

# BEAT 'EM UP GAME STARTER KIT

By Allen Tan

# Beat 'Em Up Game Starter Kit

Allen Tan

Copyright © 2012 Razeware LLC.

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

This book and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

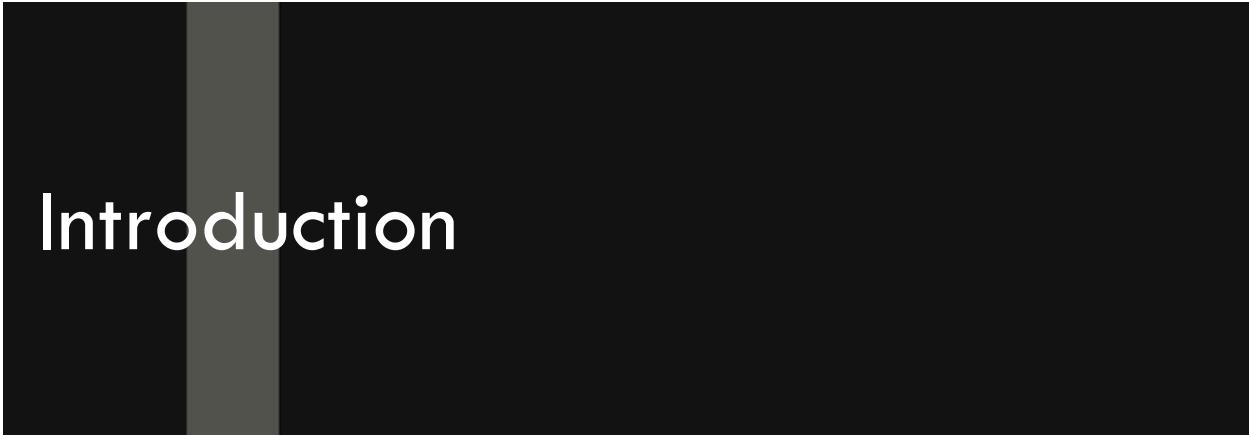
All trademarks and registered trademarks appearing in this book are the property of their respective owners.

# Table of Contents

Introduction .....	5
Chapter 1: Getting Started.....	13
Chapter 2: Walk This Way .....	49
Chapter 3: Running, Jumping, and Punching .....	84
Chapter 4: Bring On the Droids.....	129
Chapter 5: Brainy Bots .....	163
Chapter 6: Power Attacks.....	203
Chapter 7: The Droids Strike Back.....	241
Chapter 8: Final Encounter.....	281

# Dedication

To my dear departed brother, Andy.



# Introduction

There's nothing better than beating up a horde of bad guys with your trusty right and left hook.

Beat 'Em Up games have been with us since the inception of 8-bit games, and have had their share of the limelight back in the glory days. Lately however, they aren't as common, but I believe that should change! You can't deny the fact that Beat 'Em Ups are great fun.

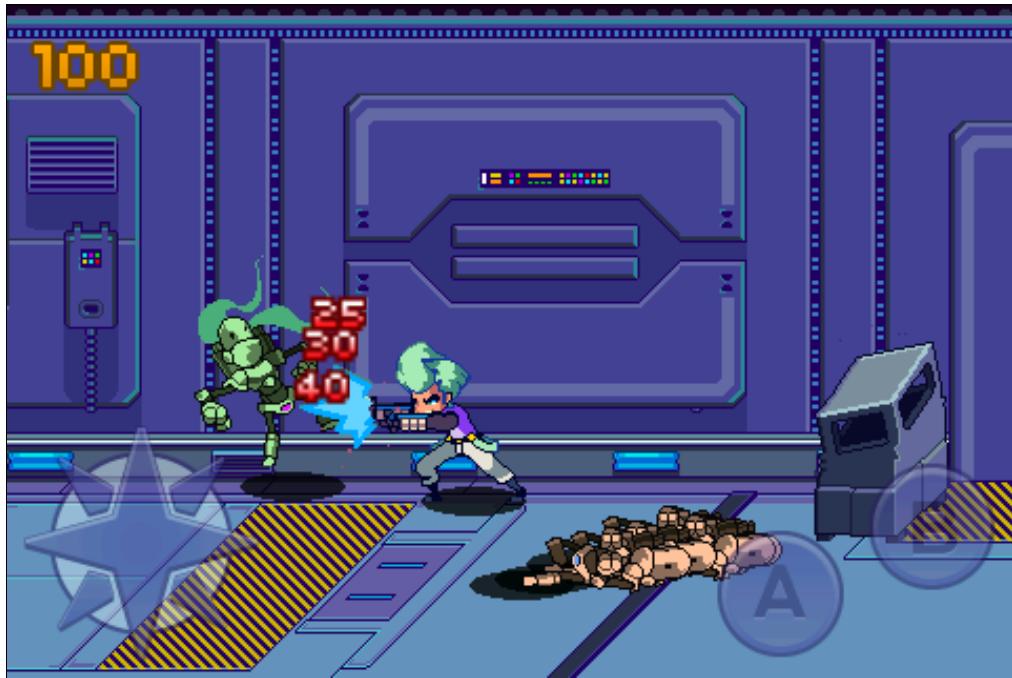
On the one hand, you have the classic Beat 'Em Ups, which have stood the test of time. Games such as Double Dragon, Teenage Mutant Ninja Turtles, Golden Axe, and Streets of Rage are still enjoyable with a group of friends.

On the other hand, modern Beat 'Em Ups appeal to the player's nostalgia while employing modern technologies in games. Games such as Castle Crashers, and Scott Pilgrim vs. the World are good examples.

If you have this kit in hand and are reading this, it means you're no stranger to the genre, and you've probably heard of most, if not all of these games - that's why you want to make your own Beat 'Em Up game!

This is where the Beat 'Em Up Game Starter Kit comes in. This starter kit equips you with the tools and skills you need to create a well polished and fun Beat 'Em Up game for the iPhone and iPad using Cocos2D.

They say a picture is worth a thousand words, so here's a sneak peek of what you will be capable of doing by the end of this book:



In just one picture, you can see a virtual D-pad, virtual buttons, animated pixel-art sprites, procedurally recolored sprites, isometric movement, scrolling tile map, combo attacks, destructible objects, weapons, hit effects, damage numbers, and collision detection.

The list goes on.

The chapters and source codes in this book were carefully crafted so that you'll learn all of these, and more, at a steady, logical, and more importantly, fun pace. Most of the things you will create were designed with reusability in mind, and this will allow you to easily customize and use them in your own game.

I sincerely hope that with the help of this starter kit, you will be able to make an awesome Beat 'Em Up game of your own.

Growing up during the golden age of Beat 'Em Ups, I've always been eager to see new games of this genre. As such, let me take this opportunity to express an advanced "thank you" to you, the reader, for keeping the Beat 'Em Up spirit alive, and for your support of raywenderlich.com.

## Who this starter kit is for

This Starter Kit is for intermediate or advanced iOS developers, who already know the basics of both Objective-C and Cocos2D, but need guidance on applying their skills and knowledge to creating their very own Beat 'Em Up game.

If you're a complete beginner, it is still possible to follow along with this book, because the chapters walk you through in a step-by-step manner. However, to get

the most of this book, I recommend going through and completing the items listed in the Prerequisites section first.

## Prerequisites

To use this starter kit, you need to have a Mac with Xcode installed. It's also helpful (but not absolutely necessary) to have an iPhone or iPod touch to test your code on.

This starter kit assumes you have some basic familiarity with Objective-C. If you are new to Objective-C, I recommend you read the book *Programming in Objective-C 2.0* by Stephen Kochan. Alternatively, we have a free Intro to Objective-C tutorial on raywenderlich.com.

This starter kit also assumes you have some basic familiarity with Cocos2D, which is the framework you'll be using to make the game. If you are new to Cocos2D, I recommend you go through our free How To Make A Simple iPhone Game with Cocos2D 2.X tutorial series available here:

<http://www.raywenderlich.com/25736/how-to-make-a-simple-iphone-game-with-cocos2d-2-x-tutorial>

After going through that tutorial, a good next step is to go through the Space Game Starter Kit if you have it. Between the introductory tutorial and the Space Game Starter Kit, you should have plenty of knowledge to follow along with this tutorial.

## How to use this starter kit

You have several ways you can make use of the Beat 'Em Up Game Starter Kit.

First, you can just look through the sample project and start using it right away. You can modify it to make your own game, or pull out snippets of code you might find useful for your own project.

As you look through the code, you can just flip through the chapters and read up on any sections of code that you're not sure how they work. The beginning of this guide has table of contents that can help with that, and the search tool is your friend!

A second way of using the Beat 'Em Up Game Starter Kit is you can go through these chapters one by one and build up the Beat 'Em Up Game from scratch. It's the best way to learn because you'll literally write each line of the main gameplay code in the game, one small piece at a time.

Note you don't necessarily have to do each tutorial – if you already know how to do everything in Tutorial 1, you can skip straight to Tutorial 2 for example. The Beat

'Em Up Game Starter Kit includes a version of the project where it leaves off after each previous tutorial that you can pick up from.

## Starter kit overview

In this book, you'll learn how to create a fun Beat 'Em Up game from scratch! Specifically, here's what you'll be learning in each chapter:

### 1 - Getting Started

Approach your game the right way. In this first chapter, you learn how to create an ARC-enabled Cocos2D project, and create useful macros to help you code effectively.

By creating the title page, you get your first taste of using pixel art in your game, and through the use of macros, learn how to support multiple devices using the same set of art.

Then, you will set the stage for succeeding chapters by using Cocos2D transitions and tile maps.

### 2 - Walk This Way

This chapter is all about movement - specifically, movement of a character across the tile map. To achieve this, you will first create AnimatedSprite – the base model of all characters in the game. AnimatedSprite is no ordinary sprite. It can move, act, and animate itself in the process through the use of a state machine. You will use this as basis to create your very first character, the pompadour hero himself.

The hero won't be able to move by himself, so you will have to assist him by creating your very own animated 8-directional virtual D-pad to control his movement. Throw in delegation, protocols, and camera scrolling in the mix and you get what is probably the most important chapter in this book.

### 3 - Running, Jumping, and Punching

This chapter expands further on the previous one by adding more actions for the hero. You start off by learning to simulate a simple directional double-tap gesture to make the hero run.

Building on your virtual control scheme, you will create and add two custom buttons to make the hero perform additional actions such as jabbing. You then learn to add a third dimension to the game by adding shadows and jumping, and also start building an army of robots to add some much needed drama.

These robots are the most interesting part of the chapter. Unlike the hero, instead of having one whole image to represent a robot, you will assemble it piece by piece. With this, you will learn how to animate a complex sprite by creating grouped

animations and actions. Finally, to spice things up, you will learn about the sprite tinting technique and apply it to get a seemingly unlimited amount of color combinations for the robots.

## 4 - Bring On The Droids

Your game can't be considered a Beat 'Em Up until there are actual fights happening. On the way to implementing these fights, you will learn to create your own collision system by marking various areas of your ActionSprites with circles. To aid you in your task, you will also learn how to programmatically show these collision shapes through debug drawing.

With the circles in place, you will get your first taste of collision handling by letting the hero beat up some helpless robots until they die. Poor, poor robots.

## 5 - Brainy Bots

It's the revenge of the fallen! In this chapter, you will work on artificial intelligence for the robots, using a system of weighted decisions that determine each robot's action. This time, the robots won't just stand idly by as they get pummeled by the hero, and will aggressively fight back.

To control the chaos that will ensue, you will be adding game events that switch between scripted events, battle events, and free roaming events. You will also learn to control and define enemy spawning through the use of property lists. This technique paves the way to adding more levels in the next chapter.

## 6 - Power Attacks

Power-up your hero with additional moves. In this chapter, you will dig deep and learn to extend the current actions system by giving the hero the ability to do a chained attack with the 1-2-3 punch combo, and be able to perform combination actions such as the jump punch, and a running kick.

Afterwards, you expand on your usage of property lists by adding support for multiple levels. Lastly, you will learn a very important lesson about ARC and memory management by experiencing memory issues first hand. You should make sure not to miss this important section!

## 7 - The Droids Strike Back

You will begin the chapter by creating HUD indicators for the player to show him his current hit points, and where to go. Then, you will enhance the overall polish of the game by adding floating damage numbers and explosive hit animations to each attack.

Finally, to balance the hero's newfound abilities from the previous chapter, you will create stronger robots, and also introduce the hero's most badass enemy yet – the ape-like Mohawk Leader.

## 8 - Final Encounter

It's the last stretch! In this chapter, you will complete the game by adding the final pieces of content. You will re-balance the playing field by creating power gauntlets that the hero can equip, and then get to stash these weapons inside destructible trashcans located across the map.

To bring the game closer to a production-ready level, you will add a scripted ending event, and top everything off with gratuitous music and sound effects. By the time you're done, you'll have a full and polished Beat 'Em Up Game!

## Source code and forums

This Starter Kit comes with the source code for each of the chapters – it's shipped with the PDF. Some of the chapters have starter projects or required resources, so you'll definitely want to have them on hand as you go through the Starter Kit.

We've also set up an official forum for the Starter Kit here:

- <http://www.raywenderlich.com/forums>

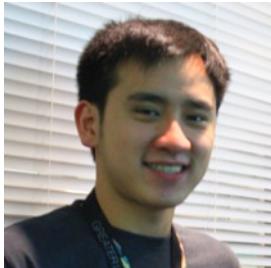
This is a great place to ask any questions you have about the book or making Beat 'Em Up games in general, and to submit any errata you may find. You can also find the latest download link there.

## Acknowledgements

I would like to thank several people for their assistance making this Starter Kit possible:

- **To Ray and the tutorial team:** Thanks for the great tutorials that kick started my iOS development in the early days, and for giving me the opportunity to write this starter kit.
- **To my family, and friends:** Thanks for the support, and for letting me work on this undisturbed.
- **To Grace:** Thanks for believing in me, and for the moral support and patience during those busy and work-focused days.

## About the author

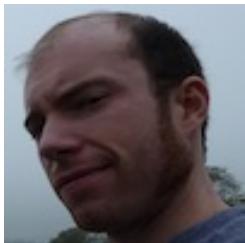


**Allen Tan** is a developer and co-founder of White Widget, an indie game and app development studio. An avid gamer and technology enthusiast, Allen is bent on showing the world how awesome Philippine game development can be, one game at a time.

## About the editors



**Fahim Farook** is a developer with over two decades of experience in developing in over a dozen different languages. Fahim's current focus is on the mobile app space with over 60 apps developed for iOS (iPhone and iPad). He's the CTO of [RookSoft Pte Ltd](#) of Singapore. Fahim has lived in Sri Lanka, USA, Saudi Arabia, New Zealand, and Singapore and enjoys science fiction and fantasy novels, TV shows, and movies. You can follow Fahim on [Twitter](#).

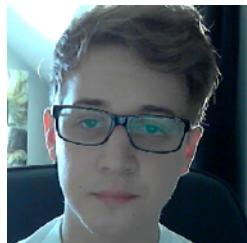


**B.C. Phillips** is an independent researcher and editor who splits his time between New York City and the Northern Catskills. He has many interests, but particularly loves cooking, eating, being active, thinking about deep questions, and working on his cabin and land in the mountains (even though his iPhone is pretty useless up there).



**Ray Wenderlich** is an iPhone developer and gamer, and the founder of [Razeware LLC](#). Ray is passionate about both making apps and teaching others the techniques to make them. He and the Tutorial Team have written a bunch of tutorials about iOS development available at <http://www.raywenderlich.com>.

## About the artist



**Hunter Russell** is a pixel artist with 7 years of experience. He has experience with game developers on every end of the scale and would love to help out with your next project.

Take a look at his portfolio [<http://hunter.digitalhaven-ent.net>], Tumblr [<http://professionalmanlyguy69.tumblr.com>] or contact him via email [hunter@digitalhaven-ent.net].

# Chapter 1: Getting Started

This first chapter will get you well on your way into making your own Beat 'Em Up. It bears more than a passing resemblance to the [How To Make A Side-Scrolling Beat 'Em Up Game](#) tutorial available on Ray's website.

Don't be fooled, though. This Starter Kit offers a whole lot more, and also contains some major differences in approach compared to the tutorial, so you should still go through this from the beginning!

What are you waiting for? Whenever you're ready... get set, go! ☺

**Note:** If you are already familiar with installing and configuring Cocos2D and enabling Automatic Reference Counting (ARC) for a Cocos2D project, then you might want to skip ahead to the section called "Creating a title scene" to get straight into the action. We'll have a starter project ready for you to pick up from.

But if you're the type who likes to start from scratch so you understand every step of the way, keep reading! ☺

## Installing Cocos2D

You will build this game using Cocos2D, a free and widely-used open-source iOS 2D game development framework.

**Note:** Over the years, Objective-C and Apple's Cocoa Framework have grown to include tons of built-in libraries and automation features, making application development significantly easier.

Unfortunately, these libraries and features cater to generic development needs, while game development is somewhat specialized. Instead of figuring

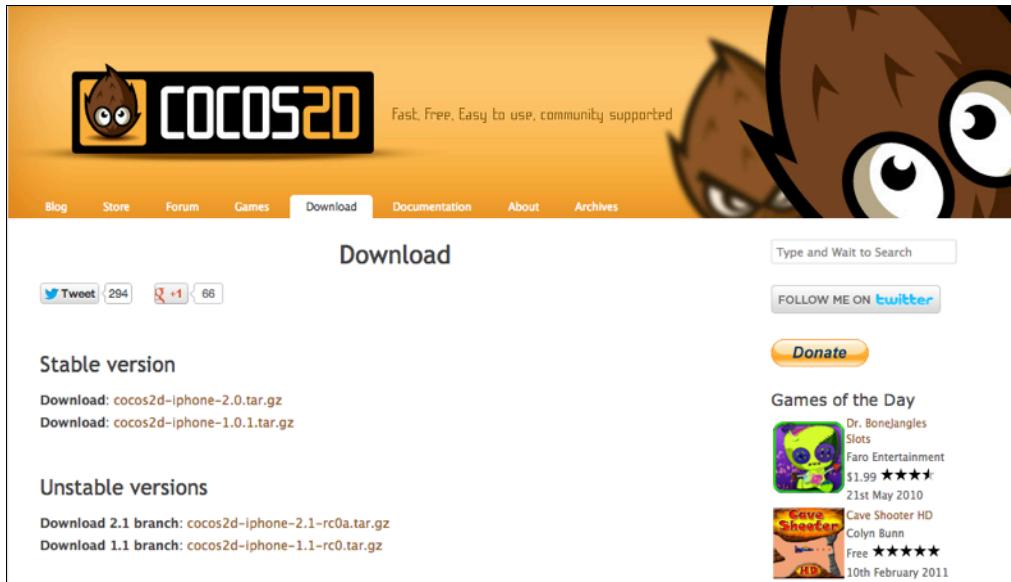
out how to draw something on the screen, you want to focus more on programming game mechanics.

This is where Cocos2D comes in. Built on top of Apple's frameworks and the OpenGL ES drawing engine, it provides game-related objects, such as sprites and particle systems, for you to work with. If you're interested in making 2D games, then Cocos2D is going to be your best friend.

If you already have Cocos2D installed, feel free to skip to the next section. Otherwise, read on to learn how to set it up.

First, load your web browser and navigate to the [Cocos2D download page](#).

You'll see something similar to the following:



You'll notice there are several choices of which version of Cocos2D to download:

- **Stable vs unstable.** The latest versions of Cocos2D are called "unstable versions", whereas older (but battle-worn) versions are called "stable versions." With Cocos2D, don't be afraid to get the latest version, even if it does say unstable, as the stable versions tend to get left behind. Case in point: at the time of writing, it's been a couple of months since iOS 6 was released, but the stable 2.0 version still doesn't support some straight-from-the-box iOS 6-related changes.
- **1.X vs 2.X.** You'll also notice there is a 1.X and 2.X version listed for each option. The 1.X line uses *OpenGL ES 1.X*, while the Cocos2D 2.x line uses *OpenGL ES 2.X*, which allows you to do some advanced and fancy things like writing custom shaders for your game.

**Note:** If you'd like to learn more about OpenGL ES 2.X and custom shader development, please visit [Ray's OpenGL ES Tutorial](http://www.raywenderlich.com/3664/opengl-es-2-0-for-iphone-tutorial) for more information on the subject:

```
http://www.raywenderlich.com/3664/opengl-es-2-0-for-iphone-tutorial
```

For this Starter Kit, you will be using the **latest unstable 2.X** version of Cocos2D. At the time of writing this was 2.1-rc0a but if a more recent version is out now, feel free to use it.

Go ahead and download the **latest unstable 2.X** version of Cocos2D and extract it to a location of your choice.

Next, you'll install the Cocos2D templates so you can easily create a new project that includes the Cocos2D library via Xcode.

To do this, go to **Applications\Utilities** and run your **Terminal** app. Use the **cd** command to switch to the directory where you extracted the Cocos2D archive, like this:

```
cd ~/Downloads/cocos2d-iphone-2.1-rc0a
```

Then install the templates using this command:

```
./install-templates.sh -f -u
```

It should look like this (the highlighted parts are the user commands):

```
$ cd /Users/rwenderlich/Downloads/cocos2d-iphone-2.1-rc0a
$ ./install-templates.sh -f -u
cocos2d-iphone template installer

Installing cocos2d templates
-----
removing old libraries: /Users/rwenderlich/Library/Developer/Xcode/Templates/cocos2d v2.x/
...creating destination directory: /Users/rwenderlich/Library/Developer/Xcode/Templates/cocos2d v2.x/
...copying cocos2d files
...copying CocosDenshion files
...copying Kzmath files
...copying template files
done!

Installing Physics Engines templates
-----
...copying Box2d files
...copying Chipmunk files
done!

Installing JS Bindings templates
-----
...copying JSBindings files
...copying JSBindigns Support files
...copying SpiderMonkey files
...copying JR Swizzle files
...copying CocosBuilderReader files
done!

Installing CCNode file templates...
-----
removing old libraries: /Users/rwenderlich/Library/Developer/Xcode/Templates/File Templates/cocos2d v2.x/
...creating destination directory: /Users/rwenderlich/Library/Developer/Xcode/Templates/File Templates/cocos2d v2.x/
done!
$
```

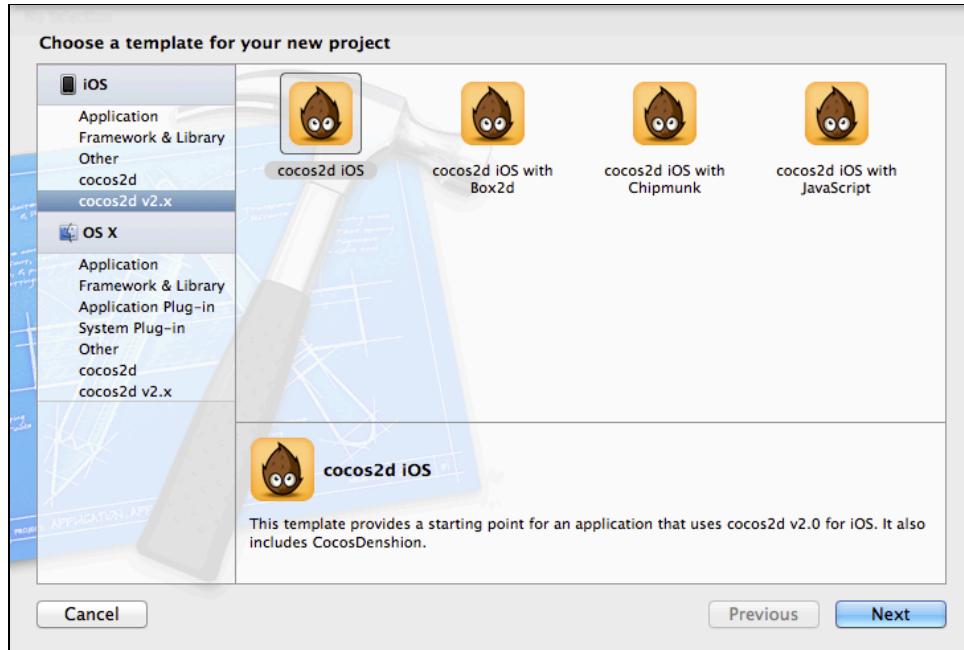
If you get a permission error when executing the above command, then run the command with a `sudo` in front and the `-u` part removed and enter your password when prompted.

That's it! Next time you start up Xcode, there will be a new option to create a Cocos2D project in the project templates.

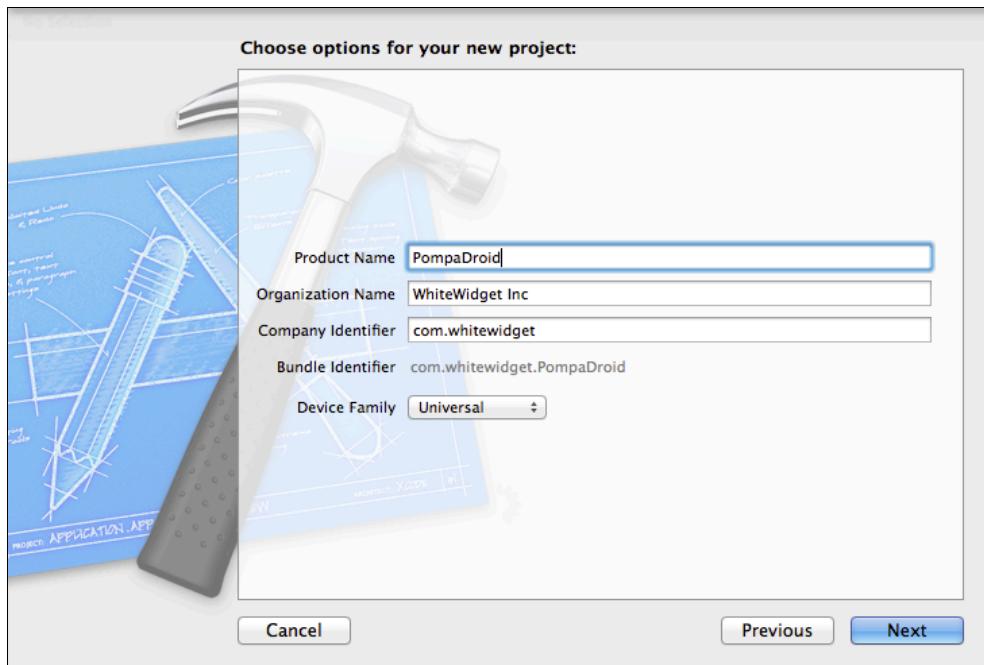
## Creating a Cocos2D project

Let's try out these new templates!

Start up Xcode, go to **File\New\Project...**(Shift + Command + N) and choose the **iOS\cocos2d v2.x\cocos2d iOS** template, as shown below:



Tap **Next**, enter **PompaDroid** for the Product Name, select **Universal** for Device Family and then save the project to a folder of your choice.



**Note:** Are you wondering why you named this project **PompaDroid**? Well, the main character in this game has a funky haircut in the pompadour style, and he's about to beat up on a bunch of droids. My apologies to all the Android devs out there! ;]

You now have a standard “Hello World” Cocos2D template project. To see what it looks like, just build and run the project – in the left portion of your Xcode Toolbar, choose your iPhone Simulator from the Scheme dropdown and click the Run button.



Upon seeing the result, you will instantly know why it is called a “Hello World” project.



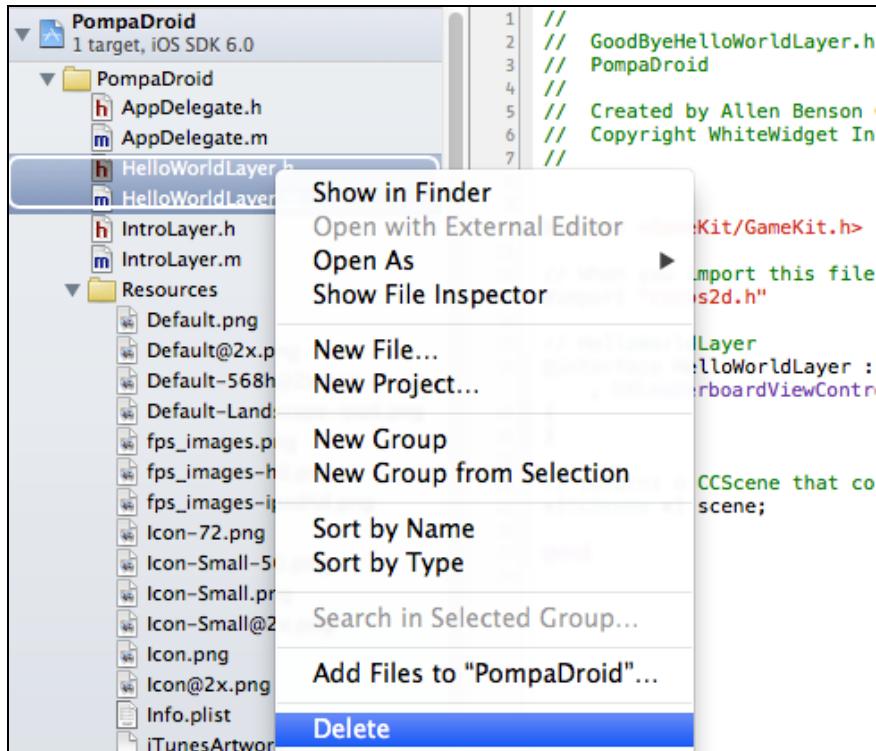
The numbers in the lower left are Cocos2D’s diagnostic/statistic numbers. From bottom to top, these are:

1. **Frames Per Second (FPS):** Maxed at 60, this is the rate at which the display is refreshed every second. Higher means a faster refresh rate.
2. **Milliseconds Per Frame (MPF):** The time it took to render a frame in milliseconds.
3. **Draw Calls:** The number of OpenGL draws on the screen. This is usually one draw call per object displayed on the screen, but by using batch nodes (which you’ll learn more about later) you can reduce this number. The more draw calls, the greater the risk of poor performance due to a heavy processor load.

Just keep these in mind. If you ever forget them, you can refer back to this section.

There’s no room for a “Hello World” message in a Beat ‘Em Up game, so you have to clean up that screen.

The code responsible for this is in **HelloWorldLayer.h** and **HelloWorldLayer.m**. You won't need these files going forward, so select and delete them (either via the right/Control+click context menu or by tapping the Delete key). When the warning comes up, choose Move To Trash.



Since those files are gone, you need to remove references to them. Open **IntroLayer.m** and remove this line at the top of the file:

```
#import "HelloWorldLayer.h" //remove this line
```

Also comment out this line in `onEnter`:

```
//[[CCDirector sharedDirector] replaceScene:[CCTransitionFade
transitionWithDuration:1.0 scene:[HelloWorldLayer scene]]];
```

Build and run to make sure that everything still works. You should end up with the Cocos2D splash screen permanently onscreen, with the diagnostics visible on the lower left:



You now have a clean starting point for your project. By this time, you must be itching to code the game, but there's still one last thing to do first – enable Automatic Reference Counting (ARC).

## ARC-ifying your project

In the old days of iOS development, you had to handle all memory management yourself. This involved adding calls to retain and release references to objects at the appropriate place in your code. It was a notorious source of errors, and could be quite confusing to beginning and seasoned programmers alike.

In iOS 5, Apple introduced ARC, or Automatic Reference Counting, which means that as a developer, you no longer have to handle the task of retaining and releasing memory. Instead, the compiler will do it for you automatically.

