Community Experience Distilled

Creating Games with cocos2d for iPhone 2

Master cocos2d through building nine complete games for the iPhone

Paul Nygard

[PACKT] PUBLISHING open source* community experience distilled

Creating Games with cocos2d for iPhone 2

Master cocos2d through building nine complete games for the iPhone

Paul Nygard

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Creating Games with cocos2d for iPhone 2

First published: December 2012

Production Reference: 1171212

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84951-900-7

www.packtpub.com

Cover Image by Abhishek Pandey (abhishek.pandey1210@gmail.com)

Credits

Author
Paul Nygard

Reviewers
Dave Hersey
Marcio Valenzuela P

Acquisition Editor
Usha Iyer

Lead Technical Editor
Llewellyn F. Rozario

Technical Editors
Sharvari Baet
Komal Chheda
Kirti Pujari
Brandt D'Mello

Project Coordinator
Abhishek Kori

Proofreaders
Bob Phillips
Bernadette Watkins

Indexer
Monica Ajmera Mehta

Production Coordinator
Shantanu Zagade

Cover Work
Shantanu Zagade

About the Author

Paul Nygard has been a computer and technology enthusiast since he was introduced to his first computer at the age of six. He has spent most of his professional life building reporting and analytical systems for a large company. After teaching himself Objective-C in 2010, he has focused most of his attention on iOS programming. Paul created a personal development label, Troll Cave Games (`http://www.trollcavegames.com`), to release his mobile gaming efforts. Paul also works as a consultant and developer-for-hire for visionaries in need of a coder. In his leisure time, Paul is passionate about games, books, and music.

I would like to thank my family for their support and for allowing me to hide in my troll cave and pursue my passions. This book is for you!

I would also like to thank Ricardo Quesada and Steve Oldmeadow, for all their hard work on cocos2d and cocosDenshion. They have helped to mold a new generation of developers by giving us all a wonderful world to play in. And to the cocos2d community: I have never seen such a large group of developers so eager to help each other along. Thanks for the support, advice, and humor over the years.

About the Reviewers

Dave Hersey has over 35 years of experience in Apple software development, dating back to the Apple II personal computer in 1977. After a successful career of more than 6 years at Apple Computer, Inc., Dave started Paracoders, Inc. in 2000 focusing on custom Mac OS X-based application and driver development. In 2008, Dave's business expanded into iOS (iPhone) mobile applications, followed by Android applications soon after. Some bigger-named clients include Paramount Home Entertainment, Lionsgate Entertainment, Seagate, Creative Labs, and Kraft. Most recently, Dave's business expansion has included additional mobile and server-side platforms as well as support services. This has led to rebranding his company as "Torchlight Apps" (`http://www.torchlightapps.com`).

Dave is also a moderator and active participant on the `cocos2d-iphone.org` forums under the username `clarus` when he's not busy with his wife raising three children, three dogs, three parakeets, four ducks, and seven ducklings at last count.

Marcio Valenzuela P is a biochemist who has studied programming as a hobby for over 12 years. He is perseverant, auto-didactic, and is always looking into the latest technologies. Marcio started by picking up ASP back in the early 1990s as Chief Web Developer for a consulting company that developed web applications for private companies. He also delved into PHP applications with a MySQL database backend. Then in 2008 he started his path down iOS and has had experience developing applications and games for the platform. His experience is mostly in business applications where there exists a cloud-based web service to interact with and more recently in games created in cocos2d.

Marcio is cofounder of `Activasolutions.com` and currently runs a small iOS project called `Santiapps.com`, which programs for companies wishing to enter the iOS platform. Marcio is a forum moderator at `RayWenderlich.com`.

I would like to acknowledge the time I have taken from raising my son to dedicate to this book. I just hope someday Santiago follows in the programming tradition as it fosters critical skills such as problem solving and innovation which is something we share.

---

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Cocos2d for iPhone is a robust framework for developing 2D games for any iOS device. It is powerful, flexible, and best of all it is free to use for your own projects. Thousands of apps, including many top selling games, have been written using cocos2d.

Creating Games with cocos2d for iPhone 2 will take you on a tour of nine very different games, guiding you through the designing process and code needed to build each game. All of the games were specifically selected to highlight different approaches to solving the challenges inherent in designing a game.

The games covered in this book are all classic game styles. By focusing on games you probably already know, you can see how everything works "under the hood" with cocos2d.

What this book covers

Chapter 1, *Thanks for the Memory Game*, covers the design and building of a memory tile game. It covers basic concepts such as grid layouts, using cocos2d actions, and using CocosDenshion for sound effects.

