [self.contentView addSubview:self.imageView];

    self.maskView = [[UIView alloc]
        initWithFrame:CGRectMake(0, 0,
            CGRectGetWidth(frame),
            CGRectGetHeight(frame))];
    self.maskView.backgroundColor = [UIColor blackColor];
    self.maskView.autoresizingMask =
        UIViewAutoresizingFlexibleWidth |
        UIViewAutoresizingFlexibleHeight;
    self.maskView.alpha = 0.0f;
    [self.contentView insertSubview:self.maskView
        aboveSubview:self.imageView];

    return self;
}

-(void)prepareForReuse
{
    [super prepareForReuse];
```

```
    [self setImage:nil];
}

-(void)applyLayoutAttributes:
    (UICollectionViewLayoutAttributes *)layoutAttributes
{
    [super applyLayoutAttributes:layoutAttributes];

    self.layer.shouldRasterize = YES;
    self.layer.shadowColor = [[UIColor blackColor] CGColor];
    self.layer.shadowOffset = CGSizeMake(0, 3);
    self.maskView.alpha = 0.0f;
    if ([layoutAttributes
        isKindOfClass:[AFCollectionViewLayoutAttributes class]])
    {
        self.layer.shadowOpacity =
            [(AFCollectionViewLayoutAttributes *)layoutAttributes
                shadowOpacity];
        self.maskView.alpha =
            [(AFCollectionViewLayoutAttributes *)layoutAttributes
                maskingValue];
    }
}

-(void)setImage:(UIImage *)image
{
    [self.imageView setImage:image];
}
```

As you can see, we're going to use a custom `UICollectionViewLayoutAttributes` class. Define it using two properties—`shadowOpacity` and `maskingValue`—both `CGFloat` values. Don't forget to implement the `NSCopying` method, shown in Listing 5.24.

Listing 5.24 `NSCopying` **Method for Custom Layout Attributes Class**

```
@implementation AFCollectionViewLayoutAttributes

-(id)copyWithZone:(NSZone *)zone
{
    AFCollectionViewLayoutAttributes *attributes = [super
        copyWithZone:zone];

    attributes.shadowOpacity = self.shadowOpacity;
    attributes.maskingValue = self.maskingValue;
```

```
    return attributes;
}

@end
```

We're almost to the stage of running our app. We're going to define three layout objects: The first will stack an entire section's cells on top of one another, the second will display all the cells in a plain flow layout grid, and the third will use a layout similar to Cover Flow.

I again want to take the opportunity to thank Mark Pospesel for his work on IntroducingCollectionViews. The code for the stacked layout and the Cover Flow layout come almost wholesale from his work.

Some of the properties in the stacked layout only make sense in the context of a more interactive collection view, so we'll be saving discussion of those aspects of the layout for the next chapter. If you're curious, you can poke around the `AFCollectionViewStackedLayout` implementation.

An important thing I'll mention now is the `hidden` property on `UICollectionViewLayoutAttributes`. This is an optimization that `UICollectionView` performs to stop itself from rendering cells which don't need to be rendered. In our case, cells "under" the stack don't need to be rendered, because they're covered, so we use the `hidden` property.

When `prepareLayout` is called, we invoke `prepareStacksLayout`, an internal method to set up our instance variables. This method calculates the position for our stacks, and the code looks very familiar to what I think `UICollectionViewFlow` layout looks. Although this is a line-based, breaking layout, I don't believe it could be implemented with `UICollectionViewFlowLayout`, because it deals with sections at a time instead of items (see Listing 5.25).

Listing 5.25 Calculating Section Stack Positions

```
- (void)prepareStacksLayout
{
    self.numberOfStacks = [self.collectionView numberOfSections];
    self.pageSize = self.collectionView.bounds.size;

    CGFloat availableWidth =
        self.pageSize.width - (self.stacksInsets.left +
        self.stacksInsets.right);
    self.numberOfStacksAcross =
        floorf((availableWidth + self.minimumInterStackSpacing) /
            (self.stackSize.width + self.minimumInterStackSpacing));
    CGFloat spacing =
        floorf((availableWidth - (self.numberOfStacksAcross *
```

```
            self.stackSize.width)) / (self.numberOfStacksAcross - 1));
    self.numberOfStackRows =
        ceilf(self.numberOfStacks / (float)self.numberOfStacksAcross);

    self.stackFrames = [NSMutableArray array];
    int stackColumn = 0;
    int stackRow = 0;
    CGFloat left = self.stacksInsets.left;
    CGFloat top = self.stacksInsets.top;

    for (int stack = 0; stack < self.numberOfStacks; stack++)
    {
        CGRect stackFrame = (CGRect){{left, top}, self.stackSize};
        [self.stackFrames addObject:[NSValue valueWithCGRect:stackFrame]];

        left += self.stackSize.width + spacing;
        stackColumn += 1;

        if (stackColumn >= self.numberOfStacksAcross)
        {
            left = self.stacksInsets.left;
            top +=
                self.stackSize.height + STACK_FOOTER_GAP +
                STACK_FOOTER_HEIGHT + self.minimumLineSpacing;
            stackColumn = 0;
            stackRow += 1;
        }
    }

    self.contentSize =
        CGSizeMake(self.pageSize.width,
            MAX(self.pageSize.height,
            self.stacksInsets.top + (self.numberOfStackRows *
                (self.stackSize.height + STACK_FOOTER_GAP +
                STACK_FOOTER_HEIGHT)) + ((self.numberOfStackRows - 1) *
                self.minimumLineSpacing) + self.stacksInsets.bottom));
}
```

The relevant parts of `layoutAttributesForItemAtIndexPath:` are found in Listing 5.26. Again, note the use of the `hidden` property.

Listing 5.26 Calculating Item Attributes for a Stacked Layout

```
- (UICollectionViewLayoutAttributes
*)layoutAttributesForItemAtIndexPath:(NSIndexPath *)path
```

```
{
    CGRect stackFrame = [self.stackFrames[path.section] CGRectValue];

    AFCollectionViewLayoutAttributes* attributes =
        [AFCollectionViewLayoutAttributes
            layoutAttributesForCellWithIndexPath:path];
    attributes.size = CGSizeMake(ITEM_SIZE, ITEM_SIZE);
    attributes.center =
        CGPointMake(CGRectGetMidX(stackFrame), CGRectGetMidY(stackFrame));
    CGFloat angle = 0;

    if (path.item == 1) angle = 5;
    else if (path.item == 2) angle = -5;

    attributes.transform3D =
        CATransform3DMakeRotation(angle * M_PI / 180, 0, 0, 1);
    attributes.alpha = path.item >= VISIBLE_ITEMS_PER_STACK? 0 : 1;
    attributes.zIndex =
        path.item >= VISIBLE_ITEMS_PER_STACK ?
            0 : VISIBLE_ITEMS_PER_STACK - path.item;
    attributes.hidden = path.item >= VISIBLE_ITEMS_PER_STACK;
    attributes.shadowOpacity =
        path.item >= VISIBLE_ITEMS_PER_STACK? 0 : 0.5;
}
```

The `layoutAttributesForItemAtIndexPath:` method is called by our `layoutAttributesForElementsInRect`, after it determines the index paths of the elements visible in that `rect`. This is typical of subclasses of `UICollectionView` itself; they need to calculate which items are in the `rect` because they can't rely on super's implementation, like with `UICollectionViewFlowLayout`. See Listing 5.27 for our implementation.

Listing 5.27 Calculating Item Index Paths in a Given `rect`