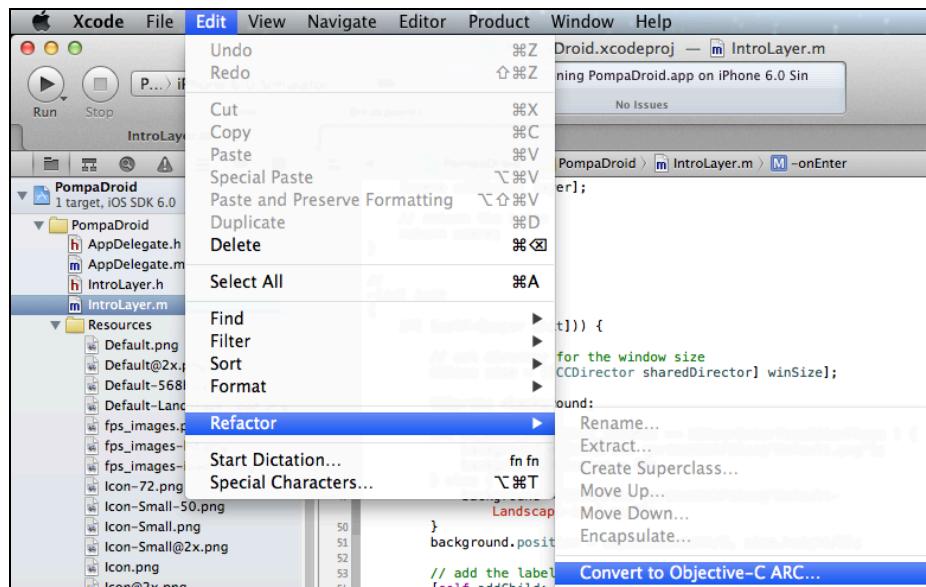
This makes iOS development much easier and your code more stable by eliminating a lot of potential memory leaks, so it's a good idea to use it in your projects when possible!

**Note:** To learn more about ARC and how it works, check out our tutorial on the subject here:

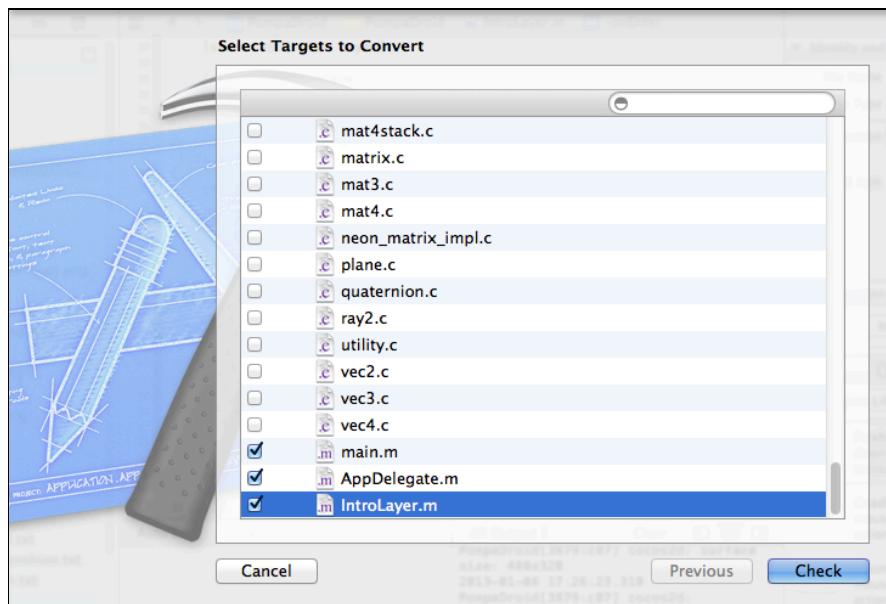
<http://www.raywenderlich.com/5677/beginning-arc-in-ios-5-part-1>

Unfortunately, the standard Cocos2D templates are not set up to be ARC-compliant. Luckily, this is quite easy to fix yourself – and that's what you're going to do now.

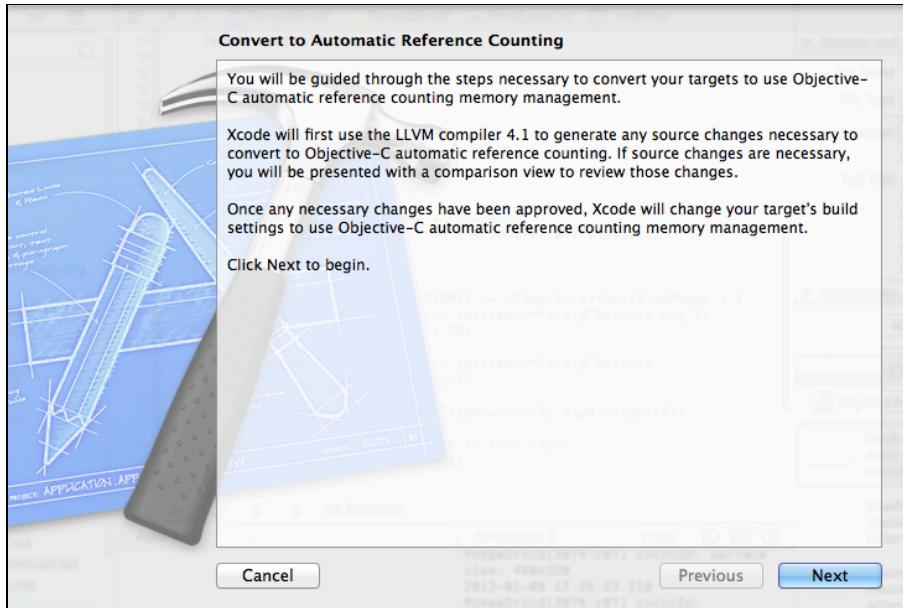
In Xcode, go to **Edit\Refactor\Convert to Objective-C ARC**:



In the dialog that appears, expand **PompaDroid.app** and mark only the files that are a part of your project: **main.m**, **AppDelegate.m**, and **IntroLayer.m** (those outside the libs folder):



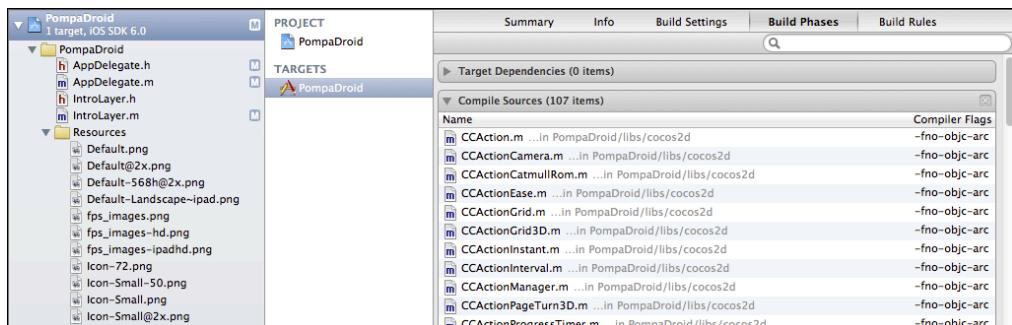
Click **Check**. After the check, a new dialog should appear:



Choose **Next**, then **Save** to finish converting your project to use ARC. If a dialog appears asking about snapshots, choose **Disable**.

## Now what exactly happened there?

In the Project Navigator, click on the project root and your center panel should show the Summary page. Make sure **PompaDroid** is highlighted under TARGETS and select the **Build Phases** tab. Expand **Compile Sources** and you should see a list of files, similar to what was just shown in the ARC-conversion dialog above:



Scroll further down until you see the three files you marked:

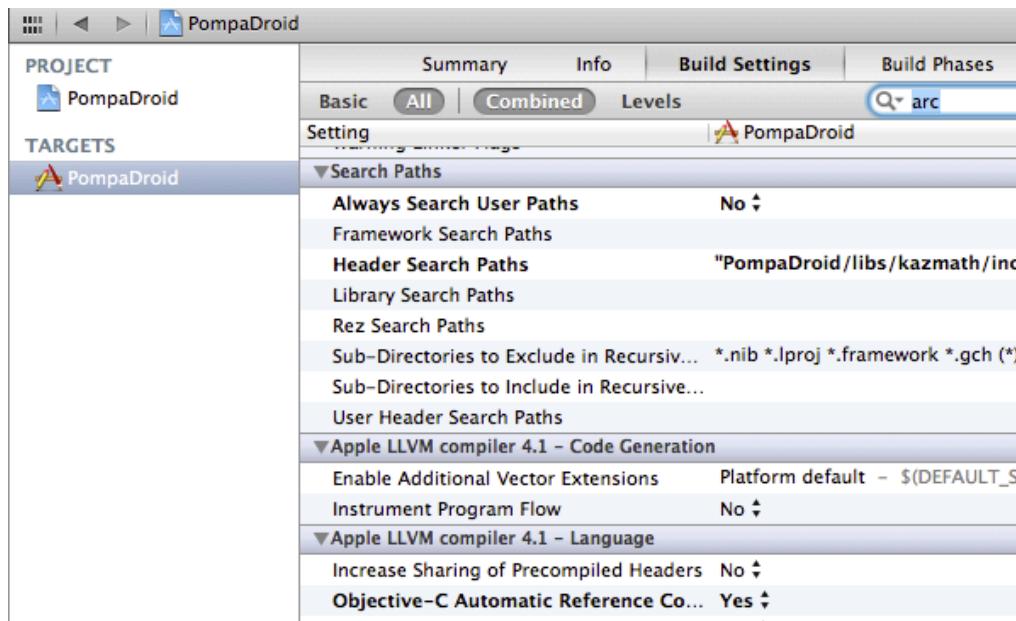
c ray2.c ...in PompaDroid/libs/kazmath/src	-fno-objc-arc
c utility.c ...in PompaDroid/libs/kazmath/src	-fno-objc-arc
c vec2.c ...in PompaDroid/libs/kazmath/src	-fno-objc-arc
c vec3.c ...in PompaDroid/libs/kazmath/src	-fno-objc-arc
c vec4.c ...in PompaDroid/libs/kazmath/src	-fno-objc-arc
m main.m ...in PompaDroid	
m AppDelegate.m ...in PompaDroid	
m IntroLayer.m ...in PompaDroid	

Can you spot the difference? The text in the right-hand column is a big clue.

The value `-fno-objc-arc` in the Compiler Flags column informs the compiler that these components (in this case, the Cocos2D library code) aren't using ARC.

If you ever add another non-ARC-compliant file to your project, you can just look for the file here, and add the flag yourself – simple, Yes?

The other change the refactor process did for you can be found in the **Build Settings** tab. Just look for the line **Objective-C Automatic Reference Counting**:



In a fresh Cocos2D project, the value of this setting would be **NO**.

While you're here, select the **Summary** tab and make sure **Deployment Target** is at least 5 or higher. This indicates the minimum Xcode version for your app to run on, and you are choosing this because this is the first version of iOS that fully supports ARC.

**Note:** ARC is actually backwards compatible to iOS 4, but there is one exception: it does not support zeroing weak references in iOS 4. For more details, see Apple's Objective-C Feature Availability Index:

<http://developer.apple.com/library/ios/#releasenotes/ObjectiveC/ObjCAvailabilityIndex/index.html>

In this Starter Kit you'll choose iOS 5 so you're sure all features of ARC are supported. Also, these days less than 5% of devices are running an iOS version prior to iOS 5 anyway, see this page for more details:

<http://david-smith.org/iosversionstats/>

And with that, you are done with the project set-up – it's time to code!

## Creating a title scene

**Note:** If you skipped ahead from the beginning of this chapter, we have a starter project ready for you. It's **PompaDroid – Starter Project.zip** – unzip it and open up the Xcode project inside.

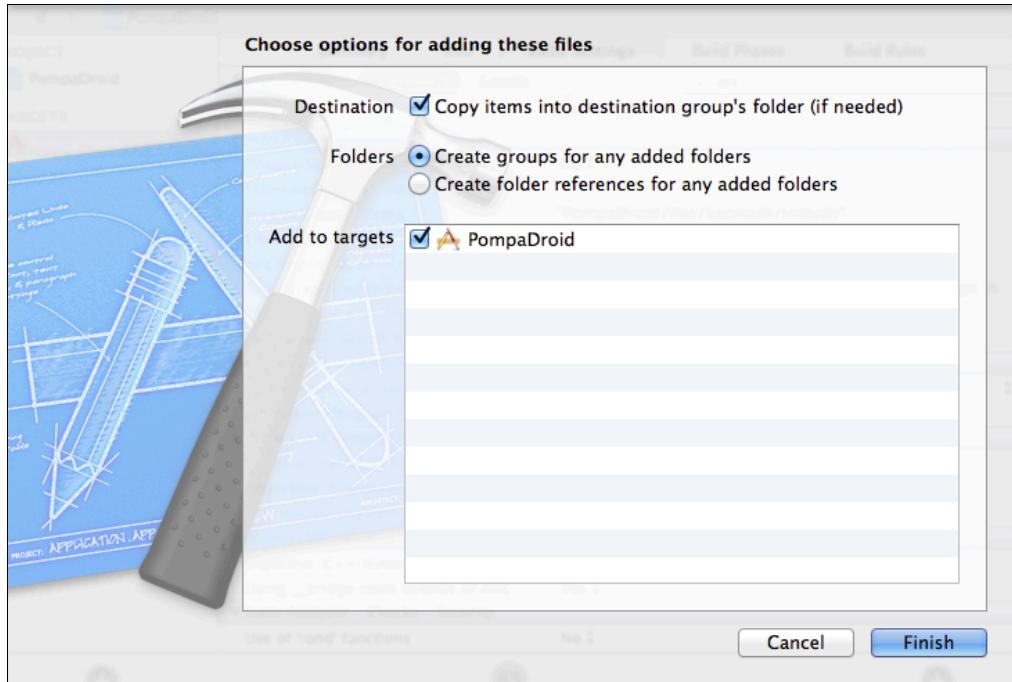
Build and run and you'll see it's just an empty Cocos2D template, but ARC enabled. Now you can dive straight into the action!

Now that your Xcode project is ARC-enabled, your first step in building the game is to create a title screen. You don't want the game to drop the player into the middle of the action before introducing itself!

You won't need a fancy title screen, so simply displaying a title and start message is sufficient.

Before you begin, you need the files containing the images used for the title scene. This Starter Kit includes a **Resources** directory, and inside of it there is a subdirectory named **Images**.

Find the **Images** folder and drag it onto the **Resources** folder in your Xcode project. In the pop-up dialog, make sure that **Copy items into destination group's folder (if needed)** is checked, **Create groups for any added folders** is selected, and that the **PompaDroid** target is checked, then click **Finish**.



The above options will apply for all other files and folders that you add to this project (except C-header files).

Take a look through the files if you'd like. You'll use these files for the title menu:

- **bg\_title.png**: The background image for the title scene.
- **txt\_title.png**: An image with the word, "PompaDroid".
- **txt\_touchtostart.png**: An image with text that says, "Touch to Start".

Don't be fooled by the "txt" prefix in the file names. These are still sprites, not labels, and will be treated as such.

**Note:** These images were prepared using Adobe Photoshop but could very well have been done in another image editor. There's nothing special about them – they are simply static images that will be displayed as sprites. You can also open the **TitleScreen.psd** that comes in the **Raw** folder included in the Starter Kit to see how these images were created.

Control-click (or right-click) on the **PompaDroid** group in Xcode, select **New Group** and give it the name **Menus**.

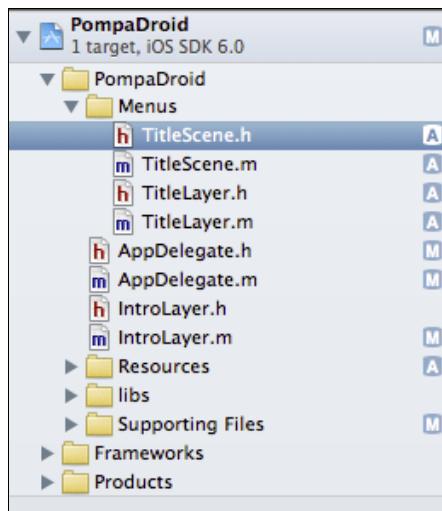
Right-click (or control-click) on the newly-created **Menus** group, but instead of selecting New Group, you should now select **New File**. Choose the **iOS\cocos2d v2.x\CCNode class** and click **Next**. Enter **CCScene** for Subclass of, click **Next** and then name the new file **TitleScene**.

Now create another file in the same fashion as before, but make it a Subclass of **CCLayer** and name it **TitleLayer**.

**Note:** A **ccscene**, which we'll refer to simply a scene (also known as a screen or a stage), is crucial to Cocos2D's application structure. You can think of a scene as an *independent* piece of the workflow, in the sense that *only one scene can be active, or shown, at any given time*. Each scene can be composed of one or more layers (**cclayer**), and these layers handle appearance and behavior.

Don't be confused by the existence of the **IntroLayer** file without an accompanying **IntroScene**. **IntroLayer** combines the scene and layer in one file, but structurally the **IntroLayer** itself is still added to a **ccscene** within that file. You can do it this way if you prefer, but for this Starter Kit you'll treat each object as a separate file for clarity.

Your **PompaDroid** group should now look something like this:



Open **TitleScene.m** and make the following changes:

```
//add to top of file
#import "TitleLayer.h"

//add inside @implementation
-(id)init
{
    if ((self = [super init]))
    {
        TitleLayer *titleLayer = [TitleLayer node];
        [self addChild:titleLayer];
```

```
    }
    return self;
}
```

You import `TitleLayer.h` at the top of the file so that you can create an instance of `TitleLayer`.

Importing header files (`.h`) allows you to use functions and objects defined in those files on the class into which you've imported them. Take a look at `TitleScene.h`. It contains these lines at the top:

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

The first line imports the core of all Objective-C libraries: the Foundation framework. The second line imports `cocos2d.h`, so you can use all of Cocos2D's classes and functions.

In `init` above, you create a new instance of `TitleLayer` and add it as a child of `TitleScene`. Every object in Cocos2D is a subclass of `ccNode`, which has a class method named `node` that simply allocates the object and calls `init` for that instance. Now that `TitleScene` exists, you can take steps to transition the game into this scene. Right now when the game boots up, the first scene it displays is `IntroLayer`'s scene. Its job is simply to display the splash image and move on to the next scene.

Go to `IntroLayer.m` and add this to the top of the file:

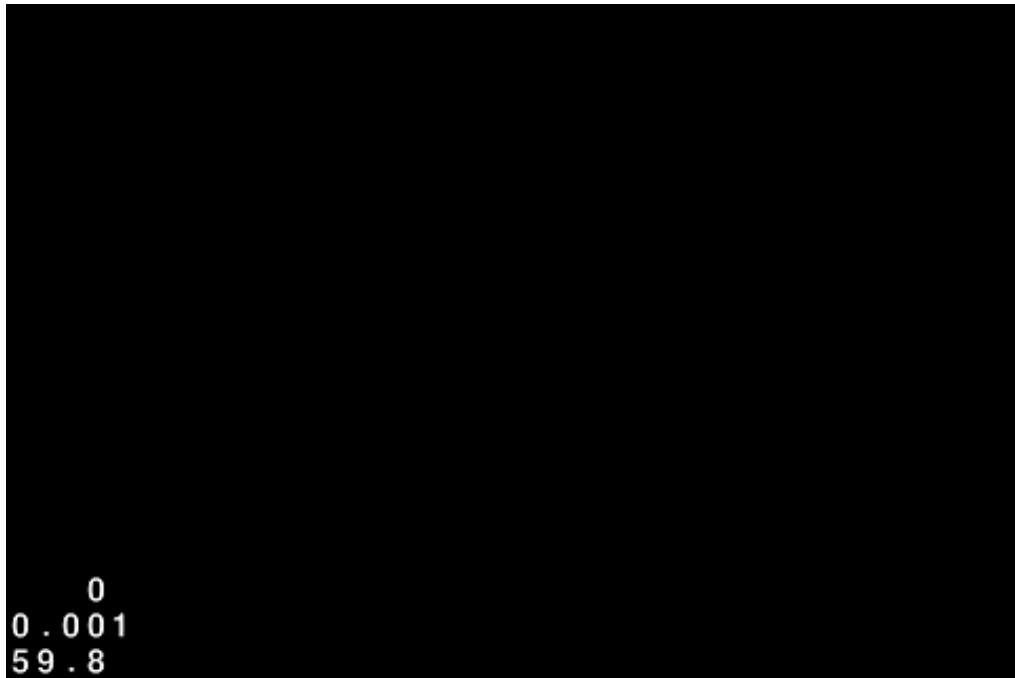
```
#import "TitleScene.h"
```

This allows you to call instances of `TitleScene` from `IntroLayer.m`. You want `IntroLayer` to transition to `TitleScene` after a few seconds of showing the splash image.

In `onEnter`, uncomment the line you commented before, and change it as follows:

```
//uncomment and change HelloWorldLayer scene to TitleScene node
[[CCDirector sharedDirector] replaceScene:[CCTransitionFade
transitionWithDuration:1.0 scene:[TitleScene node]]];
```

Build and run your project, and you should see the following displayed onscreen after the splash image:



It's a whole lot of nothing, but that's what you want – a new clean scene to work with! Remember that it's the `CCLayer`'s job to populate a scene with appearance and behavior, so it's time to work with `TitleLayer`.

Go to `TitleLayer.m` and add the following methods inside `@implementation`:

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

-(void)setupTitle
{
    CGSize winSize = [CCDirector sharedDirector].winSize;

    CCSprite *titleBG = [CCSprite spriteWithFile:@"bg_title.png"];
    titleBG.position = ccp(winSize.width/2, winSize.height/2);
    [self addChild:titleBG];
}
```

You call `setupTitle` from `init`. It gets the screen dimensions from `CCDirector`, creates a `CCSprite` using the background image you added previously, positions it at the center of the screen and adds it as a child of the layer.

`ccsprites`, by default, are anchored at the center, so when you position a sprite you are moving it from its center. Since the background image is the same size as the screen, here you position it at the center of the screen.

**Note:** Cocos2D positions and sizes are in points, not pixels. A point scales up according to the screen of the device. A point can be 1 pixel on non-retina display devices, and 2 pixels on retina display devices.

This makes it easier to position objects using hard values. If you tell an object to move 100 points to the right, it will move by the same relative amount on both retina and non-retina displays.

Getting the center of the screen from the screen dimensions, as you did in the above code, might seem a bit tedious. You can just imagine how many times you might be doing this throughout a full game project.



Thankfully, there is a way to define shortcuts so this becomes much quicker and easier.

Control-click on the **Supporting Files** group and select **New File**. Choose the **iOS\C and C++\Header File** and click **Next**. Give it the name **Defines** and make sure that **PompaDroid** in **Targets** is unchecked. You don't need to include header files in your target – only implementation files or resources like images and sounds.

For the sake of uniformity and ease-of-use, you will be keeping all of your definitions in this file.

Open **Defines.h** and add the following after `#define PompaDroid_Defines_h`:

```
#define SCREEN [[CCDirector sharedDirector] winSize]
#define CENTER ccp(SCREEN.width/2, SCREEN.height/2)
#define OFFSCREEN ccp(-SCREEN.width, -SCREEN.height)
#define IS_RETINA() ([[UIScreen mainScreen]
    respondsToSelector:@selector(scale)]) && [[UIScreen mainScreen]
    scale] == 2)
```

```
#define IS_IPHONE5() ([[CCDirector sharedDirector] winSize].width  
== 568)  
#define IS_IPAD() ([[UIDevice currentDevice] userInterfaceIdiom]  
== UIUserInterfaceIdiomPad)  
#define CURTIME CACurrentMediaTime()
```

The first word after `#define` is the keyword/macro, and the next part after that is the value returned by the macro.

The macros you just defined are:

- **SCREEN:** The screen's dimension in points.
- **CENTER:** The center of the screen in points.
- **OFFSCREEN:** Coordinates that you're sure will never be visible.
- **IS\_RETINA():** A Boolean that identifies whether the device display is retina or not.
- **IS\_IPHONE5():** A Boolean that identifies whether the device display is 4.0-inch or not.
- **IS\_IPAD():** A Boolean that identifies whether the current device is an iPad.
- **CURTIME:** Shorthand for `CACurrentMediaTime()`, which returns the current device time. Yes, I'm that lazy. ;]

You want `Defines.h` to be present in every file that you work on from here on out, but it's a hassle to have to enter `#import "Defines.h"` for every new file. The quick solution is to include it in the pre-compiled header file instead.

The pre-compiled header file (`Prefix.pch`) is a special file that automatically gets imported in every file in your project without you needing to manually import it. You can import your `Defines.h` in this file to make it available everywhere as well.

In the **Supporting Files** folder, open **Prefix.pch** and do the following:

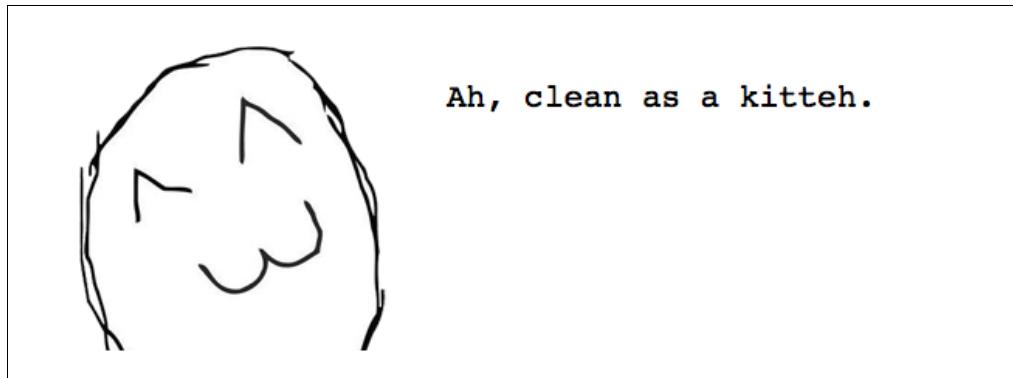
```
//add after #import <Foundation/Foundation.h>  
#import "Defines.h"
```

Build so that `Defines.h` gets populated throughout your project, but don't run just yet.

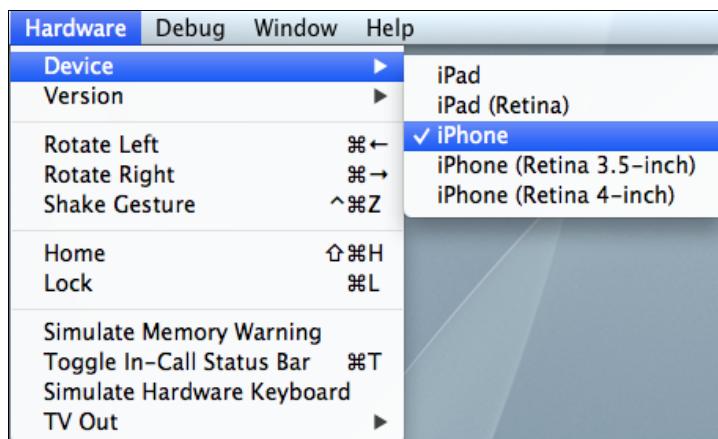
Go back to **TitleLayer.m** and replace `setupTitle` with the following:

```
-(void)setupTitle  
{  
    CCSprite *titleBG = [CCSprite spriteWithFile:@"bg_title.png"];  
    titleBG.position = CENTER;  
    [self addChild:titleBG];  
}
```

As you can see, that simplifies the code by making use of the **CENTER** macro. Now doesn't that look cleaner? ☺



Open the iPhone Simulator and go to **Hardware\Device\iPhone** to switch your simulator to be a **non-retina display iPhone simulator**:



I'll explain why you had to choose the non-retina iPhone simulator soon.

Then build and run your project. You should see the following:



Now one last little but important detail: add the title and start message to the scene.

Still in **TitleLayer.m**, add the following to the end of **setupTitle**:

```
//add after [self addChild:titleBG]
CCSprite *title = [CCSprite spriteWithFile:@"txt_title.png"];
title.position = ccp(CENTER.x, CENTER.y + 66);
[self addChild:title];

CCSprite *start = [CCSprite
spriteWithFile:@"txt_touchtostart.png"];
start.position = ccp(CENTER.x, CENTER.y - 37.5);
[self addChild:start];
```

That's pretty simple code – you add the remaining two images as sprites and position them from the center.

Build and run the game and you will see:



The two sprites appear on top of the background image because they were added after the background image, in code order. Optionally, you could have added these sprites first and then manipulated the z-order to change the order in which the sprites are drawn. Keep this in mind – you may need to know this in your own projects!

## Supporting universal display

In the previous section, I specifically asked you to choose the **non-retina display iPhone Simulator** because, well, you hadn't done anything to support the other displays yet.

If you tried running the game on a retina iPhone display, or on an iPad of any resolution, you would have gotten very different results. For example, if you had run it on a **retina display iPhone Simulator**, you would have seen the following:



By default in Cocos2D, each device needs its own version of an image, with different suffixes to signify their purpose. If you take `bg_title.png` as an example, there should also have been the following:

- **`bg_title-hd.png`**: For iPhone retina and (optionally) iPad non-retina (at 2x the size of the non-retina iPhone scale).
- **`bg_title-ipad.png`**: For iPad non-retina.
- **`bg_title-ipadhd.png`**: For iPad retina (2x the size of the iPad non-retina scale).
- **`bg_title-wide.png`**: For iPhone 5 non-retina. This device actually doesn't exist but is there for cleanliness.
- **`bg_title-widehd.png`**: For iPhone 5 retina.

This would be true for all images supporting universal display.

**Note:** You do not necessarily need to include all of these options if you can figure out a way to re-use images. For example, in our games we often like to use the same art for the iPad non-retina and iPhone retina displays, since they are about the same size. Also, to support the iPhone 5 we often use the same sprites as the normal iPhone – just include different background images that are wider.

You can override the extensions used for each display in **`AppDelegate.m`** – search for the call to `setPhoneRetinaDisplaySuffix:` to see where.

Your current situation is a special case however, because this Starter Kit uses pixel art. For pixel art, you don't need to include higher resolution images for bigger screen sizes, because the entire point is to look pixelated!

So instead of adding higher resolution images, you will just scale the artwork. Not only is this easier, but it also saves memory and can get you kudos points for being stylish. For example, check out the pixelated cuteness of the game [Tiny Tower](#):



**Note:** If you want to learn how to make pixel art for games, this tutorial by Glauber Kotaki is a good introduction:

<http://www.raywenderlich.com/14865/introduction-to-pixel-art-for-games>

For any image that uses pixel art, you only need the lowest resolution version of the graphic, sometimes even lower than the non-retina display resolution you would have normally used. You handle the scaling in the program itself. This will help you keep texture memory at a minimum.

The downside to doing it this way, though, is that you cannot take direct advantage of Cocos2D's automation in selecting the proper resolution art for each device.

You could just disable retina display support altogether so that Cocos2D automatically scales, but this would also be a problem if you had some regular art that had high-resolution versions. Plus, you would still have to handle the iPad device family.

One way to handle the pixel art would be to scale each image as you create it, like so:

```
if (IS_RETINA())
{
    titleBG.scale *= 2.0;
    title.scale *= 2.0;
    start.scale *= 2.0;
}

if (IS_IPAD())
{
    titleBG.scale *= 2.0;
    title.scale *= 2.0;
    start.scale *= 2.0;
}
```

If the device has a retina display, then double the scale of the sprites. Further, if the device is an iPad, double the scale again. However, it'd be a bit tedious to do these two checks every time you create a sprite. Fortunately, there is a cleaner way that involves defining some macros.



Switch to **Defines.h** and add these lines right after the existing code (but above the final `#endif`):

```
#define kPointFactor (IS_IPAD() ? 2.0 : 1.0)
#define kScaleFactor (kPointFactor * (IS_RETINA() ? 2.0 : 1.0))
```

The syntax on the value side of the macro might be new to you. The `? :` is a ternary operator for a conditional expression – which is just a fancy way of saying that it is a shorter form of an `if-else` statement.

If the condition on the left side of the `?` is `True`, then the first value is returned; otherwise, the second value is returned. I know it looks unpleasant, and I personally don't like seeing this kind of syntax, but that's why you put it in `Defines.h`. ☺

You defined two very useful macros:

1. **kPointFactor:** Automatic point-pixel conversion is only done when going from non-retina to retina, so you will need this to convert points from iPhone to iPad.
2. **kScaleFactor:** This will be used to scale for retina and iPad devices.

Now go to **TitleLayer.m** and replace **setupTitle** with this:

```
-(void)setupTitle
{
    CCSprite *titleBG = [CCSprite spriteWithFile:@"bg_title.png"];
    titleBG.position = CENTER;
    [self addChild:titleBG];