Chapter 2, *Match 3 and Recursive Methods*, walks through the design and building of a match-3 game. This chapter covers two different approaches to checking for matches, as well as an extensive section on predictive matching and how to generate artificial randomness.

Chapter 3, *Thumping Moles for Fun*, provides the basic concepts of how to design a mole thumping game. This game uses Z-ordering to "trick the eye", and uses cocos2d actions extensively to give a very polished game with very little coding needed.

Chapter 4, *Give a Snake a Snack...*, follows the design and building of a snake game. Some of the topics covered in this chapter include overriding methods, making sprites follow each other, and implementing increasing difficulty levels.

Chapter 5, *Brick Breaking Balls with Box2D*, covers the building of a brick-breaking game using the Box2D physics engine. In this chapter, you will find a basic primer on how to use Box2D, using plists to store level data, and implementing power-ups.

Chapter 6, *Cycles of Light*, takes us to an iPad only multiplayer game. This game allows two players to compete head-to-head on the same iPad, or by using GameKit's Bluetooth connectivity to play against each other on two iPads. This chapter also walks through how we can use a single pixel to draw almost everything in the game.

Chapter 7, *Playing Pool, Old School*, revisits the Box2D physics engine to build a top-down pool game. The focus on this chapter is to implement a simple "rules engine" into the game, as well as how to easily build multiple control methods into the same game.

Chapter 8, *Shoot, Scroll, Shoot Again*, walks through the building of a top-down scrolling shooter. This chapter walks you through how to use readily available outside tools and resources, including Sneaky Joystick and the Tiled tile map editor. It also covers two different forms of enemy AI, including A* Pathfinding.

Chapter 9, *Running and Running and Running...*, brings us to the most ambitious game of all, the endless runner. The primary topics covered are how to create random terrain that characters can walk on, parallax scrolling backgrounds, and implementing a lot of different types of enemies.

What you need for this book

The book and code bundle contain the complete source code you will need to run all nine games. You will only need a few items to run the games:

- An Intel-based Macintosh running OS X Lion (or later)
- Xcode version 4.5 (or higher)
- To run any games on a real device (iPhone, iPad, or iPod Touch), you will need to be enrolled in the Apple's iOS Developer program. You will be able to run the games in the iOS Simulator without enrolling, but the tilt controls in *Chapter 8*, *Shoot, Scroll, Shoot Again* and the Bluetooth multiplayer mode in *Chapter 6*, *Cycles of Light* will only work on a real iOS device.

Who this book is for

This book is written for people who have basic experience with cocos2d, but want some guidance on how to approach real-world design issues. Although the book does revisit some basic concepts, we hit the ground running, so having a basic understanding of cocos2d is recommended. At least some knowledge of Objective-C is also strongly recommended.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
-(void) makeTransition:(ccTime)dt
{
  [[CCDirector sharedDirector] replaceScene:
     [CCTransitionFade transitionWithDuration:1.0
        scene:[MTMenuScene scene] withColor:ccWHITE]];
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
-(void) makeTransition:(ccTime)dt
{
  [[CCDirector sharedDirector] replaceScene:
     [CCTransitionFade transitionWithDuration:1.0
        scene:[MTMenuScene scene] withColor:ccWHITE]];
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output.

You can download this file from `http://www.packtpub.com/sites/default/files/downloads/9007OS_ColoredImages.pdf`

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Thanks for the Memory Game

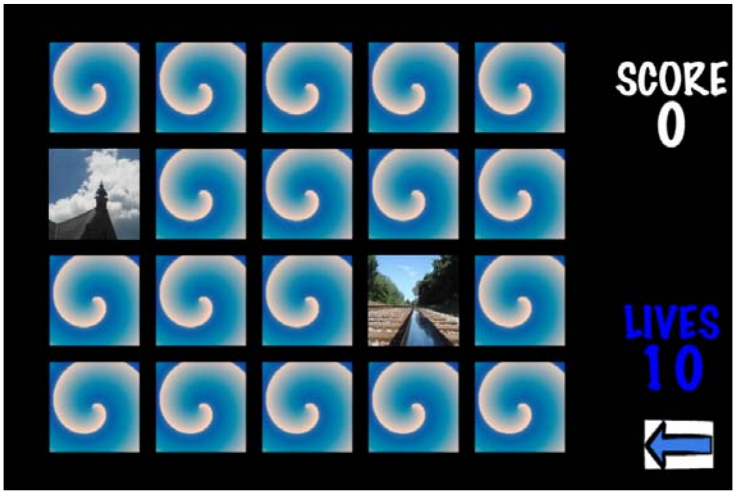
As children, we learn many useful skills by playing games. We learn coordination, strategy, and memory skills. These are all skills we take with us throughout our lives. The perfect place to start is a traditional childhood game.

