Mastering Cocos2d Game Development

Cocos2d is one of the world's leading 2D game development engines, letting game developers create awesome games with ease. Its multiplatform capabilities meet the demands of modern game development. If you want to dive into Cocos2d game development and create exciting, high-quality games, this book is an advanced and reliable resource. Covering the most important game development tools, and guiding you through the development cycle of a complete 2D game, Mastering Cocos2d Game Development will build your confidence and inspire you to go further in game development.

Combining practical steps with detailed insights, this book will help you discover and learn a powerful approach to game development to create exceptional iOS games, tips to create cool content, and how to harness the unique power of Swift.

Who this book is written for
If you are a developer who is experienced with Cocos2d and Objective-C, and want to take your game development skills to the next level, this book is going to help you achieve your goal.

What you will learn from this book
- Prototype a game quickly and easily
- Use the physics engine to apply forces and detect collisions
- Engage your user with the best sound effects a game can have
- Create interesting, engaging mechanics
- Polish your game and make it worth playing
- Explore the full functionality of Cocos2d through easy-to-follow examples and practical guidance designed to help you complete your own game development projects
- Make use of Apple's new Swift language
- Develop a high quality game for iOS
- Understand the Cocos2d-Swift architecture and its main classes
- Make use of Apple's new Swift language
- Develop a high quality game for iOS
- Understand the Cocos2d-Swift architecture and its main classes

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Focusing on Physics'
- A synopsis of the book’s content
- More information on Mastering Cocos2d Game Development

About the Author

Alex Ogorek is a 4-year veteran of iOS development, who has developed everything from traditional apps to games, with over 12 apps published on Apple’s App Store under the name of KeitGames (http://www.keitgames.com/). All of his apps and games are self-developed and published, including graphics. He has been coding since he was 16, and has loved games since he was young. Before attending USC for computer science (games) and entrepreneurship, he worked at the Cleveland Clinic Foundation as an iOS developer, working on medical iPad apps that help further research with concussions in students and athletes.
Mastering Cocos2d
Game Development

If you've ever wanted to know what goes on in a great game from start to finish, this book will guide you through that process. You will discover that your games are just a few steps away from becoming the best games possible.

You'll be guided through a project and see what makes a game stand out when hundreds, or even thousands, of similar games fail to make any lasting impact. You'll experience the simplicity of Cocos2d, Chipmunk, OALSimpleAudio, and much more. You'll be given code for some unique game mechanics that will make your game stand out from the masses. This book ensures that no matter what scale of game you're working on is—whether it is indie or for a large publisher—you'll get the skills you need to create a highly polished and enjoyable experience for your players.

What This Book Covers

Chapter 1, Refreshing Your Cocos2d Knowledge, describes some tools you can use when developing your Cocos2d game, how to create an Apple Developer account, and also the importance of setting flexible goals when designing your project.

Chapter 2, Failing Faster with Prototypes, talks about the need for fast iterations and getting things up and running quickly. This chapter sets up the initial outline, or wireframe, for the book's project and goes into detail about using some of the tools introduced in the first chapter.

Chapter 3, Focusing on Physics, introduces the Chipmunk physics engine and the power behind it with only a few lines of code. This chapter shows you how you can integrate touch controls, physics, and the accelerometer all within a single project with ease.

Chapter 4, Sound and Music, covers the addition of sound and music, which are integral to a gaming experience. This chapter explains how to best use OALSimpleAudio and the various ways you can use sound within your game.

Chapter 5, Creating Cool Content, shows the use of particle effects, Bézier curves, and parallax scrolling. This chapter has three mini projects as well as some additions to this book's main project.

Chapter 6, Tidying Up and Polishing, explains the important process of polishing your game. There are many topics in this chapter, ranging from button animations to graphical changes. It explains in depth what you can do to make your game progress from a concept to a finished product.
Chapter 7, *Reaching Our Destination*, finalizes the book’s project and prepares everything for launch, including the creation of the app on iTunes Connect, setting up the loading screen and app icon, and adding analytics. This chapter also covers how to actually submit the game to the App Store and what to do after you submit it.