    CCSprite *title = [CCSprite spriteWithFile:@"txt_title.png"];
    title.position = ccp(CENTER.x, CENTER.y + 66 * kPointFactor);
    [self addChild:title];

    CCSprite *start = [CCSprite
spriteWithFile:@"txt_touchtostart.png"];
    start.position = ccp(CENTER.x, CENTER.y - 37.5 *
kPointFactor);
    [self addChild:start];

    titleBG.scale *= kScaleFactor;
    title.scale *= kScaleFactor;
    start.scale *= kScaleFactor;
}
```

The new code first multiplies the position values with **kPointFactor** so that they get adjusted for iPad, and then it multiplies all the sprite sizes by **kScaleFactor** so that they get scaled accordingly.

In your iPhone Simulator, select **Hardware\Device\iPhone (Retina 4-inch)**. Build and run your project. It should look like this:



Now try it for different devices: iPhone (Retina 3.5-inch), iPad and iPad (Retina).

Notice that the background image covers up the entire screen regardless of the resolution. This is because this image was drawn at a unique resolution of 568x384.

568 pixels is the longest needed width. Doubling this value results in 1136, which is the 4-inch iPhone's pixel width, even longer than the 1024 pixel width of the iPad.

384 pixels is the longest needed height. Doubling this will be 768, which is the pixel height of the iPad.

In other words, the background image is sized to be the maximum size necessary on any device, and on devices with smaller screens might overlap a bit. Here's a picture to show you what I mean:



From now on, whenever you build and run, you can test on any device (or devices) you'd like.

You may have noticed one minor problem: the art seems blurred at higher resolutions. This is because by default, when images are scaled up, Cocos2D applies anti-aliasing to the images. To disable this, add these lines at the end of `setupTitle`:

```
[titleBG.texture setAliasTexParameters];
[title.texture setAliasTexParameters];
[start.texture setAliasTexParameters];
```

This code turns anti-aliasing off for the textures of the three sprites, in order to make the pixel art scale at crisp pixel boundaries (rather than blurring the edges). Build and run again, and you should now have crisp pixel graphics even on retina displays:



It may be hard to see the difference from this screenshot or on the simulator, but if you have a retina device I recommend seeing what it looks like with and without these lines so you can see the difference. It should look a lot crisper once you add these lines.

Now you know how to make pixel artwork for all devices. And since you didn't disable retina graphics, you can mix retina and regular graphics to your heart's content!

## Transitioning to the game

The title scene says "Touch to Start", but there's nothing to start yet! You'd better create the next scene.

Select the **PompaDroid** group in Xcode, go to **File\New\Group** and name the new group **Game**.

Next, select the **Game** group and go to **File\New\New File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **CCScene** for Subclass of, click **Next** and name the new file **GameScene**.

Create two more files the same way, entering **CCLayer** for Subclass of and naming them **GameLayer** and **HudLayer**, respectively.

Open **GameScene.m** and make the following changes:

```
//add to top of file
#import "GameLayer.h"
#import "HudLayer.h"
```

```
//add inside @implementation
-(id)init
{
    if ((self = [super init]))
    {
        GameLayer *gameLayer = [GameLayer node];
        [self addChild:gameLayer z:0];

        HudLayer *hudLayer = [HudLayer node];
        [self addChild:hudLayer z:1];
    }
    return self;
}
```

The code creates one instance of `GameLayer` and one instance of `HudLayer`, and adds them to the `GameScene`. Additionally, their z-order dictates the order in which these two layers will be drawn on the scene. `GameLayer` has the lower z-order, so it will be drawn first.

As you may have guessed from the names, `GameLayer` will contain game elements such as the characters and the stage, while `HudLayer` will contain display elements such as the directional pad and the A and B buttons. `HudLayer` needs to be drawn last because it has to be on top of everything and always visible.

Next, you'll make the game switch to this new scene when the user taps the main menu. Go back to `TitleLayer.m` and make the following changes:

```
//add to top of file
#import "GameScene.h"

//add inside init, after [self setupTitle]
self.touchEnabled = YES;

//add inside @implementation
-(void)ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    [[CCDirector sharedDirector] replaceScene:[CCTransitionFade
transitionWithDuration:1.0 scene:[GameScene node]]];
}
```

This is very straightforward. In `init`, you enable touches for this layer. This allows for `ccTouchesBegan:` to work, and inside `ccTouchesBegan:`, you create a transition to `GameScene`.

There's one last thing to do before you check your work, and that is to add a bit of life to `TitleScene` by making the "Touch to Start" sprite blink.

Add these lines at the end of `setupTitle`:

```
start.tag = 1;
CCBlink * blink = [CCBlink actionWithDuration:5.0 blinks:10];
CCRepeat * repeat = [CCRepeatForever actionWithAction:blink];
[start runAction:repeat];
```

The first thing you do is give the start sprite a tag. A tag is just a number that you associate with a sprite so you can easily find it again later (the tag defaults to 0). It can be handy if you want a quick way to get a reference to a sprite that you add to the layer in another method. Alternatively, you can create an instance variable to keep track of it, but sometimes this is easier.

Next, you create a few Cocos2D **actions**. Actions are handy ways to make your sprites do things like move, blink, or jump. You first create the action by passing in the appropriate parameters, then you specify which Cocos2D node will perform the action.

The above `CCBlink` tells the performer of the action to blink 10 times in 5 seconds, while `CCRepeatForever` tells the performer to keep on repeating the inner action until it is told to stop.

Then add these lines at the end of `ccTouchesBegan`:

```
CCSprite *start = (CCSprite *) [self getChildByTag:1];
[start stopAllActions];
start.visible = NO;
```

When the screen is touched, `start` should disappear. You assigned `start` a **tag** of 1 in `setupTitle`, and retrieved `start` again in `ccTouchesBegan:` using its tag number.

Once you have a reference to the sprite, you use the `stopAllActions` method to stop the blinking, and also make it invisible.

**Note:** The `ccsprite *` inside the parenthesis tells the compiler that whatever is returned by the method is going to be an object of type `ccsprite`. This is called typecasting.

Typecasting is allowed in instances where the object type that you cast on the object is a subclass of the expected object. In this case, `getChildByTag` returns an object of type `CCNode`, and `ccsprite` is a subclass of `CCNode`.

Build and run, and you should see the message blink. Try to touch the screen and it should transition to the `GameScene` like so:



Onward... into nothingness!

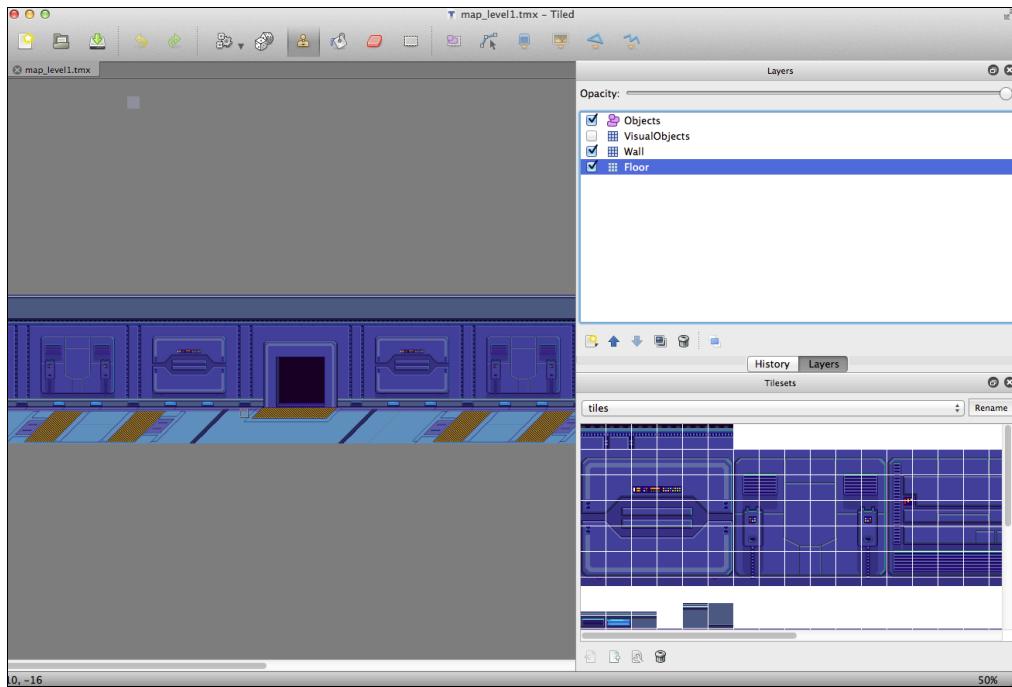
## Loading the tile map

You get a blank screen after starting the game because there is nothing on the `GameScene`. So it's time to put some content into `GameScene`! You will start with the item that appears behind everything else: the tile map.

For this section, you will use the following files from the Images folder:

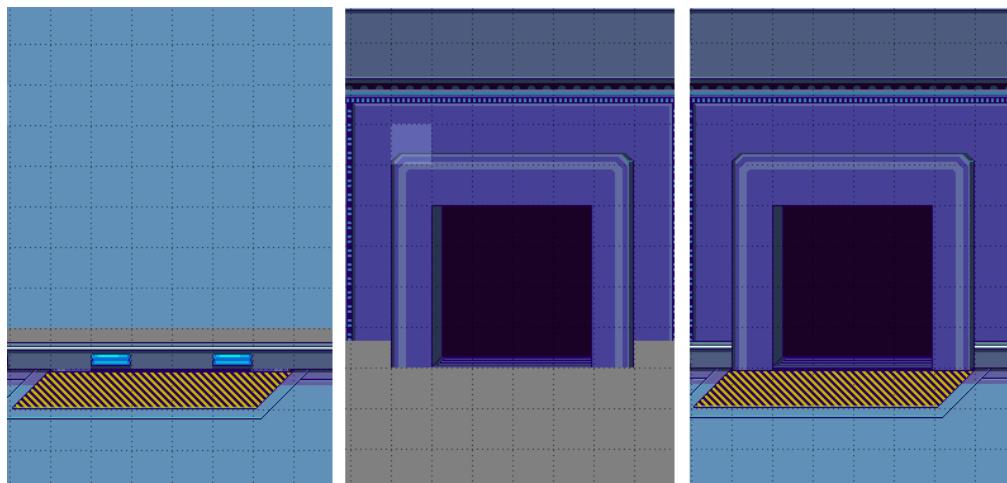
- **tiles.png**: The tiles used by the tile map. Each tile is 32x32 pixels in size.
- **map\_level1.tmx**: The TMX tile map file that contains the layout for the first level of the game.

The tile map was made using the [Tiled map editor](http://www.mapeditor.org) (<http://www.mapeditor.org>). If you have this application installed, you can open `map_level1.tmx` to see what's inside:



There are a few important things to keep in mind here:

- There are three tile layers (**Wall**, **Floor**, and **VisualObjects**) and one object layer (**Objects**).
- The **Wall** and **Floor** layers contain the wall and floor tiles, respectively.
- Try hiding the **Wall** and **Floor** layers one at a time by un-ticking the checkbox beside the layer's name. You will see that the fourth row of tiles from the bottom (to see tile rows, enable the grid view by selecting View>Show Grid from the menu) is comprised of tiles from both the Wall and Floor layers. Layers can be useful for a lot of things, and in this case, they're being used to create variations of tiles by combining two in one spot.



- The walkable floor tiles are in the first three rows from the bottom.

- The **VisualObjects** layer is just meant to be a guide for where to place objects on the map, and that's why it's turned off.
- The **Objects** layer is responsible for determining where to place objects. You'll get to that later.

**Note:** If you wish to know more about creating tile maps, check out our tutorial on how to make a simple tile-based game with Cocos2D 2.X using the Tiled map editor:

<http://www.raywenderlich.com/29458/how-to-make-a-tile-based-game-with-cocos2d-2-x>

Similar to the background image in the title scene, to support both the iPhone and iPad the tile map's height is set to 384 pixels (12 tiles, where each tile is 32 pixels in height).

Let's get coding again. Open **GameLayer.h** and add the following variable inside the curly braces after the `@interface`:

```
CCTMXTiledMap *_tileMap;
```

Switch to **GameLayer.m** and add these methods inside the `@implementation` section:

```
-(id)init
{
    if ((self = [super init]))
    {
        [self initTileMap:@"map_level1.tmx"];
    }
    return self;
}

-(void)initTileMap:(NSString *)fileName
{
    _tileMap = [CCTMXTiledMap tiledMapWithTMXFile:fileName];

    for (CCTMXLayer *child in [_tileMap children]) {
        [[child texture] setAliasTexParameters];
    }

    _tileMap.scale *= kScaleFactor;

    [self addChild:_tileMap z:-6];
}
```

You create a `_tileMap` instance variable so that you can access the tile map from anywhere in the class. You'll use this later to check the properties of the tile map. You could also access the tile map using the `tag` attribute as you did before, but when you frequently need a reference to something I prefer making an instance variable.

**Note:** If you're wondering why there is an underscore before the instance variable name, this is just a naming convention used by a lot of developers, as well as by Apple. It helps distinguish instance variables from local variables and properties.

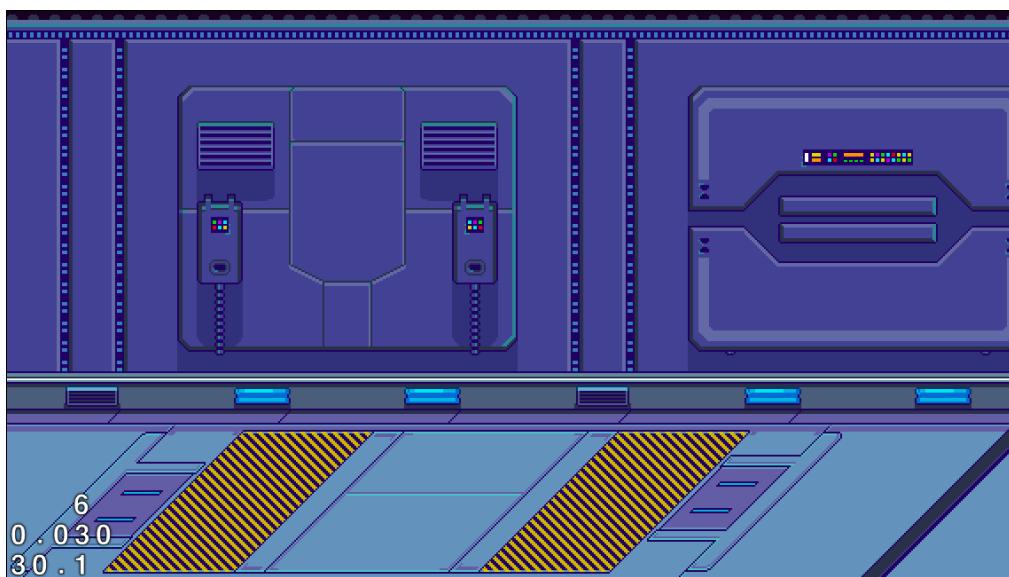
The usefulness of this technique will be especially apparent once you begin taking advantage of Xcode's syntax automation features.

`initTileMap`: creates the tile map based on the filename provided. It then iterates over all members of the tile map and disables anti-aliasing.

The code also makes sure that the tile map scales according to the device by multiplying the map's scale property by `kScaleFactor`.

Finally, it adds the tile map to the layer. Keep in mind that the z-value of the tile map is -6. You want to make sure that the tile map is always behind everything, so from now on, there shouldn't be any children added to `GameLayer` with a z-value lower than -6.

Build and run on all devices, and you should see the tile map displayed when you enter `GameScene`:



Congratulations, you now have a main menu and the start of your main level!

At this point, you've accomplished a lot – you've made an ARC-enabled Cocos2D project, made a main menu with some simple Cocos2D actions, have transitioned between different scenes, and have even loaded a tile map into your game scene.

Feel free to take a well-deserved break. When you return, you'll add the hero into the game – and his funky hair-do.

**Challenge:** Download the latest version of the Tiled map editor from this site if you haven't already:

<http://www.mapeditor.org/>

Then open map\_level1.tmx and modify the beginning part of the level to have a slightly different look (use the Wall and Floor layers). Build and run and make sure it shows up OK in your game



# 2

## Chapter 2: Walk This Way

Now that you have a setting for your game, how about adding a hero?

It's easy enough to add a sprite for the hero to the layer, just like you did for the menu background and text. But a hero's job is not so simple!

An iOS Beat 'Em Up Game typically has the following requirements:

- Display some joypad controls on the screen
- Move the hero around the screen with joypad controls
- Animate the hero as he moves around
- Scroll the layer as the hero moves around

And this is exactly what you'll accomplish in this chapter! Your hero will be strutting his stuff in no time.

### Walking in style

It's easy enough to add a hero, but hardly worthwhile unless your hero looks alive rather than wooden – and that means animations.

In most 2D side-scrolling games, characters have various animations, each portraying an action. It's very easy to run an animation using Cocos2D actions, similar to how you made the main menu text blink with a Cocos2D action.

But knowing how to run animations isn't enough – you also need to know the proper time to play each animation. For example, when the hero is walking you want to play the walking animation, and when he's jumping you want to play the jumping animation.

### The all-important state machine

One possible answer is to use a state machine.

A state machine is simply something that changes behavior by switching states. A *single state machine can have only one state active at a time, but it can transition from one active state to another.*

To understand this better, imagine the base character for the game, and list the things s/he can do:

- Stay Put / Idle
- Walk
- Punch

Then list the requirements for each, assuming, for simplicity's sake, that only one activity can happen at a time:

- If s/he is idle, then s/he cannot be walking or punching.
- If s/he is walking, then s/he cannot be idle or punching.
- If s/he is punching, then he cannot be idle or walking.

Expand this list to include two more actions, those that the player cannot control – getting hurt and dying – and you have five base actions in total:



The hero has more actions than the five illustrated, but for the sake of this example, given these base actions, you could say that the character, if s/he were a state machine, would switch between an Idle State, a Walking State, a Punching State, a Hurt State and a Dead State.

To have complete flow between the states, each state should have a requirement and a result. The walking state cannot suddenly transition to the dead state, for example, since your hero must first go through being hurt in order to die.

The result of each state also helps solve the problem presented earlier. For the game's implementation, when you switch states, the character will also switch animations.

That's enough theory for now. It's time to continue coding!

## The Great Wall of Code

The first step is to create a base template for a sprite that is able to switch between actions.

**Note:** For the purposes of this tutorial, since each action represents a state, the words action and state will be used interchangeably.

First, you need to create some new definitions. Open **Defines.h** and add the following before the `#endif`:

```
typedef enum _ActionState
{
    kActionStateNone = 0,
    kActionStateIdle,
    kActionStateAttack,
    kActionStateAttackTwo,
    kActionStateAttackThree,
    kActionStateWalk,
    kActionStateRun,
    kActionStateRunAttack,
    kActionStateJumpRise,
    kActionStateJumpFall,
    kActionStateJumpLand,
    kActionStateJumpAttack,
    kActionStateHurt,
    kActionStateKnockedOut,
    kActionStateRecover,
    kActionStateDead,
    kActionStateAutomated,
} ActionState;

typedef struct _ContactPoint
{
    CGPoint position;
    CGPoint offset;
    CGFloat radius;
} ContactPoint;
```

Here you added an enumeration and a structure:

1. `ActionState` is an enumeration of the states that the class you're about to make can have. These are simply integers, with values from 0 to 16, but the enumeration names make your code much more readable.

2. `ContactPoint` is a structure of information you'll need to keep track of a circle shape. It includes a radius and an origin/position, which is computed from the position of the owner based on the value of the `offset` variable. You'll use this later for collision handling.

Now it's time to create your animated sprite class. Select the **PompaDroid** group in **Xcode**, go to **File\New\Group** and name the new group **Characters**.

Next, select the **Characters** group, go to **File\New\New File**, choose the **iOS\cocos2D v2.x\CCNode class** template, and click **Next**. Enter **CCSprite** for Subclass of, click **Next** and name the new file **ActionSprite**.

Prepare for a wall of code!

Go to **ActionSprite.h** and add the following before the `@end`:

```
//attachments
@property(nonatomic, strong)CCSprite *shadow;

//actions
@property(nonatomic, strong)id idleAction;
@property(nonatomic, strong)id attackAction;
@property(nonatomic, strong)id walkAction;
@property(nonatomic, strong)id hurtAction;
@property(nonatomic, strong)id knockedOutAction;
@property(nonatomic, strong)id recoverAction;
@property(nonatomic, strong)id runAction;
@property(nonatomic, strong)id jumpRiseAction;
@property(nonatomic, strong)id jumpFallAction;
@property(nonatomic, strong)id jumpLandAction;
@property(nonatomic, strong)id jumpAttackAction;
@property(nonatomic, strong)id runAttackAction;
@property(nonatomic, strong)id dieAction;

//states
@property(nonatomic, assign)ActionState actionState;
@property(nonatomic, assign)float directionX;

//attributes
@property(nonatomic, assign)float walkSpeed;
@property(nonatomic, assign)float runSpeed;
@property(nonatomic, assign)float hitPoints;
@property(nonatomic, assign)float attackDamage;
@property(nonatomic, assign)float jumpAttackDamage;
@property(nonatomic, assign)float runAttackDamage;
@property(nonatomic, assign)float maxHitPoints;
@property(nonatomic, assign)float attackForce;
```

This declares some basic properties for `ActionSprite`. Separated by section, these are:

1. **Attachments:** Any other object that is stuck to the `ActionSprite` like its shadow.
2. **Actions:** These are the `CCActions` (Cocos2D actions) to be executed for each state. The `CCActions` will be a combination of executing sprite animations and other events triggered when the character switches states. The `id` data type is a dynamic data type, meaning that you can put any object into it.
3. **States:** Holds the current action/state of the sprite using a type named `ActionState` that you will define soon. You also have a variable to determine if the sprite is facing left or right.
4. **Attributes:** Properties that define the character's statistics such as speed, hit/health points, damage and knockback force.

But wait, there's more!



Still in `ActionSprite.h`, add these lines before `@end`:

```
//movement
@property(nonatomic, assign)CGPoint velocity;
@property(nonatomic, assign)float jumpVelocity;
@property(nonatomic, assign)float jumpHeight;
@property(nonatomic, assign)CGPoint desiredPosition;
@property(nonatomic, assign)CGPoint groundPosition;

//measurements
@property(nonatomic, assign)float centerToSides;
@property(nonatomic, assign)float centerToBottom;

//collision
@property(nonatomic, assign)ContactPoint *contactPoints;
@property(nonatomic, assign)ContactPoint *attackPoints;
@property(nonatomic, assign)int contactPointCount;
@property(nonatomic, assign)int attackPointCount;
@property(nonatomic, assign)float detectionRadius;
```

```
- (CGRect)feetCollisionRect;
```

These are still more declarations for things that you will need later. You must be a forward-thinking person, huh?

1. **Movement:** These are dynamic values used to determine how the `ActionSprite` moves around the map.
2. **Measurements:** These hold useful measurement values regarding the actual image of the `ActionSprite`. You need these values because the sprites you will use have a canvas size that is much larger than the image contained inside.
3. **Collision:** These hold information related to collision detection for the `ActionSprite`.

There's still one last section to add before you move on to `ActionSprite.m`.



Again, add these lines to `ActionSprite.h`:

```
//action methods
-(void)idle;
-(void)attack;
-(void)hurtWithDamage:(float)damage force:(float)force
direction:(CGPoint)direction;
-(void)knockoutWithDamage:(float)damage
direction:(CGPoint)direction;
-(void)die;
-(void)recover;
-(void)getUp;
-(void)moveWithDirection:(CGPoint)direction;
-(void)runWithDirection:(CGPoint)direction;
-(void)walkWithDirection:(CGPoint)direction;
-(void)enterFrom:(CGPoint)origin to:(CGPoint)destination;
-(void)jumpRiseWithDirection:(CGPoint)direction;
-(void)jumpCutoff;
-(void)jumpFall;
```

```
- (void)jumpLand;
- (void)jumpAttack;
- (void)runAttack;
- (void)reset;

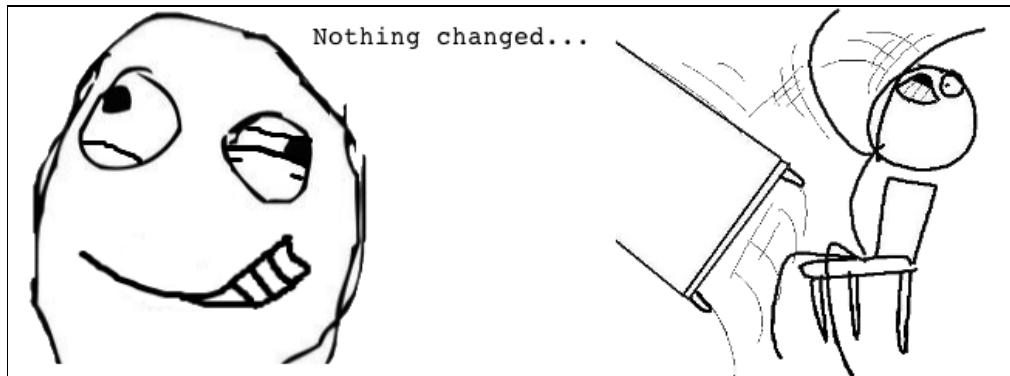
//contact point methods
- (void)modifyContactPointAtIndex:(const NSUInteger)pointIndex
    offset:(const CGPoint)offset radius:(const float)radius;
- (void)modifyAttackPointAtIndex:(const NSUInteger)pointIndex
    offset:(const CGPoint)offset radius:(const float)radius;
- (void)modifyPoint:(ContactPoint *)point offset:(const
    CGPoint)offset radius:(const float)radius;
- (ContactPoint)contactPointWithOffset:(const CGPoint)offset
    radius:(const float)radius;

//factory methods
- (CCAnimation *)animationWithPrefix:(NSString *)prefix
    startFrameIdx:(NSUInteger)startFrameIdx
    frameCount:(NSUInteger)frameCount delay:(float)delay;
```

Don't worry. All of this advanced planning will make your work smoother and faster later on! Treat it like an initial checklist of what needs to be defined.

1. **Action Methods:** You won't use the `ccActions` (from the actions section) directly. Instead you'll use these methods to trigger each state.
2. **Contact Point Methods:** These are helper methods to create and adjust collision points.
3. **Factory Methods:** These are methods that make it easier to create animations.

That's it. Build and run your code to make sure that everything still works as expected.



**Pompodour go!**

Stay with me now, and you'll see some results soon!

Select the **Characters** group, and go to **File\New\New File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **ActionSprite** for Subclass of, click **Next** and name the new file **Hero**.

Add this at the top of **Hero.h**:

```
#import "ActionSprite.h"
```

Now switch to **Hero.m** and add the following within the `@implementation` section:

```
- (id) init {
    if ((self = [super
initWithSpriteFrameName:@"hero_idle_00.png"])) {
        int i;

        //idle animation
        CCArray *idleFrames = [CCArray arrayWithCapacity:6];
        for (i = 0; i < 6; i++) {
            CCSpriteFrame *frame = [[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:[NSString
stringWithFormat:@"hero_idle_%02d.png", i]];
            [idleFrames addObject:frame];
        }
        CCAnimation *idleAnimation = [CCAnimation
animationWithSpriteFrames:[idleFrames getNSArray] delay:1.0/12.0];
        self.idleAction = [CCRepeatForever
actionWithAction:[CCAnimate actionWithAnimation:idleAnimation]];
    }
    return self;
}
```

You create the hero character with an initial sprite frame, prepare a **ccArray** containing all the other sprite frames belonging to the idle animation and create the **ccAction** that plays this animation.

Some important points to note:

1. You're initializing the hero sprite differently from how you did the sprites for the title scene. Instead of using a filename directly, you're using the name of the sprite's frame. You will learn more about this later – for now, just keep this in mind.
2. A **ccArray** is an indexed collection of objects. It's similar to **NSMutableArray**, but optimized for performance. In this case, you use it to keep track of the sprite frames in the idle animation.
3. A **ccAnimation** changes the displayed frame of a sprite using an array of sprite frames at every interval defined by the delay parameter. **1.0/12.0** means 12 frames per second.

4. **CCAnimate** is the action version of **CCAnimation**. Think of it this way: **CCAnimation** contains an animation's details, while **CCAnimate** is the action that allows a sprite to use a **CCAnimation**.

To test out your hero, switch to **GameLayer.h** and make the following changes:

```
//add to top of file
#import "Hero.h"

//add inside the curly braces
CCSpriteBatchNode *_actors;

//add before @end
@property(nonatomic, strong) Hero *hero;
```

Next switch to **GameLayer.m** and make the following changes:

```
//replace init
-(id)init
{
    if ((self = [super init]))
    {
        [[CCSpriteFrameCache sharedSpriteFrameCache]
        addSpriteFramesWithFile:@"sprites.plist"];
        _actors = [CCSpriteBatchNode
        batchNodeWithFile:@"sprites.pvr.ccz"];
        [_actors.texture setAliasTexParameters];
        [self addChild:_actors z:-5];
        [self initTileMap:@"map_level1.tmx"];
        [self initHero];
    }
    return self;
}

//add before @end
-(void)initHero
{
    self.hero = [Hero node];
    [_actors addChild:_hero];
    _hero.scale *= kScaleFactor;
    _hero.position = ccp(100 * kPointFactor, 100 * kPointFactor);
    [_hero idle];
}
```

You use **ccspriteFrameCache** to load the list of frames from the sprite sheet, which is an efficient way to make one image contain several images.

**Note:** We'll cover more about sprite sheets in the next section, but in the meantime if you'd like to learn more about sprite sheets here is a fun and educational video:

<http://www.codeandweb.com/what-is-a-sprite-sheet>

Any sprite frame that is in `ccSpriteFrameCache` can now be used by any sprite, which is why the hero could be instantiated using `initWithSpriteFrameName:` instead of `initWithFile:`.

You then create a batch node, add it to the layer and add the hero to it.

**Note:** A batch node's technical purpose is to reduce the amount of OpenGL draw calls (see the note earlier about Cocos2D diagnostic statistics). If you were to add five sprites to a layer, then there would be five more draw calls, but if you added those five sprites to the `ccSpriteBatchNode`, then there would only be one more draw call, and that would be for the `ccSpriteBatchNode` itself.

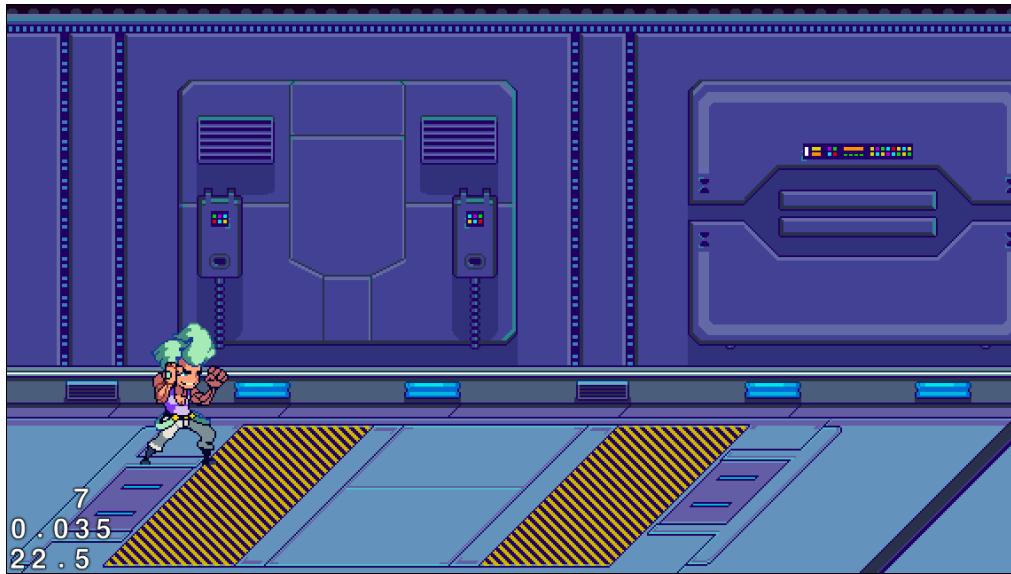
Of course, the limitation is that each of those five sprites must be using sprite frames from the same sprite sheet – the one with which you initialized the `CCSpriteBatchNode`.

Next you need to define what the `idle` method does. Go to `ActionSprite.m` and add the following inside `@implementation`:

```
-(void)idle
{
    if (_actionState != kActionStateIdle)
    {
        [self stopAllActions];
        [self runAction:_idleAction];
        _velocity = CGPointMakeZero;
        self.actionState = kActionStateIdle;
    }
}
```

If the `ActionSprite` isn't already in the idle state, it executes the idle action, changes the current action state to `kActionStateIdle`, and zeroes out the velocity.

Build and run, and you should now see your hero doing his idle animation:



## The almighty sprite sheet

Your hero is now in the house! Or at least on the game scene. Going from nothing to an animated sprite seems like magic if you don't know the origin of the content, so let's backtrack a bit.

The animation is all thanks to the sprite sheet that was part of the Images folder. The sprite sheet consists of `sprites.plist` and `sprites.pvr.cc2`.

But where did these two files come from, and what's their purpose, exactly?

These files were generated using [TexturePacker](#), a sprite sheet creation tool. For each sprite sheet, there should be a:

- 1. Texture File:** The texture atlas that contains all the sprite frames crammed together. In this case, it is **sprites.pvr.cc2**. The PVR image format (not PNG) is best used for the iPhone, and pvr.cc2 is just a compressed version of it.
- 2. Definition File:** The property list (**sprites.plist**) file contains information about each sprite frame, such as where the sprite frame is located in the texture and what its dimensions are.

If you want (and you have TexturePacker installed), open **Raw\Sprites.tps** to see how the sprite sheet for this game was made.

**Note:** Sprite sheets are most useful when you want to squeeze all the performance you can get from the device.

First, if you use batch nodes with sprite sheets, then you reduce the number of drawing calls. Second, you also are able to reduce the texture memory used

when each image's canvas size is a lot bigger than the actual drawing, by trimming out the blank spaces from the texture atlas.

Each frame is numbered in such a way that it is easy to add the sprite frames by their name using loops.

It's also important to know how each sprite frame was drawn.

In the Raw folder is a file named **HeroFrames.psd**. If you take a look at the drawing, the canvas size is bigger than the hero. Each frame for the hero was created on a 280x150-pixel canvas, but the actual drawing only takes up a fraction of that space.



The additional space is there to accommodate all of the hero's actions.

The basic idea is that all images in an animation need to be the same size, so that when you play an animation it's as simple as replacing the current animation frame with the next, like a flipbook.

Sometimes the character needs to move a bit in an animation – like in a jump animation or falling down animation. That's why you need to make the canvas bigger – so you can put the character at the right spot in those animations that require relative movement.

However, note that when the character is moving around the screen (like when you hold down the right button), you move the entire character canvas within the layer.

The animations should only cover the movements of the sprite relative to its current position – such as moving the character's head and limbs.

## Creating a directional pad

Now that your hero's on the scene, what's next?

The logical thing is to make the hero move around the map. Originally, Beat 'Em Ups were made with console gaming in mind, and most console games use

directional pads to control the player characters. So, for this Starter Kit, you will make your own virtual 8-directional animated D-pad to move the hero.

You'll start by creating the D-pad class. Select the **PompaDroid** group in Xcode, go to **File\New\Group** and name the new group **Controls**.

Next select the **Game** group, go to **File\New\New File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **CCSprite** for Subclass of, click **Next** and name the new file **ActionDPad**.

Open **ActionDPad.h** and replace its contents with the following:

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

typedef enum _ActionDPadDirection
{
    kActionDPadDirectionCenter = 0,
    kActionDPadDirectionUp,
    kActionDPadDirectionUpRight,
    kActionDPadDirectionRight,
    kActionDPadDirectionDownRight,
    kActionDPadDirectionDown,
    kActionDPadDirectionDownLeft,
    kActionDPadDirectionLeft,
    kActionDPadDirectionUpLeft
}ActionDPadDirection;

@class ActionDPad;

@protocol ActionDPadDelegate <NSObject>

-(void)actionDPad:(ActionDPad *)actionDPad
didChangeDirectionTo:(ActionDPadDirection)direction;
-(void)actionDPad:(ActionDPad *)actionDPad
isHoldingDirection:(ActionDPadDirection)direction;
-(void)actionDPadTouchUpInside:(ActionDPad *)actionDPad;

@end

@interface ActionDPad : CCSprite <CCTouchOneByOneDelegate>{
    ActionDPadDirection _previousDirection;
    float _radius;
    NSString *_prefix;
}

@property(nonatomic, assign)ActionDPadDirection direction;
```

```
@property(nonatomic, weak)id <ActionDPadDelegate> delegate;
@property(nonatomic, assign)BOOL isHeld;

+(id)dPadWithPrefix:(NSString *)filePrefix radius:(float)radius;
-(id)initWithPrefix:(NSString *)filePrefix radius:(float)radius;

@end
```

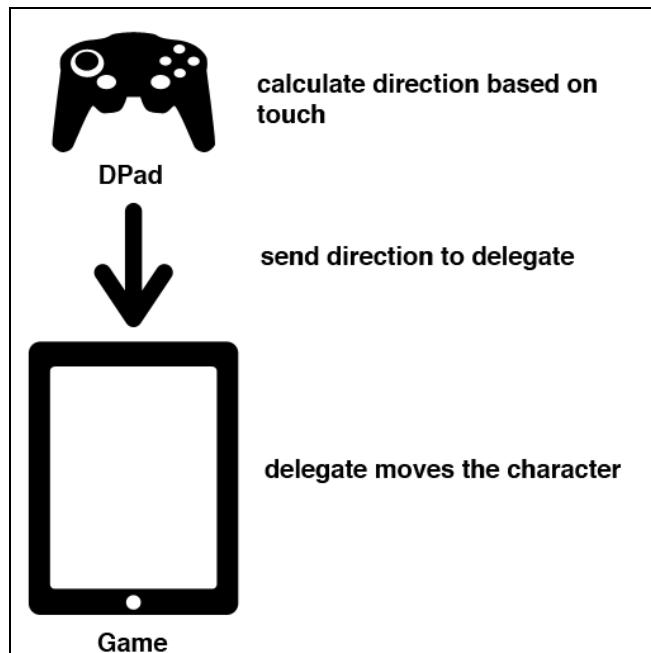
There are a lot of declarations above, but here are the important ones:

1. **ActionDPadDirection**: This value represents the D-pad's nine directions (including center). You store both the current and previously pressed direction of the D-pad.
2. **radius**: The radius of the circle formed by the D-pad.
3. **prefix**: You'll use the filename prefix for the D-pad image file for animating the D-pad.
4. **delegate**: The delegate of the D-pad, explained in detail below.
5. **isHeld**: A Boolean that returns YES as long as the player is touching the D-pad.

For the `ActionDPad`, you are going to use a coding design pattern called **delegation**. It means that a delegate class (other than `ActionDPad`) will handle some of the tasks started by the delegated class (`ActionDPad`). At certain points that you specify, `ActionDPad` will pass on responsibility to the delegate class, mostly when it comes to handling any game-related stuff.

This keeps `ActionDPad` ignorant of the game logic, thus allowing you to re-use it in any other game that you may want to develop!

This diagram shows you what happens when the user touches the D-pad:



When **ActionDPad** detects a touch that is within the radius of the D-pad, it calculates the direction (**ActionDPadDirection**) of that touch and sends a message to its delegate indicating the direction. Anything else after that is not the concern of **ActionDPad**.

To enforce this pattern, **ActionDPad** needs to at least know something about its delegate, specifically the methods to be called to pass the direction value to the delegate. This is where another design pattern comes in: **protocols**.

Go back to the code above and look at the section inside the `@protocol` block. This defines methods that any delegate of **ActionDPad** should have, and acts somewhat like an indirect header file for the delegate class. In this way, **ActionDPad** enforces its delegate to have the three specified methods, so that it can be sure that it can call any of these methods whenever it wants to pass something onto the delegate.

Actually, **ActionDPad** itself follows a protocol, as can be seen in this bit of code:

```
<CCTouchOneByOneDelegate>
```

The above is a protocol for making a touch-enabled class capable of claiming touches, disallowing any other class from receiving that touch. When the **ActionDPad** is touched, it should claim the touch so that **GameLayer** can't.

Now switch to **ActionDPad.m** and add the following inside `@implementation`:

```
+ (id)dPadWithPrefix:(NSString *)filePrefix radius:(float)radius
{
    return [[self alloc] initWithPrefix:filePrefix radius:radius];
}

-(id)initWithPrefix:(NSString *)filePrefix radius:(float)radius
{
    NSString *filename = [filePrefix
stringByAppendingString:@"_center.png"];

    if ((self = [super initWithSpriteFrameName:filename]))
    {
        _radius = radius;
        _direction = kActionDPadDirectionCenter;
        _isHeld = NO;
        _prefix = filePrefix;
        [self scheduleUpdate];
    }
    return self;
}
```

There's nothing new here – only a bunch of initialization methods. Just take note that you use `initWithSpriteFrameName:`, so expect to have a sprite sheet for the D-pad images.

**Note:** You may notice something weird about the `_direction` variable. For one thing, you never made an instance variable named `_direction` – you just declared a property named `direction` (note the missing underscore). What's more, usually declaring a property requires you to use the `@synthesize` directive in the implementation file so that the property generates getter and setter methods – but not here. Why does this work?

It's all because in the latest version of Xcode, properties are automatically synthesized. They also get their own instance variable, complete with accompanying underscore before the name, all without you having to write a line of code. For more information, check out Chapter 2 in [iOS 6 by Tutorials](#), "Modern Objective-C Syntax."

Pretty neat, right? Just remember that if you do things this way, the instance variable stays private to that class, such that even its subclasses won't be able to access it directly.

There are still a few exceptions to this rule, but unless you specifically get warnings or error messages from Xcode, you can assume that the above method is allowed (as long as you're using the latest Xcode).

Still in `ActionDPad.m`, add the following methods:

```
- (void)onEnterTransitionDidFinish
{
    [[[CCDirector sharedDirector] touchDispatcher]
    addTargetedDelegate:self priority:1 swallowsTouches:YES];
}

-(void) onExit
{
    [[[CCDirector sharedDirector] touchDispatcher]
    removeDelegate:self];
}

-(void)update:(ccTime)delta
{
    if (_isHeld)
    {
        [_delegate actionDPad:self isHoldingDirection:_direction];
    }
}
```

```
}
```

The first two methods register and remove `ActionDPad` as a delegate class that swallows (or claims) touches it receives, while `update:` constantly passes on the direction value to the delegate, as long as `ActionDPad` is being touched.

Don't leave `ActionDPad.m` just yet! You still need to add the following methods:

```
// 1
-(BOOL)ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
    CGPoint location = [[CCDirector sharedDirector]
convertToGL:[touch locationInView:[touch view]]];

    float distanceSQ = ccpDistanceSQ(location, _position);
    if (distanceSQ <= _radius * _radius)
    {
        //get angle 8 directions
        [self updateDirectionForTouchLocation:location];
        _isHeld = YES;
        return YES;
    }
    return NO;
}

// 2
-(void)ccTouchMoved:(UITouch *)touch withEvent:(UIEvent *)event
{
    CGPoint location = [[CCDirector sharedDirector]
convertToGL:[touch locationInView:[touch view]]];
    [self updateDirectionForTouchLocation:location];
}

// 3
-(void)ccTouchEnded:(UITouch *)touch withEvent:(UIEvent *)event
{
    _direction = kActionDPadDirectionCenter;
    _isHeld = NO;
    [self setDisplayFrame: [[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:[NSString
stringWithFormat:@"%@_center.png", _prefix]]];
    [_delegate actionDPadTouchEnded:self];
}

// 4 & 5
-(void)updateDirectionForTouchLocation:(CGPoint)location
```

```
{  
    float radians = ccpToAngle(ccpSub(location, _position));  
    float degrees = -1 * CC_RADIANS_TO_DEGREES(radians);  
    NSString *suffix = @"_center";  
  
    _previousDirection = _direction;  
  
    if (degrees <= 22.5 && degrees >= -22.5)  
    {  
        _direction = kActionDPadDirectionRight;  
        suffix = @"_right";  
    }  
    else if (degrees > 22.5 && degrees < 67.5)  
    {  
        _direction = kActionDPadDirectionDownRight;  
        suffix = @"_downright";  
    }  
    else if (degrees >= 67.5 && degrees <= 112.5)  
    {  
        _direction = kActionDPadDirectionDown;  
        suffix = @"_down";  
    }  
    else if (degrees > 112.5 && degrees < 157.5)  
    {  
        _direction = kActionDPadDirectionDownLeft;  
        suffix = @"_downleft";  
    }  
    else if (degrees >= 157.5 || degrees <= -157.5)  
    {  
        _direction = kActionDPadDirectionLeft;  
        suffix = @"_left";  
    }  
    else if (degrees < -22.5 && degrees > -67.5)  
    {  
        _direction = kActionDPadDirectionUpRight;  
        suffix = @"_upright";  
    }  
    else if (degrees <= -67.5 && degrees >= -112.5)  
    {  
        _direction = kActionDPadDirectionUp;  
        suffix = @"_up";  
    }  
    else if (degrees < -112.5 && degrees > -157.5)  
    {  
        _direction = kActionDPadDirectionUpLeft;
```

```
        suffix = @"_upleft";
    }
    [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:[NSString
stringWithFormat:@"%@%@.png", _prefix, suffix]]];

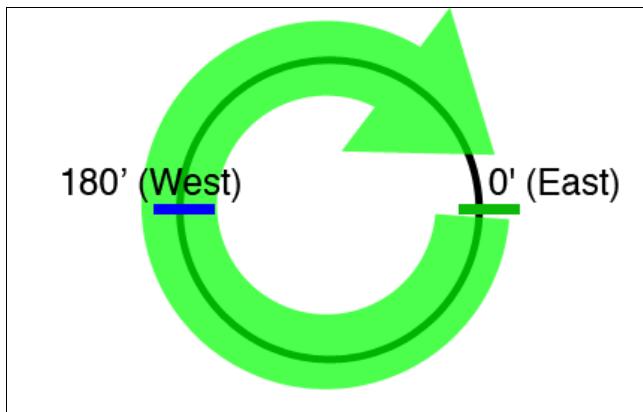
    if (_isHeld)
    {
        if (_previousDirection != _direction)
        {
            [_delegate actionDPad:self
didChangeDirectionTo:_direction];
        }
    }
    else
    {
        [_delegate actionDPad:self
didChangeDirectionTo:_direction];
    }
}
```

Don't be intimidated by the long `if-else` block; it's very simple once broken down:

1. `ccTouchBegan` checks if the touch location is inside the D-pad's circle. If yes, it switches on the `_isHeld` Boolean and triggers an update to the direction value. It also returns `YES` to claim the touch.
2. `ccTouchMoved` simply triggers an update of the direction value every time the touch is moved.
3. `ccTouchEnded` switches off the `_isHeld` Boolean, centers the direction, resets the displayed frame to the center frame and notifies the delegate that the touch has ended.
4. `updateDirectionForTouchLocation` calculates the location of the touch against the center of the D-pad by getting the angle of difference between the two. It then assigns the correct value for direction based on the resulting angle. This direction value is passed on to the delegate whenever a new direction is selected.
5. `updateDirectionForTouchLocation` also sets a suffix that corresponds to the filename of each sprite frame. There is a different image for each pressed direction, plus the neutral direction (nine images in all). `setDisplayFrame:` changes the displayed frame for the sprite by combining the prefix and suffix value taken from the direction and appending the ".png" extension.

The angle values, in degrees, may look weird to you, since the common misconception is that 0 degrees means north.

Take a look at this circle:



This illustration should help you understand why in mathematics, 0 degrees is east of the circle, becoming positive in the counterclockwise direction (opposite to the arrow direction in the diagram). Since you multiply the angle by -1, it becomes positive in the clockwise direction (to match the diagram).

This means if the touch happens in an angle between -22.5 to 22.5 degrees, then the touch was on the right side of the D-pad, and so on.

OK, that's the D-pad class implemented. Now you need to add it to your game and use it. The D-pad needs to be on top of everything else, so it has to be added to the Heads Up Display (HUD).

Open **HudLayer.h** and make the following changes:

```
//add to top of file
#import "ActionDPad.h"

//add after the closing curly bracket but before the @end
@property(nonatomic, weak)ActionDPad *dPad;
```

Switch to **HudLayer.m**, and add the following method:

```
-(id)init
{
    if ((self = [super init]))
    {
        [[CCSpriteFrameCache sharedSpriteFrameCache]
        addSpriteFramesWithFile:@"joypad.plist"];

        float radius = 64.0 * kPointFactor;
        _dPad = [ActionDPad dPadWithPrefix:@"dpad" radius:radius];
        _dPad.position = ccp(radius, radius);
        _dPad.opacity = 128;
        [self addChild:_dPad];
    }
}
```

```
    return self;
}
```

The code instantiates an **ActionDPad** and sets its radius. The radius is multiplied by **kPointFactor** so that it automatically converts for iPad.

Before creating the **ActionDPad**, you load the **joypad.plist** sprite sheet. This sprite sheet contains the D-pad image files. If you look inside, you'll see that each direction's filename has a prefix of "dpad".

One noticeable difference between the D-pad and the hero sprite is that you didn't multiply the former's scale by **kScaleFactor**. This is because your D-pad art isn't drawn in the pixel art style, and so instead you have a sprite sheet for each device resolution:

- **joypad.plist** for iPhone non-retina.
- **joypad-hd.plist** for iPhone retina and iPad non-retina.
- **joypad-ipadhd.plist** for iPad retina.

By default, Cocos2D will look for **joypad-ipad.plist** for the iPad version, but in this case, you want it to use **joypad-hd.plist** for both the retina iPhone and non-retina iPad. Cocos2D has a fallback mechanism: when it is unable to find the file for a certain resolution, it will look for the file that is one resolution lower.

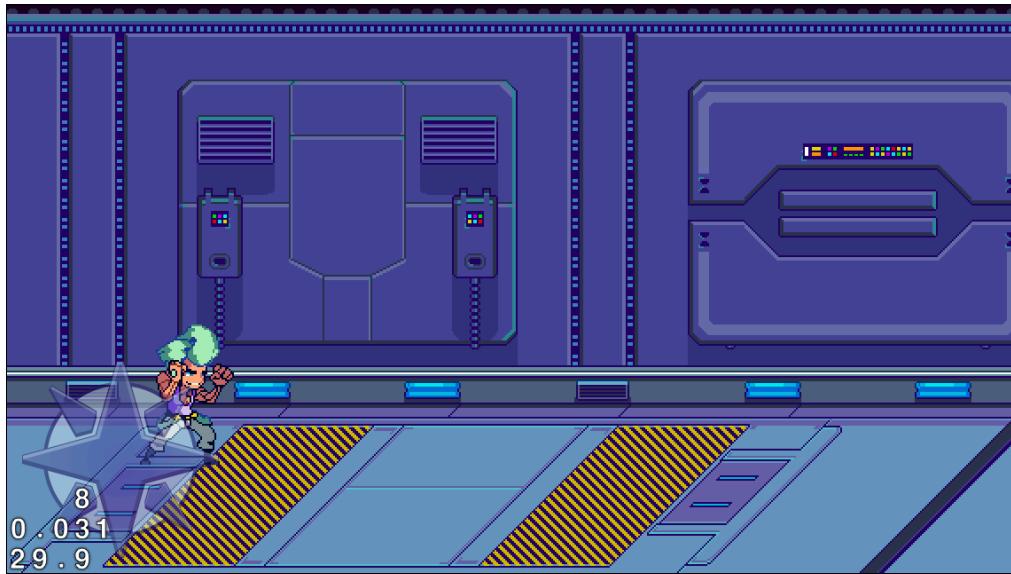
However, this mechanism is turned off by default, so go to **AppDelegate.m** and look for this line:

```
[sharedFileUtils setEnableFallbackSuffixes:NO];
```

Change it to:

```
[sharedFileUtils setEnableFallbackSuffixes:YES];
```

That's it. You already added the **HudLayer** to **GameScene** in one of the earlier sections, so just build and run your code to see:



If you try pressing the D-pad, you will see different areas shaded based on where you touch. Neat!

## Moving the hero

So what now? You have a hero and a D-pad. Is that it for this chapter?

This chapter is called “Walk This Way” for a reason, and that’s because your final goal is to walk the hero across the map. Get ready for the final stretch – of this chapter, at least!

### He's got the moves

The first step is to create a movement state for the hero.

The process of creating an animation action for movement is very similar to that of creating the idle animation action. In fact, all other actions will take the same route.

To avoid retyping the same code over and over, it would be better to have a sort of factory method for producing animations. You already declared one earlier:

`animationWithPrefix:startFrameIdx:frameCount:delay:`. Now it’s time to implement it.

Go to **ActionSprite.m** and add this method before `@end`:

```
- (CCAnimation *)animationWithPrefix:(NSString *)prefix
startFrameIdx:(NSUInteger)startFrameIdx
frameCount:(NSUInteger)frameCount delay:(float)delay
{
```

```
int idxCount = frameCount + startFrameIdx;
CCArray *frames = [CCArray arrayWithCapacity:frameCount];
int i;
CCSpriteFrame *frame;
for (i = startFrameIdx; i < idxCount; i++)
{
    frame = [[CCSpriteFrameCache sharedSpriteFrameCache]
spriteFrameByName:[NSString stringWithFormat:@"%@_%02d.png",
prefix, i]];
    [frames addObject:frame];
}

return [CCAnimation animationWithSpriteFrames:[frames
getNSArray] delay:delay];
}
```

This method is very similar to how you created the idle animation. The difference is that this one is reusable because of its parameters.

It creates an animation with sprite frames that have the format prefix\_XX, where XX is a two-digit number starting at `startFrameIdx` and incremented `frameCount` times.

With this method, you no longer have to create a `CCArray` and a `for` loop every time you want to create an animation.

Switch to `Hero.m` and replace `init` with the following:

```
-(id)init {
    if ((self = [super
initWithSpriteFrameName:@"hero_idle_00.png"])) {
        //idle animation
        CCAnimation *idleAnimation = [self
animationWithPrefix:@"hero_idle" startFrameIdx:0 frameCount:6
delay:1.0/12.0];
        self.idleAction = [CCRepeatForever
actionWithAction:[CCAnimate actionWithAnimation:idleAnimation]];

        //walk animation
        CCAnimation *walkAnimation = [self
animationWithPrefix:@"hero_walk" startFrameIdx:0 frameCount:8
delay:1.0/12.0];
        self.walkAction = [CCRepeatForever
actionWithAction:[CCAnimate actionWithAnimation:walkAnimation]];

        self.walkSpeed = 80 * kPointFactor;
        self.directionX = 1.0;
```

```
    }
    return self;
}
```

This changes the syntax of the idle animation to use the method you just created. Additionally, it uses the same method to quickly create the walk animation. It also defines the walk speed of the hero and his starting direction.

Go back to **ActionSprite.m** and add the following methods:

```
-(void)walkWithDirection:(CGPoint)direction
{
    if (_actionState == kActionStateIdle || _actionState ==
kActionStateRun)
    {
        [self stopAllActions];
        [self runAction:_walkAction];
        self.actionState = kActionStateWalk;
        [self moveWithDirection:direction];
    }
    else if (_actionState == kActionStateWalk)
    {
        [self moveWithDirection:direction];
    }
}

-(void)moveWithDirection:(CGPoint)direction
{
    if (_actionState == kActionStateWalk)
    {
        _velocity = ccp(direction.x * _walkSpeed, direction.y *
_walkSpeed);
        [self flipSpriteForVelocity:_velocity];
    }
    else if (_actionState == kActionStateRun)
    {
        _velocity = ccp(direction.x * _runSpeed, direction.y *
_walkSpeed);
        [self flipSpriteForVelocity:_velocity];
    }
    else if (_actionState == kActionStateIdle)
    {
        [self walkWithDirection:direction];
    }
}
```

**walkWithDirection:** triggers the walk action if the previous state was idle or run. If the previous state was already a walk state, then it simply passes on responsibility to **moveWithDirection:**.

**moveWithDirection:** sets the velocity of the sprite by multiplying a vector (direction) by the sprite's movement speed. It also calls **flipSpriteForVelocity:**, which you will define next.

Still in **ActionSprite.m**, add the new method:

```
- (void)flipSpriteForVelocity:(CGPoint)velocity
{
    if (velocity.x > 0)
    {
        self.directionX = 1.0;
    }
    else if (velocity.x < 0)
    {
        self.directionX = -1.0;
    }

    self.scaleX = _directionX * kScaleFactor;
}
```

**flipSpriteForVelocity:** simply sets the value of **directionX** to 1.0 (east), or to -1.0 (west), and multiplies it by **kScaleFactor** to determine the **scaleX** value for the sprite.

**Note:** In Cocos2D, a negative scale will reverse the drawing of the sprite.

## Taking directions

Now that you've got the methods in place, you want to call on them when the D-pad is pressed. Right now, though, there's no connection between the D-pad and the hero.

To fix this, go to **GameLayer.h** and do the following:

```
//add to top of file
#import "HudLayer.h"

//add in between @interface GameLayer : CCLayer and the opening
//curly bracket
<ActionDPadDelegate>

//add after the closing curly bracket and before the @end
```

```
@property(nonatomic, weak)HudLayer *hud;
```

You just added a weak reference to a `HudLayer` instance within `GameLayer`. You also made `GameLayer` follow the protocol created by `ActionDPad`.

Now you can tie it all together within `GameScene`! So go to `GameScene.m` and do the following:

```
//add to init inside if ((self = [super init])) right after [self
addChild:_hudLayer z:1]
hudLayer.dPad.delegate = gameLayer;
gameLayer.hud = hudLayer;
```

The code simply makes `GameLayer` the delegate of `HudLayer`'s `ActionDPad`, and also connects `HudLayer` to `GameLayer`.

Now go to `GameLayer.m` and add the delegate methods:

```
-(void)actionDPad:(ActionDPad *)actionDPad
didChangeDirectionTo:(ActionDPadDirection)direction
{
    CGPoint directionVector = [self vectorForDirection:direction];
    [_hero walkWithDirection:directionVector];
}

-(void)actionDPad:(ActionDPad *)actionDPad
isHoldingDirection:(ActionDPadDirection)direction
{
    CGPoint directionVector = [self vectorForDirection:direction];
    [_hero moveWithDirection:directionVector];
}

-(void)actionDPadTouchUpInside:(ActionDPad *)actionDPad
{
    if (_hero.actionState == kActionStateWalk || _hero.actionState
== kActionStateRun)
    {
        [_hero idle];
    }
}
```

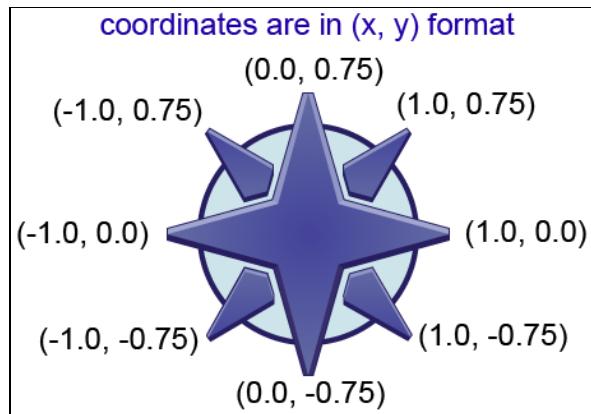
You trigger the hero's `walkWithDirection:` and `moveWithDirection:` methods every time the `ActionDPad` is pressed, and trigger the hero's `idle` method every time the touch on `ActionDPad` stops.

ActionDPad sends an enum value for direction, but ActionSprite expects a normal vector value (-1.0 to 1.0 in x and y coordinates), so you use a conversion method, which you will write next.

Still in **GameLayer.m**, add this method:

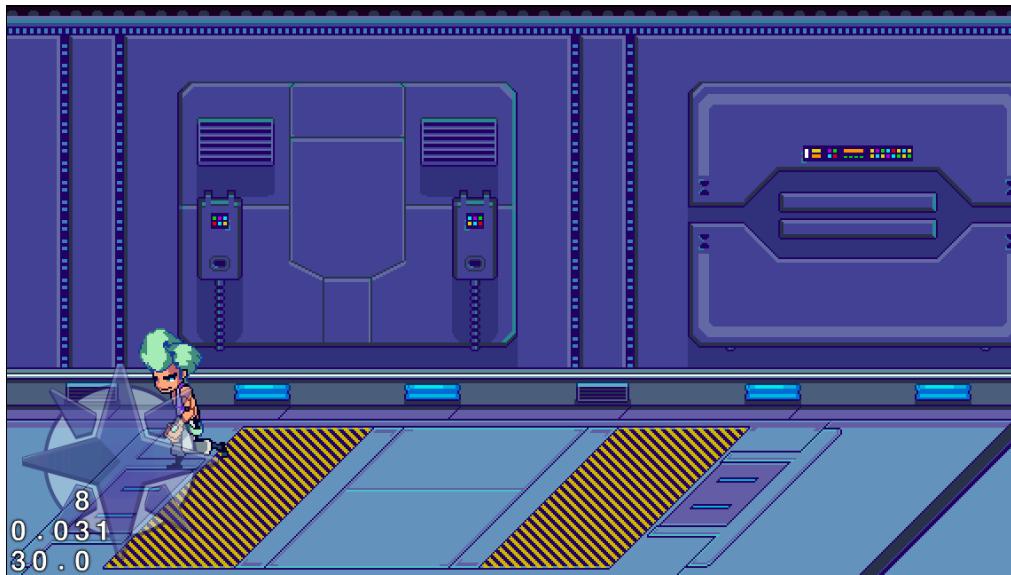
```
- (CGPoint)vectorForDirection:(ActionDPadDirection)direction
{
    float maxX = 1.0;
    float maxY = 0.75;
    switch (direction) {
        case kActionDPadDirectionCenter:
            return CGPointMakeZero;
            break;
        case kActionDPadDirectionUp:
            return ccp(0.0, maxY);
            break;
        case kActionDPadDirectionUpRight:
            return ccp(maxX, maxY);
            break;
        case kActionDPadDirectionRight:
            return ccp(maxX, 0.0);
            break;
        case kActionDPadDirectionDownRight:
            return ccp(maxX, -maxY);
            break;
        case kActionDPadDirectionDown:
            return ccp(0.0, -maxY);
            break;
        case kActionDPadDirectionDownLeft:
            return ccp(-maxX, -maxY);
            break;
        case kActionDPadDirectionLeft:
            return ccp(-maxX, 0.0);
            break;
        case kActionDPadDirectionUpLeft:
            return ccp(-maxX, maxY);
            break;
        default:
            return CGPointMakeZero;
            break;
    }
}
```

To understand this method, take a look at the following diagram:



For each direction, the method just assigns the corresponding x and y value. The maximum value for the y-axis is capped at 0.75 to make the vertical movement slower than the horizontal movement. You can change this if you want. Making the values higher will make the hero move faster, while making the values lower will have the opposite effect.

Build and run, and try moving the hero using the D-pad.



All right, he's walking! Er, wait a minute... he's not actually moving. What gives?

## A little sprite navigation

Take a look at `moveWithDirection:` again, and you'll notice that it doesn't do anything except change the velocity of the hero. Where is the code for changing the hero's position?

Changing the hero's position is the responsibility of both `ActionSprite` and `GameLayer`. An `ActionSprite` never really knows where it is located on the map. Hence, it doesn't know when it has reached the map's edges, or when it has

collided with other objects. It only knows where it wants to go – the desired position.

It is **GameLayer**'s responsibility to translate that desired position into an actual position.

First go to **ActionSprite.m** and add this method:

```
- (void)update:(ccTime)delta
{
    if (_actionState == kActionStateWalk)
    {
        _desiredPosition = ccpAdd(_position, ccpMult(_velocity,
delta));
    }
}
```

This method should be called every time the game updates the scene, and it in turn updates the desired position of the sprite only when the sprite is in the walking state. It adds the value of velocity to the current position of the sprite, but before that, velocity is multiplied by delta time so that the time interval is factored into the equation.