```
-(NSArray*)layoutAttributesForElementsInRect:(CGRect)rect
{
    NSMutableArray* attributes = [NSMutableArray array];
    for (int stack = 0; stack < self.numberOfStacks; stack++)
    {
        CGRect stackFrame = [self.stackFrames[stack] CGRectValue];
        stackFrame.size.height +=
            (STACK_FOOTER_GAP + STACK_FOOTER_HEIGHT);

        if (CGRectIntersectsRect(stackFrame, rect))
        {
```

```
            NSInteger itemCount = [self.collectionView
                numberOfItemsInSection:stack];
            for (int item = 0; item < itemCount; item++)
            {
                NSIndexPath* indexPath = [NSIndexPath
                    indexPathForItem:item inSection:stack];
                [attributes addObject:[self
                    layoutAttributesForItemAtIndexPath:indexPath]];
            }
        }
    }

    return attributes;
}
```

Our `AFCollectionViewFlowLayout` is incredibly simple, as shown in Listing 5.28.

Listing 5.28 Calculating Item Index Paths in a Given `rect`

```
@implementation AFCollectionViewFlowLayout

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

    self.itemSize = CGSizeMake(200, 200);
    self.sectionInset = UIEdgeInsetsMake(13.0f, 13.0f, 13.0f, 13.0f);
    self.minimumInteritemSpacing = 13.0f;
    self.minimumLineSpacing = 13.0f;

    return self;
}

@end
```

Finally, our Cover Flow layout is copied directly from the previous chapter (see Listing 5.29). The only difference is the size of the items, which is specified in the controller.

Listing 5.29 Specifying the Item Size for the Cover Flow Layout

```
-(UIEdgeInsets)collectionView:(UICollectionView *)collectionView
layout:(UICollectionViewLayout *)collectionViewLayout
insetForSectionAtIndex:(NSInteger)section
{
```

```
if (collectionViewLayout == self.coverFlowLayout)
{
    CGFloat margin = 0.0f;

    if (UIInterfaceOrientationIsPortrait(self.interfaceOrientation))
    {
        margin = 130.0f;
    }
    else
    {
        margin = 280.0f;
    }

    UIEdgeInsets insets = UIEdgeInsetsZero;

    if (section == 0)
    {
        insets.left = margin;
    }
    else if (section == [collectionView numberOfSections] - 1)
    {
        insets.right = margin;
    }

    return insets;
}
else if (collectionViewLayout == self.flowLayout)
{
    return self.flowLayout.sectionInset;
}
else
{
    // Should never happen.
    return UIEdgeInsetsZero;
}
```

This code might look a little perplexing. Unlike the Cover Flow example from the preceding chapter, we now have more than one section, so we need to use different insets for different sections. Figure 5.7 illustrates the problem. The diagram on top shows what would happen if each section had left and right edge insets set to `margin`; we would have a gap in between the orange and blue sections. Instead, we only want a left margin on the first section and a right margin on the last section.

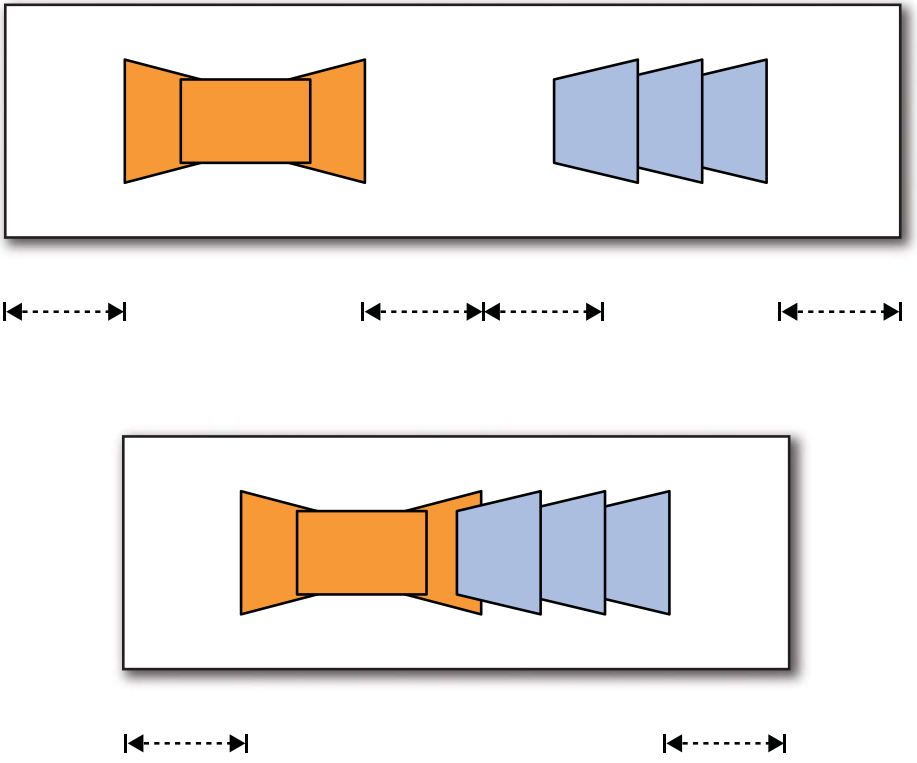


Figure 5.7 The need for different section insets

Because this delegate method belongs to `UICollectionViewDelegateFlowLayout`, it will only get called for our flow layout and our Cover Flow layout.

If we were to run the application right now, it would look like Figure 5.8.

That's nice, but pretty boring. Let's add code that will change layouts when we select an item, as shown in Listing 5.30.

Figure 5.8 The stacked layout

Listing 5.30 Changing Layout on Cell Selection

```
-(void)collectionView:(UICollectionView *)collectionView
didSelectItemAtIndexPath:(NSIndexPath *)indexPath
{
    if (self.collectionView.collectionViewLayout == self.stackLayout)
    {
        [collectionView deselectItemAtIndexPath:indexPath animated:NO];

        [self.flowLayout invalidateLayout];
        [self.collectionView
            setCollectionViewLayout:self.flowLayout animated:YES];

        [self.collectionView scrollToItemAtIndexPath:indexPath
            atScrollPosition:
                UICollectionViewScrollPositionCenteredVertically |
                UICollectionViewScrollPositionCenteredHorizontally
            animated:YES];

        [self.navigationItem
            setLeftBarButtonItem:[[UIBarButtonItem alloc]
                initWithTitle:@"Back"
                style:UIBarButtonItemStyleBordered
```