Chapter 8, *Exploring Swift*, introduces the Swift programming language (and the use of Swift with Cocos2d) through the creation of a very simple game. This chapter also explains the basics of how Swift works and the main differences between Swift and Objective-C.

Chapter 9, *Simple Swift App*, covers the basics of creating a nongame app using Swift as the main language. Although this chapter is not available in the book, it is available for download at https://www.packtpub.com/sites/default/files/downloads/6718OS_Chapter9.pdf.
This chapter is for those of you who want to incorporate physics into your games. Whether you’re building a mini golf game that uses realistic wall bounces or a platformer with endless gravity, this chapter is for you. It will have tutorials on the physics side of games, as well as show you how to use a physics engine without gravity.

In this chapter, we will cover the following topics:

- How Chipmunk works
- Setting up a project and creating basic objects
- Setting gravity by tilting the device
- Handling collisions in Chipmunk

You must use Chipmunk for collision detection only (and not physics). Not all games need (or should even consider using) a physics engine. Sometimes, it’s better to leave it out. However, if you feel your game will either be more polished or be produced faster, then by all means use it. That being said, the project in this book will not need a physics engine. So, instead of following the project here, we will create a small project that has many modular examples that can be adapted in your other projects. The book’s main project will continue in the next chapter.

You may have been used to using the Box2D physics engine, but ever since version 3.0 of Cocos2d, there is no longer any support from the developers to make Box2D work out of the box as it did in previous versions. That being said, this chapter will focus on Chipmunk. If this chapter does not cover everything you need as far as physics is concerned, feel free to check out the documentation at http://chipmunk-physics.net/documentation.php. It also has a variety of online tutorials.
Learn how Chipmunk works

As previously mentioned, Chipmunk is the physics engine that's integrated with Cocos2d, and is the main physics engine from version 3.0 onwards. The good news for both newcomers and fans of Box2D alike is that Chipmunk is very simple to use. Let's get down to how Chipmunk basically works.

Overall structure of Chipmunk

Chipmunk is a physics engine in Cocos2d that simulates real-world physics, that is, making use of gravity, collisions, objects bouncing off each other, and so on.

Chipmunk uses a "bodies within a world" way of doing things. This means that, as shown in the following diagram, there's a physics simulation happening (called a world), and anything with physics applicable to it is a body. You just create a world that will simulate physics on the bodies within, and off you go. Each world you create will have its own gravity attached to it.

This is a simple representation of bodies within a world. Note that the entire green rectangle is the world, and the individual squares are bodies within it.
Each body has a type (explained in the next section) as well as properties such as density, mass, friction, elasticity, velocity, and more. In Cocos2d, you can attach a physics body to a sprite with a single line of code, and the sprite will move around to the place where the body is located.

When these objects have their boundaries touching/intersecting one another, it's a collision. When a collision occurs, you can handle it however you wish.

**Types of bodies**
Chipmunk has three types of physics bodies that can be added to the world. They are static, dynamic, and kinematic:

- **Static bodies**: These are the walls, ground, immovable rocks, and other objects in the game. They will not be affected by any gravity or other forces that try to interact with them.
- **Dynamic bodies**: These are the default when creating a `CCPhysicsBody` object. These are the objects that will go flying around, colliding with other objects, and have forces act upon them.
- **Kinematic bodies**: These are a sort of hybrid body type that cannot be influenced by forces or gravity, but can still be moved by `CCActions` and other methods.

In general, you will only be using static and dynamic bodies (and so is the case with this chapter). If you feel that you need more help with kinematic body types, check out the Cocos2d documentation about physics bodies at [http://www.cocos2d-swift.org/docs/api/Constants/CCPhysicsBodyType.html](http://www.cocos2d-swift.org/docs/api/Constants/CCPhysicsBodyType.html).