In this chapter, we cover the following topics:

- Scenes versus layers
- Sprites and sprite sheets
- Loading sequential files
- Random playfield generation
- Touch handlers
- Using actions
- Basic matching strategies
- Scoring
- Tracking lives
- Game over conditions
- SimpleSoundEngine

The project is...

We will begin with classic **memory game**. Not just any memory game – *the* memory game that is the source of joy and frustration of children everywhere. In case you've never been exposed to this game (really?), the gameplay is simple. The playing field is a set of tiles with pretty pictures on the front, and a generic back image. You turn over two tiles to see if you made a match. If not, you turn them back over. Pick another pair of tiles and see if they match. Repeat this till all the tiles are matched. Let's take a look at the finished game:



Our memory game needs to be flexible enough to allow different skill levels in the game. We will create different skill levels by varying the number of memory tiles on the board. If there are four tiles (two each of two designs), that's pretty easy. Creating a 4 x 5 grid of tiles is a lot more challenging (20 tiles, 10 designs). We will build a single project that will handle these variations dynamically.

Our game will be a little different from the traditional version in two ways: it is single player only, and we will add a way to lose the game, to make it more exciting. We'll go into detail on that aspect later.

We will detail several foundational design approaches in this chapter that will be used throughout the book. To avoid repetitive code, later chapters will omit some of the boilerplate detail that we cover here.

Let's build a menu

We'll start building the project from the default cocos2d v2.x - cocos2d iOS template. Once the project is created, we first remove the `HelloWorldLayer.h/.m` files. `HelloWorld` is a good starting point for learning the code structure, but we don't really want (or need) this boilerplate class for anything (don't forget to remove the `#import "HelloWorldLayer.h"` at the top of the `IntroLayer.m` class). For now we'll leave the reference in the bottom of the `IntroLayer.m`'s `makeTransition` class.

One of the most commonly used classes in the cocos2d framework is probably the `CCLayer`. A `CCLayer` is the object that is (usually) represented on the screen, and acts as our "canvas" for our game. We use the `CCLayer` object as a basis, and then create subclasses of it to add our own game code.

There is another often-used class, the `CCScene` class. A `CCScene` class can be thought of as a "container" for `CCLayer` objects. A `CCScene` object is rarely used for much more than adding `CCLayer`s as children. A good comparison is like the creation of cartoons before the age of computers. Each scene was assembled from a stack of transparent plastic sheets, each with a different part of the scene on it: each main character would have their own layer, another for the background, another for each different element of the scene. Those plastic sheets are the equivalent of a `CCLayer` objects, and the `CCScene` class is where these are stacked up to display on screen.

We will start with a basic `CCLayer` subclass, `MTMenuLayer`. We create a title, and a basic menu. We need to pay attention to how we call the `MTPlayfieldScene` class (our main game screen) from the menu.

Filename: `MTMenuLayer.m`

```
-(void) startGameEasy {
    [[CCDirector sharedDirector] replaceScene:
     [MTPlayfieldScene sceneWithRows:2 andColumns:2]];
}

-(void) startGameMedium {
    [[CCDirector sharedDirector] replaceScene:
     [MTPlayfieldScene sceneWithRows:3 andColumns:4]];
}

-(void) startGameHard {
    [[CCDirector sharedDirector] replaceScene:
     [MTPlayfieldScene sceneWithRows:4 andColumns:5]];
}
```

You will notice that the `startGameXXX` methods are calling a custom constructor for the scene, rather than the normal `[MyLayer scene]` that is commonly used. We will explain the `sceneWithRows:andColumns:` method shortly.

This book will not include the complete code within the text. Portions that aren't interesting for the discussion will be omitted.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Where is the scene?

Oh, you noticed? The boilerplate cocos2d template includes a class method `+(id)` scene inside the layer (in `HelloWorldLayer`). While this approach works, it can lead to confusion as we build more complex scenes with multiple layers. Using the template-based approach may seem odd when you call a method that takes a `CCScene` object as a parameter, yet you pass it a value like `[MySpecialLayer scene]`. So are you referencing a `CCScene` or `CCLayer` object? It makes a lot more logical sense to us that you would, in this example, pass a value like `[MySpecialScene scene]`. It is less confusing to pass a scene object when a `CCScene` is requested. A `CCScene` object is a higher-level container that was designed to contain `CCLayer` objects, so why not keep it as its own class? Let's go ahead and examine our approach:

Filename: `MTMenuScene.h`

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "MTMenuLayer.h"

@interface MTMenuScene : CCScene {
}

+(id)scene;

@end
```

Filename: `MTMenuScene.m`

```
#import "MTMenuScene.h"