```
                target:self
                action:@selector(goBack)]
            animated:YES];
    }
    else if (self.collectionView.collectionViewLayout == self.flowLayout)
    {
        [collectionView deselectItemAtIndexPath:indexPath animated:NO];

        [self.coverFlowLayout invalidateLayout];
        [self.collectionView
            setCollectionViewLayout:self.coverFlowLayout animated:YES];
        [self.collectionView
            scrollToItemAtIndexPath:indexPath
            atScrollPosition:
                UICollectionViewScrollPositionCenteredVertically |
                UICollectionViewScrollPositionCenteredHorizontally
            animated:YES];
    }
    else
    {
        [collectionView deselectItemAtIndexPath:indexPath animated:YES];
    }
}

-(void)goBack
{
    if (self.collectionView.collectionViewLayout == self.coverFlowLayout)
    {
        [self.flowLayout invalidateLayout];
        [self.collectionView
            setCollectionViewLayout:self.flowLayout
            animated:YES];
    }
    else if (self.collectionView.collectionViewLayout == self.flowLayout)
    {
        [self.stackLayout invalidateLayout];
        [self.collectionView
            setCollectionViewLayout:self.stackLayout
            animated:YES];

        [self.navigationItem setLeftBarButtonItem:nil animated:YES];
    }
}
```

The implementation of `collectionView:didSelectItemAtIndexPath:` should be fairly familiar. The only addition to our circle layout example from earlier is that we also need to scroll to the selected index, which could be offscreen after the change to our layout. `UICollectionViewScrollPosition` is a bit mask, so we can supply both a horizontal and vertical position; we choose center for both dimensions. We also add a "back" button when we're not displaying the stacked layout. Figure 5.9 shows the transitions between the layouts.

Figure 5.9 Our three layouts

Notice the lovely animations between the layout changes. That's really awesome! And it comes with `UICollectionView` for free; we just had to scroll to our selected index after changing layout.

Now let's contextualize our content a little more. When users first launch the app, they won't know that the first section is Popular. We should add a supplementary view to let them know which section is which.

Create a new supplementary view class that contains a label, as shown in Listing 5.31.

Listing 5.31 Supplementary View for Stack Layout

```
static NSString *kind = @"AFCollectionViewHeaderView";

@interface AFCollectionViewHeaderView ()

@property (nonatomic, strong) UILabel *label;

@end

@implementation AFCollectionViewHeaderView

- (id)initWithFrame:(CGRect)frame
{
    if (!(self = [super initWithFrame:frame])) return nil;

    self.backgroundColor = [UIColor orangeColor];

    self.label = [[UILabel alloc]
        initWithFrame:CGRectMake(0, 0,
            CGRectGetWidth(frame),
            CGRectGetHeight(frame))];
    self.label.autoresizingMask =
        UIViewAutoresizingFlexibleHeight |
        UIViewAutoresizingFlexibleWidth;
    self.label.backgroundColor = [UIColor clearColor];
    self.label.textAlignment = NSTextAlignmentCenter;
    [self addSubview:self.label];

    return self;
}

-(void)setText:(NSString *)text
{
    self.label.text = text;
}

+(NSString *)kind
{
    return kind;
}

@end
```

Register the supplementary view with the collection view in the controller's `loadView` method (see Listing 5.32).

Listing 5.32 Supplementary View Registration

```
[collectionView registerClass:[AFCollectionViewHeaderView class]
    forSupplementaryViewOfKind:[AFCollectionViewHeaderView kind]
    withReuseIdentifier:HeaderIdentifier];
```

Now we need to provide context for the header view in the controller, as shown in Listing 5.33.

Listing 5.33 Supplementary Configuration

```
(UICollectionReusableView *)collectionView:
    (UICollectionView *)collectionView
    viewForSupplementaryElementOfKind:(NSString *)kind
    atIndexPath:(NSIndexPath *)indexPath
{
    AFCollectionViewHeaderView *headerView = [collectionView
        dequeueReusableSupplementaryViewOfKind:kind
        withReuseIdentifier:HeaderIdentifier
        forIndexPath:indexPath];

    switch (indexPath.section) {
        case AFViewControllerPopularSection:
            [headerView setText:@"Popular"];
            break;
        case AFViewControllerEditorsSection:
            [headerView setText:@"Editors' Choice"];
            break;
        case AFViewControllerUpcomingSection:
            [headerView setText:@"Upcoming"];
            break;
    }

    return headerView;
}
```

Finally, we need to modify our stacked layout class to insert a supplementary view for each stack, as shown in Listing 5.34:

Listing 5.34 Supplementary View for Each Stack

```
-(NSArray*)layoutAttributesForElementsInRect:(CGRect)rect
```

```
{
    NSMutableArray* attributes = [NSMutableArray array];
    for (int stack = 0; stack < self.numberOfStacks; stack++)
    {
        CGRect stackFrame = [self.stackFrames[stack] CGRectValue];
        stackFrame.size.height +=
            (STACK_FOOTER_GAP + STACK_FOOTER_HEIGHT);
        if (CGRectIntersectsRect(stackFrame, rect))
        {
            NSInteger itemCount = [self.collectionView
                numberOfItemsInSection:stack];
            for (int item = 0; item < itemCount; item++)
            {
                NSIndexPath* indexPath = [NSIndexPath
                    indexPathForItem:item inSection:stack];
                [attributes addObject:[self
                    layoutAttributesForItemAtIndexPath:indexPath]];
            }

            // add small label as footer
            [attributes addObject:[self
                layoutAttributesForSupplementaryViewOfKind:
                    [AFCollectionViewHeaderView kind]
                atIndexPath:[NSIndexPath
                    indexPathForItem:0 inSection:stack]]];
        }
    }

    return attributes;
}

(UICollectionViewLayoutAttributes *)
    layoutAttributesForSupplementaryViewOfKind:(NSString *)kind
    atIndexPath:(NSIndexPath *)indexPath
{
    if (![kind isEqualToString:[AFCollectionViewHeaderView kind]])
        return nil;

    UICollectionViewLayoutAttributes* attributes =
        [UICollectionViewLayoutAttributes
            layoutAttributesForSupplementaryViewOfKind:kind
            withIndexPath:indexPath];

    attributes.size =
        CGSizeMake(STACK_WIDTH, STACK_FOOTER_HEIGHT);
    CGRect stackFrame =
```