Now that you know how Chipmunk works from a technical perspective, let's actually get down to the coding so that we can see these physics bodies for ourselves.

**Setting up a project and creating basic objects**
The Chipmunk physics engine is fairly well integrated within the Cocos2d library. It works within SpriteBuilder as well as programatically within Cocos2d. As with the rest of this book, we will be focusing on creating the project with code, and only using SpriteBuilder as the tool for project creation. So, create a new project in SpriteBuilder and publish it. If you've forgotten how to do this, feel free to go back and reference the project creation in the first chapter.
Setting up Cocos2d for use with physics

First, we need to lay down the groundwork so that we can begin to code with it, as the published SpriteBuilder project uses the SpriteBuilder files and not the actual coded scenes. Similar to the previous chapter, open AppDelegate.m and add the following line of code to the top of the file:

```
#import "MainScene.h"
```

Then, in the `startScene` method of the AppDelegate.m file, replace that one line of code that already exists with the following:

```
return [MainScene scene];
```

With that in place, open MainScene.h and add code such that your file looks something like this (make sure you change the `CCNode` inheritance to `CCScene`, or else some later methods won’t work):

```
@interface MainScene : CCScene {
    CGSize winSize;
}
+(CCScene*)scene;
@end
```

Finally, open MainScene.m and add this block of code between the `@implementation` and `@end` lines:

```
+(CCScene*)scene
{
    return [[self alloc] init];
}

-(id)init
{
    if ((self=[super init]))
    {
        winSize = [[CCDirector sharedDirector] viewSize];

        //these values range 0 to 1.0, so use float to get ratio
        CCNode *background = [CCNodeColor nodeWithColor:[CCColor colorWithRed:58/255.f green:138/255.f blue:88/255.f]];
        [self addChild:background];
    }
    return self;
}
```
If you run the project at this point (in any simulator or on any device), you'll see a full, green screen. If you don't, go back and make sure you copy all of the code as instructed. If you do see the green color, then you're set to move on to the next section.

**Building a world for physics to exist**

Cocos2d is simply a graphics engine, and we need to create a Chipmunk physics simulation environment so that we can make use of the library. This sounds a lot more complicated than it actually is. Basically, we create a `CCPhysicsNode` object, and then add our sprites and nodes to this object, instead of `self`.

Remember, `self` is a reference to the current object. In the past, we've added objects to the screen using `[self addChild:]`, but with the `CCPhysicsNode` object, we're going to add objects using `[world addChild:]`, since `world` will be the name of our `CCPhysicsNode` object.

So, open `MainScene.h` and add the declaration for the `world` variable below the `winSize` declaration:

```c
CCPhysicsNode *world;
```

Then open `MainScene.m` and add this block of code after the background is created in the `init` method:

```c
//create the physics simulation world
world = [CCPhysicsNode node];
world.debugDraw = YES;
world.gravity = ccp(0, -300);
[self addChild:world];
```

And that's it! The preceding code will create a physics world that allows physics to be simulated. All we need to do next is create some `CCNode` objects, add physics bodies to the `CCNode` objects, then add the `CCNode` objects to the physics world (instead of `self`).

**Downloading the example code**

You can download the example code files from your account at [http://www.packtpub.com](http://www.packtpub.com) for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit [http://www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files e-mailed directly to you.
Setting `debugDraw` to `YES` ensures that whenever we attach any physics body to a `CCNode` object and add it to the world, we will have an outline of that body's shape drawn over our objects. Thus, we will know exactly where each body is and how it's colliding. Normally, when `debugDraw` is not turned on, we won't see these shapes. The collision will still work the same whether you have `debugDraw` on or not. The main purpose is to, well, debug the project to make sure the physics bodies are added and colliding as intended.