Multiplying by delta time makes the hero move at the same rate, no matter the current frame rate. Position + Velocity \* Delta Time really just means, “move x and y (velocity) points each second (1 delta).”

**Note:** This way of integrating position is called Euler's Integration. It's known for being an approach that's easy to understand and implement, though not as one that is extremely accurate. But since this isn't a physics simulation, Euler's Integration is close enough for your purposes.

Before **GameLayer** can check if the hero is colliding with the map's bounds, it needs to know the bounds of the hero himself.

Go to **Hero.m** and add the following:

```
//add inside the curly braces of if ((self = [super
initWithSpriteFrameName:@"hero_idle_00.png"]))
self.centerToBottom = 39.0 * kPointFactor;
self.centerToSides = 29.0 * kPointFactor;
```

Then go to **ActionSprite.m** and add this method:

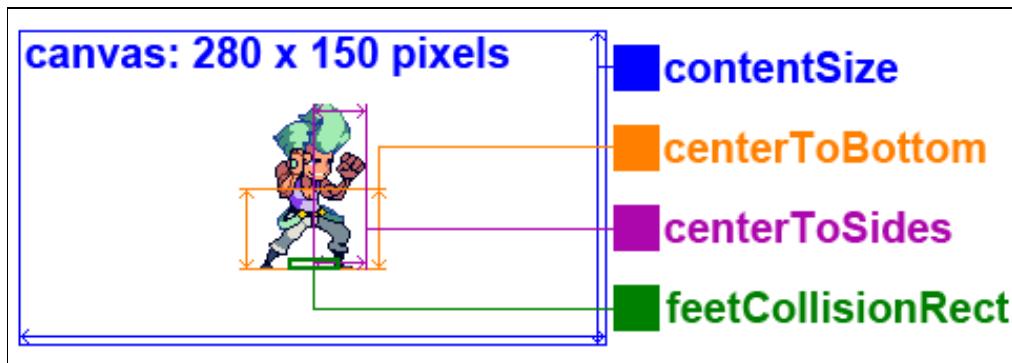
```
-(CGRect)feetCollisionRect
{
```

```
    CGRect feetRect = CGRectMake(_desiredPosition.x -  
    _centerToSides, _desiredPosition.y - _centerToBottom,  
    _centerToSides * 2, 5.0 * kPointFactor);  
    return CGRectInset(feetRect, 15.0 * kPointFactor, 0);  
}
```

In Hero.m, you define some measurements from the center of the sprite to the sides and bottom, and then in ActionSprite.m, you define a rectangle that represents the colliding part of each sprite's feet.

**Note:** You could have defined `feetCollisionRect` in Hero.m. In this Starter Kit, you're putting it in ActionSprite.m because all characters will use the same definition.

To better understand these measurements, take a look at the following diagram:



Each frame of the hero sprite was created on a 280x150 pixel canvas, but the actual hero sprite only takes up a fraction of that space. Each ccsprite has a property named `contentSize` that gives you the size of the frame. `contentSize` is useful in cases where the sprite takes up the whole frame, but not very useful here. Therefore it's a good idea to store some measurements so that you can easily position the sprite – in this case, the hero.

The last value in the above diagram, `feetCollisionRect`, represents a rectangle for, as the name implies, the feet of the character.

The above code uses `centerToSides` and `centerToBottom` to define a starting rectangle, and uses `CGRectInset` to shrink it by 15 points in width (7.5 points from each side) so that there can be a bit of overlap allowance. The rectangle's height is 5 because that's my estimation of the height (or width, from the hero's perspective) of his feet.

**Note:** A lot of the things you define, like the `feetCollisionRect` method, you declared earlier, when you first created `ActionSprite`. This way, you won't

need to go back and forth between ActionSprite.h and ActionSprite.m unless you have to add something totally new.

Switch to **GameLayer.m** and make the following changes:

```
//add to init inside if ((self = [super init])) right after [self
initHero];
[self scheduleUpdate];

//add these methods inside @implementation and before @end
-(void)dealloc
{
    [self unscheduleUpdate];
}

-(void)update:(ccTime)delta
{
    [_hero update:delta];
    [self updatePositions];
}

-(void)updatePositions
{
    float mapWidth = _tileMap.mapSize.width *
_tileMap.tileSize.width * kPointFactor;
    float floorHeight = 3 * _tileMap.tileSize.height *
kPointFactor;
    float posX = MIN(mapWidth -
_hero.feetCollisionRect.size.width/2,
MAX(_hero.feetCollisionRect.size.width/2,
_hero.desiredPosition.x));
    float posY = MIN(floorHeight + (_hero.centerToBottom -
_hero.feetCollisionRect.size.height), MAX(_hero.centerToBottom,
_hero.desiredPosition.y));
    _hero.position = ccp(posX, posY);
}
```

First you schedule `GameLayer`'s `update:` method, which acts as the main run loop for the game. Here you will see how `GameLayer` and `ActionSprite` cooperate in setting `ActionSprite`'s position.

For every loop, `GameLayer` asks the hero to update its desired position, and then it takes that desired position and checks if it is within the bounds of the tile map's floors by using these values:

- **mapSize:** This is the number of tiles in the tile map. There are 12x100 tiles total, but only 3x100 for the floor.

- **tileSize:** This contains the dimensions of each tile, 32x32 points in this case. You multiply by **kPointFactor** so that it scales up for iPad.

GameLayer also makes a lot of references to `ActionSprite`'s measurement values. If an `ActionSprite` wants to stay within the scene, its position shouldn't go past the actual sprite bounds. Remember that the canvas you're using for the sprites is much bigger than the actual sprite area, in at least some cases.

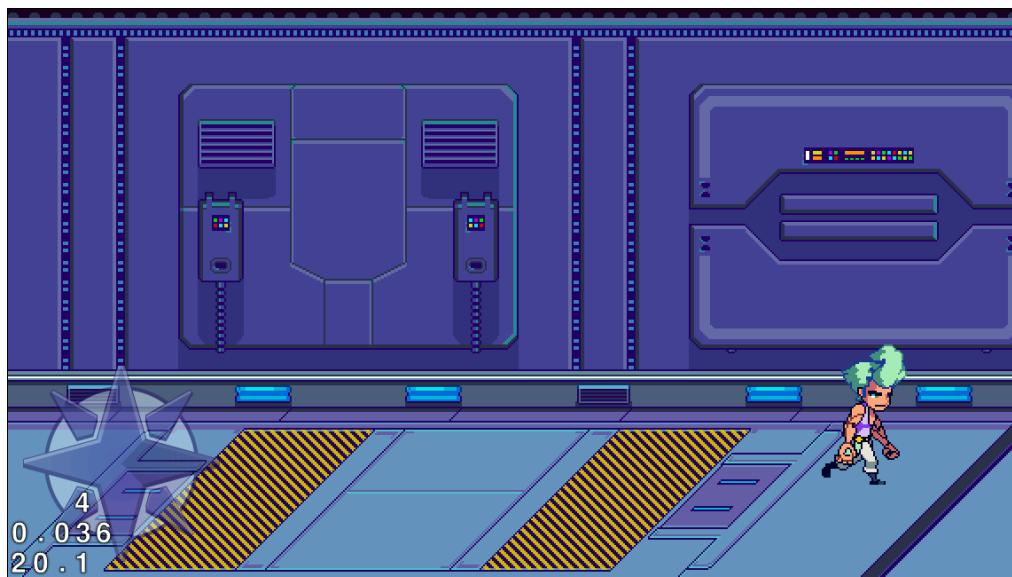
If you take it per side, it's quite simple:

- The left and right sides of the sprite's `feetCollisionRectangle` cannot exceed the leftmost point of the map (0.0 x-coordinate) and the rightmost point of the rightmost floor tile, respectively.
- The highest point of the sprite's `feetCollisionRectangle` cannot exceed the highest point on the highest floor tile, while the lowest point of the sprite cannot exceed the lowest point of the map (0.0 y-coordinate).

If the position of the `ActionSprite` is within the boundaries that have been set, GameLayer gives it the desired position. If not, GameLayer asks it to stay in its current position.

**Note:** The `MIN` function compares two values and returns the lower value, while the `MAX` function returns the higher of two values. Using these two in conjunction clamps a value to a minimum and maximum number. Cocos2D also comes with a convenience function for `cgPoints` that is similar to the above: `ccpClamp`.

Build and run, and you should now be able to move your hero across the map.



## Staying on camera

You'll notice two things:

1. The hero now begins at the lower left of the map instead of at (100, 100). This is because the value of `desiredPosition` is still (0, 0), and `GameLayer` positions the hero accordingly.
2. The hero can walk past the right edge of the map, vanishing from view.

Ignore the first point for now (we'll address that in another chapter) and focus on the second one since it's more important. To make the camera seem to follow the hero, you can scroll the whole `GameLayer` based on the hero's position.

Still in `GameLayer.m`, make the following changes:

```
//add this in update:(ccTime)dt, right after [self
updatePositions];
[self setViewpointCenter:_hero.position];

//add this method
-(void)setViewpointCenter:(CGPoint) position {
    int x = MAX(position.x, CENTER.x);
    int y = MAX(position.y, CENTER.y);
    x = MIN(x, (_tileMap.mapSize.width * _tileMap.tileSize.width *
kPointFactor)
        - CENTER.x);
    y = MIN(y, (_tileMap.mapSize.height * _tileMap.tileSize.height
* kPointFactor)
        - CENTER.y);
    CGPoint actualPosition = ccp(x, y);

    CGPoint viewPoint = ccpSub(CENTER, actualPosition);
    self.position = viewPoint;
}
```

This code centers the screen on the hero's position except when he's at the edge of the map. There is no real camera movement here since you are just moving `GameLayer`. Of course, when `GameLayer` is moved, all of its children are taken along for the ride.

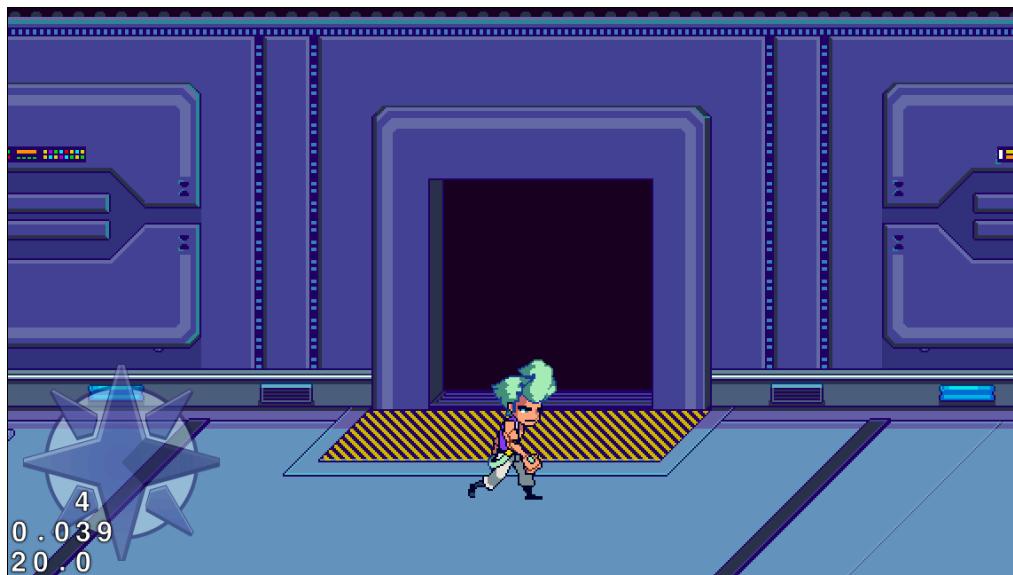
The `CENTER` macro, you might recall, simply returns half of the screen's dimensions, and not the actual center of the entire map. Subtracting from `CENTER` makes sure that the "camera" doesn't move out of bounds.

Confused? OK, pretend the image below shows the entire map:



If the position of the camera were less than `CENTER`, then part of the view would be off the screen, hence the need to adjust using `CENTER`.

Build and run. The hero should now be visible at all times and the background will scroll as the hero walks towards the right edge of the screen (or the left edge if he's walked more than half a screen-width to the right).



All right, this is really starting to feel like a real game!

Now's a great time to take another break and review what you've done. In this chapter, you've accomplished a lot. You've:

- Added your hero to the scene.
- Created a state machine to track and control your characters' activities.
- Created a directional pad to move your hero.
- Animated your hero to stand and walk.

But if the structure of `ActionSprite` is any indication, this is just the tip of the iceberg. After all, this is supposed to be a Beat 'Em Up Game, but right now there is nothing to beat up!

Stay tuned for the next chapter, where the fists will begin to fly!

**Challenge:** Try replacing the idle and walk animations with your own hand-drawn images. It's OK if you aren't a great artist – stick figures will do fine!

You can find the raw images inside **Raw\Sprites\Hero**, and you can replace those with your own versions. You can then build the sprite sheet with Texture Packer, using **Sprites.tps**.

Maybe you could make an animated version of your friend, family member, or favorite celebrity. Your imagination is the only limit!

# Chapter 3: Running, Jumping, and Punching

So far your hero can move around – but he's a bit slow for an action hero, don't you think? It's time to add some pep into his step and add the ability for him to run.

Then, you'll add the typical controls for a Beat 'Em Up Game – jumping and punching.

And of course, you gotta give something for our hero to punch! So you'll learn how to add some enemies into the scene, and color them in a dynamic fashion.

By the time you're done with this chapter, your hero will be ready for the fight of his life!

## Run sprite, run!

In the last chapter, you left the hero all alone in a long empty corridor with nothing to do but walk. It gets boring pretty fast.

You could increase the hero's `walkSpeed` attribute to make him move faster, but there's a better option – allow him to run!

Open **ActionSprite.m** and take a quick look at `moveWithDirection:` again:

```
- (void)moveWithDirection:(CGPoint)direction
{
    if (_actionState == kActionStateWalk)
    {
        _velocity = ccp(direction.x * _walkSpeed, direction.y * _walkSpeed);
        [self flipSpriteForVelocity:_velocity];
    }
    else if (_actionState == kActionStateRun)
    {
```

```

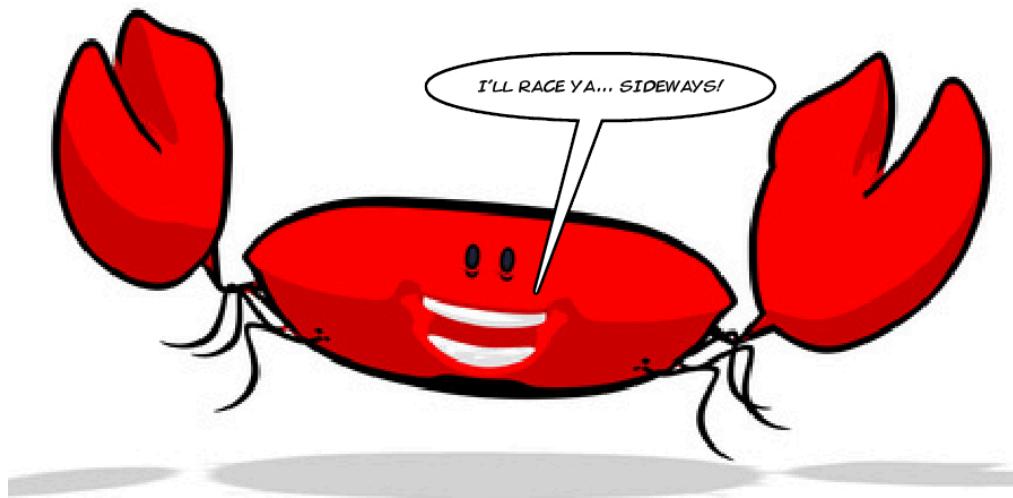
        _velocity = ccp(direction.x * _runSpeed, direction.y *
_walkSpeed);
        [self flipSpriteForVelocity:_velocity];
    }
    else if (_actionState == kActionStateIdle)
    {
        [self walkWithDirection:direction];
    }
}

```

You may have noticed something peculiar about this method when you created it originally. In the second `if` statement, you already check for the run state!

The run action works similarly to the walk action, and the two differ only in speed, animation and trigger condition. Of course, if the sprite is running, it is expected to move at a faster rate than when it's walking.

To achieve this, instead of using the `walkSpeed` variable, you use `runSpeed`. You only apply it to straight (forward and backward) movement, since nobody can run sideways, with the exception of crabs.



Still in **ActionSprite.m**, make the following changes:

```

// Add this new method
-(void)runWithDirection:(CGPoint)direction
{
    if (_actionState == kActionStateIdle || _actionState ==
kActionStateWalk)
    {
        [self stopAllActions];
        [self runAction:_runAction];
        self.actionState = kActionStateRun;
    }
}

```

```

        [self moveWithDirection:direction];
    }

//replace the if statement inside update:(ccTime)delta with this
if (_actionState == kActionStateWalk || _actionState ==
kActionStateRun)

```

Similar to `walkWithDirection:`, `runWithDirection:` ensures that the previous state was either `kActionStateIdle` or `kActionStateWalk`. If yes, it triggers the run action, sets the state correctly and passes on the responsibility to `moveWithDirection::`.

Then in `update::`, you also allow `desiredPosition` to be changed when the current state is `kActionStateRun`.

Next go to **Hero.m** and make the following changes:

```

//add inside the curly braces of if ((self = [super
initWithSpriteFrameName:@"hero_idle_00.png"])), right after
self.walkAction

//run animation
CCAnimation *runAnimation = [self animationWithPrefix:@"hero_run"
startFrameIdx:0 frameCount:8 delay:1.0/12.0];
self.runAction = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:runAnimation]];

self.runSpeed = 160 * kPointFactor;

```

You create the run animation action with a range of sprite frames at 12 frames per second. Then you set the hero's `runSpeed` to be double the value of `walkSpeed`.

Now that the hero knows what to do when the player wants him to run, you merely need to trigger the run action. The D-pad already triggers the hero to walk, so you have to come up with some intuitive way to make him run with the same controls.

Here's the behavior you'll implement:

- If the player taps the left or right arrows twice quickly, then the hero will run.
- If the hero is running and the player suddenly changes his direction, then he should stop running and start walking.

**GameLayer** will control both of these rules, as it is the mediator between the hero and the D-pad. To achieve this, **GameLayer** needs to store two things:

- The time interval between taps. If the interval is within an allotted time, then the previous and current taps are counted as a double tap.
- The previous direction pressed. If the previous and current directions pressed are the same, then the hero should run.

You can use these two variables combined to determine if the same direction arrow was tapped twice.

Go to **GameLayer.h** and add the following instance variables:

```
float _runDelay;
ActionDPadDirection _previousDirection;
```

`runDelay` will store the window of opportunity for running. As long as it contains a positive non-zero time value, the hero can run. `previousDirection` simply stores the last direction returned by the D-pad.

Switch to **GameLayer.m** and make the following changes:

```
//add inside update: right after [self updatePositions]
if (_runDelay > 0)
{
    _runDelay -= delta;
}

//replace this method
-(void)actionDPad:(ActionDPad *)actionDPad
didChangeDirectionTo:(ActionDPadDirection)direction
{
    CGPoint directionVector = [self vectorForDirection:direction];

    // 1
    if (_runDelay > 0 && _previousDirection == direction &&
(direction == kActionDPadDirectionRight || direction ==
kActionDPadDirectionLeft))
    {
        [_hero runWithDirection:directionVector];
    }
    // 2
    else if (_hero.actionState == kActionStateRun &&
abs(_previousDirection - direction) <= 1)
    {
        [_hero moveWithDirection:directionVector];
    }
    // 3
    else
    {
        [_hero walkWithDirection:directionVector];
        _previousDirection = direction;
        _runDelay = 0.2;
    }
}
```

You make sure that `runDelay` counts down to 0 by decrementing the delta time at every game update. Then you make three significant changes to how you handle D-pad direction changes:

1. Check if the same direction was selected in the allowed time interval and call `runWithDirection:` if so.
2. Allow the hero to run in his current direction only if the direction is the same. You'll revisit this in a bit.
3. Make the hero walk, either because it is the first time the hero moved, or because he was running but abruptly changed direction. Whenever the hero walks, you store the previous direction and set a new time window for running, which is 0.2 seconds.

The second `if` statement looks weird in how it determines if the direction has changed or not. It gets the difference between the previous and current directions and decides that if the difference is 1 or less, then the hero's moving in the same direction.



To make sense of this, open **ActionDPad.h** and check the definition of the `ActionDPadDirection` enumeration:

```
typedef enum _ActionDPadDirection
{
    kActionDPadDirectionCenter = 0,
    kActionDPadDirectionUp,
    kActionDPadDirectionUpRight,
    kActionDPadDirectionRight,
    kActionDPadDirectionDownRight,
    kActionDPadDirectionDown,
    kActionDPadDirectionDownLeft,
    kActionDPadDirectionLeft,
    kActionDPadDirectionUpLeft
}ActionDPadDirection;
```

An enumeration is simply a bunch of numbers grouped together using words. It helps make code readable, and helps limit the numbers that you are allowed to use for a given situation. The first number, 0, is given the keyword `kActionDPadDirectionCenter`. If the succeeding keywords don't explicitly specify a number, then their value is automatically set to the previous value plus one.

This means `kActionDPadDirectionUp` gets a value of 1, `kActionDPadDirectionUpRight` is 2, `kActionDPadDirectionRight` is 3 and so on.

Label your D-pad with these numbers and you will have:

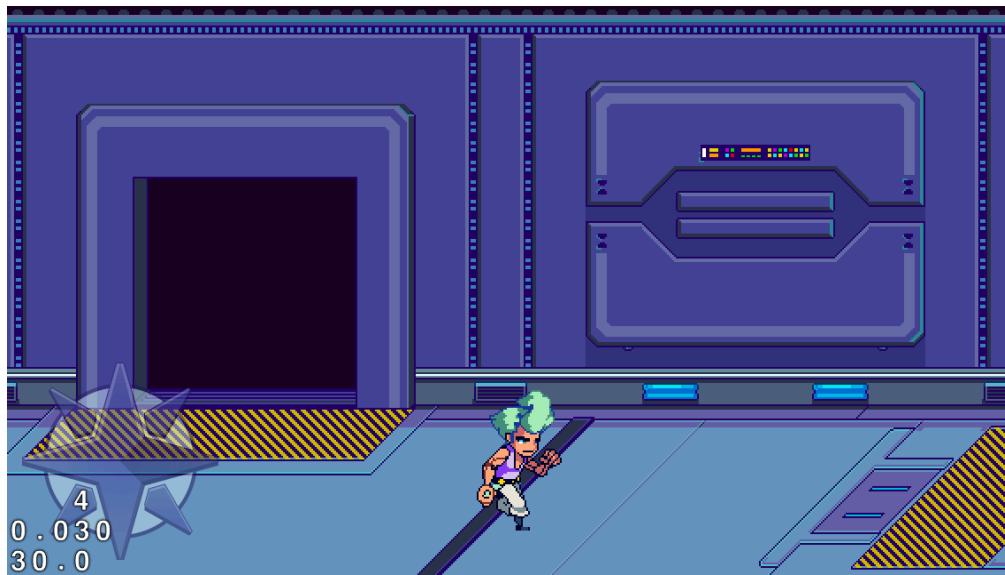


If the hero is moving forward, or to the right, then the possible direction values are 2, 3 and 4. If the hero is moving backward, or to the left, then the direction values are 6, 7 and 8.

`previousDirection` only stores the first direction pressed, and for the run sequence to even be considered, this direction has to either be left (value 7) or right (value 3). Since the diagonal directions for each of these sides only differ by 1, you just check if the difference between the previous and current direction is less than or equal to 1.

If running left, `previousDirection` will have a value of 7, and if `currentDirection` has a value other than 6, 7 or 8, then the hero stops running. If running right, `previousDirection` will be 3, and if `currentDirection` has a value other than 2, 3 or 4, then the hero also stops running.

That's it! Build and run, literally!



## Adding animated buttons

You've pretty much exhausted all possible functions of the D-pad. So before moving on to creating more actions for the hero, you need to have some additional control mechanisms to trigger these actions.

No game controller is complete without buttons, so next you'll create some to accompany the D-pad!

Select the **Controls** group in Xcode, go to **File\New\New File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **CCSprite** for Subclass of, click **Next** and name the new file **ActionButton**.

Open **ActionButton.h** and replace its contents with the following:

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

@class ActionButton;

@protocol ActionButtonDelegate <NSObject>

-(void)actionButtonWasPressed:(ActionButton *)actionButton;
-(void)actionButtonIsHeld:(ActionButton *)actionButton;
-(void)actionButtonWasReleased:(ActionButton *)actionButton;

@end

@interface ActionButton : CCSprite <CCTouchOneByOneDelegate> {
```

```
    float _radius;
    NSString *_prefix;
}

@property(nonatomic, weak)id <ActionButtonDelegate> delegate;
@property(nonatomic, assign)BOOL isHeld;

+(id)buttonWithPrefix:(NSString *)filePrefix radius:(float)radius;
-(id)initWithPrefix:(NSString *)filePrefix radius:(float)radius;

@end
```

This is very similar to how you set up `ActionDPad` – much simpler, even! You declare protocol methods that `ActionButton`'s delegate should implement. This way, `ActionButton` can be reused in any other game you want to create.

**Note:** Please refer to the “Creating a Directional Pad” section of Chapter 2 for a more detailed explanation of the delegation pattern.

You also create the following variables:

- **radius:** The radius of the action button. You'll use this to determine if the button was touched.
- **prefix:** The prefix of the sprite filename for the button. You'll use this when animating the action button.
- **delegate:** `ActionButton` will notify the delegate class when it is pressed, held or released.
- **isHeld:** A Boolean that returns `true` as long as the action button is being touched.

Finally, you declare initializer methods for creating new `ActionButton` instances.

Switch to `ActionButton.m` and add these methods after `@implementation` and before `@end`:

```
+ (id)buttonWithPrefix:(NSString *)filePrefix radius:(float)radius
{
    return [[self alloc] initWithPrefix:filePrefix radius:radius];
}

-(id)initWithPrefix:(NSString *)filePrefix radius:(float)radius
{
    NSString *filename = [filePrefix
    stringByAppendingString:@"_normal.png"];

    if ((self = [super initWithSpriteFrameName:filename]))
```

```

    {
        _radius = radius;
        _prefix = filePrefix;
        _isHeld = NO;
        [self scheduleUpdate];
    }
    return self;
}

-(void)update:(ccTime)delta
{
    if (_isHeld)
    {
        [_delegate actionButtonIsHeld:self];
    }
}
}

```

You initialize a new **ActionButton** with a sprite frame whose name is equal to the string formed by combining the prefix string and “\_normal.png”. If the prefix string were “button”, then the sprite frame’s name would be “button\_normal.png”.

After storing the important values to the instance variables, you schedule the update: method. update: effectively calls the delegate’s **actionButtonIsHeld:** method as long as the button is being touched.

Still in **ActionButton.m**, add the following methods:

```

-(void)onEnterTransitionDidFinish
{
    [[[CCDirector sharedDirector] touchDispatcher]
    addTargetedDelegate:self priority:1 swallowsTouches:YES];
}

-(void) onExit
{
    [[[CCDirector sharedDirector] touchDispatcher]
    removeDelegate:self];
}

-(BOOL)ccTouchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
    CGPoint location = [[CCDirector sharedDirector]
    convertToGL:[touch locationInView:[touch view]]];

    float distanceSQ = ccpDistanceSQ(location, _position);
    if (distanceSQ <= _radius * _radius)
}

```

```

{
    _isHeld = YES;
    [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:[NSString
stringWithFormat:@"%@_selected.png", _prefix]]];
    [_delegate actionButtonWasPressed:self];
    return YES;
}
return NO;
}

-(void)ccTouchEnded:(UITouch *)touch withEvent:(UIEvent *)event
{
    _isHeld = NO;
    [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:[NSString
stringWithFormat:@"%@_normal.png", _prefix]]];
    [_delegate actionButtonWasReleased:self];
}

```

As in `ActionDPad`, the first two methods register and remove `ActionButton` as a delegate class that claims the touches it receives.

`ccTouchBegan:` checks if the touch location is within the bounds of the button. If the button was touched, then the sprite frame of the button changes, `_isHeld` is turned on and the delegate's `actionButtonWasPressed:` is called.

Once the touch leaves the button, `ccTouchEnded:` turns off the `_isHeld` Boolean, switches the sprite frame back to the original and calls the delegate's `actionButtonWasReleased:`.

With this, you now have a reliable and reusable action button to accompany your D-pad. You will be adding two buttons: the classic A and B. One button will be used to make the hero attack, while the other will be used to make the hero jump.

To add these two buttons, go first to `HudLayer.h` and make the following changes:

```

//add to top of file
#import "ActionButton.h"

//add after the closing curly bracket but before the @end
@property(nonatomic, weak)ActionButton *buttonA;
@property(nonatomic, weak)ActionButton *buttonB;

```

The code simply declares two `ActionButton` instances: `buttonA` and `buttonB`.

Go to `Defines.h` to add some new definitions that you will need:

```
//add after #define PompaDroid_Defines_h and before #endif
#define kTagButtonA 1
#define kTagButtonB 2
```

These two values will help distinguish the two buttons from one another later.

Switch to **HudLayer.m** and make the following changes:

```
//add inside curly braces of if ((self = [super init])) after
[self addChild:_dPad];
float buttonRadius = radius / 2.0;
float padding = 8.0 * kPointFactor;

_buttonB = [ActionButton buttonWithType:@"button_b"
radius:buttonRadius];
_buttonB.position = ccp(SCREEN.width - buttonRadius - padding,
buttonRadius * 2 + padding );
_buttonB.opacity = 128;
_buttonB.tag = kTagButtonB;
[self addChild:_buttonB];

_buttonA = [ActionButton buttonWithType:@"button_a"
radius:buttonRadius];
_buttonA.position = ccp(_buttonB.position.x - radius - padding,
buttonRadius + padding);
_buttonA.opacity = 128;
_buttonA.tag = kTagButtonA;
[self addChild:_buttonA];
```

You create and add two buttons using two different images, and then position them diagonally beside each other. This is similar to how you added ActionDPad, save for one tiny detail: the tag.

You are using the `tag` property to hold a unique identifying number for each object. This will be especially important later because the delegate will need to distinguish between these two buttons when it implements the protocol methods. The D-pad doesn't need to have a tag value because there is only one D-pad in the scene.

Build and run, and you should see your two action buttons on the screen:



It'll take a few more coding maneuvers before these buttons see some action, which is what you'll take up next.

## A simple jab attack

This wouldn't be much of a Beat 'Em Up Game if the hero didn't have the means to beat up enemies, right? 😊 This is where the two action buttons you added come in.

For now, you will make the hero do a quick jab or a weak punch, but later you will expand this to a 1-2-3 punch combo!

To create new actions, just follow the same procedure you used to create the idle, walk and run actions. You create an action's method in `ActionSprite` and prepare the sprite frames and `ccAction` in `Hero`.

Do the latter first. Go to `Hero.m` and make the following changes:

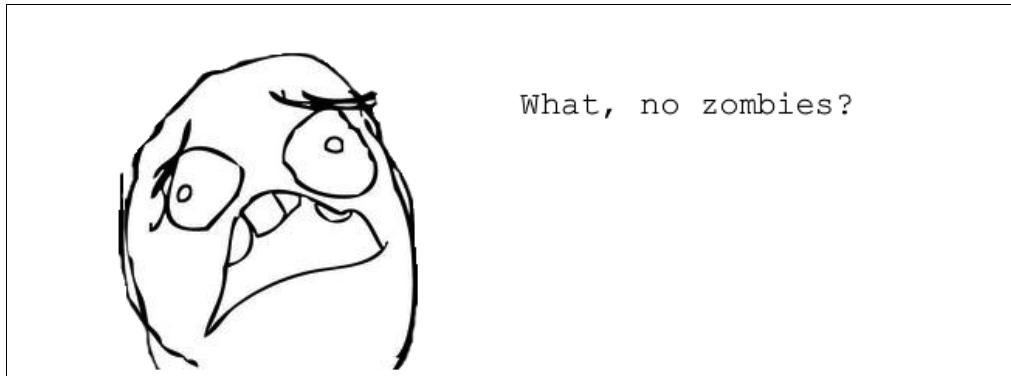
```
//add right after self.runAction = [CCRepeatForever
actionWithAction:[CCAnimate actionWithAnimation:runAnimation]];
CCAnimation *attackAnimation = [self
animationWithPrefix:@"hero_attack_00" startFrameIdx:0 frameCount:3
delay:1.0/15.0];
self.attackAction = [CCSequence actions:[CCAnimate
actionWithAnimation:attackAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];
```

There are differences here compared to before. The attack animation runs faster than the idle animation at 15 frames per second. Plus, the attack action only animates the attack animation once, and quickly switches back to the idle state by calling `idle`.

Go to **ActionSprite.m** and add the following method:

```
- (void)attack
{
    if (_actionState == kActionStateIdle || _actionState ==
kActionStateWalk || _actionState == kActionStateAttack)
    {
        [self stopAllActions];
        [self runAction:_attackAction];
        self.actionState = kActionStateAttack;
    }
}
```

Here you see a few more restrictions to the attack action than to the idle action. The hero can only attack if his previous state was idle, attack or walk. This is to ensure that the hero will not be able to attack while he's being hurt or when he's dead.



After the initial checks, the code changes the action state to `kActionStateAttack` and executes the attack action.

To trigger the attack, go to **GameLayer.h** and make the following changes:

```
//change <ActionDPadDelegate> to
<ActionDPadDelegate, ActionButtonDelegate>
```

Switch to **GameLayer.m** and add these methods:

```
- (void)actionButtonWasPressed:(ActionButton *)actionButton
{
    if (actionButton.tag == kTagButtonA)
    {
        [_hero attack];
    }
}
```

```
- (void)actionButtonIsHeld:(ActionButton *)actionButton{  
}  
  
- (void)actionButtonWasReleased:(ActionButton *)actionButton{  
}
```

You make `GameLayer` implement the `ActionButtonDelegate` protocol so that it can act as a delegate for the buttons. You then implement the protocol methods that `GameLayer`, as `ActionButton`'s delegate, needs to implement. `actionButtonIsHeld:` and `actionButtonWasReleased:` are executed when any button is held or released, respectively.

You just add empty methods for the time being since they are not optional. You will flesh out these methods later.



If either of the two buttons is pressed, then you execute `actionButtonWasPressed:`. You have to know which button is being pressed, so you use the `tag` property that you set earlier. The hero will execute his `attack` method when the player presses the A button.

To complete the connection between `GameLayer` and the buttons, go to `GameScene.m` and make the following changes:

```
//add to init inside if ((self = [super init])) right after  
gameLayer.hud = hudLayer  
hudLayer.buttonA.delegate = gameLayer;  
hudLayer.buttonB.delegate = gameLayer;
```

You set `GameLayer` as the buttons' delegate. When either button is pressed, the appropriate `ActionButton` instance can use this `delegate` property to see who to pass the action to (the `GameLayer`).

Before you build your game, go to `AppDelegate.m` and quickly make the following changes:

```
//add this inside -(BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions, right  
after CCGLView *glView is created  
[glView setMultipleTouchEnabled:YES];
```

This enables multiple touches for your game. One touch is for the directional pad, while the other touch is for the buttons. Without this, only one touch at any given time would be detected, and a player couldn't attack while walking, for example.

Build and run, and jab away!

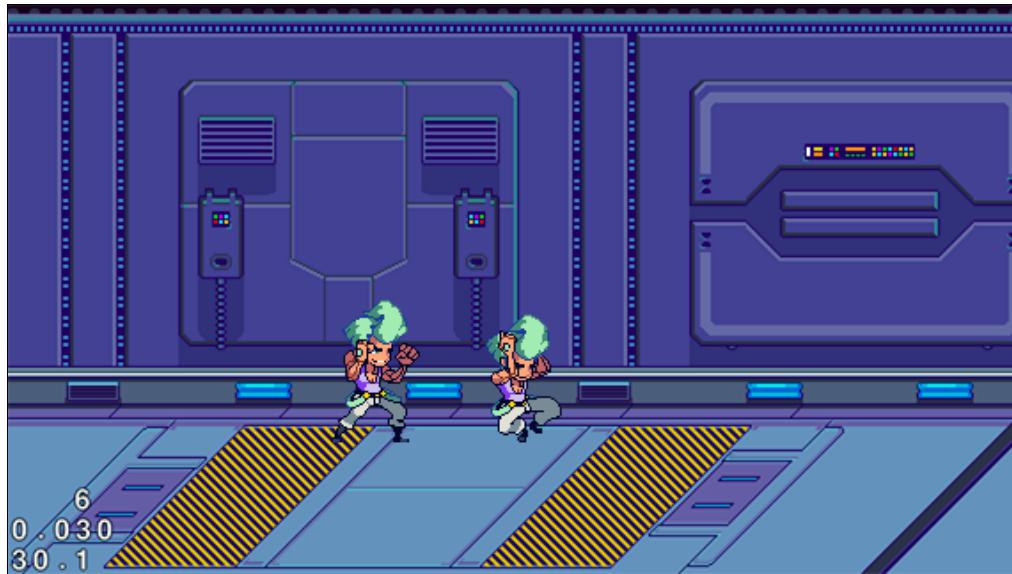


## Jump up and get down!

With the game's current 2D isometric-like perspective, making a character jump is a little tricky. In a 2D game, there are only two dimensions, x and y. The x-axis is already used to move the sprite left and right on the screen, while the y-axis already handles the up and down movement.

Jumping should move the character higher, so the only choice is to move the character's position higher along the y-axis. However, if you do this, it will look as if he just walked up instead of jumping into the air.

Picture these two characters on the map:



The character on the right is jumping, but he looks like he is just squatting on the same plane as the character on the left. It also looks like these two characters should collide when they bump into each other, but they should not. What's a sprite to do?

## The shadow effect

Visually, this is easy to remedy: you can add some fake three-dimensional depth to the scene by using simple shadows. The shadows will always be positioned on the ground position of the character – for the standing character, this is right below his feet, for the jumping one, it is farther down.

With shadows, the scene above will look like this:



When you add shadows, these two characters no longer appear to be on the same y-coordinate, so it's more obvious that the character to the right is jumping. Plus you get a rough idea as to where he really is on a two-dimensional plane.

Back in the first chapter, aside from the jump action itself, you added several variables to `ActionSprite` to help you achieve this depth effect:

- **shadow:** The circular shadow sprite. Moves independently of the `ActionSprite`.
- **groundPosition:** The current ground position of the `ActionSprite`. This is where you'll place the shadow.
- **jumpHeight:** The current height of the jump. This will help decide the sprite's y-coordinate position.
- **jumpVelocity:** The velocity relative to the jump (e.g., how fast the character rises when jumping).

Since you've already created the code that positions the hero on the screen, you're going to need to refactor in some of these changes. One big change is how the sprite's position is decided on the screen.

First, set up some constant values that you'll need. Open `Defines.h` and add the following:

```
#define kGravity 1000.0 * kPointFactor
#define kJumpForce 340.0 * kPointFactor
#define kJumpCutoff 130.0 * kPointFactor
#define kPlaneHeight 7.0 * kPointFactor
```

For jumping, you'll use a very simple simulation of forces. When the hero jumps, his velocity starts with an initial jumping force, `kJumpForce`. There's also a gravity force, `kGravity`, which constantly drags down the hero's velocity. This way, the hero leaps up with a high velocity and then gradually slows down until he eventually falls back down to the ground.

The height of the hero's jump will depend on how long the player holds onto the jump button. `kJumpForce` allows the hero to reach the peak of his jump, while `kJumpCutoff` allows him to at least always reach the minimum height of his jump. You will see more about how this dynamic works in a short while.

Go to `ActionSprite.m` and add these methods:

```
-(void)setGroundPosition:(CGPoint)groundColor
{
    _groundColor = groundPosition;
    _shadow.position = ccp(groundColor.x, groundPosition.y -
_centerToBottom);
}

-(void)jumpRiseWithDirection:(CGPoint)direction
```

```

{
    if (_actionState == kActionStateIdle)
    {
        [self jumpRise];
    }
    else if (_actionState == kActionStateWalk || _actionState ==
kActionStateJumpLand)
    {
        _velocity = ccp(direction.x * _walkSpeed, direction.y *
_walkSpeed);
        [self flipSpriteForVelocity:_velocity];
        [self jumpRise];
    }
    else if (_actionState == kActionStateRun)
    {
        _velocity = ccp(direction.x * _runSpeed, direction.y * *
_walkSpeed);
        [self flipSpriteForVelocity:_velocity];
        [self jumpRise];
    }
}

-(void)jumpRise
{
    if (_actionState == kActionStateIdle || _actionState ==
kActionStateWalk || _actionState == kActionStateRun ||
_actionState == kActionStateJumpLand)
    {
        [self stopAllActions];
        [self runAction:_jumpRiseAction];
        _jumpVelocity = kJumpForce;
        self.actionState = kActionStateJumpRise;
    }
}
}

```

You override the `setGroundPosition:` method and make sure that the shadow's position is always relative to the `groundPosition` of the ActionSprite.

Then you create the two jump methods:

- 1. `jumpRiseWithDirection:`**: Decides the appropriate jump action based on the current state. If the state is idle, then the sprite just jumps vertically. If the sprite is moving, then the sprite jumps forward or backward depending on the direction. The move velocity also differs if the sprite was walking, running or landing when it jumped.

**2. jumpRise:** This simply sets the `jumpVelocity` to the `kJumpForce` constant, changes the state to `kActionStateJumpRise` and runs the `jumpRiseAction` action.

Unlike other action states, jumping will cover multiple states:

- `kActionStateJumpRise` for when the `ActionSprite` lifts off from the ground and rises;
- `kActionStateJumpFall` for when the `ActionSprite` starts falling down until it hits the ground; and
- `kActionStateJumpLand` for the exact moment the `ActionSprite` lands.

For the first state, you have the `jumpRise` methods. To complete the jump sequence, you still have to handle falling and landing.

Still in `ActionSprite.m`, add the following methods:

```
- (void)jumpCutoff
{
    if (_actionState == kActionStateJumpRise)
    {
        if (_jumpVelocity > kJumpCutoff)
        {
            _jumpVelocity = kJumpCutoff;
        }
    }
}

-(void)jumpFall
{
    if (_actionState == kActionStateJumpRise || _actionState ==
kActionStateJumpAttack)
    {
        self.actionState = kActionStateJumpFall;
        [self runAction:_jumpFallAction];
    }
}

-(void)jumpLand
{
    if (_actionState == kActionStateJumpFall || _actionState ==
kActionStateRecover)
    {
        _jumpHeight = 0;
        _jumpVelocity = 0;
        self.actionState = kActionStateJumpLand;
        [self runAction:_jumpLandAction];
    }
}
```

```
}
```

`jumpCutoff` checks if the current jump velocity is greater than the cutoff, and if so, it instantly reduces it to the cutoff value. This allows the player to interrupt the jump by releasing the jump button and make the sprite fall quicker.

`jumpFall` simply switches the state to `kActionStateJumpFall` and executes `jumpFallAction` only when the sprite is rising or attacking in the air.

`jumpLand` zeroes out both `jumpHeight` and `jumpVelocity`, changes the state to `kActionStateJumpLand` and executes `jumpLandAction`.

Both `groundColor` and `jumpHeight` have been introduced, but they do not yet affect the actual position of the sprite on the screen. Currently, both `ActionSprite` and the `GameLayer` determine an `ActionSprite`'s position, so there should be changes in both.

In `ActionSprite.m`, replace `update:` with the following:

```
- (void)update:(ccTime)delta
{
    if (_actionState == kActionStateWalk || _actionState ==
kActionStateRun)
    {
        _desiredPosition = ccpAdd(_groundColor,
ccpMult(_velocity, delta));
    }
    else if (_actionState == kActionStateJumpRise)
    {
        _desiredPosition = ccpAdd(_groundColor,
ccpMult(_velocity, delta));
        _jumpVelocity -= kGravity * delta;
        _jumpHeight += _jumpVelocity * delta;

        if (_jumpVelocity <= kJumpForce/2)
        {
            [self jumpFall];
        }
    }
    else if (_actionState == kActionStateJumpFall)
    {
        _desiredPosition = ccpAdd(_groundColor,
ccpMult(_velocity, delta));
        _jumpVelocity -= kGravity * delta;
        _jumpHeight += _jumpVelocity * delta;

        if (_jumpHeight <= 0)
```

```
        {
            [self jumpLand];
        }
    }
```

The change here is simple. When the hero is walking or running, instead of moving `desiredPosition` based on the position variable, `ActionSprite` now moves it based on the `groundPosition` value.

When the hero is rising during a jump, his `jumpHeight` increases by the `jumpVelocity` for every frame. To make sure that his upward velocity slows down and that the hero eventually falls, his `jumpVelocity` is also decreased by gravity at every frame.

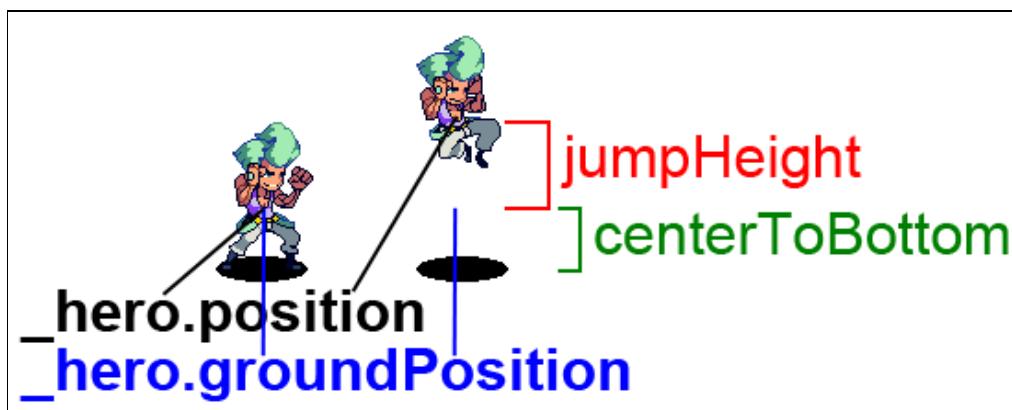
Once `jumpVelocity` falls below half of its original value, then the `jumpFall` method is executed.

When the hero is falling, his `jumpVelocity` and `jumpHeight` continue to deplete, and this goes on until `jumpHeight` goes down to zero, when the hero transfers control to the `jumpLand` method.

Switch to `GameLayer.m` and make the following changes in `updatePositions`:

```
//replace _hero.position = ccp(posX, posY) with the following:
_hero.groundPosition = ccp(posX, posY);
_hero.position = ccp(_hero.groundPosition.x,
_hero.groundPosition.y + _hero.jumpHeight);
```

To better visualize what is happening, check out this diagram of the hero in his `idle` and `jumpRise` states:



Instead of directly affecting position, which dictates the onscreen coordinates of the hero, the code now just influences `groundPosition`. The onscreen position is then a combination of both the `groundPosition` and `jumpHeight` of the hero.

This effectively splits the 2D left/right/up/down movement of the hero from his pseudo 3D vertical jumping movement. Additionally, the shadow is always placed below the `groundPosition` using the `centerToBottom` measurement you defined in the first chapter.

## B is for jump

The next thing to do is to map the B button to the hero's jump action.

Still in `GameLayer.m`, replace `actionButtonWasPressed:` and `actionButtonWasReleased:` with the following:

```
- (void)actionButtonWasPressed:(ActionButton *)actionButton
{
    if (actionButton.tag == kTagButtonA)
    {
        [_hero attack];
    }
    else if (actionButton.tag == kTagButtonB)
    {
        CGPoint directionVector = [self
vectorForDirection:_hud.dPad.direction];
        [_hero jumpRiseWithDirection:directionVector];
    }
}

-(void)actionButtonWasReleased:(ActionButton *)actionButton
{
    if (actionButton.tag == kTagButtonB)
    {
        [_hero jumpCutoff];
    }
}
```

If the player presses the A button, the hero attacks as before, but now when the player presses the B button, the hero jumps in the direction of the D-pad. Once the B button is released, the hero cuts off his jump.

While here, you might as well add the hero's shadow to the scene so that it gets drawn.

Go to `initHero` and add this line:

```
//add before [_actors addChild:_hero]
[_actors addChild:_hero.shadow];
_hero.shadow.scale *= kScaleFactor;
```

You add the shadow to the batch node before the hero himself. This is to make sure that the shadow gets drawn first, and appears behind the hero. You multiply by `kScaleFactor` so that the image scales according to the device.

You're almost there! All that's left now is to create the jump animations and the actual shadow sprite.

Go to `Hero.m` and make the following changes:

```
//add these to the end of init, inside the curly braces of the if
statement
self.shadow = [CCSprite
spriteWithSpriteFrameName:@"shadow_character.png"];
self.shadow.opacity = 190;

CCArray *jumpRiseFrames = [CCArray arrayWithCapacity:2];
[jumpRiseFrames addObject:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"hero_jump_05.png"]];
[jumpRiseFrames addObject:[[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"hero_jump_00.png"]];
self.jumpRiseAction = [CCAnimate actionWithAnimation:[CCAnimation
animationWithSpriteFrames:[jumpRiseFrames getNSArray]
delay:1.0/12.0]];

self.jumpFallAction = [CCAnimate actionWithAnimation:[self
animationWithPrefix:@"hero_jump" startFrameIdx:1 frameCount:4
delay:1.0/12.0]];

self.jumpLandAction = [CCSequence actions:[CCCallFunc
actionWithTarget:self selector:@selector(setLandingDisplayFrame)],
[CCDelayTime actionWithDuration:1.0/12.0], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];

//add this method
-(void)setLandingDisplayFrame
{
    [self setDisplayFrame:[[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"hero_jump_05.png"]];
}
```

You just did the following:

1. Created the `shadow` sprite using the `shadow_character.png` sprite frame that was included in the sprite sheet.
2. Created the `jumpRise` animation and action using two specific frames from the sprite sheet. You couldn't use the convenience function for creating animations here because the two sprite frames aren't in sequence.

3. Created the `jumpFall` animation and action using the now-familiar method.
4. Created the `jumpLand` animation and action. This action doesn't use the conventional animation action you've used in the past, but instead uses a new method named `setLandingDisplayFrame`. It then transitions to the idle action after 1/12<sup>th</sup> of a second.
5. `setLandingDisplayFrame` simply changes the displayed frame of the hero to `hero_jump_05.png`. You do this because you only have a single frame to show that the hero is landing, as opposed to other animations that have two or more frames. So there's no need to create a `CCAnimation` for it.

That's it. Build and run, and get jumping!



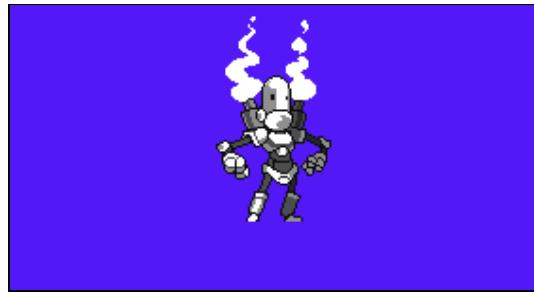
## This is the droid you're looking for

Yes, being able to walk, run and jump around is fun, but exploring a huge, empty corridor gets pretty boring, pretty fast. It's up to you to give your intrepid hero some company.

You already have a base model for the enemy sprite: `ActionSprite`. Now that you've seen how the `Hero` class was put together, it should be easy to create a new character using different images. For the next character that you create, though, you'll spice things up by assembling him piece-by-piece.

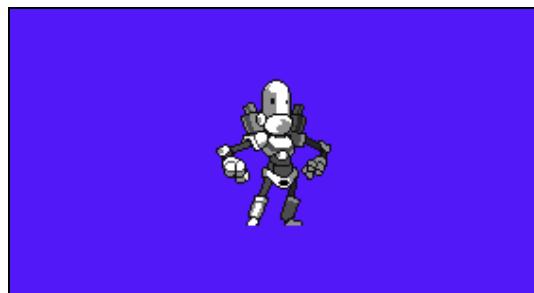
### Assembling the robot

Introducing the `Robot`:

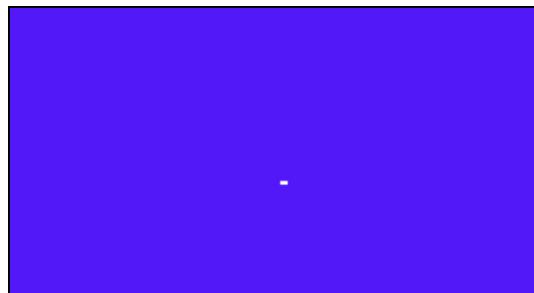


What you see here is the robot's complete form. In reality, it is drawn using three separate sprite components.

The base:



The orb of the belt:



The smoke:



All of these components are drawn within the same canvas size so that their positions remain consistent when combined.

You may be wondering why the robot sprite is split into three different pieces – and why separate the belt orb and the smoke? The explanation is closely related to why

the robot is drawn in a white hue. Suffice it to say, doing it this way will allow you to make many variations of colored robots later on.

**Note:** To see the different pieces of the robot and its animation frames, open the RobotFrames.psd file that comes in the Raw folder included in your Starter Kit.

Select the **Characters** group in Xcode, go to **File\New\New File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **ActionSprite** for Subclass of, click **Next** and name the new file **Robot**.

Open **Robot.h** and make the following changes:

```
//add to top of file
#import "ActionSprite.h"

//add after the curly braces, before @end
@property(nonatomic, strong)CCSprite *belt;
@property(nonatomic, strong)CCSprite *smoke;
```

Switch to **Robot.m** and add this method:

```
-(id)init
{
    if ((self = [super
initWithSpriteFrameName:@"robot_base_idle_00.png"]))
    {
        self.belt = [CCSprite
spriteWithSpriteFrameName:@"robot_belt_idle_00.png"];
        self.smoke = [CCSprite
spriteWithSpriteFrameName:@"robot_smoke_idle_00.png"];

        self.shadow = [CCSprite
spriteWithSpriteFrameName:@"shadow_character.png"];
        self.shadow.opacity = 190;

        self.walkSpeed = 80 * kPointFactor;
        self.runSpeed = 160 * kPointFactor;
        self.directionX = 1.0;
        self.centerToBottom = 39.0 * kPointFactor;
        self.centerToSides = 29.0 * kPointFactor;
    }
    return self;
}
```

The above creates the robot with a base sprite and then creates two other sprites – the belt and the smoke. Then it sets the same attributes that were set before for the hero, this time using measurements for the robot.

Whenever the base sprite's properties change, the belt and smoke sprites should change along with it. You can easily achieve this through method overriding, which is just a fancy way of saying that you will replace the implementation of a method that already exists in the superclass (`ccsprite` in this case).

Still in **Robot.m**, add the following methods:

```
- (void) setPosition:(CGPoint)position
{
    [super setPosition:position];
    _belt.position = position;
    _smoke.position = position;
}

-(void)setScaleX:(float)scaleX
{
    [super setScaleX:scaleX];
    _belt.scaleX = scaleX;
    _smoke.scaleX = scaleX;
}

-(void)setScaleY:(float)scaleY
{
    [super setScaleY:scaleY];
    _belt.scaleY = scaleY;
    _smoke.scaleY = scaleY;
}

-(void)setScale:(float)scale
{
    [super setScale:scale];
    _belt.scale = scale;
    _smoke.scale = scale;
}

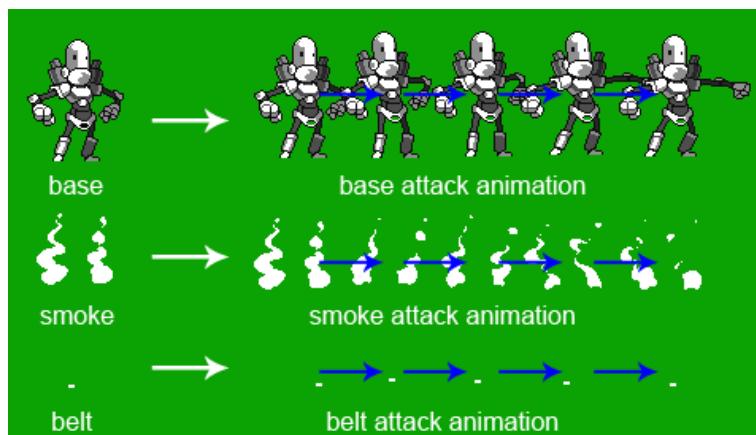
-(void)setVisible:(BOOL)visible
{
    [super setVisible:visible];
    _belt.visible = visible;
    _smoke.visible = visible;
}
```

All of these methods are already available in `ccSprite`. When you set the position of a sprite through `sprite.position`, internally the `setPosition:` method is executed. The same goes for the `scale` and `visible` properties.

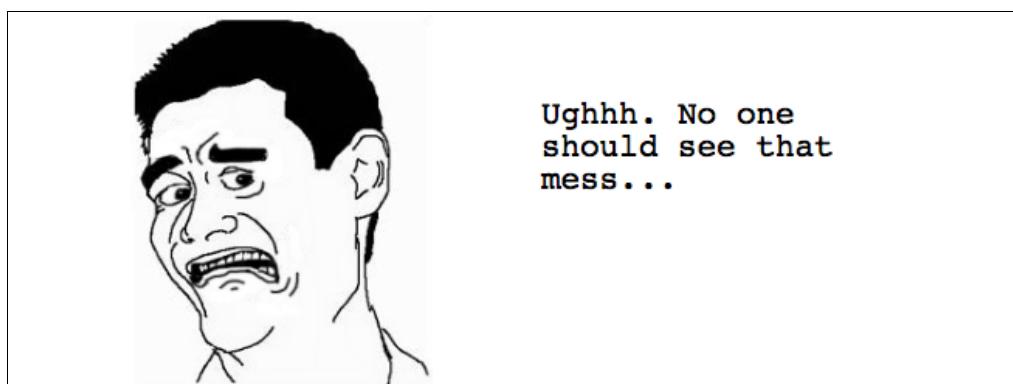
Now, whenever the base sprite's properties change, the belt and smoke sprite's properties change along with it.

## Do the robot dance!

To complete the robot, you need to create its animation actions. However, animation actions can only run for a single sprite. Since the robot is made up of three different sprites – base, belt and smoke – it needs to run three different animations, like this:



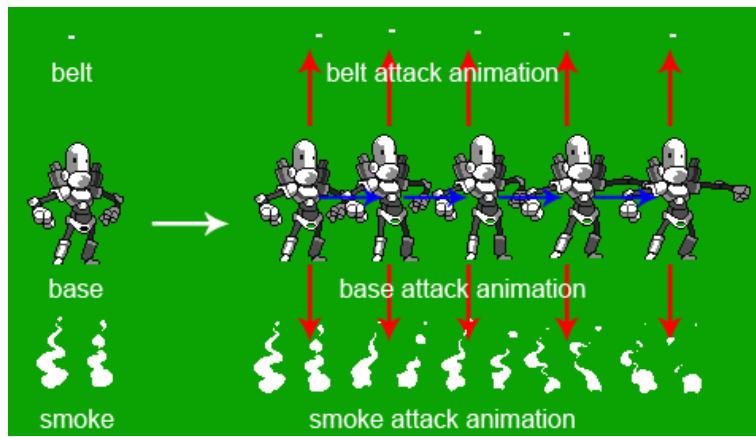
But there's a problem: if you create three separate animation actions and have each of these sprites executing their own animations, there is a big possibility that they will go out of sync with one another, especially since you are dealing with very short intervals per frame.



To overcome this, there should be a grouping mechanism that ensures that the other two sprites – belt and smoke – only switch frames whenever the base sprite switches frames.

In the same way that `setPosition:` also sets the position of the belt and smoke sprites, each action step of the base sprite should also trigger the equivalent action step for the other two sprites.

Running actions should correspond like this:



The base sprite only needs to run the base animation action, and then for every step taken by that animation action, it triggers the same step on the other two animation actions. This ensures that all three actions will be synchronized at all times.

The first step in implementing the above is to make actions that can be grouped. Thankfully, Cocos2D actions can be easily extended and copied.

Select the **Controls** group in Xcode, go to **File\New\New File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **NSObject** for Subclass of, click **Next** and name the new file **AnimationMember**.

**AnimationMember** will contain a **ccAnimation** and it will belong to an animation group. In this case, you'll use it for the belt and smoke animations.

Open **AnimationMember.h** and replace its contents with the following:

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

@interface AnimationMember : NSObject {
    id _origFrame;
    CCSprite *_target;
}

@property(nonatomic, strong)CCAnimation *animation;
@property(nonatomic, strong)CCSprite *target;

+(id)memberWithAnimation:(CCAnimation *)animation target:(CCSprite *)target;
```

```
- (id)initWithAnimation:(CCAnimation *)animation target:(CCSprite *)target;
-(void)start;
-(void)stop;
-(void)setFrame:(NSUInteger)frameIndex;

@end
```

Switch to **AnimationMember.m** and add these methods:

```
+ (id)memberWithAnimation:(CCAnimation *)animation target:(CCSprite *)target
{
    return [[self alloc] initWithAnimation:animation
target:target];
}

-(id)initWithAnimation:(CCAnimation *)animation target:(CCSprite *)target
{
    if ((self = [super init]))
    {
        self.animation = animation;
        self.target = target;
        _origFrame = nil;
    }
    return self;
}

-(void)start
{
    _origFrame = _target.displayFrame;
}

-(void)stop
{
    if (_animation.restoreOriginalFrame)
    {
        [_target setDisplayFrame:_origFrame];
    }
}

-(void)setFrame:(NSUInteger)frameIndex
{
    NSArray *frames = [_animation frames];
```

```
CCAnimationFrame *frame = [frames objectAtIndex:frameIndex];
CCSpriteFrame *spriteFrame = [frame spriteFrame];
[_target setDisplayFrame:spriteFrame];
}
```

**AnimationMember** just acts as the commander of the **ccAnimation** stored within. You initialize it with a **ccAnimation** and the target sprite that the **ccAnimation** should affect.

When the animation starts, you store the **ccsprite**'s original frame so that you can restore it when the animation stops.

The **setFrame:** method is able to pick out a specific frame from the **ccanimation** and it changes the **ccsprite**'s image to the retrieved frame.

**Note:** If you take a look at Cocos2D's **ccAnimate** action, you will notice that **AnimationMember** is written in a similar way, only simpler. You can look at **AnimationMember** as a sort of dumbed-down and manually-controlled **CCAnimate** action.

Next, select the **Controls** group in Xcode, go to **File\New\New File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **CCAnimate** for Subclass of, click **Next** and name the new file **AnimateGroup**.