```
        [self.stackFrames[indexPath.section] CGRectValue];
    attributes.center =
        CGPointMake(CGRectGetMidX(stackFrame),
            CGRectGetMaxY(stackFrame) + STACK_FOOTER_GAP +
            (STACK_FOOTER_HEIGHT/2));

    return attributes;
}
```

Perfect. This places the header directly beneath each stack (see Figure 5.10). Let's run the application.

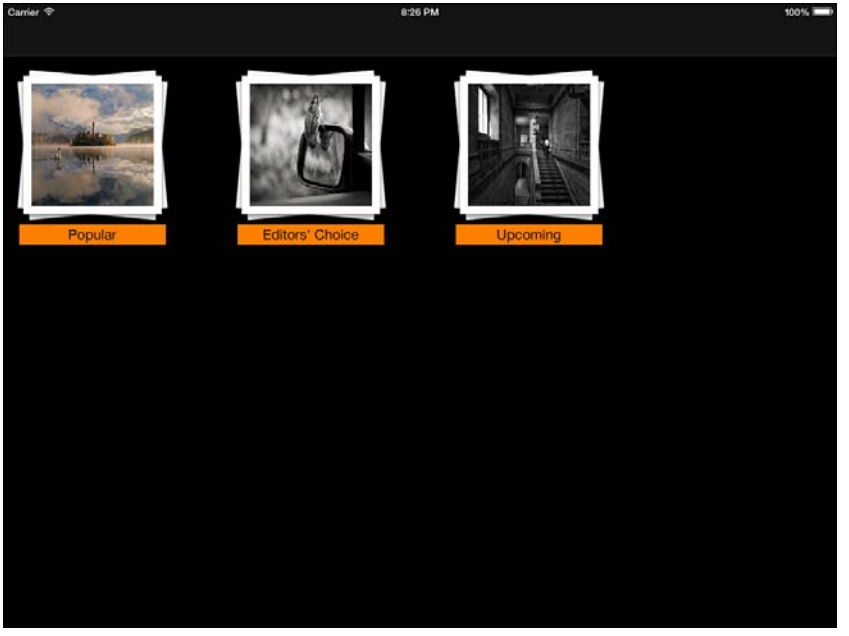


Figure 5.10 Stacked layout headers

That wraps up this chapter. `UICollectionViewLayout` offers a lot of power, but you have to do a lot of work to access that power. If you enjoyed this chapter, and I sincerely hope you have, then I think you'll really like the next one. We're going to look back at the examples through this book and add some interactivity to them.

6

Adding Interactivity to `UICollectionView`

So far in this book, we've concentrated on every aspect of collection views except interactivity. With the exception of some basic copy/paste support provided to you by `UICollectionViewDelegate`, we have not focused on how the user interacts with the collection view beyond basic content offset changes. Sure, we've changed layouts in response to user interaction, but that's not what I'm talking about. I'm asking this: How can we add interactivity directly to the collection view? How can we make truly immersive interfaces? The answer is gesture recognizers, which are the main focus of this chapter.

We're going to take a look back at four code examples and augment them with new user interactivity. This is the polish that makes truly exceptional applications. After reading this chapter, you'll have the skills to apply these same techniques to your own collection views.

Basic Gesture Recognizer

Let's look back at our Better Survey example; the code for this chapter can be found under the Improved Better Survey project. We'll add code to display a custom menu controller above a cell when it's been long-pressed, as shown in Figure 6.1.

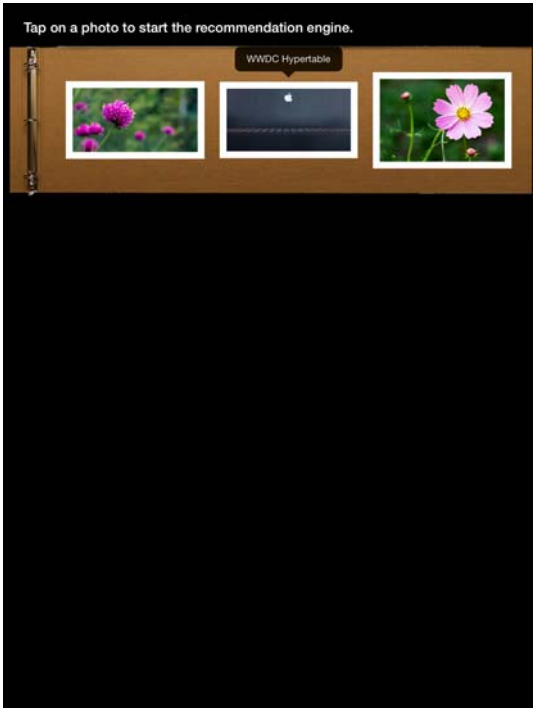


Figure 6.1 Custom menu controller with long-press gesture

Remove the old `UICollectionViewDelegate` methods for cut/copy/paste support; we don't need it anymore. Add the code shown in Listing 6.1 to your `viewDidLoad` implementation.

Listing 6.1 Setting Up the Long-Press Gesture Recognizer

```
    gesutureRecognizer =
    [[UILongPressGestureRecognizer alloc]
        initWithTarget:self
        action:@selector(handleLongPress:)];
[self.collectionView addGestureRecognizer:longPressGestureRecognizer];
```

This creates a long-press gesture recognizer and attaches it to our collection view. It's important to set up your gesture recognizers after your view has been set up. We could do this in `loadView`, but I prefer `viewDidLoad` because this is the sort of behavior we want only after the view has been loaded, and that's exactly what `viewDidLoad` is for.

Note also that we've added the gesture recognizer to the collection view itself instead of each individual cell. This is less code and fewer objects in memory. As you'll see

shortly, `UICollectionView` offers an easy way to determine which cell (if any) is located at a certain point.

Now that we have our gesture recognizer set up, we need to respond to it. Add the method shown in Listing 6.2 to your view controller implementation.

Listing 6.2 Long-Press Gesture Recognizer Method

```
-(void)handleLongPress:(UILongPressGestureRecognizer *)recognizer
{
    if (recognizer.state != UIGestureRecognizerStateBegan) return;

    // Grab the location of the gesture and use it to locate the
    // cell it was made on.
    CGPoint point = [recognizer locationInView:self.collectionView];
    NSIndexPath *indexPath = [self.collectionView
        indexPathForItemAtPoint:point];

    // Check to make sure the long press was performed on a cell.
    if (!indexPath)
    {
        return;
    }

    // Update our ivar for the menuAction: method
    lastLongPressedIndexPath = indexPath;

    // Grab our cell to display the menu controller from
    UICollectionViewCell *cell = [self.collectionView
        cellForItemAtIndexPath:indexPath];

    // Create a custom menu item to hold the name of the model the
    // cell is presenting
    UIMenuItem *menuItem = [[UIMenuItem alloc]
        initWithTitle:[[self photoModelForIndexPath:indexPath] name]
        action:@selector(menuAction:)];

    // Configure the shared menu controller and display it
    UIMenuController *menuController = [UIMenuController
        sharedMenuController];
    menuController.menuItems = @[menuItem];
    [menuController setTargetRect:cell.bounds inView:cell];
    [menuController setMenuVisible:YES animated:NO];
}
```

This method is invoked whenever there is a change to the gesture recognizer. You want to note a couple of important things about this method. First, it checks for the state of the recognizer. Even though this is a long-press gesture recognizer and it doesn't have a "changed" state, we need to make sure that it's in its "began" state (instead of, for example, "possible"). In general, this is good practice when working with gesture recognizers.

Next, it finds the point under which the user is tapping. We have to check the index path returned; it will be `nil` if there is no cell under the tap.

We save our index path in an instance variable for use later; we need to remember which index path was most recently long-pressed. Finally, we grab the shared `UIMenuController` singleton and display a custom `UIMenuItem` from the cell itself. This will display the cut/copy/paste menu on the cell. The `menuAction:` selector is necessary; it cannot be `nil`, but it needs to be implemented in our class somewhere. We implement `menuAction:` in Listing 6.3.

Listing 6.3 Menu Controller Selector