@implementation MTMenuScene

+(id)scene {
    return( [ [ [ self alloc ] init ] autorelease ] );
}

-(id) init
{
  if( (self=[super init])) {
        MTMenuLayer *layer = [MTMenuLayer node];
        [self addChild: layer];
  }
  return self;
}

@end
```

Here we have followed the convention that the scene method returns an `autoreleased` object. We do not explicitly call `alloc` on it (when we instantiate the class), so we don't "own" the object.

Now we can go back to the `IntroLayer.m` file, and change the `makeTransition` method to point to our new menu scene:

```
-(void) makeTransition:(ccTime)dt
{
  [[CCDirector sharedDirector] replaceScene:
     [CCTransitionFade transitionWithDuration:1.0
        scene:[MTMenuScene scene] withColor:ccWHITE]];
}
```

We also need to make sure we are importing the `MTMenuScene.h` file in the `AppDelegate.m` file. Now that our menu is complete, we can concentrate on the game itself.

It is important to note that this design of using `CCScene` as a separate class in the structure is not universally adopted. Many people choose to follow the same approach as the templates. Both ways will work, but we are of the "camp" that strongly believes these should kept separate, as we have done here. Both ways are perfectly valid coding practice, and you are free to structure your code in other way.

Building the playfield

Next, we will add a `CCScene` class to drive our main game screen, here named `MTPlayfieldScene`. Much of this looks the same as the `MTMenuScene` class we defined earlier, except here we define a method `sceneWithRows:andColumns:` instead of the simpler scene method we used in the previous code.

Filename: `MTPlayfieldScene.m`

```
+(id) sceneWithRows:(NSInteger)numRows
          andColumns:(NSInteger)numCols {
  return [[[self alloc] sceneWithRows:numRows
                            andColumns:numCols]
                                   autorelease];
}