**Enabling touch creation of our object**

Now we are going to need some objects on the screen that we want to collide with one another. Instead of programmatically spawning the objects, let's have the objects spawn wherever the user touches the screen.

So, open `MainScene.m` and add the following line of code anywhere in the `init` method. This will allow you to grab any touch data from the user:

```objective-c
[self setUserInteractionEnabled:YES];
```

Then, below the `init` method, we add this method so that we can start receiving touches in our scene. This method (as seen in the previous chapter) will grab any touch event, and we want to grab the location of the touch so that we can position the object accordingly. Then we will spawn a black square at the touch position:

```objective-c
-(void)touchBegan:(CCTouch *)touch withEvent:(CCTouchEvent *)event
{
    CGPoint touchPos = [touch locationInNode:self];
    // create a black square
    CGFloat width = winSize.width * 0.1f;
    CCNode *square = [CCNodeColor nodeWithColor:[CCColor colorWithRed:0
green:0 blue:0] width:width height:width];
    square.position = touchPos;
    square.anchorPoint = ccp(0.5f,0.5f);
    [world addChild:square];
}
```

This will create a node that's 10 percent of the screen's width and add it to the `world` object. We're adding it to `world` instead of `self` because when we add physics bodies to the square, we want the simulation to handle the object's movement. We need to set the `anchorPoint` of the node because, by default, a `CCNode` has its anchor point at (0,0).
If you run the game at this point, you will see the same green screen. As you tap around the screen, black squares will appear, centered at the touch location.

However, you'll notice that even though we've added them to the world object, they aren't falling according to the gravity we set up, and they aren't colliding and pushing each other around. What we have to do next is create a `CCPhysicsBody` object and assign it to the created square so that the physics simulation will handle the movement properly.

If you're still a bit unsure what exactly `CCPhysicsBody` is, it's essentially an object that has a shape, or perimeter, that defines where the object can collide with other objects. `CCPhysicsBody` also has other properties such as elasticity, restitution, density, and so on, and is best used when attached to a `CCNode` object so that the node can have its movement handled by the physics body.

**Making the objects fall – adding `CCPhysicsBody`**

In the `MainScene.m` file, add the following code to the bottom of the `touchBegan` method. This will add the physics body we want to the square:

```objective-c
//add a physics body to the black square
CCPhysicsBody *squareBody = [CCPhysicsBody bodyWithRect:CGRectMake(0, 0, width, width) cornerRadius:0];
squareBody.elasticity = 0.5f;
square.physicsBody = squareBody;
```

With this code in place, when you run the game, you will notice that the objects not only begin to fall but also have that debug square that was mentioned earlier. Notice that we don't have to declare the body type as `dynamic`, as that's the default. Also note the `elasticity` (which, as you might recall, is the bounciness) of 0.5. This is set so that the squares don't go bouncing all over the place. If we wanted that, we would've set the elasticity to a higher number, such as 1.0.

But oh no! The squares just fall off the screen as if it's a bottomless pit. Let's fix that.
Adding the ground and walls

Creating a ground object is easy, but since the walls and the ceiling are basically the same thing, let's define a method that will take in CGRect and create for us a static, invisible node with those dimensions.

So, anywhere in MainScene.m, add the following method:

```objective-c
-(void)addWallWithRect:(CGRect)rect
{
    CCPhysicsBody *wallBody = [CCPhysicsBody bodyWithRect:rect
        cornerRadius:0];
    wallBody.type = CCPhysicsBodyTypeStatic;
    wallBody.elasticity = .5f;

    CCNode *wall = [CCNode node];
    wall.physicsBody = wallBody;
    [world addChild:wall];
}
```

This code is fairly similar to the square's code, except for one notable difference: the type of the physics body. We set it to static because we don't want the gravity affecting it, nor do we want the squares to push the ground in any direction.