```
-(void)menuAction:(id)sender
{
    // Grab the last long-pressed index path, use it to find its
    // corresponding model, and copy that to the pasteboard

    UIPasteboard *pasteboard = [UIPasteboard generalPasteboard];
    [pasteboard setString:[[self photoModelForIndexPath:lastLongPressedIndexPath]
name]];
}
```

This method is necessary because it will be invoked when the user taps on the custom menu item.

If you were to run the app right now, it would appear broken because the menu controller never appears. Why is that? The answer is a little obscure. `UIViewController`, which our view controller is a descendent of, extends `UIResponder`. The methods in this class are used by `UIMenuController` to determine whether a given object can display a menu controller. Specifically, we need to override `canBecomeFirstResponder`, shown in Listing 6.4, because the default is `NO`, and `canPerformAction:withSender:`, because we've created a custom action.

Listing 6.4 `UIResponder` Overridden Methods

```
- (BOOL)canPerformAction:(SEL)selector withSender:(id)sender
{
    // Make sure the menu controller lets the responder chain know
    // that it can handle its own custom menu action
    if (selector == @selector(menuAction:))
```

```
    {
        return YES;
    }

    return [super canPerformAction:selector withSender:sender];
}

-(BOOL)canBecomeFirstResponder
{
    // Must override to let the menu controller know it can be handled

    return YES;
}
```

Now the long-press gesture recognizer will work as expected.

If you're rolling your own gesture recognizers, you could run into trouble with interference from the built-in recognizers of `UICollectionView`. Look in `UIScrollView.h` for the tap and `panGestureRecognizer` and `pinchGestureRecognizer` properties. If you are having trouble implementing your own gesture recognizers, set up a `UIGestureRecognizerDelegate` and set up a chain of failing recognizers using `require-GestureRecognizerToFail:`.

Responding to Taps

As you saw, a Cover Flow layout isn't that difficult. However, our current implementation lacks a little panache. It would be nice if, when tapping a noncentered cell, the collection view would center on that cell. It sounds easy, but as I found out, some strange behaviors about `UICollectionView` make it a little tricky.

The sample code for the new Cover Flow layout is called Improved Cover Flow. Let's start by setting up another basic gesture recognizer in `viewDidLoad`, as shown in Listing 6.5.

Listing 6.5 Basic Tap Gesture Recognizer Setup

```
UITapGestureRecognizer *tapGestureRecognizer =
    [[UITapGestureRecognizer alloc]
        initWithTarget:self
        action:@selector(handleTapGestureRecognizer:)];
[self.collectionView addGestureRecognizer:tapGestureRecognizer];
```

Great. Before we implement `handleTapGestureRecognizer:`, we need to have some way to determine whether a cell is centered yet. Let's go to the

`AFCoverFlowLayout` class and add the following public method to the header and implementation file, as shown in Listing 6.6.

Listing 6.6 Determining Whether a Cell Is Centered

```
-(BOOL)indexPathIsCentered:(NSIndexPath *)indexPath
{
    CGRect visibleRect = CGRectMake(self.collectionView.contentOffset.x,
        self.collectionView.contentOffset.y,
        CGRectGetWidth(self.collectionView.bounds),
        CGRectGetHeight(self.collectionView.bounds));

    UICollectionViewLayoutAttributes *attributes = [self
        layoutAttributesForItemAtIndexPath:indexPath];

    CGFloat distanceFromVisibleRectToItem =
        CGRectGetMidX(visibleRect) - attributes.center.x;

    return fabs(distanceFromVisibleRectToItem) < 1;
}
```

To determine whether a cell is centered, we rely on our existing layout's methods to determine the attributes of an item at a given index path and check if that cell's distance from the center of the currently visible frame is less than a small value (say, 1).

Next, we implement our gesture recognizer target method (see Listing 6.7).

Listing 6.7 Tap-to-Center Gesture Recognizer

```
-(void)handleTapGestureRecognizer:(UITapGestureRecognizer *)recognizer
{
    if (self.collectionView.collectionViewLayout !=
        coverFlowCollectionViewLayout) return;
    if (recognizer.state != UIGestureRecognizerStateRecognized) return;

    CGPoint point = [recognizer locationInView:self.collectionView];
    NSIndexPath *indexPath = [self.collectionView
        indexPathForItemAtPoint:point];

    if (!indexPath)
    {
        return;
    }

    BOOL centered = [coverFlowCollectionViewLayout
        indexPathIsCentered:indexPath];
```

```
    if (centered)
    {
        UICollectionViewCell *cell = [self.collectionView
            cellForItemAtIndexPath:indexPath];

        [UIView transitionWithView:cell duration:0.5f
            options:UIViewAnimationOptionTransitionFlipFromRight
            animations:^{
                cell.bounds = cell.bounds;
            } completion:nil];
    }
    else
    {
        CGPoint proposedOffset = CGPointZero;
        if (UIInterfaceOrientationIsPortrait(self.interfaceOrientation))
        {
            proposedOffset.x = indexPath.item *
                (coverFlowCollectionViewLayout.itemSize.width +
                coverFlowCollectionViewLayout.minimumLineSpacing);
        }
        else
        {
            proposedOffset.x = indexPath.item - 1) *
                (coverFlowCollectionViewLayout.itemSize.width +
                coverFlowCollectionViewLayout.minimumLineSpacing);
        }

        CGPoint contentOffset = [coverFlowCollectionViewLayout
            targetContentOffsetForProposedContentOffset:proposedOffset
            withScrollingVelocity:CGPointMake(0, 0)];

        [self.collectionView setContentOffset:contentOffset animated:YES];
    }
}
```

The first thing we do is check to make sure that the user tapped while in our Cover Flow layout. Then we check to make sure the gesture recognizer is in the correct state; this is common when dealing with gesture recognizers. We grab the point under the tap and its corresponding `indexPath` and check if it's `nil`, just like last time.

Next, we determine whether the cell is already centered. If it is, we perform a cool flip animation similar to the iTunes app's Cover Flow. If it is not centered yet, we make it centered with an animation by adjusting the content offset.

We can't rely on `UICollectionView`'s `scrollToItemAtIndexPath:scrollToItemAtIndexPath:animated:` method because it will use the item's location while uncentered to perform the calculation of where to go. The effect is that you'll never center the item. Unfortunately, the answer is to calculate where the content offset would be if the item were already centered and scroll to there; we do this when we set `proposedOffset.x`.

This is just an estimate of where the cell would be if it were centered; we only have to get close enough for the Cover Flow layout to take over and readjust it. That's why we call `targetContentOffsetForProposedContentOffset:withScrollingVelocity:`. It will go ahead and adjust our content offset to the exact offset we need to display our cell in the center of the screen.

Pinch and Pan Support

Let's move on to a more recent example we can improve: the Circle Layout. The code for the Circle Layout with gesture support is labeled Improved Circle Layout.

We're going to add a pinching gesture recognizer to the circle that will let the user reposition and resize the circle upon which our items fall. Luckily, our layout already exposes the `center` and `radius` properties, so adjusting them from the view controller is easy.

Currently, we're setting the center and radius of the circle in `prepareLayout`. We'll need to remove these because this method is called every time the layout is invalidated. Instead, we'll put these in the `viewDidLoad` method of the view controller. We'll also set up our pinch recognizer here, shown in Listing 6.8.

Listing 6.8 Updated `viewDidLoad`