-(id) sceneWithRows:(NSInteger)numRows
          andColumns:(NSInteger)numCols {

  if( (self=[super init])) {
        // Create an instance of the MTPlayfieldLayer
        MTPlayfieldLayer *layer = [MTPlayfieldLayer
                                   layerWithRows:numRows
                                   andColumns:numCols];
        [self addChild: layer];
  }
  return self;
}
```

Here we have the custom `sceneWithRows:andColumns:` method we referenced in the `MTMenuLayer` earlier. The class method handles the `alloc` and `init` methods, and identifies it as an autoreleased object, so we don't have to worry about releasing it later. The `sceneWithRows:andColumns:` method passes the rows and columns variables directly to the `MTPlayfieldLayer` class' custom `init` method, `layerWithRows:andColumns:`. This lets us pass the requested values through the scene to the layer, where we can use the values later.

We need sprites

Before we proceed with building the playing field, we need some graphics to use in our game. Our design calls for square images for the tiles, and one image to use for the common back of the tiles. Because we want them to be able to scale to different sizes (for our different skill levels), we need images large enough to look good at the simplest skill level, which is a two by two grid. Unless your goal is the "chunky pixel" look, you never want to scale images up. Based on the screen size, we want our tiles to be 150 points wide and 150 points high. Since we want to use better graphics on iPhone (and iPod Touch) Retina Display devices, our -hd version of the graphics will need to be 300 pixels by 300 pixels.

Points are the easiest way to use cocos2d effectively. On an older iOS Device, one point equals one pixel on the screen. On Retina Display devices, one point equals *two* pixels, which occupy the same physical space on the screen as the one pixel on the non-Retina screens. From a practical perspective, this means that once you provide the -hd graphics, cocos2d will treat your Retina and non-Retina layouts identically, with no extra layout code. You *can* do things in pixels if you really want to, but we wouldn't recommend making a habit of it.

For this game, we will be using a variety of photos. There was some amount of manipulation needed to arrive at the proper aspect ratio and size. This is a great place to make use of Automator that is part of Mac OS X. There is an Automator script in the source code for this chapter inside a folder called `Helpers`. When you run it, it will prompt for a folder of images. Once selected, it will create a folder on your desktop called `ch1_tiles`, and it will contain sequentially numbered images (that is `tile1.png`, `tile2.png`, and so on), with each image being exactly 300 pixels by 300 pixels.

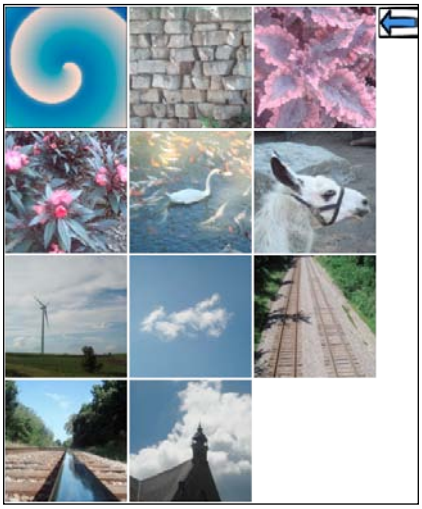
Two other images will be needed to build the game: `backButton.png` will be used for navigation and `tileback.png` will be used as the image for the back of the tiles when they are face down.

Building a sprite sheet

Sprite sheets are one of the foundations of any effective cocos2d game. Compiling all of your sprites into sprite sheets lets cocos2d load fewer files as well as giving you the ability to use a batch node. We won't delve into the "under the hood" details of the `CCSpriteBatchNode` class here except at a high level. When you load a sprite sheet into a batch node, it acts as a "parent" for all the sprites on it. When you use a batch node, the calls to draw the sprites on the screen are batched together, which gives a performance boost. This batch drawing allows the system to draw 100 sprites with about the same efficiency (and speed) as drawing one sprite. The bottom line is batch nodes allow you to draw a lot more on-screen without slowing your game down.

There are two files needed for a sprite sheet: the texture (image file) and the plist file. We don't even want to think about attempting to hand-build the sprite sheet. Fortunately, there are a number of very good tools that were built for this. The most established sprite sheet tools in the cocos2d community are TexturePacker (`http://www.texturepacker.com`) and Zwoptex (`http://zwopple.com/zwoptex`), although there are a number of newer apps that are also available. Which tool you use is a matter of personal preference. Regardless of the tool, you will need to create both the standard and -hd versions of the images. (Most current tools have built-in options to aid in this process.)

No matter which tool is used, the desired result is four files: `memorysheet.png`, `memorysheet.plist`, `memorysheet-hd.png`, and `memorysheet-hd.plist`. The -hd files include the 300 x 300 images for the iPhone Retina Display, and the others include the 150 x 150 pixel images for non-Retina iPhone Displays. We also include the `backButton.png` and `tileback.png` files in appropriate sizing in both sprite sheets as well. Let's take a look at the final sprite sheet we will use for this game:



On to the playfield

Now we're ready to get to the playfield layer itself. We know we need to keep track of the size of the game screen, how big each tile should be, how big the game board should be, and how much spacing we need between the tiles when they are laid out in a grid.

Creating the playfield header

In the header, we also have the declaration for the class method `initWithRows:andColumns:` that we called in the `MTPlayfieldScene` class.

Filename: `MTPlayfieldLayer.h`

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "MTMemoryTile.h"
#import "SimpleAudioEngine.h"
#import "MTMenuScene.h"

@interface MTPlayfieldLayer : CCLayer {
    CGSize size; // The window size from CCDirector

    CCSpriteBatchNode *memorysheet;

    CGSize tileSize; // Size (in points) of the tiles

    NSMutableArray *tilesAvailable;
    NSMutableArray *tilesInPlay;
    NSMutableArray *tilesSelected;

    CCSprite *backButton;

    NSInteger maxTiles;

    float boardWidth; // Max width of the game board
    float boardHeight; // Max height of the game board

    NSInteger boardRows; // Total rows in the grid
    NSInteger boardColumns; // Total columns in the grid

    NSInteger boardOffsetX; // Offset from the left
    NSInteger boardOffsetY; // Offset from the bottom
```

```
    NSInteger padWidth; // Space between tiles
    NSInteger padHeight; // Space between tiles

    NSInteger playerScore; // Current score value
    CCLabelTTF *playerScoreDisplay; // Score label

    NSInteger livesRemaining; // Lives value
    CCLabelTTF *livesRemainingDisplay; // Lives label
    BOOL isGameOver;
}

+(id) layerWithRows:(NSInteger)numRows
         andColumns:(NSInteger)numCols;

@end
```

One item to point out in the header is the `CGSize size` variable. This is a convenience variable we use to avoid repetitive typing. The name `size` is an abbreviation for `winSize`, which is a value that the `CCDirector` class will provide for you that identifies the size of the screen, in points. You can read the value from the `CCDirector` every time you use it, but doing so will make your code lines a bit longer. Our approach will work fine, as long as we do not support both portrait and landscape modes in the same layer. If we do allow both orientations, then the value we have cached in the `size` variable will be incorrect. Since our app only allows `LandscapeLeft` and `LandscapeRight` orientations, the size is identical in both orientations, so the `size` will be stable for our game.