**AnimateGroup** is going to be a **ccAnimate** action that is able to control a group of **AnimationMembers**.

Open **AnimateGroup.h** and replace its contents with the following:

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

@interface AnimateGroup : CCAnimate {
}

@property(nonatomic, strong)CCArray *members;

+(id)actionWithAnimation:(CCAnimation *)animation members:(CCArray *)
* )members;
-(id)initWithAnimation:(CCAnimation *)animation members:(CCArray *)
* )members;

+(id)actionWithAnimation:(CCAnimation *)animation
memberCount:(int)memberCount;
-(id)initWithAnimation:(CCAnimation *)animation
memberCount:(int)memberCount;
```

```
@end
```

Here you create a subclass of `ccAnimate`. It contains an array of `AnimationMembers` and some helper methods to create a new `AnimateGroup`.

Switch to **AnimateGroup.m** and make the following changes:

```
//add to top of file
#import "AnimationMember.h"

//add these methods
+(id)actionWithAnimation:(CCAnimation *)animation members:(CCArray *)
* )members
{
    return [[self alloc] initWithAnimation:animation
members:members];
}

-(id)initWithAnimation:(CCAnimation *)animation members:(CCArray *)
* )members
{
    if ((self = [super initWithAnimation:animation]))
    {
        self.members = members;
    }
    return self;
}

+(id)actionWithAnimation:(CCAnimation *)animation
memberCount:(int)memberCount
{
    return [[self alloc] initWithAnimation:animation
memberCount:memberCount];
}

-(id)initWithAnimation:(CCAnimation *)animation
memberCount:(int)memberCount
{
    if ((self = [super initWithAnimation:animation]))
    {
        self.members = [CCArray arrayWithCapacity:memberCount];
    }
    return self;
}
```

**AnimateGroup** is initialized as a `ccAnimate` action, but it also stores an array named `members`. `members` will contain all the `AnimationMember` instances that should run in sync with `AnimateGroup`.

Still in **AnimateGroup.m**, add these methods:

```
- (void)startWithTarget:(id)target
{
    [super startWithTarget:target];

    AnimationMember *member;
    CCARRAY_FOREACH(_members, member)
    {
        [member start];
    }
}

-(void)stop
{
    [super stop];

    AnimationMember *member;
    CCARRAY_FOREACH(_members, member)
    {
        [member stop];
    }
}

-(void)update:(ccTime)t
{
    [super update:t];

    int frameIndex = MAX(0, _nextFrame - 1);

    AnimationMember *member;
    CCARRAY_FOREACH(_members, member)
    {
        [member setFrame:frameIndex];
    }
}
```

When the main animation action starts, it also starts all of its component `AnimationMember` instances. If the main animation stops, then all the member animations also stop.

Whenever the `AnimateGroup` is updated, as an inherited behavior from `ccAnimate`, it changes animation frames. This all happens in `[super update:]`.

The important part here is what comes after. Every time an update happens, `AnimateGroup` finds the current index of its own animation frame, which is one number less than the next frame variable. It then changes the frame of all its `AnimationMembers` to the same frame index.

You will be making a lot of `AnimationMembers` and `AnimateGroups`, so it's a good idea to create helper methods to ease the process.

Go to `ActionSprite.h` and make the following changes:

```
//add to top of file
#import "AnimationMember.h"

//add before @end
-(AnimationMember *)animationMemberWithPrefix:(NSString *)prefix
startFrameIdx:(NSUInteger)startFrameIdx
frameCount:(NSUInteger)frameCount delay:(float)delay
target:(id)target;
```

Switch `ActionSprite.m` and add the new method:

```
-(AnimationMember *)animationMemberWithPrefix:(NSString *)prefix
startFrameIdx:(NSUInteger)startFrameIdx
frameCount:(NSUInteger)frameCount delay:(float)delay
target:(id)target
{
    CCAcceleration *acceleration = [self accelerationWithPrefix:prefix
startFrameIdx:startFrameIdx frameCount:frameCount delay:delay];
    return [AnimationMember memberWithAcceleration:acceleration
target:target];
}
```

`animationMemberWithPrefix:` creates a `CCAcceleration` using the helper method you defined before. It then creates the `AnimationMember` that owns this acceleration and sets the target to the correct object.

To make things even simpler, you will also make a helper method to produce `AnimateGroups` for the `Robot` class specifically.

Go to `Robot.m` and make the following changes:

```
//add to top of file
#import "AnimateGroup.h"

//add this method
-(AnimateGroup *)animateGroupWithActionWord:(NSString
*)actionKeyWord frameCount:(NSUInteger)frameCount
delay:(float)delay
```

```

{
    CCAnimation *baseAnimation = [self
animationWithPrefix:[NSString stringWithFormat:@"robot_base_%@", actionKeyWord] startFrameIdx:0 frameCount:frameCount delay:delay];

    AnimationMember *beltMember = [self
animationMemberWithPrefix:[NSString
stringWithFormat:@"robot_belt_%@", actionKeyWord] startFrameIdx:0
frameCount:frameCount delay:delay target:_belt];

    AnimationMember *smokeMember = [self
animationMemberWithPrefix:[NSString
stringWithFormat:@"robot_smoke_%@", actionKeyWord] startFrameIdx:0
frameCount:frameCount delay:delay target:_smoke];

    CCArray *animationMembers = [CCArray
arrayWithNSArray:@[beltMember, smokeMember]];

    return [AnimateGroup actionWithAnimation:baseAnimation
members:animationMembers];
}

```

This method makes it easy to create animation actions for the robot. You only need the suffix keyword for the action (e.g., idle, attack, move) and the number of frames, and the method will automatically create the following:

- A `CCAnimation` with sprite frame animations for the `base` sprite.
- An `AnimationMember` with sprite frame animations targeting the `belt` sprite.
- An `AnimationMember` with sprite frame animations targeting the `smoke` sprite.
- An `AnimateGroup` using the `CCAnimation` above as the base animation, and an array containing the two `AnimationMembers` above as the member animations.

Each animation contains a sprite frame name that starts with a descriptive prefix (`robot_base_`, `robot_smoke_`, `robot_belt_`) and combines this with the action suffix. The string formed should reflect the sprite frame name of each sprite.

You are expecting all of these animations to have the same number of frames, and the same delay between frames.

Now you can create grouped animation actions for the robot with ease!

Still in `Robot.m`, add the following:

```
//add inside init right after self.shadow.opacity = 190
AnimateGroup *idleAnimationGroup = [self
animateGroupWithActionWord:@"idle" frameCount:5 delay:1.0/12.0];
```

```

self.idleAction = [CCRepeatForever
    actionWithAction:idleAnimationGroup];

AnimateGroup *attackAnimationGroup = [self
    animateGroupWithActionWord:@"attack" frameCount:5 delay:1.0/15.0];
self.attackAction = [CCSequence actions:attackAnimationGroup,
    [CCCallFunc actionWithTarget:self selector:@selector(idle)], nil];

AnimateGroup *walkAnimationGroup = [self
    animateGroupWithActionWord:@"walk" frameCount:6 delay:1.0/12.0];
self.walkAction = [CCRepeatForever
    actionWithAction:walkAnimationGroup];

```

You create three actions – idle, attack and walk – using the helper method you just created for the robot. These work much the same way as the hero's actions, except that triggering these actions also updates animations for the belt and smoke sprites, because you used **AnimateGroup** instead of **CCAnimate**.

## Your robot army

Now jump straight to filling the game level with a horde of these robots.

First go to **Defines.h** and add these definitions:

```

#define random_range(low,high) ((arc4random()%(high-low+1))+low)
#define frandom ((float)arc4random()/UINT64_C(0x100000000))
#define frandom_range(low,high) (((high-low)*frandom)+low)
#define random_sign (arc4random() % 2 ? 1 : -1)

```

These are just random number generators that you will need:

- **random\_range**: Creates a random integer within the low and high value range.
- **frandom**: Creates a random floating point value.
- **frandom\_range**: Creates a random floating point value within the low and high value range.
- **random\_sign**: Randomly returns 1 or -1.

Switch to **GameLayer.h** and add the following property:

```
@property(nonatomic, strong)CCArray *robots;
```

Now switch to **GameLayer.m** and make the following changes:

```
//add to top of file
#import "Robot.h"
```

```

//add inside if ((self = [super init])) in init, right after [self
initHero];
[self initRobots];

//add this method
-(void)initRobots {
    int robotCount = 50;
    self.robots = [[CCArray alloc] initWithCapacity:robotCount];

    for (int i = 0; i < robotCount; i++) {
        Robot *robot = [Robot node];
        [_actors addChild:robot.shadow];
        [_actors addChild:robot.smoke];
        [_actors addChild:robot];
        [_actors addChild:robot.belt];
        [_robots addObject:robot];

        int minX = SCREEN.width + robot.centerToSides;
        int maxX = _tileMap.mapSize.width *
_tileMap.tileSize.width * kPointFactor - robot.centerToSides;
        int minY = robot.centerToBottom;
        int maxY = 3 * _tileMap.tileSize.height * kPointFactor +
robot.centerToBottom;

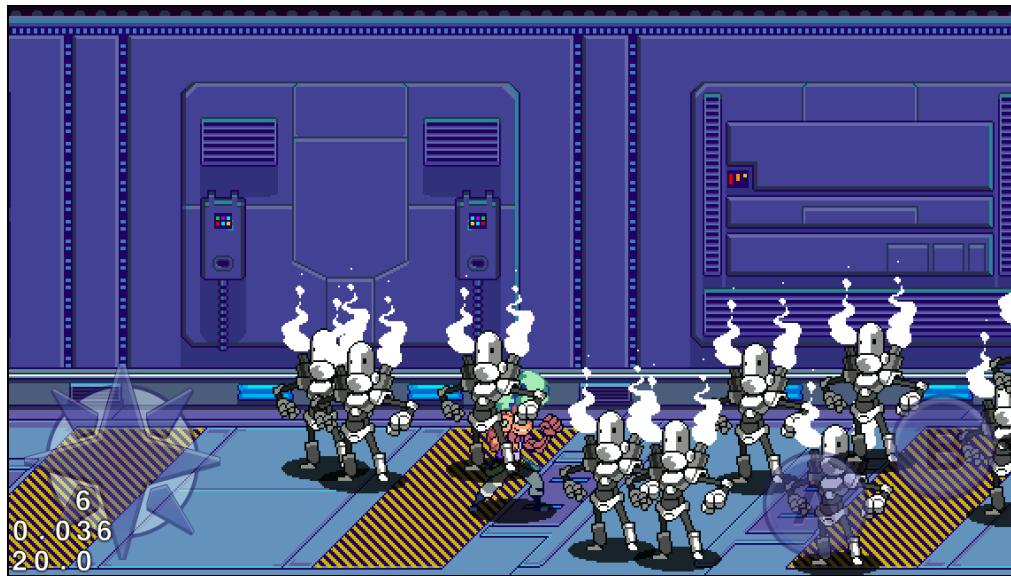
        robot.scaleX = -kScaleFactor;
        robot.scaleY = kScaleFactor;
        robot.shadow.scale *= kScaleFactor;
        robot.position = ccp(random_range(minX, maxX),
random_range(minY, maxY));
        robot.groundPosition = robot.position;
        robot.desiredPosition = robot.position;
        [robot idle];
    }
}

```

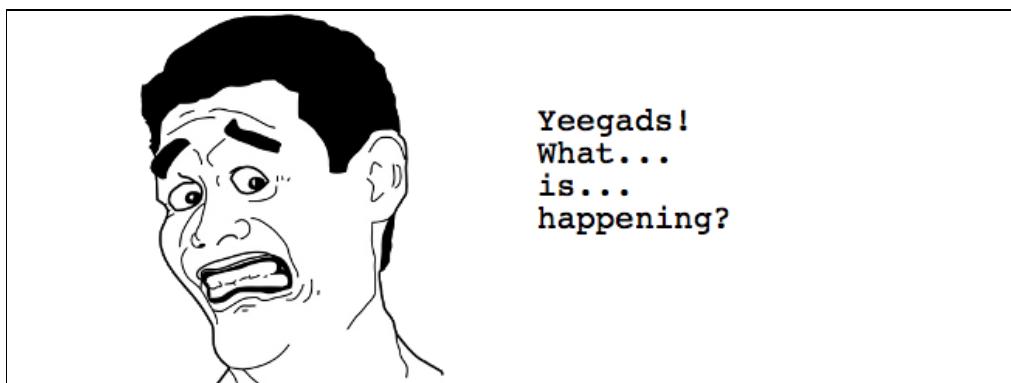
The above does the following:

1. Creates an array of 50 robots and adds all of their components to the batch node.
2. Uses the random functions in Defines.h to place the 50 robots across the tile map's floors. Also makes sure that no robots are placed at the starting point by making the minimum random value bigger than the screen's width.
3. Scales the components to support universal display. The robots' `scaleX` values are negative so that they face left initially.
4. Makes each robot perform its idle action.

Build and run, and move forward until you see robots on the map.

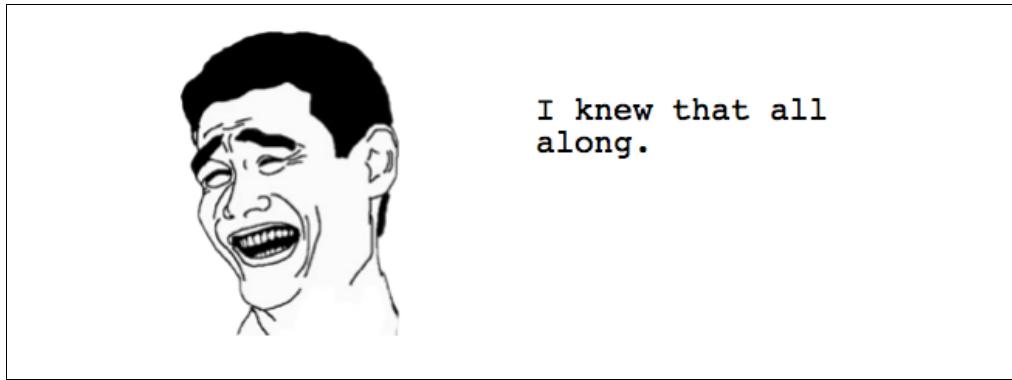


Try walking around a bit in an area with robots, and you'll notice that there's something seriously wrong with how the robots are drawn. According to the current perspective, if the hero is below a robot, then he should be drawn in front of the robot, not the other way around.



For things to be drawn in the right sequence, you need to explicitly tell the game which objects to draw first. You should already know how to do this – think back to how you made the tile map appear behind everything else, and how you made `HudLayer` appear in front of `GameLayer`.

Figured it out? If your answer was z-order/z-value, then you're 100% correct!



To refresh your memory, take a look at how you added the sprite batch node and tile map to the scene in **GameLayer.m**:

```
[self addChild:_actors z:-5];
[self addChild:_tileMap z:-6];
```

Now take a look at how you added the hero and the robots:

```
[_actors addChild:_hero];
[_actors addChild:robot];
```

There are two differences:

1. You added the `SpriteBatchNode` and the `cctmxtiledMap` both as direct children of `GameLayer`, while you added the hero and the robots as children of the `CCSpriteBatchNode`. `GameLayer` is responsible for drawing the `CCSpriteBatchNode` and `cctmxtiledMap` in the proper sequence, while it is the `CCSpriteBatchNode`'s responsibility to draw the hero and the robots in the correct sequence within itself.
2. You didn't explicitly assign a z-value to the hero and the robots. By default, the object added last will have a higher z-value than the previous objects – and that's why all the robots are drawn in front of the hero.

To fix the broken drawing sequence, you need to handle the z-order dynamically. Every time a sprite moves across the screen vertically, its z-order should be changed. The higher a sprite is on the screen, the lower its z-value should be.

Still in **GameLayer.m**, make the following changes:

```
//add inside update:, after [self updatePositions];
[self reorderActors];

//add these methods
-(NSInteger)getZFromYPosition:(float)yPosition
{
    return (_tileMap.mapSize.height * _tileMap.tileSize.height *
kPointFactor) - yPosition;
```

```
}

-(void)reorderActors
{
    NSInteger spriteZ = [self
getZFromYPosition:_hero.groundPosition.y];

[_actors reorderChild:_hero.shadow z:spriteZ];
[_actors reorderChild:_hero z:spriteZ];

Robot *robot;
CCARRAY_FOREACH(_robots, robot)
{
    spriteZ = [self getZFromYPosition:robot.groundPosition.y];
    [_actors reorderChild:robot.shadow z:spriteZ];
    [_actors reorderChild:robot.smoke z:spriteZ];
    [_actors reorderChild:robot z:spriteZ];
    [_actors reorderChild:robot.belt z:spriteZ];
}
}
```

`getZFromYPosition:` returns an integer ranging from 0 to the total map height. Every time the sprite positions are updated, `reorderActors` makes the `ccSpriteBatchNode` reorder the z-value of each of its children based on how far the child is from the bottom of the map. As the child goes higher, the resulting z-value goes down.

You use `groundPosition` instead of `position` so that jumping does not affect the z-value of the characters.

**Note:** Each `ccNode` has its own property named `zOrder`, but changing this won't give you the same effect as calling `reorderChild` from the parent. It's the parent's responsibility to draw its children in order, so it should also be the parent's responsibility to set the order of its children.

Build and run, and the drawing sequence should now be correct.



## Color tinting

A corridor full of robots looks exciting! But it's a little monotonous with all the robots looking exactly the same, isn't it?

In the previous section, I hinted that there was a special reason why the robots were colored the way they are, and why they were created using three different parts.

If you've ever seen games like *Tiny Tower* or *Pocket Planes*, you may be wondering how they were able to achieve a seemingly infinite number of color combinations for their characters, or Bitizens, as the games call them.

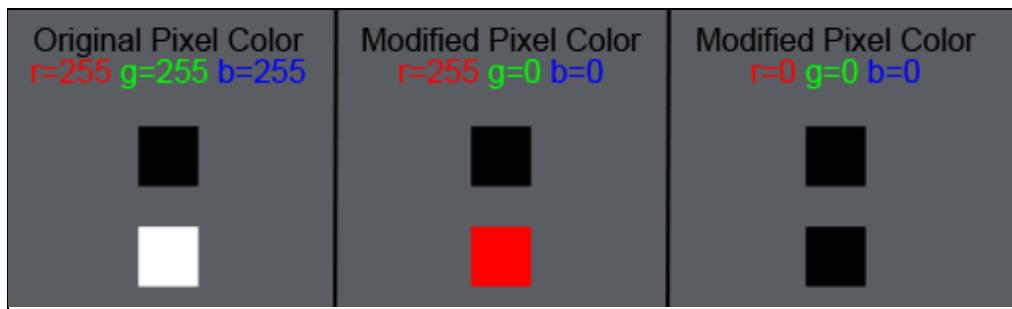
In the old days, developers could have achieved this through a technique called palette swapping, where the same sprite is drawn using different color sets or palettes. In Cocos2D, changing the color of a sprite is much easier thanks to the `color` property of `ccsprite`.

Changing the `color` property of a `ccsprite` doesn't simply change the sprite's color, though. What actually happens is a process called tinting. With tinting, the original color of the texture still matters. Each pixel's color is influenced by the value of the `color` property to give a different resulting color.

A color is split into three components – red, green, and blue. Each has a value that ranges from 0 to 255, where 0 is no color and 255 is full color for that particular color component (red, green, or blue).

By default, a `ccsprite`'s `color` value is 255 red, 255 green, 255 blue, meaning that all three represent the `ccsprite` in its original full color. When it is 0 red, 0 green, 0 blue, the `ccsprite` loses its color.

Take a look at this diagram:



An originally white pixel changes color depending on the `color` property, while an originally black pixel doesn't change at all.

Because `color` already starts at 255, 255, 255 for the original pixel color, you cannot change the color of a `ccsprite` to become lighter than the original image's colors – you can only change it to become darker. For example, you cannot change an originally black or even yellow pixel to white, but you can change a white pixel to black or yellow.

The whiter the color, the more it gets affected by color tinting.

This is why the robot is colored the way it is in the game:



Based on what `color` value you set, the white parts will change completely, while the gray parts will change partially, resulting in a darker version of the same color so that the shading stays consistent.

As you might suspect, if you change the `color` property of a `ccsprite` it will affect the whole image. This is why you created three different components. This way, the belt, smoke and base colors can all be different so that if the robot's body is green, the smoke doesn't necessarily need to be green as well.

Though stay tuned – there will be green smoke! Right now it's time to test out all that theory.

Go to **Defines.h** and add this definition:

```
typedef enum _ColorSet
{
```

```
kColorLess = 0,  
kColorCopper,  
kColorSilver,  
kColorGold,  
kColorRandom,  
} ColorSet;
```

You will use these definitions as presets to determine how to color each robot.

Switch to **Robot.h** and add this property:

```
@property(nonatomic, assign)ColorSet colorSet;
```

Switch to **Robot.m** and add this method override:

```
-(void)setColorSet:(ColorSet)colorSet  
{  
    _colorSet = colorSet;  
    if (colorSet == kColorLess)  
    {  
        self.color = ccWHITE;  
        _belt.color = ccWHITE;  
        _smoke.color = ccWHITE;  
    }  
    if (colorSet == kColorCopper)  
    {  
        self.color = ccc3(255, 193, 158);  
        _belt.color = ccc3(99, 162, 255);  
        _smoke.color = ccc3(220, 219, 182);  
    }  
    else if (colorSet == kColorSilver)  
    {  
        self.color = ccWHITE;  
        _belt.color = ccc3(99, 255, 128);  
        _smoke.color = ccc3(128, 128, 128);  
    }  
    else if (colorSet == kColorGold)  
    {  
        self.color = ccc3(233, 177, 0);  
        _belt.color = ccc3(109, 40, 25);  
        _smoke.color = ccc3(222, 129, 82);  
    }  
    else if (colorSet == kColorRandom)  
    {  
        self.color = ccc3(random_range(0, 255), random_range(0,  
255), random_range(0, 255));  
    }  
}
```

```
        _belt.color = ccc3(random_range(0, 255), random_range(0, 255), random_range(0, 255));
        _smoke.color = ccc3(random_range(0, 255), random_range(0, 255), random_range(0, 255));
    }
}
```

Whenever you change the `colorSet` property of a robot, the base, belt and smoke colors are changed according to the preset.

`kColorLess` will change all colors to `ccWHITE`, which is just a shortcut definition for an all 255 RGB value. With this, everything returns to its original whitish color.

`kColorCopper`, `kColorSilver` and `kColorGold` change the colors to different predetermined combinations.

`kColorRandom` sets all the colors to a random value, resulting in a different color combination each time.

Try it out! Go to **GameLayer.m** and add this line inside `initRobots`:

```
//add after [robot idle]; before the closing curly brace
robot.colorSet = kColorRandom;
```

This will make all robots choose a random color for their base, belt and smoke sprites.

Build and run. The robots now look very psychedelic!



Building on the first two chapters, you added buttons, punching, jumping, and colorful enemies. Not bad for one chapter!

Stay tuned for next chapter, where your hero will beat these robots to scrap metal!

**Challenge:** If you did the challenge from Chapter 2, you have replaced the idle and walk animations for the hero. In this chapter, replace the rest of the animations with your own drawings as well so your hero can be fully animated.

When you're done, try experimenting with the jump variables a bit. What would you change to make the hero jump higher, or float in the air more?

# Chapter 4: Bring On the Droids

In the last chapter, you made your hero run, jump, and punch, and added an army of colorful robots into the scene.

However, there's a big problem with the current implementation. Your hero's punches don't connect with anything! And if this is a Beat 'Em Up Game, he should be able to beat somebody up, right?

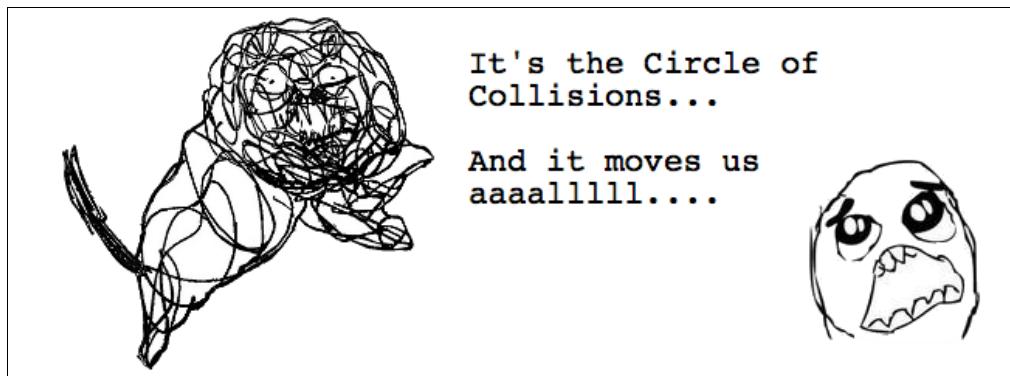
So in this chapter, you will implement collision detection so punching will actually damage the robots. Along the way, you'll learn how to handle some tricky challenges such as making the collision detection vary based on the frame of the animation that is running, adding debug drawing, and more.

So get ready to lay some smack down!

## Your collision strategy

For the hero to be able to punch and actually hit the robots, you need to implement a strategy for detecting collisions. One way of doing this is to create collision boxes or rectangles to represent each character, and check if these rectangles intersect.

For this game, however, you'll use a more versatile way of checking collisions – circles. You'll set up each character with a collection of circles attached to parts of their bodies. These circles will aid in the collision detection.



For this system, you will define three different sets of circles for each character:

1. **Contact Circles:** These circles will represent the body of the sprite – the parts that can be hurt.
2. **Attack Circles:** These circles will represent the parts of the sprite that can hurt another sprite. For example, a hand when it punches or a short energy burst.
3. **Detection Circle:** There will be several of each of the above circle types for each character, and it would be inefficient to have to check all of these for collisions when the sprites aren't even remotely near one another. The detection circles define the minimum area within which a collision is allowable for two sprites.

If two sprites have intersecting detection circles, and an attack circle of one sprite collides with a contact circle of another at the exact time of attack, then a collision occurs. This distinction between circles will help you decide who hit whom.

You already defined a structure for a circle in **Defines.h**. For reference, here it is again:

```
typedef struct _ContactPoint
{
    CGPoint position;
    CGPoint offset;
    CGFloat radius;
} ContactPoint;
```

Each circle stores three values:

- **position:** This holds the x and y coordinates of the center of the circle relative to world space or the scene. As the sprite moves, `position` changes. Think of it as the location of the circle as the `GameLayer` sees it.
- **offset:** This is the x and y distance of the circle's center from the center of the sprite. This value never changes once it is set, and it's used to calculate the position of the circle. Think of it as the internal position of the circle as the sprite sees it.
- **radius:** This is simply the radius of the circle.

In Chapter 1, you already declared properties for the aforementioned circles of the `ActionSprite`. For reference, here they are again:

```
@property(nonatomic, assign)ContactPoint *contactPoints;
@property(nonatomic, assign)ContactPoint *attackPoints;
@property(nonatomic, assign)int contactPointCount;
@property(nonatomic, assign)int attackPointCount;
@property(nonatomic, assign)float detectionRadius;
```

These are as follows:

- **contactPoints**: An array of contact circles. You cannot use `ccArrays` for the `ContactPoint` type because `ccArrays` can only hold Objective-C objects, while these are C-language structures.
- **contactPointCount**: The number of contact circles.
- **attackPoints**: An array of attack circles.
- **attackPointCount**: The number of attack circles.
- **detectionRadius**: The radius of the detection circle. There's no need to use `ContactPoint` for the detection circle because its position will always be the position of the sprite.

Let's set up some of these values. Go to **Hero.m** and add the following:

```
//add to the end of init inside the curly braces of if ((self =
[super initWithSpriteFrameName:@"hero_idle_00.png"]))
self.detectionRadius = 100.0 * kPointFactor;
self.attackPointCount = 3;
self.attackPoints = malloc(sizeof(ContactPoint) *
self.attackPointCount);
self.contactPointCount = 4;
self.contactPoints = malloc(sizeof(ContactPoint) *
self.contactPointCount);

self.maxHitPoints = 200.0;
self.hitPoints = self.maxHitPoints;
self.attackDamage = 5.0;
self.attackForce = 4.0 * kPointFactor;
```

You set the `detectionRadius` of the hero to 100 points. You can see how I arrived at the 100-point in the "Optional Exercise" section that is coming soon.

You also create three `ContactPoints` as attack points (the `attackPoints` array) and four `ContactPoints` as contact points (the `contactPoints` array). You create these arrays with the old school C `malloc` function, which allocates a block of memory for three `ContactPoints` with zeroed out values for offset, position and radius.

Finally, you set the following attributes for the hero:

- **maxHitPoints**: The full health value of the hero.
- **hitPoints**: The current health value of the hero. When he gets hit, this goes down. If it gets to zero, then the hero dies.
- **attackDamage**: The damage inflicted by the hero's jab attack.
- **attackForce**: The knockback force, in points, of the hero's attack.

Switch to **Robot.m** and make the following changes:

```
//add to the end of init, inside the curly braces of the if
statement
self.detectionRadius = 50.0 * kPointFactor;
self.contactPointCount = 4;
self.contactPoints = malloc(sizeof(ContactPoint) *
self.contactPointCount);
self.attackPointCount = 1;
self.attackPoints = malloc(sizeof(ContactPoint) *
self.attackPointCount);

self.maxHitPoints = 100.0;
self.hitPoints = self.maxHitPoints;
self.attackDamage = 4;
self.attackForce = 2.0 * kPointFactor;
```

This is a repeat of what you did for the hero. Notice that the robot doesn't have as much hit points, attack damage, or attack force as the hero. This is logical, because otherwise your hero will never be able to defeat a horde of robots. ☺

You still need to set the offset, position and radius of each contactPoint created in the attackPoints and contactPoints arrays. The first thing to do is create helper methods to make it easier.

Go to **ActionSprite.m** and add these methods:

```
// 1
-(ContactPoint)contactPointWithOffset:(const CGPoint)offset
radius:(const float)radius
{
    ContactPoint contactPoint;
    contactPoint.offset = ccpMult(offset, kPointFactor);;
    contactPoint.radius = radius * kPointFactor;
    contactPoint.position = ccpAdd(_position,
contactPoint.offset);
    return contactPoint;
}

// 2
-(void)modifyPoint:(ContactPoint *)point offset:(const
CGPoint)offset radius:(const float)radius
{
    point->offset = ccpMult(offset, kPointFactor);
    point->radius = radius * kPointFactor;
    point->position = ccpAdd(_position, point->offset);
}
```

```
// 3
-(void)modifyAttackPointAtIndex:(const NSUInteger)pointIndex
offset:(const CGPoint)offset radius:(const float)radius
{
    ContactPoint *contactPoint = &_attackPoints[pointIndex];
    [self modifyPoint:contactPoint offset:offset radius:radius];
}