```
    // ... continued

    // Set up circle layout

    CGSize size = self.collectionView.bounds.size;
    self.circleLayout.center = CGPointMake(size.width / 2.0,
        size.height / 2.0);
    self.circleLayout.radius = MIN(size.width, size.height) / 2.5;

    // Set up gesture recognizers
    UIPinchGestureRecognizer* pinchRecognizer = [[UIPinchGestureRecognizer alloc]
initWithTarget:self action:@selector(handlePinchGesture:)];
    [self.collectionView addGestureRecognizer:pinchRecognizer];
}
```

We should also override the setters for the `center` and `radius` properties to automatically invalidate the layout for us, as shown in Listing 6.9.

Listing 6.9 Overridden Setters for Invalidating the Layout

```
-(void)setRadius:(CGFloat)radius
{
    _radius = radius;

    [self invalidateLayout];
}

-(void)setCenter:(CGPoint)center
{
    _center = center;

    [self invalidateLayout];
}
```

Great, we're almost done. Really. The layout uses the `center` and `radius` properties to lay out the cells already, so we don't have to change any more layout code. All we need to do is write the gesture recognizer method, which is shown in Listing 6.10.

Listing 6.10 Pinch Gesture Recognizer Method

```
- (void)handlePinchGesture:(UIPinchGestureRecognizer *)recognizer
{
    static CGPoint initialLocation;
    static CGPoint initialPinchLocation;
    static CGFloat initialRadius;

    if (self.collectionView.collectionViewLayout != self.circleLayout)
       return;

    if (recognizer.state == UIGestureRecognizerStateBegan)
    {
        initialLocation = self.circleLayout.center;
        initialPinchLocation = [recognizer
            locationInView:self.collectionView];
        initialRadius = self.circleLayout.radius;
    }
    else if (recognizer.state == UIGestureRecognizerStateChanged)
    {
        CGPoint newLocation = [recognizer
            locationInView:self.collectionView];
```

```
        CGPoint translation;
        translation.x = initialPinchLocation.x - newLocation.x;
        translation.y = initialPinchLocation.y - newLocation.y;

        CGFloat newScale = [recognizer scale];

        self.circleLayout.center = CGPointMake(
            initialLocation.x - translation.x,
            initialLocation.y - translation.y);
        self.circleLayout.radius = initialRadius * newScale;
    }
}
```

When the user begins the gesture, we "remember" the original pinching location, circle center, and circle radius in some static variables. These will retain their values over each iteration of the method invocation. Then, whenever the gesture recognizer changes value, we recalculate the new values for `center` and `radius`, relying on the overridden setters to invalidate the layout for us.

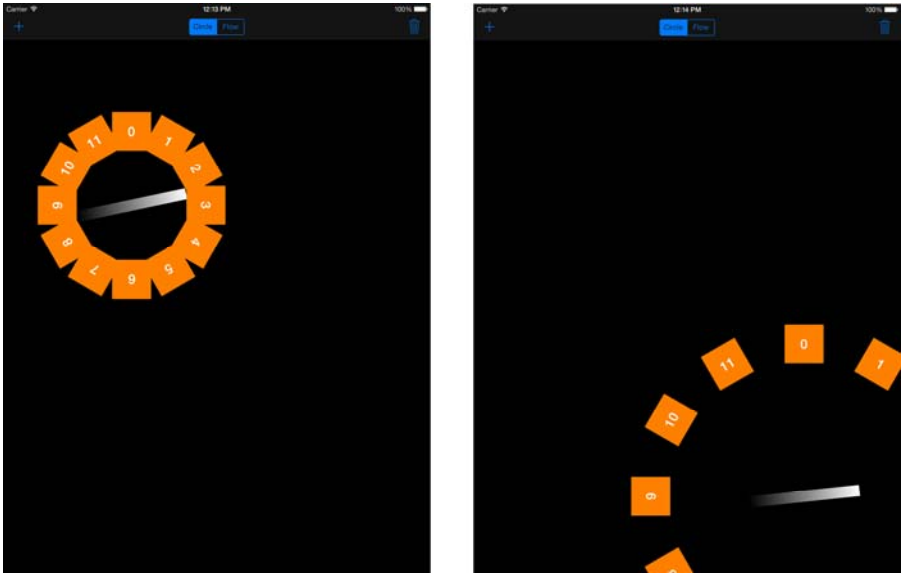


Figure 6.2 Variable circle layout

Let's look at one more pinch layout. The stack layout that Mark Pospesel wrote already contains code for pinching open a stack (just like the Photos app on the iPad does with photo albums).

Listing 6.11 shows a bug with `UICollectionView` where it won't remove supplementary views and decoration views when changing layouts; so for now, we need to do it ourselves.

Listing 6.11 Pinch Gesture Recognizer Method for Stacks

```
-(void)handlePinch:(UIPinchGestureRecognizer *)recognizer
{
    if (self.collectionView.collectionViewLayout != self.stackLayout)
        return;

    if (recognizer.state == UIGestureRecognizerStateBegan)
    {
        CGPoint initialPinchPoint = [recognizer
            locationInView:self.collectionView];
        NSIndexPath* pinchedCellPath = [self.collectionView
            indexPathForItemAtPoint:initialPinchPoint];
        if (pinchedCellPath)
        {
            [self.stackLayout
                setPinchedStackIndex:pinchedCellPath.section];
        }
    }
    else if (recognizer.state == UIGestureRecognizerStateChanged)
    {
        self.stackLayout.pinchedStackScale = recognizer.scale;
        self.stackLayout.pinchedStackCenter = [recognizer
            locationInView:self.collectionView];
    }
    else
    {
        if (self.stackLayout.pinchedStackIndex >= 0)
        {
            if (self.stackLayout.pinchedStackScale > 2.5)
            {
                [self.collectionView
                    setCollectionViewLayout:self.flowLayout animated:YES];
                [self.navigationItem
                    setLeftBarButtonItem:[[UIBarButtonItem alloc]
                    initWithTitle:@"Back"
                    style:UIBarButtonItemStyleBordered
                    target:self action:@selector(goBack)] animated:YES];
            }
            else
            {
                // collapse items back into stack
```