Creating the playfield layer

In the `MTPlayfieldLayer.m` file, we implement our custom `layerWithRows:andColumns:` and `initWithRows:andColumns:` methods as follows:

Filename: `MTPlayfieldLayer.m`

```
+(id) layerWithRows:(NSInteger)numRows
         andColumns:(NSInteger)numCols {
  return [[[self alloc] initWithRows:numRows
                   andColumns:numCols] autorelease];
}

-(id) initWithRows:(NSInteger)numRows
        andColumns:(NSInteger)numCols {
```

```
if (self == [super init]) {

    self.isTouchEnabled = YES;

    // Get the window size from the CCDirector
    size = [[CCDirector sharedDirector] winSize];

    // Preload the sound effects
    [self preloadEffects];

    // make sure we've loaded the spritesheets
    [[CCSpriteFrameCache sharedSpriteFrameCache]
     addSpriteFramesWithFile:@"memorysheet.plist"];
    memorysheet = [CCSpriteBatchNode
            batchNodeWithFile:@"memorysheet.png"];

    // Add the batch node to the layer
    [self addChild:memorysheet];

    // Add the back Button to the bottom right corner
    backButton = [CCSprite spriteWithSpriteFrameName:
                  @"backbutton.png"];
    [backButton setAnchorPoint:ccp(1,0)];
    [backButton setPosition:ccp(size.width - 10, 10)];
    [memorysheet addChild:backButton];

    // Maximum size of the actual playing field
    boardWidth = 400;
    boardHeight = 320;

    // Set the board rows and columns
    boardRows = numRows;
    boardColumns = numCols;

    // If the total number of card positions is
    // not even, remove one row
    // This against an impossible board
    if ( (boardRows * boardColumns) % 2 ) {
        boardRows--;
    }

    // Set the number of images to choose from
    // We need 2 of each, so we halve the total tiles
    maxTiles = (boardRows * boardColumns) / 2;
```

```
            // Set up the padding between the tiles
            padWidth = 10;
            padHeight = 10;

            // We set the desired tile size
            float tileWidth = ((boardWidth -
                                (boardColumns * padWidth))
                               / boardColumns) - padWidth;
            float tileHeight = ((boardHeight -
                                 (boardRows * padHeight))
                                / boardRows) - padHeight;

            // We force the tiles to be square
            if (tileWidth > tileHeight) {
                tileWidth = tileHeight;
            } else {
                tileHeight = tileWidth;
            }

            // Store the tileSize so we can use it later
            tileSize = CGSizeMake(tileWidth, tileHeight);

            // Set the offset from the edge
            boardOffsetX = (boardWidth - ((tileSize.width +
                           padWidth) * boardColumns)) / 2;
            boardOffsetY = (boardHeight - ((tileSize.height+
                           padHeight) * boardRows)) / 2;

            // Set the score to zero
            playerScore = 0;

            // Initialize the arrays

            // Populate the tilesAvailable array
            [self acquireMemoryTiles];

            // Generate the actual playfield on-screen
            [self generateTileGrid];

            // Calculate the number of lives left
            [self calculateLivesRemaining];

            // We create the score and lives display here
            [self generateScoreAndLivesDisplay];
        }
    return self;
}
```

The class method `layerWithRows:andColumns:` is the method we saw in the `MTPlayfieldScene` class earlier. The class method calls the `alloc` and `initWithRows: andColumns:` methods, and then wraps it all in an `autorelease` call since it is a convenience method. The instance method `initWithRows:AndColumns:` (called by the class method) sets up a few of the variables we established in the header, including the assignment of our passed `numRows` and `numColumns` parameters into the instance variables `boardRows` and `boardColumns`.

Memory games are traditionally played with a square or rectangular layout. They also need an even number of tiles in the game, since there will be two of each type of tile. Because we are allowing flexible parameters for the number of rows and the number of columns, there are certain combinations that will not work. Requesting five rows and five columns means we will have 25 tiles on the board, which is impossible to win. To protect our game from these invalid values, we multiply the `boardRows` times the `boardColumns`. If the result is not even (using the % 2 check), then we remove one `boardRow` from the game. From the prior example, if we requested a five by five board (resulting in 25 tiles), the code would alter it to a four by five grid, which has 20 tiles.

We also set the `tileSize` value here, based on an even spacing of the tiles, along with the extra pad space we will be using between the tiles. Because we need square tiles, there is also the additional check to force the tiles to be square if the source images are not. This will distort the images, but it won't disrupt the game mechanics. Additionally, the `boardOffsetX` and `boardOffsetY` variables simply ensure the board will be nicely centered in the available board space.