Just because we declared a method doesn't mean it will automatically generate walls for us. So, go to your init method and add the following block of code to add the walls around the edges of the device:

```objective-c
/***** Add Ground, Walls, and Ceiling *****/

//ground
[self addWallWithRect:CGRectMake(0, 0, winSize.width, 1)];

//left wall
[self addWallWithRect:CGRectMake(0, 0, 1, winSize.height)];

//right wall
[self addWallWithRect:CGRectMake(winSize.width, 0, 1, winSize.height)];

//ceiling
[self addWallWithRect:CGRectMake(0, winSize.height, winSize.width, 1)];
```
This code is fairly self-explanatory in terms of positioning. The bottom-left corner of every Cocos2d scene is (0,0), and the `CGRectMake` function takes in the x and y coordinates, and then the width and height of the rectangle.

That's all for creating a simple object with some physics acting on it! Feel free to mess around with some of the variables such as elasticity, gravity, and so on.

At this point, if you notice that your objects seem to be lagging when they move across the screen—even though Xcode says the game is running at 60 FPS—add the following line of code at the top of the `startScene` method in `AppDelegate.m`:

```objc
[CCDirector sharedDirector].fixedUpdateInterval = 1.0f/120.0f;
```

### Setting gravity by tilting the device

When your user tilts the device, the device’s accelerometer will pick up that information at even the slightest, most minute values. This is beneficial to those who wish to use the accelerometer (or tilting) within their game, especially with regards to moving the character or manipulating gravity, which is what we’ll cover in this section.

Ever since iOS 5.0, UIAccelerometer has been replaced by the Core Motion framework data. Thankfully, it’s no more complicated, so let’s get started.

If you aren’t a registered developer and can’t use your actual device to test (and have been doing so on the simulator), note that it’s not possible to test the accelerometer with the simulator unless the data can be sent to the simulator. Here’s a link that might be useful in this situation: [http://www.vimov.com/isimulate/](http://www.vimov.com/isimulate/).

However, even if you can’t test the accelerometer on the simulator by default, it will still work as intended on anyone’s device, so if you plan to include the accelerometer in your game, feel free to still follow along.
Setting up the accelerometer

Open MainScene.h and import the Core Motion framework:

```objective-c
#import <CoreMotion/CoreMotion.h>
```

Then add a variable for something called the `CMMotionManager`. This object will calculate the accelerometer data, and we can collect this data when we need it. Add the following line of code under the `world` variable:

```objective-c
CMMotionManager *motionManager;
```

Now open MainScene.m, and anywhere in the `init` method, add the following block of code so that the motion manager we just declared will begin to grab the accelerometer's data:

```objective-c
// 60 times per second, in theory once per frame
CGFloat interval = 1/60.f;
motionManager = [[CMMotionManager alloc] init];
motionManager.accelerometerUpdateInterval = interval;
[motionManager startAccelerometerUpdates];
```

Right now, if you run the game, you won't notice any changes in the way it plays or any debug information. Even though the motion manager is initialized and is grabbing the data, we must set up a method to collect the data so that we can do something with it.

Reading the data

In your `init` method of MainScene.m, after you've initialized the motion manager, add the following line of code. It will run the specified method at the given interval (60 times per second):

```objective-c
[self schedule:@selector(getAccelerometerData:) interval:interval];
```

Then, anywhere in MainScene.m, we add the method we want to be called at the specified interval so that we can read the accelerometer data:

```objective-c
- (void)getAccelerometerData:(CCTime)delta
{
    NSLog(@"%f	%f	%f",
```
motionManager.accelerometerData.acceleration.x, motionManager.accelerometerData.acceleration.y, motionManager.accelerometerData.acceleration.z);

Now, if you run the game, you'll see a lot of information being printed in the console output in Xcode. If the numbers are changing as you tilt the device, it means everything's working. Yay! If not, go back and make sure you add everything correctly.

Next, we're going to actually set the game's gravity based on how the device is rotated at any given moment.