```
                NSMutableArray *leftoverViews = [NSMutableArray array];
                for (UIView *subview in self.collectionView.subviews)
                {
                    // Find all the supplementary views
                    if ([subview isKindOfClass:[AFCollectionViewHeaderView
                        class]])
                    {
                        [leftoverViews addObject:subview];
                    }
                }
                self.stackLayout.collapsing = YES;
                [self.collectionView performBatchUpdates:^{
                    self.stackLayout.pinchedStackIndex = -1;
                    self.stackLayout.pinchedStackScale = 1.0;
                } completion:^(BOOL finished) {
                    self.stackLayout.collapsing = NO;
                    // remove them from the view hierarchy
                    for (UIView *subview in leftoverViews)
                        [subview removeFromSuperview];
                }];
            }
        }
    }
}
```

Figure 6.3 shows the intermediate state of opening a stack with a pinching gesture.

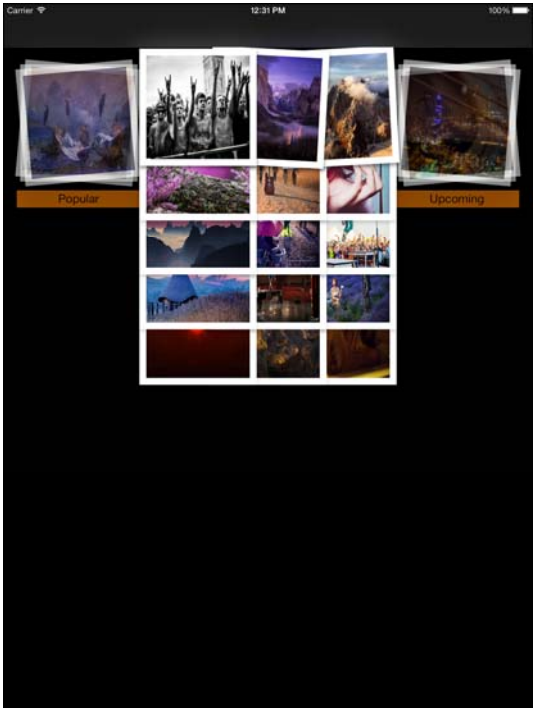


Figure 6.3 Pinching open a stack layout

Layout-to-Layout Transitions

iOS 7 introduced a new concept with `UICollectionViewController`: layout-to-layout transitions. These are handy, easy ways to interpolate between one collection view layout and another. The use case for this technique is when you need a noninteractive "push"-type transition (that is, tapping on a cell to see more detail). Note that the iOS 7 swipe-from-left-edge gesture will still work in its interactive manner.

To use layout-to-layout transitions, you'll need to be using two `UICollectionViewControllers` within a navigation controller. One will push the other onto the navigation stack. Just before pushing that second view controller, set `useLayoutToLayoutNavigationTransitions` to `YES`. The second view controller's collection view will use the same data source and delegate as the first. (As will be the case in our example, this is usually the first view controller itself.) The delegate outlet itself is set to the first view controller's collection view's delegate, but messages are forwarded to the second view controller itself. How Apple is doing this is unclear, and the documentation doesn't specify much.

Our implementation is rather simple. We're going to have a basic view controller with a basic flow layout set up in the app delegate, as shown in Listing 6.12.

Listing 6.12 Layout-to-Layout App Delegate

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Override point for customization after application launch.

    AFPrimaryLayout *layout = [[AFPrimaryLayout alloc] init];
    AFPrimaryViewController *viewController = [[AFPrimaryViewController alloc]
initWithCollectionViewLayout:layout];

    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen]
bounds]];
    self.window.rootViewController = [[UINavigationController alloc]
initWithRootViewController:viewController];

    [self.window makeKeyAndVisible];

    return YES;
}
```

Next is the layout object. Its implementation is simple, as shown in Listing 6.13.

Listing 6.13 Layout-to-Layout Primary Layout

```
@implementation AFPrimaryLayout

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

    self.itemSize = CGSizeMake(140, 140);

    return self;
}

@end
```

Now that you've defined the app delegate and the layout, it's time for our primary view controller itself. This is shown in Listing 6.14.

Listing 6.14 Layout-to-Layout Primary View Controller

```
@implementation AFPrimaryViewController
```

```
static NSString *CellIdentifier = @"Cell";

-(void)viewDidLoad {
    [super viewDidLoad];

    [self.collectionView registerClass:[UICollectionViewCell class]
forCellWithReuseIdentifier:CellIdentifier];
}

-(NSInteger)collectionView:(UICollectionView *)collectionView
numberOfItemsInSection:(NSInteger)section {
    return 100;
}

-(UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
cellForItemAtIndexPath:(NSIndexPath *)indexPath {
    UICollectionViewCell *cell = [collectionView
dequeueReusableCellWithReuseIdentifier:CellIdentifier forIndexPath:indexPath];

    cell.backgroundColor = [UIColor purpleColor];

    return cell;
}

-(void)collectionView:(UICollectionView *)collectionView
didSelectItemAtIndexPath:(NSIndexPath *)indexPath {
    [collectionView deselectItemAtIndexPath:indexPath animated:YES];

    AFSecondaryViewController *viewController = [[AFSecondaryViewController
alloc] initWithCollectionViewLayout:[[AFSecondaryLayout alloc] init]];
    viewController.useLayoutToLayoutNavigationTransitions = YES;
    [self.navigationController pushViewController:viewController animated:YES];
}

@end
```

Note the bolded code. The secondary view controller's implementation is *completely empty*. That's because its collection view is relying on the primary view controller as the data source.

UIKit Dynamics

UIKit Dynamics are a new iOS 7 technology that uses a two-dimensional physics simulation to drive animations. They can also be used to drive collection view layouts, as discussed here.

The general use of UIKit Dynamics is beyond the scope of this book, but you can read more about them here. The crux of it is that a `UIDynamicAnimator` object drives the physics simulation, updating the center, size, and two-dimensional transform of a `UIView` or a `UICollectionViewLayoutAttribute`. We're going to take a look at an open source example I've written. The source code is available on GitHub here.

This example uses UIKit Dynamics to reproduce the bouncy spring effect present in iOS 7's Messages app.

When we initialize our dynamic animator, we pass it our collection view layout. This is important because the dynamic animator is going to be responsible for invalidating our layout whenever the underlying physics simulation changes. We're going to subclass a flow layout so that we can rely on some logic in the superclass. We're going to rely on this logic to update some spring-like *behaviors* in our animator. Each behavior is going to represent a layout element of our collection view.

So basically, we have a collection view layout that is going to own a dynamic animator. (Someone needs to own a strong reference to it.) That animator is going to contain spring-like attachment behaviors representative of the layout elements of our collection view. When we scroll, we're going to rely on the logic in `UICollectionViewFlowLayout` to update our behaviors.

Start with a basic application with a collection view controller, as shown in Listing 6.15.