The flow of the game

We will need several arrays in the game to help track the tiles. The first, `tilesAvailable`, will be used in the loading and building of the playfield. The second, `tilesInPlay`, will contain all of the tiles that have not yet been matched. The third, `tilesSelected`, will be used for the match detection methods. Since we are handling a relatively small number of tiles, using this multiple array structure will work fine for our purposes without any performance concerns. Let's add the code for the arrays now:

Filename: `MTPlayfieldLayer.h` (already in variable declarations)

```
NSMutableArray *tilesAvailable;
NSMutableArray *tilesInPlay;
NSMutableArray *tilesSelected;
```

Filename: `MTPlayfieldLayer.m` (`initWithRows`, add after "Initialize the arrays")

```
        tilesAvailable = [[NSMutableArray alloc]
                          initWithCapacity:maxTiles];
        tilesInPlay = [[NSMutableArray alloc]
                       initWithCapacity:maxTiles];
        tilesSelected = [[NSMutableArray alloc]
                         initWithCapacity:2]; MTPlayfieldLayer.m:
- (void) dealloc
{
    // Release of the arrays
    [tilesAvailable release];
    [tilesInPlay release];
    [tilesSelected release];

  [super dealloc];
 }
```

Here we established the three `NSMutableArray` arrays in the header as variables, instantiated them in the `initWithRows:andColumns:` method, and added them to a new `dealloc` method. The `dealloc` method releases the three arrays. The `[super dealloc]` call is always required, and it should be the last line of the `dealloc` method. This call to `super dealloc` tells the parent class of the current class to do whatever it needs to clean up. This is important to call because our current class doesn't have to worry about the details of any clean up that is done by the parent `CCLayer` class.

A stack of tiles

Now we need to define the class for the tiles themselves. We have a few variables we need to track for the tiles and we will use the `MTMemoryTile` class to handle some of the touch detection and tile animation.

The memory tile class

For this, we will be subclassing `CCSprite`. This will allow us to still treat it like a `CCSprite`, but we will enhance it with other methods and properties specific to the tile.

Filename: `MTMemoryTile.h`

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "SimpleAudioEngine.h"

// MemoryTile is a subclass of CCSprite
```

```
@interface MTMemoryTile : CCSprite {
    NSInteger _tileRow;
    NSInteger _tileColumn;

    NSString *_faceSpriteName;

    BOOL isFaceUp;
}

@property (nonatomic, assign) NSInteger tileRow;
@property (nonatomic, assign) NSInteger tileColumn;
@property (nonatomic, assign) BOOL isFaceUp;
@property (nonatomic, retain) NSString *faceSpriteName;

// Exposed methods to interact with the tile
-(void) showFace;
-(void) showBack;
-(void) flipTile;
-(BOOL) containsTouchLocation:(CGPoint)pos;

@end
```

Here we are declaring the variables with an underscore prefix, but we set the corresponding property without the underscore prefix. This is usually done to avoid accidentally setting the variable value directly, which would bypass the getter and setter methods for the property. This split-naming is finalized in the `@synthesize` statements in the `.m` file, where the property will be set to the variable. These statements will be of the basic format:

```
@synthesize propertyName = _variableName;
```

We're planning ahead with this class, including the headers for three methods that we will use for the tile animation: `flipTile`, `showFace`, and `showBack`. This class will be responsible for handling its own animation.

All animation in our game will be done using cocos2d actions. Actions are essentially transformations of some sort that can be "run" on most types of cocos2d objects (for example, `CCLayer`, `CCSprite`, and so on). There are quite a number of different actions defined in the framework. Some of the most commonly used are actions such as `CCMoveTo` (to move an object), `CCScaleTo` (to change the scale of the object), and `CCCallFunc` (to call another method). Actions are a "fire and forget" feature. Once you schedule an action, unless you explicitly change the action (such as calling `stopAllActions`), the actions will continue until complete. This is further extended by "wrapping" several actions together in a `CCSequence` action, which allows you to chain several actions together, to be run in the order specified.

We will use `CCSequence` "chaining" extensively throughout the book. Actions can be run on most cocos2d objects, but they are most commonly called (via the `runAction:` method) on the `CCSprite` and `CCLayer` objects.

Filename: `MTMemoryTile.m`

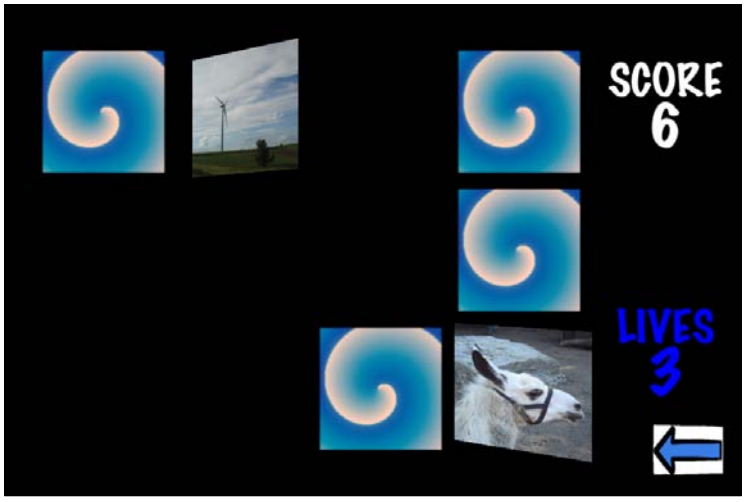
```
@implementation MTMemoryTile