**Manipulating gravity to your heart's content**

Instead of typing that long line of text every time, let's make a method with a relevant name and pass the accelerometer data. That being said, modify your `getAccelerometerData` method to the following, and add this new method, which will set the gravity of the physics world based on the accelerometer's data:

```objective-c
-(void)getAccelerometerData:(CCTime)delta
{
    //NSLog(@"%f%f%f", motionManager.accelerometerData.acceleration.x, motionManager.accelerometerData.acceleration.y, motionManager.accelerometerData.acceleration.z);
    [self setGravityFromAcceleration: motionManager.accelerometerData.acceleration];
}

-(void)setGravityFromAcceleration:(CMAcceleration)accel
{
    CGFloat xGravity = 500 * accel.y;
    CGFloat yGravity = -500 * accel.x;
    world.gravity = ccp(xGravity, yGravity);
}
```
Now, if you run the game and spawn a few blocks, you’ll see them floating, falling down, sliding sideways, or in whichever direction your iOS device perceives as "down." You’ll probably see something similar to the following screenshot:

![Screenshot of blocks falling and sliding](image)

From here onwards, feel free to turn off debug draw, maybe mess around with how much gravity is set, or even change the axis the gravity is affected by.

You might notice that a square stops moving on a wall, then doesn’t move when you rotate the device; that’s because the object’s body is sleeping. This is done to help conserve CPU processing time as well as energy when an object has no active collisions and no forces acting on it.

However, if you need them to be constantly moving, add the following line of code to your init method:

```swift
world.sleepTimeThreshold = 100000; //100,000 seconds, or about 27 hours
```

This defaults to 0.5 seconds, and why the sleeping happens when the squares stop moving against a wall. Setting it to a relatively large value, such as 100,000, will ensure that they never stop moving (that is, unless the square is sitting still for that duration, but the higher the threshold is, the less likely it is to happen).
Handling collisions in Chipmunk

It's relatively easy to handle collisions within Cocos2d when using Chipmunk (as is pretty much everything else in Cocos2d). That is why so many developers use Cocos2d. To do so, we need to do a few different steps so that Cocos2d can properly detect and handle our collisions.

Setting the collision delegate

The delegate pattern is a common way to handle messages sent by a class to any of its potential parent classes. For example, if you want to use UITableView (which is the standard table in UIKit), you must set the delegate of the table view to the class you're adding it to so that when the table view tries to refresh the data in the table, it knows which class' methods to call.

That being said, we need to tell MainScene that it will be a delegate for our collision handler, so open up MainScene.h and add CCPPhysicsCollisionDelegate to the @interface line. This will allow the CCPPhysicsNode object to set collision detection events on our MainScene class:

```objective-c
@interface MainScene : CCScene <CCPhysicsCollisionDelegate>
```

Then open MainScene.m, and in the init method, set the world object's collision delegate to self, like this:

```objective-c
world.collisionDelegate = self;
```

Recall that world is our CCPPhysicsNode object, so any collision detection events that happen within that world (or simulation) will need to be sent somewhere to be handled further. We're setting it to self because self refers to the current MainScene instance (our currently running scene). Finally, this helps us determine which objects collide with other objects. Without this line of code, we wouldn't be able to see in the code when two objects collide, let alone tell which objects they actually are.

Although this doesn't do anything directly if you run the game now, it properly sets up your physics simulation to be able to detect and handle collisions.

Setting collision tags on game objects

Before we can create the method that will detect our collisions, we need to specify which objects will be colliding with one another. For now, we'll only be considering squares colliding with other squares.
Therefore, in the `touchBegan` method of the `MainScene.m` file, add the following line of code so that the collision detection delegate knows what object is colliding:

```objective-c
squareBody.collisionType = @"square";
```

You can do this with any physics body, but for now, this is our only object. With that in place, let's add the code to detect the actual collision between two squares.