Listing 6.15 Basic Collection View Controller

```
@implementation ASHCollectionViewController

static NSString * CellIdentifier = @"CellIdentifier";

-(void)viewDidLoad
{
    [super viewDidLoad];
    [self.collectionView registerClass:[UICollectionViewCell class]
            forCellWithReuseIdentifier:CellIdentifier];
}

-(UIStatusBarStyle)preferredStatusBarStyle
{
    return UIStatusBarStyleLightContent;
}

-(void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
```

```
    [self.collectionViewLayout invalidateLayout];
}

#pragma mark - UICollectionView Methods

-(NSInteger)collectionView:(UICollectionView *)collectionView
    numberOfItemsInSection:(NSInteger)section
{
    return 120;
}

-(UICollectionViewCell *)collectionView:(UICollectionView *)collectionView
                 cellForItemAtIndexPath:(NSIndexPath *)indexPath
{
    UICollectionViewCell *cell = [collectionView
        dequeueReusableCellWithReuseIdentifier:CellIdentifier
                                  forIndexPath:indexPath];

    cell.backgroundColor = [UIColor orangeColor];
    return cell;
}

@end
```

We're invalidating the layout as soon as our view appears because the application this demo is in uses storyboards; this isn't necessary if you set your collection views up using code. Let's look at Listing 6.16, the private interface for our collection view layout.

Listing 6.16 Collection View Layout Interface

```
@interface ASHSpringyCollectionViewFlowLayout ()

@property (nonatomic, strong) UIDynamicAnimator *dynamicAnimator;

@end
```

Nothing fancy here; just keeping a reference to the dynamic animator. Let's also set up our basic properties in the initializer, as shown in Listing 6.17.

Listing 6.17 Collection View Layout Initializer

```
- (id)init
```

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

    self.minimumInteritemSpacing = 10;
    self.minimumLineSpacing = 10;
    self.itemSize = CGSizeMake(44, 44);
    self.sectionInset = UIEdgeInsetsMake(10, 10, 10, 10);

    self.dynamicAnimator = [[UIDynamicAnimator alloc]
initWithCollectionViewLayout:self];

    return self;
}
```

Let's next implement our prepare layout method. We can call the superclass's implementation to lay out our collection view layout attributes according to the properties we set in the initializer. After we've prepared the layout in our superclass, we can use its implementation for determining the layout attributes in a given `rect`. Let's look at the attributes in the `rect` defined by our entire content size, shown in Listing 6.18.

Listing 6.18 `prepareLayout` Implementation

```
[super prepareLayout];

CGSize contentSize = self.collectionView.contentSize;
NSArray *items = [super layoutAttributesForElementsInRect:
    CGRectMake(0.0f, 0.0f, contentSize.width, contentSize.height)];
```

Note that this is incredibly inefficient. (Imagine if our collection view was even a little bigger; the number of items would take up a lot of memory simultaneously.) Iterating over all of them, as we're about to do, would take up a lot of CPU time, as well.

We'll need to check whether our dynamic animator already has behaviors for our items. If it does, and we add duplicate behaviors, we'll get a runtime exception. Listing 6.19 shows our implementation.

Listing 6.19 Collection View Layout Interface

```
if (self.dynamicAnimator.behaviors.count == 0) {
    [items enumerateObjectsUsingBlock:^(id<UIDynamicItem> obj, NSUInteger idx,
BOOL *stop) {
        UIAttachmentBehavior *behaviour = [[UIAttachmentBehavior alloc]
initWithItem:obj

attachedToAnchor:[obj center]];
```

```
        behaviour.length = 0.0f;
        behaviour.damping = 0.8f;
        behaviour.frequency = 1.0f;

        [self.dynamicAnimator addBehavior:behaviour];
    }];
}
```

For each item in the full content rect of the collection view, we create an attachment behavior based on that item, configure it, and add it to the dynamic animator. I chose those values for the properties on the behaviors because they seemed nice, experimentally.

Next, we need to forward inquiries about the state of our collection view layout attributes to our dynamic animator (see Listing 6.20). This is relatively straightforward, because dynamic animators were designed specifically to work with collection views.

Listing 6.20 Forwarding Messages to the Dynamic Animator

```
-(NSArray *)layoutAttributesForElementsInRect:(CGRect)rect
{
    return [self.dynamicAnimator itemsInRect:rect];
}

-(UICollectionViewLayoutAttributes
*)layoutAttributesForItemAtIndexPath:(NSIndexPath *)indexPath
{
    return [self.dynamicAnimator layoutAttributesForCellAtIndexPath:indexPath];
}
```

The next step is to respond to scrolling events. We're going to do this in a slightly roundabout way: We're going to override the super implementation of `shouldInvalidateLayoutForBoundsChange`. This method is called whenever the bounds of the collection view changes, such as when it's scrolled by the user's finger (shown in Listing 6.21).

Listing 6.21 Responding to Scrolling

```
-(BOOL)shouldInvalidateLayoutForBoundsChange:(CGRect)newBounds
{
    UIScrollView *scrollView = self.collectionView;
    CGFloat delta = newBounds.origin.y - scrollView.bounds.origin.y;

    CGPoint touchLocation = [self.collectionView.panGestureRecognizer
locationInView:self.collectionView];
```

```
    [self.dynamicAnimator.behaviors
enumerateObjectsUsingBlock:^(UIAttachmentBehavior *springBehaviour, NSUInteger
idx, BOOL *stop) {
        CGFloat yDistanceFromTouch = fabsf(touchLocation.y -
springBehaviour.anchorPoint.y);
        CGFloat xDistanceFromTouch = fabsf(touchLocation.x -
springBehaviour.anchorPoint.x);
        CGFloat scrollResistance = (yDistanceFromTouch + xDistanceFromTouch) /
1500.0f;

        UICollectionViewLayoutAttributes *item =
springBehaviour.items.firstObject;
        CGPoint center = item.center;
        if (delta < 0) {
            center.y += MAX(delta, delta*scrollResistance);
        }
        else {
            center.y += MIN(delta, delta*scrollResistance);
        }
        item.center = center;

        [self.dynamicAnimator updateItemUsingCurrentState:item];
    }];

    return NO;
}
```

There's a lot of math in there; don't worry, though, we'll tease it apart. First, we calculate the change in content offset y (that is, how much the user has scrolled by since the last time this method was called). Next, we determine where the user is touching on the collection view. This is important because we want items closer to the user's finger to scroll more rapidly and want items farther away to lag behind a bit more.

For each behavior in our dynamic animator, we divide the sum of the x and y deltas by a denominator of 1500, a value determined experimentally. Use a smaller denominator to make the collection view react with more "spring." This is like a "resistance" to the scrolling of the collection view. We then, finally, cap that product at a min or max of the delta. This prevents the delta from being negative and having items really far away from the user's finger scrolling in the opposite direction than they're supposed to.

Finally, notice that we return `NO` to the method. Because the dynamic animator is going to take care of invalidating our layout, we don't have to do so here.

That's really all there is to it. You can build and run the application, or you can download an animated GIF of the collection view in action here. This rather naïve approach works for collection views with up to a few hundred items. This is sufficient for the

scope of this book. If you'd like to learn more about how to *tile* the behaviors, take a look at a tutorial I wrote on the topic at obj.io.

Now that you understand how gesture recognizers can be used to interact with collection views—via their layouts—and understand the basics of using UIKit Dynamics to back a collection view layout, you're ready to create truly immersive, awesome applications. Good luck!