@synthesize tileRow = _tileRow;
@synthesize tileColumn = _tileColumn;
@synthesize faceSpriteName = _faceSpriteName;
@synthesize isFaceUp;

-(void) dealloc {
    // We set this to nil to let the string go away
    self.faceSpriteName = nil;

    [super dealloc];
}

-(void) showFace {
    // Instantly swap the texture used for this tile
    // to the faceSpriteName
    [self setDisplayFrame:[[CCSpriteFrameCache
                            sharedSpriteFrameCache]
                spriteFrameByName:self.faceSpriteName]];

    self.isFaceUp = YES;
}

-(void) showBack {
    // Instantly swap the texture to the back image
    [self setDisplayFrame:[[CCSpriteFrameCache
                            sharedSpriteFrameCache]
                spriteFrameByName:@"tileback.png"]];

    self.isFaceUp = NO;
}

-(void) changeTile {
    // This is called in the middle of the flipTile
    // method to change the tile image while the tile is
    // "on edge", so the player doesn't see the switch
    if (isFaceUp) {
        [self showBack];
    } else {
        [self showFace];
```

```
    }
}

-(void) flipTile {
    // This method uses the CCOrbitCamera to spin the
    // view of this sprite so we simulate a tile flip

    // Duration is how long the total flip will last
    float duration = 0.25f;

    CCOrbitCamera *rotateToEdge = [CCOrbitCamera
                actionWithDuration:duration/2 radius:1
                deltaRadius:0 angleZ:0 deltaAngleZ:90
                angleX:0 deltaAngleX:0];
    CCOrbitCamera *rotateFlat = [CCOrbitCamera
                actionWithDuration:duration/2 radius:1
                deltaRadius:0 angleZ:270 deltaAngleZ:90
                angleX:0 deltaAngleX:0];
    [self runAction:[CCSequence actions: rotateToEdge,
                      [CCCallFunc actionWithTarget:self
                      selector:@selector(changeTile)],
                      rotateFlat, nil]];

    // Play the sound effect for flipping
    [[SimpleAudioEngine sharedEngine] playEffect:
                                    SND_TILE_FLIP];
}

- (BOOL)containsTouchLocation:(CGPoint)pos
{
    // This is called from the CCLayer to let the object
    // answer if it was touched or not
  return CGRectContainsPoint(self.boundingBox, pos);
}
@end
```

We will not be using a touch handler inside this class, since we will need to handle the matching logic in the main layer anyway. Instead, we expose the `containsTouchLocation` method, so the layer can "ask" the individual tiles if they were touched. This uses the tile's `boundingBox`, which is baked-in functionality in cocos2d. A `boundingBox` is a `CGRect` representing the smallest rectangle surrounding the sprite image itself.

We also see the `showFace` and `showBack` methods. These methods will set a new display frame for the tile. In order to retain the name of the sprite frame that is used for the face of this tile, we use the `faceSpriteName` variable to hold the sprite frame name (which is also the original image filename). We don't need to keep a variable for the tile back, since all tiles will be using the same image, so we can safely hardcode that name.

The `flipTile` method makes use of the `CCOrbitCamera` to deform the tile by rotating the "camera" around the sprite image. This is a bit of visual trickery and isn't a perfect flip (some extra deformation occurs nearer the edges of the screen), but it gives a fairly decent animation without a lot of heavy coding or prerendered animations. Here we use a `CCSequence` action to queue three actions. The first action, `rotateToEdge`, will rotate the tile on its axis until it is edge-wise to the screen. The second calls out to the `changeFace` method, which will do an instant swap between the front and back of the tile. The third action, `rotateFlat`, completes the rotation back to the original "flat" orientation. The same `flipTile` method can be used for flipping to the front and flipping to the back, because the `isFaceUp` Boolean being used allows the `changeTile` method to know whether front or back should be visible. Let's look at following screenshot, which shows the tile flips, in mid-flip:



Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output.

You can download this file from `http://www.packtpub.com/sites/default/files/downloads/9007OS_ColoredImages.pdf`