### Detecting collision

The way Chipmunk handles collisions in Cocos2d is by detecting all collisions and sending method calls to the respective functions for each collision type. So, since we're going to be detecting the collision between two squares, both parameters have to be named `square`. Otherwise, the method won't get called properly. The actual names of the variables of the parameters (`firstSquare` and `secondSquare`) don't matter for the sake of detection.

So, anywhere in `MainScene.m`, add the following method:

```objective-c
-(BOOL)ccPhysicsCollisionBegin:(CCPhysicsCollisionPair *)pair
square:(CCNode *)firstSquare square:(CCNode *)secondSquare
{
    NSLog(@"squares collided!");
    return YES;
}
```

If you run the game at this point, you should see the `squares collided!` text being printed to the console output every time a square collides with another. If it doesn't, go back and make sure you add everything correctly. From here onwards, you can do whatever you want when the two objects collide, as you have a pointer to both objects as well as their types.

As another example, suppose we want to detect collisions between the squares and a wall; it's very easy. First, add `wall` as `collisionType` to the body in the `addWallWithRect` method:

```objective-c
wallBody.collisionType = @"wall";
```

Then, add the method to detect square-wall collisions (notice the name change in the parameter from the previous `collisionBegan` method we added):

```objective-c
-(BOOL)ccPhysicsCollisionBegin:(CCPhysicsCollisionPair *)pair
square:(CCNode *)nodeA wall:(CCNode *)nodeB
{
    NSLog(@"square-wall collision!");
    return YES;
}
```
If you run the game at this point and spawn a square, as soon as it hits a wall, you should see the output being printed. And that's it for detecting collisions! Just set the collisionDelegate property, set the collisionType property, and add the collision methods.

What if you want to detect collisions on objects, but don't want them to be bouncing with other objects? In Chipmunk, you can do that.

**Using Chipmunk for just collision detection**

Here is an important note: if you're going to just detect one of the following, you do not need to use a physics engine for your collision detection, but rather the listed method:

- For rectangles intersecting rectangles, use CGRectIntersectsRect
- For points inside rectangles, use CGRectContainsPoint
- For radius/distance intersection, use ccpDistance

If you're going to be detecting collisions with non-rectangular and noncircular objects, feel free to read on.

Sometimes, all you're doing is sending objects across the screen, or rotating them with your custom actions, and you just want to know when two objects collide, but don't want the bouncing, pushing, and colliding that comes with the entire physics engine. Luckily, we can do that in Chipmunk.

**Turning the physics body into a sensor**

Sensors, with respect to Chipmunk, are basically bodies that can detect collisions, but pass right through other bodies. These are helpful when you have areas or sections of the screen that trigger certain events, but don't actually cause any physics-based interactions.

For example, if you're making a top-down mini golf game with a physics engine and you want to include ramps, the best way to do this is to set the ramp as a sensor. When the ball and ramp collide, set the gravity in a certain direction.
To make a body a sensor, simply set the sensor variable to true, like this:

```swift
[squareBody setSensor:YES];
```

If you run the game now, you'll notice that the blocks fall right through one another, as well as right through the floor.

And that's it! All you need is the collision handling (in the previous section), and the sensor variable set to true. Any physics body that's a sensor will trigger collision detection events, but not cause any movement or changes to other bodies.

**Summary**

Throughout this chapter, you learned how to create a physics simulation, add bodies to the simulation, set the gravity of the world with the accelerometer, handle collisions, and even use the simulation for just collision detection.

When it comes to really complex physics engine mechanics, such as ropes, joints, and pivots, the Cocos2d documentation at [http://www.cocos2d-swift.org/docs/api/index.html](http://www.cocos2d-swift.org/docs/api/index.html) explains a lot. At the time of writing this book, there are very few (if any) tutorials on such complex mechanics available.

In the next chapter, you'll learn about sound in Cocos2d and some of the cool things you can do to sound effects within the engine.
Where to buy this book

You can buy Mastering Cocos2d Game Development from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.