-(void)modifyContactPointAtIndex:(const NSUInteger)pointIndex
offset:(const CGPoint)offset radius:(const float)radius
{
    ContactPoint *contactPoint = &_contactPoints[pointIndex];
    [self modifyPoint:contactPoint offset:offset radius:radius];
}
```

Let's go over this section by section:

1. **contactPointWithOffset:** creates a new ContactPoint given an offset position and a radius. It gets the initial position value by adding the offset coordinates to the sprite's current position.
2. **modifyPoint:** allows you to adjust the offset and radius of a ContactPoint. You'll use this when you need to move circles around.
3. To make **modifyPoint:** even easier, you add **modifyAttackPointAtIndex:** and **modifyContactPointAtIndex:,** which modify the ContactPoint from a specific array – the `attackPoints` array or the `contactPoints` array.

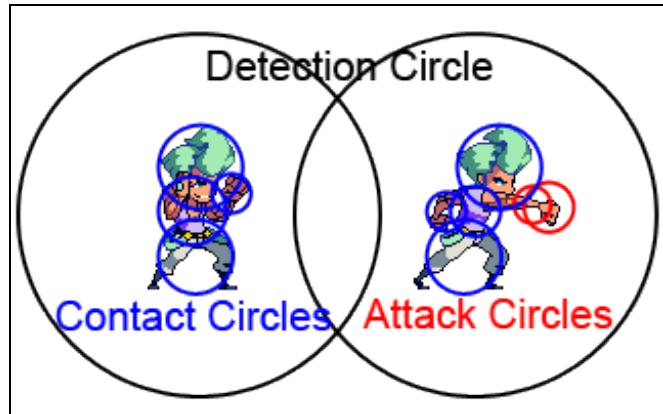
You also multiply **kPointFactor** for iPad conversion so you don't have to do the calculations individually later on.

## Optional Exercise: Finding Contact Points

In this Starter Kit, I've already figured out the contact points for each frame of the animations for you. However, you may be curious to learn how I figured these out so you can do the same with your own artwork.

So if you'd like to learn how I figured out the contact points, keep reading! But if you want to just get back to the code, feel free to skip to the next section.

To define the contact points for the hero and robot, you need to know the offsets and radii of the circles you need to create. Think of it visually and you will have this:



Each character needs to have a detection circle originating from the center of the sprite, contact circles marking the hittable parts of the body and attack circles marking the damage-inflicting parts of the body.

These need not be exact, and if you look at the illustration above, you'll see that the blue circles don't cover the entire body. If there's a part of the body you're sure will never get hit, then there is no need to put a circle over it.

The circles drawn on the diagram make it look so easy. Wouldn't it be nice to have a visual editor to help you draw these circles and figure out their offsets and radii?

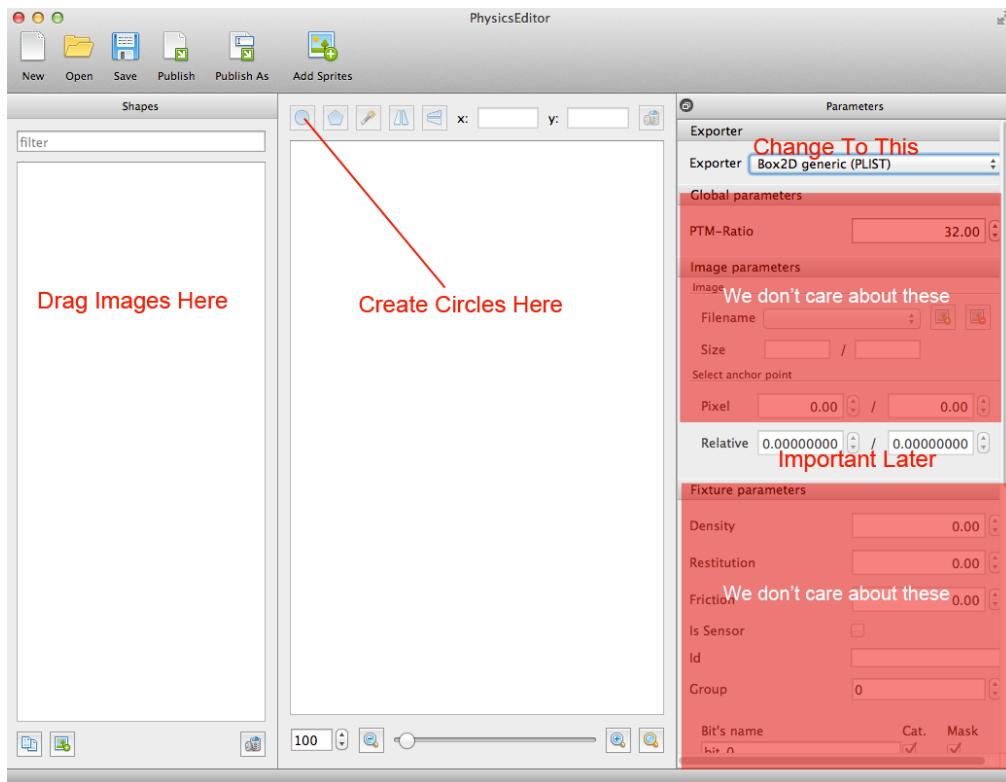
Well, in fact, there is such a program! Though it wasn't intended for this purpose, [PhysicsEditor](#) fits the bill pretty nicely.

**Note:** PhysicsEditor is a tool for creating collision shapes for different physics engines such as Box2D and Chipmunk, two commonly used Cocos2D physics engines.

If you haven't already done so, download PhysicsEditor and install it, then fire it up. You will get a blank project with three panels/columns.

Working with PhysicsEditor is pretty straightforward. On the left, you put all the images with which you want to work. In the middle, you visually define a polygon for your image. On the right, you have various physics parameters, most of which don't matter for what you're trying to do here (just figure out the offset and radius for the shapes).

First select **Box2D generic (PLIST)** as the Exporter from the right panel:

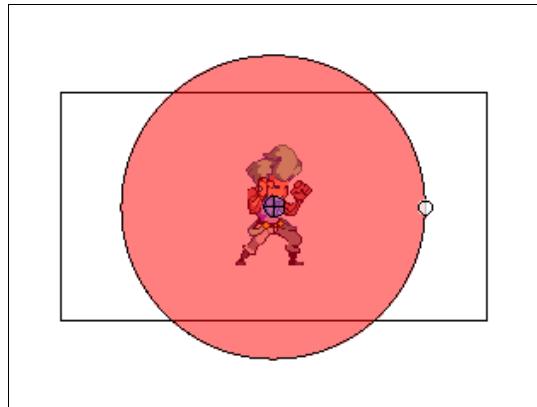


This will open up the Relative options – the anchor point of the sprite, or in other words, the x and y origin of the sprite. In this Starter Kit, you've been using the center of the sprite, coordinates (0.5, 0.5), as the origin, so in order for PhysicsEditor to give out correct offset values, it also needs to know that the origin of your sprite is at (0.5, 0.5). Go to the **Raw/Sprites/Hero** folder of your Starter Kit and drag **hero\_idle\_00.png** to the left panel of PhysicsEditor. You should now see the idle frame of the hero in the center panel.

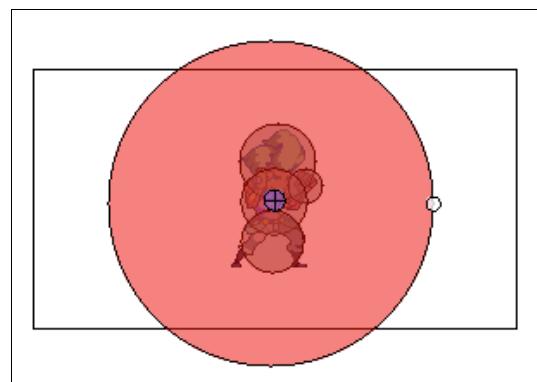
Go to the right panel and change the value of Relative to 0.5 / 0.5.

Next, in the center panel, change the magnification, configurable via a slider at the bottom of the panel, to a comfortable level, and then tap on the circle button on the upper part of this panel to create a circle shape. A new circle will appear at the lower left portion of the frame.

The first circle you create will be the detection circle, so expand it to cover the hero's body completely for all possible actions that the hero can perform. Consider looking at the attack and jump frames of the hero, as they should also be within this circle. A good rule of thumb is to make the detection circle about three times the size of the sprite.



Next create four circles representing the four contact circles of the hero and arrange and resize them like this:



For the attack circle of the hero, drag **hero\_attack\_00\_01.png** to the left panel, change Relative to 0.5 / 0.5, and create a circle near the fist of the hero.



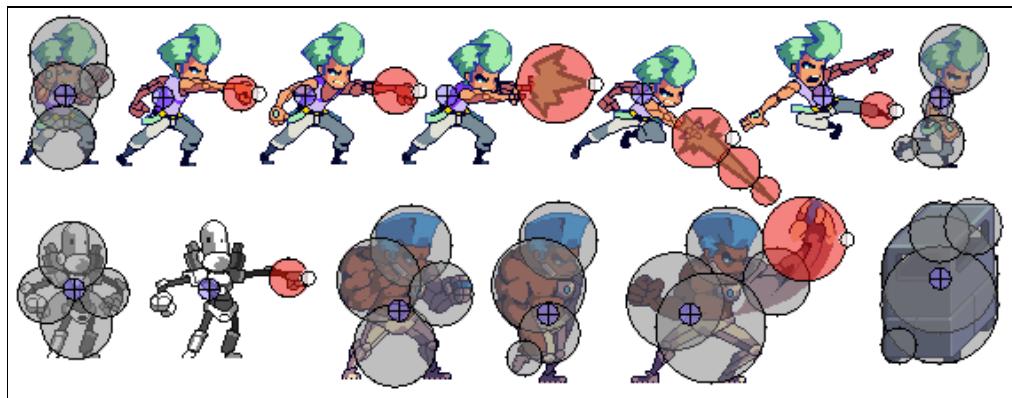
Add all the other relevant images from the Raw folder in the Starter Kit to the left panel, and define their circles one-by-one.

Here's the full list (including the two just covered):

- hero\_idle\_00.png
- hero\_attack\_00\_01.png
- hero\_attack\_01\_01.png

- hero\_attack\_02\_02.png
- hero\_jumpattack\_02.png
- hero\_runattack\_03.png
- robot\_base\_idle\_00.png
- robot\_base\_attack\_03.png
- boss\_idle\_00.png
- boss\_attack\_01.png
- trashcan.png

There's no one correct way to create the circles, so you will just have to use your best judgment. For reference, here are some of my own mappings (excluding detection circles):



There are two frames present above which are not included in the previous list: the walk frame of the hero and the walk frame of the Mohawk dude (more about him in Chapter 7). If you want to be extremely accurate, you can change the positions of the contact circles for every action, but if you feel that you can keep using the same positions for some actions, there is no need to define circles for them. In this case, if you didn't define circles for the walk frame, it will simply use the same positions as the action that came before it, which most likely will be the idle action.

When you're done, hit **Publish** or **Publish As** to export the PLIST file containing the circle information. Save the file as **Circles.plist**.

You'll see how to use this file in the next section – but now you know the general process if you want to use your own artwork!

## Using The Contact Points

**Note:** If you skipped the optional section, you can go to the **Raw** folder of your Starter Kit to find the Circles.plist that I created for you. You can use this to continue on from this point.

You only want to look at the information contained in the **Circles.plist**, so do not add this file to your project. Just open **Circles.plist** with Xcode to view its contents.

Click on the triangle icon beside **bodies** to expand this section, and you will see the list of images for which you defined shapes. You need to drill down to the deepest level to get the circle shape definitions, like so:

Key	Type	Value
Root	Dictionary	(2 items)
metadata	Dictionary	(2 items)
format	Number	1
ptm_ratio	Number	32
bodies	Dictionary	(13 items)
robot_base_idle_00	images	
anchorpoint	String	{ 0.5000,0.5000 }
circles	Array	(4 items)
item 0	Dictionary	(10 items)
density	Number	2
friction	Number	0.0
restitution	Number	0.0
filter_categoryBits	Number	1
filter_groupIndex	Number	0
filter_maskBits	Number	65,535
isSensor	Boolean	NO
id	String	
fixture_type	String	CIRCLE
circle	circle details	
radius	Number	20 ← radius
position	String	{ 1.728,-19.498 } ← offset
Item 1	Dictionary	(10 items)
Item 2	circles	
Item 3	Dictionary	(10 items)
robot_base_attack_03	Dictionary	(2 items)
hero_attack_01_01	Dictionary	(2 items)
hero_attack_02_02	Dictionary	(2 items)
hero_runattack_03	Dictionary	(2 items)
hero_jumpattack_02	Dictionary	(2 items)
trashcan	Dictionary	(2 items)
boss_idle_00	images	

Inside the **image** section, expand **fixtures**, and you'll see Item 0, Item 1, Item 2, Item 3, etc. These are your circles. Expand each to see a row named **circle**. Expand circle and you can see the radius and the offset of that circle.

For each image, you need to get the circle details for all the circles.

Now that you have the means to get the exact radius and offset values of all the circles, you can create the contact circles and attack circles of the hero and robots with ease!

Ideally, every action should have its own arrangement of contact circles and attack circles. This means you want every `ActionSprite` to rearrange these circles whenever it changes actions.

Go to `ActionSprite.h` and add this method definition:

```
-(void)setContactPointsForAction:(ActionState)actionState;
```

Switch to **ActionSprite.m** and add these methods:

```
// 1
-(void)setActionState:(ActionState)actionState
{
    _actionState = actionState;
    [self setContactPointsForAction:actionState];
}

// 2
-(void)setContactPointsForAction:(ActionState)actionState
{
    //override this
}

// 3
-(void)transformPoints
{
    float pixelScaleX = _scaleX / CC_CONTENT_SCALE_FACTOR();
    float pixelScaleY = _scaleY / CC_CONTENT_SCALE_FACTOR();
    int i;
    for (i = 0; i < _contactPointCount; i++)
    {
        _contactPoints[i].position = ccpAdd(_position,
        ccp(_contactPoints[i].offset.x * pixelScaleX,
        _contactPoints[i].offset.y * pixelScaleY));
    }
    for (i = 0; i < _attackPointCount; i++)
    {
        _attackPoints[i].position = ccpAdd(_position,
        ccp(_attackPoints[i].offset.x * pixelScaleX,
        _attackPoints[i].offset.y * pixelScaleY));
    }
}
```

Let's go over this method by method:

1. **setActionState**: executes **setContactPointsForAction**: whenever an **ActionSprite** changes its action state.
2. **setContactPointsForAction**: is supposed to rearrange all the circles based on the action state of the **ActionSprite**. This method is empty here because each **ActionSprite** subclass should have its own implementation of this method.
3. **transformPoints** moves the world position of all **ContactPoints** based on their offset and the sprite's current x direction. This method is executed every time the position of **ActionSprite** changes. Doing this ensures that all the contact circles and attack circles will follow **ActionSprite** wherever it goes (or faces).

Now go to **Hero.m** and add the **setContactPointsForAction:** method:

```
- (void) setContactPointsForAction: (ActionState) actionState
{
    if (actionState == kActionStateIdle)
    {
        [self modifyContactPointAtIndex:0 offset:ccp(3.0, 23.0)
radius:19.0];
        [self modifyContactPointAtIndex:1 offset:ccp(17.0, 10.0)
radius:10.0];
        [self modifyContactPointAtIndex:2 offset:CGPointZero
radius:19.0];
        [self modifyContactPointAtIndex:3 offset:ccp(0.0, -21.0)
radius:20.0];
    }
    else if (actionState == kActionStateWalk)
    {
        [self modifyContactPointAtIndex:0 offset:ccp(8.0, 23.0)
radius:19.0];
        [self modifyContactPointAtIndex:1 offset:ccp(12.0, 4.0)
radius:4.0];
        [self modifyContactPointAtIndex:2 offset:CGPointZero
radius:10.0];
        [self modifyContactPointAtIndex:3 offset:ccp(0.0, -21.0)
radius:20.0];
    }
    else if (actionState == kActionStateAttack)
    {
        [self modifyContactPointAtIndex:0 offset:ccp(15.0, 23.0)
radius:19.0];
        [self modifyContactPointAtIndex:1 offset:ccp(24.5, 4.0)
radius:6.0];
        [self modifyContactPointAtIndex:2 offset:CGPointZero
radius:16.0];
        [self modifyContactPointAtIndex:3 offset:ccp(0.0, -21.0)
radius:20.0];

        [self modifyAttackPointAtIndex:0 offset:ccp(41.0, 3.0)
radius:10.0];
        [self modifyAttackPointAtIndex:1 offset:ccp(41.0, 3.0)
radius:10.0];
        [self modifyAttackPointAtIndex:2 offset:ccp(41.0, 3.0)
radius:10.0];
    }
    else if (actionState == kActionStateAttackTwo)
    {
```

```
[self modifyAttackPointAtIndex:0 offset:ccp(51.6, 2.4)
radius:13.0];
    [self modifyAttackPointAtIndex:1 offset:ccp(51.6, 2.4)
radius:13.0];
        [self modifyAttackPointAtIndex:2 offset:ccp(51.6, 2.4)
radius:13.0];
}
else if (actionState == kActionStateAttackThree)
{
    [self modifyAttackPointAtIndex:0 offset:ccp(61.8, 6.2)
radius:22.0];
    [self modifyAttackPointAtIndex:1 offset:ccp(61.8, 6.2)
radius:22.0];
        [self modifyAttackPointAtIndex:2 offset:ccp(61.8, 6.2)
radius:22.0];
}
else if (actionState == kActionStateRunAttack)
{
    [self modifyAttackPointAtIndex:0 offset:ccp(31.2, -8.8)
radius:10.0];
    [self modifyAttackPointAtIndex:1 offset:ccp(31.2, -8.8)
radius:10.0];
        [self modifyAttackPointAtIndex:2 offset:ccp(31.2, -8.8)
radius:10.0];
}
else if (actionState == kActionStateJumpAttack)
{
    [self modifyAttackPointAtIndex:2 offset:ccp(70.0, -55.0)
radius:8.0];
    [self modifyAttackPointAtIndex:1 offset:ccp(55.0, -42.0)
radius:12.0];
        [self modifyAttackPointAtIndex:0 offset:ccp(34.0, -25.0)
radius:17.0];
}
}
```

This moves all the contact points and attack points of the hero based on the offsets and radii taken from Circles.plist for each `actionState`. In this code, some values differ from Circles.plist slightly, since the actual values used are up to you.

**Note:** Some of the other action states aren't included in the movement of contact and attack circles.

You can still change them if you want, but it's not necessary for this Starter Kit because the sprite's positions are very similar per action. Adding them here will also unnecessarily lengthen the Starter Kit. The important thing is that you understand how to do this on your own. ☺

Switch to **Robot.m** and add the same method but with different values:

```
-(void)setContactPointsForAction:(ActionState)actionState
{
    if (actionState == kActionStateIdle)
    {
        [self modifyContactPointAtIndex:0 offset:ccp(1.7, 19.5)
radius:20.0];
        [self modifyContactPointAtIndex:1 offset:ccp(-15.5, 3.5)
radius:16.0];
        [self modifyContactPointAtIndex:2 offset:ccp(17.0, 2.1)
radius:14.0];
        [self modifyContactPointAtIndex:3 offset:ccp(-0.8, -18.5)
radius:19.0];
    }
    else if (actionState == kActionStateWalk)
    {
        [self modifyContactPointAtIndex:0 offset:ccp(8.0, 23.0)
radius:19.0];
        [self modifyContactPointAtIndex:1 offset:ccp(12.0, 4.0)
radius:4.0];
        [self modifyContactPointAtIndex:2 offset:CGPointZero
radius:10.0];
        [self modifyContactPointAtIndex:3 offset:ccp(0.0, -21.0)
radius:20.0];
    }
    else if (actionState == kActionStateAttack)
    {
        [self modifyContactPointAtIndex:0 offset:ccp(15.0, 23.0)
radius:19.0];
        [self modifyContactPointAtIndex:1 offset:ccp(24.5, 4.0)
radius:6.0];
        [self modifyContactPointAtIndex:2 offset:CGPointZero
radius:16.0];
        [self modifyContactPointAtIndex:3 offset:ccp(0.0, -21.0)
radius:20.0];
        [self modifyAttackPointAtIndex:0 offset:ccp(45.0, 6.5)
radius:10.0];
    }
}
```

{

This is more of the same. The robot contains fewer arrangements because it has fewer actions than the hero.

Build and run. For now just make sure that the code compiles correctly and that everything still works on all devices. You have not yet implemented actual collision detection, so you won't see much of a difference in the game functionality.



## Debug drawing

In the previous section, you defined a lot of circles that are very important for collision handling. The problem is, right now there's no way to confirm whether you did the right thing. You don't want to have to wait until detecting and resolving collisions to find out that you'd placed your circles incorrectly!



As always, it's a good idea to make sure everything is working at each step. So you're going to draw all the circles and rectangles you've been creating via debug drawing.

Go to **Defines.h** and add this definition:

```
#define DRAW_DEBUG_SHAPES 1
```

**DRAW\_DEBUG\_SHAPES** will be your on/off switch for debug mode. If set to 1, then debug drawing is turned on. When it is 0, then debug drawing is off.

Switch to **GameLayer.m** and add the following code:

```
#if DRAW_DEBUG_SHAPES

-(void)draw
{
    [super draw];
    [self drawShapesForActionSprite:_hero];

    ActionSprite *actionSprite;
    CCARRAY_FOREACH(_robots, actionSprite)
    {
        [self drawShapesForActionSprite:actionSprite];
    }
}

-(void)drawShapesForActionSprite:(ActionSprite *)sprite
{
    if (sprite.visible)
    {
        int i;

        ccDrawColor4B(0, 0, 255, 255);
        ccDrawCircle(sprite.position, sprite.detectionRadius, 0,
16, NO);

        ccDrawColor4B(0, 255, 0, 255);
        for (i = 0; i < sprite.contactPointCount; i++)
        {
            ccDrawCircle(sprite.contactPoints[i].position,
sprite.contactPoints[i].radius, 0, 8, NO);
        }

        ccDrawColor4B(255, 0, 0, 255);
        for (i = 0; i < sprite.attackPointCount; i++)
        {

```

```

        ccDrawCircle(sprite.attackPoints[i].position,
sprite.attackPoints[i].radius, 0, 8, NO);
    }

    ccDrawColor4B(255, 255, 0, 255);
    ccDrawRect(sprite.feetCollisionRect.origin,
ccp(sprite.feetCollisionRect.origin.x +
sprite.feetCollisionRect.size.width,
sprite.feetCollisionRect.origin.y +
sprite.feetCollisionRect.size.height));
}
}

#endif

```

Note the `#if DRAW_DEBUG_SHAPES` and `#endif` directives before and after the methods – these are important. They tell the compiler whether or not to include the code contained within the `#if-#endif` block. If `DRAW_DEBUG_SHAPES` has its value set to 0, then as far as Xcode is concerned, the code inside the block does not exist.

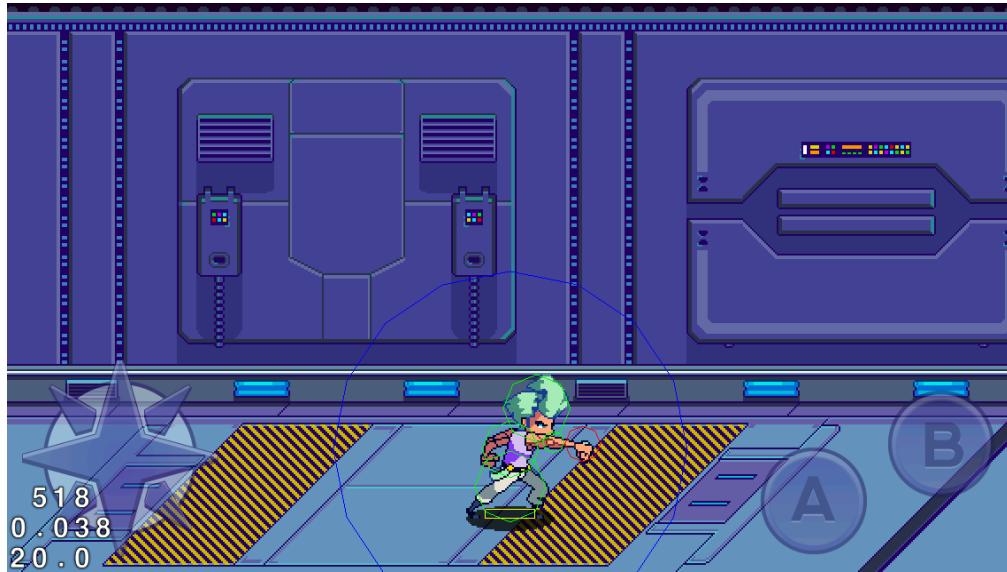
Every Cocos2D node object (`ccNode` – the root class of all Cocos2D objects) has a draw method. Any OpenGL-related drawing code must be executed inside this method, or else it won't work as expected. In this case, the draw method executes `drawShapesForActionSprite:` for the hero and all the robots.

`drawShapesForActionSprite:` draws an `ActionSprite`'s detection, contact and attack circles. It also draws the rectangle representing the `ActionSprite`'s feet.

These are the steps for drawing the shapes:

1. `ccDrawColor4B` changes the color of all the shapes that are drawn after it. Detection circles will be blue, contact circles green, attack circles red and feet rectangles yellow.
2. `ccDrawCircle` draws a circle given an origin, radius, angle, vertices and a Boolean to decide if a line will be drawn from the center to the edge of the circle. Since the angle of the circle doesn't matter, the third parameter is 0, and the Boolean is set to NO. The number of vertices dictates how smoothly the circle will be drawn. Since the Detection Circle is bigger, you give it 16 vertices, while the rest have only 8.
3. `ccDrawRect` draws a rectangle starting from the lower-left corner to the upper-right corner.

Build and run, and you should see all the shapes on the screen!



You should try out all of the different states (walking, running, jumping, and punching) and make sure the shapes are OK for each state.

**Note:** The way debug drawing works at this point isn't very efficient. Just take one look at the topmost number in the lower-left corner – the number of OpenGL draw calls being executed – and you will notice that it has shot up to a very high value.

Every `ccDraw` call that's executed is one more draw call. Ideally, you would batch all your shapes and draw them in one go, but this requires a bit of OpenGL knowledge, and may be more effort than it's worth here.

The good thing is that your debug drawing doesn't need to be efficient. With the switch you created in `Defines.h`, you can turn it off anytime! You certainly won't leave it on once you've finished building the game.

## Collision detection and resolution

If you haven't done so, disable debug drawing mode by going to **Defines.h** and changing the value of `DRAW_DEBUG_SHAPES` to 0. You already know that the shapes have been positioned correctly, so there's no need to keep debug drawing enabled.

Now that the collision strategy is in place, you need to come up with a sound plan for detecting and resolving collisions.

You need to implement the following:

- **Collision Trigger:** When to check for collisions.
- **Collision Detection:** Check intersecting circles.

- **Collision Resolution:** The appropriate response for each type of collision.

When you implemented moving the hero using the D-pad, you actually went through these same three steps.

There, the collision trigger is when the hero sets a `desiredPosition`. The collision detection is when the `desiredPosition` intersects with the bounds of the allowable movement area. The collision resolution is when the hero's `position` is restricted to the allowed area.

For this section, you will go about these steps in reverse order – starting with writing the `ActionSprite` response to collisions. This way, you can be sure that a given `ActionSprite` will react properly when a collision is detected, and you also know how to detect collisions when a call for collision detection is triggered.

## Resolving collisions

It's time to bring on the pain! Go to `ActionSprite.m` and add the following method:

```
-(void)hurtWithDamage:(float)damage force:(float)force
direction:(CGPoint)direction
{
    if (_actionState > kActionStateNone && _actionState <
kActionStateKnockedOut)
    {
        if (_jumpHeight > 0)
        {
            [self knockoutWithDamage:damage direction:direction];
        }
        else
        {
            [self stopAllActions];
            [self runAction:_hurtAction];
            self.actionState = kActionStateHurt;
            _hitPoints -= damage;
            self.desiredPosition = ccp(self.position.x +
direction.x * force, self.position.y);
            if (_hitPoints <= 0)
            {
                [self knockoutWithDamage:0 direction:direction];
            }
        }
    }
}
```

If the sprite is in the middle of a jump, then you want to knock out the character via `knockoutWithDamage:`. If not, then `hurtWithDamage:` will subtract the damage

amount from the current `hitPoints`, move the `desiredPosition` using the force value multiplied by the x and y directions, and execute the hurt action.

If `hitPoints` reach or fall below zero, then the sprite will be knocked out, but this time with zero damage because the damage has already been applied.

Still in `ActionSprite.m`, add `knockoutWithDamage::`:

```
-(void)knockoutWithDamage:(float)damage
direction:(CGPoint)direction
{
    if (_actionState != kActionStateKnockedOut && _actionState != kActionStateDead && _actionState != kActionStateRecover && _actionState != kActionStateNone)
    {
        _hitPoints -= damage;
        [self stopAllActions];
        [self runAction:_knockedOutAction];
        _jumpVelocity = kJumpForce / 2.0;
        _velocity = ccp(direction.x * _runSpeed, direction.y * _runSpeed);
        [self flipSpriteForVelocity:ccp(-_velocity.x, _velocity.y)];
        self.actionState = kActionStateKnockedOut;
    }
}
```

`knockoutWithDamage:` subtracts the damage from the current `hitPoints` and executes the damage action.

When a sprite is knocked out, its body will not only be forced back, it will also rise from the ground. To achieve this effect, you set the `jumpVelocity` to half the default jump force (`kJumpForce`) and set the movement `velocity` to `runSpeed` in a certain direction. You also make the sprite face the direction opposite to where he's headed.

There are two paths to take after being knocked out. A sprite can either die or recover from being knocked out.



Dying won't just be `ActionSprite`'s responsibility because `GameLayer` also has to be informed. When the hero dies, for example, he needs to tell the `GameLayer` that he has died, so that `GameLayer` executes the necessary game logic.

Similar to `ActionDPad` and `ActionButton`, you will use the delegation pattern with `ActionSprite`.

Go to **ActionSprite.h** and make the following changes:

```
//add these before @interface
@class ActionSprite;

@protocol ActionSpriteDelegate <NSObject>

-(BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite;
-(BOOL)actionSpriteDidDie:(ActionSprite *)actionSprite;

@end

//add this property to ActionSprite
@property(nonatomic, weak)id <ActionSpriteDelegate> delegate;
```

This should be familiar by now. When the `ActionSprite` attacks, then the `delegate` should execute `actionSpriteDidAttack:`, passing in a reference to which `ActionSprite` performed the action. Similarly, when an `ActionSprite` dies, then the `delegate` should execute `actionSpriteDidDie:`.

Now all `ActionSprites` can die peacefully, knowing that the game can attend to their death. ☺

Switch to **ActionSprite.m** and add these methods:

```
-(void)die
{
    self.actionState = kActionStateDead;
    _velocity = CGPointMakeZero;
    _jumpHeight = 0;
```

```

    _jumpVelocity = 0;
    _hitPoints = 0.0;
    [_delegate actionSpriteDidDie:self];
    [self runAction:_dieAction];
}

-(void)recover
{
    if (_actionState == kActionStateKnockedOut)
    {
        self.actionState = kActionStateNone;
        _velocity = CGPointMakeZero;
        _jumpVelocity = 0;
        [self performSelector:@selector(getUp) withObject:nil
afterDelay:0.5];
    }
}

-(void)getUp
{
    self.actionState = kActionStateRecover;
    [self runAction:_recoverAction];
}

```

When the `ActionSprite` dies, it simply changes the state to `kActionStateDead`, zeroes out all movement values, executes the death animation action and informs the delegate that it has died.

If the `ActionSprite` is knocked out but did not die, then it goes through a two-step recovery process. First, in `recover`, it has absolutely no action state and zero movement values. After half a second, `setUp` is executed, the `ActionSprite`'s state changes to `kActionStateRecover`, and the recover animation action is executed.

Still in `ActionSprite.m`, add this clause to the `if-else` block in `update`:

```

else if (_actionState == kActionStateKnockedOut)
{
    _desiredPosition = ccpAdd(_groundPosition,
ccpMult(_velocity, delta));
    _jumpVelocity -= kGravity * delta;
    _jumpHeight += _jumpVelocity * delta;

    if (_jumpHeight <= 0)
    {
        if (_hitPoints <= 0)
        {

```

```
        [self die];
    }
else
{
    [self recover];
}
}
```

When an `ActionSprite` gets knocked out, you don't want to instantly trigger the die or recover actions, especially while the sprite is still in mid-air.

In the above code, the update method updates the `desiredPosition` of the `ActionSprite` when it is knocked out so that it moves across the screen, and it will wait until `jumpHeight` goes back to zero. When this happens, it checks if the `ActionSprite` has enough `hitPoints` to recover or if it should die, and triggers the appropriate method for each situation.

To complete these actions, you still have to retrofit both the `Hero` and `Robot` classes with their respective actions.

Go to `Hero.m` and add the following inside `init` (below the existing animation blocks):

```
//hurt animation
CCAnimation *hurtAnimation = [self
animationWithPrefix:@"hero_hurt" startFrameIdx:0 frameCount:3
delay:1.0/12.0];
self.hurtAction = [CCSequence actions:[CCAnimate
actionWithAnimation:hurtAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];

//knocked out animation
CCAnimation *knockedOutAnimation = [self
animationWithPrefix:@"hero_knockout" startFrameIdx:0 frameCount:5
delay:1.0/12.0];
self.knockedOutAction = [CCAnimate
actionWithAnimation:knockedOutAnimation];

//die action
self.dieAction = [CCBlink actionWithDuration:2.0 blinks:10.0];

//recover animation
CCAnimation *recoverAnimation = [self
animationWithPrefix:@"hero_setup" startFrameIdx:0 frameCount:6
delay:1.0/12.0];
```

```
self.recoverAction = [CCSequence actions:[CCAnimate
actionWithAnimation:recoverAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(jumpLand)], nil];
```

Switch to **Robot.m** and also add the following inside `init` (again, below the existing animation blocks):

```
//hurt animation
AnimateGroup *hurtAnimationGroup = [self
animateGroupWithActionWord:@"hurt" frameCount:3 delay:1.0/12.0];
self.hurtAction = [CCSequence actions:hurtAnimationGroup,
[CCCallFunc actionWithTarget:self selector:@selector(idle)], nil];

//knocked out animation
self.knockedOutAction = [self
animateGroupWithActionWord:@"knockout" frameCount:5
delay:1.0/12.0];

//die action
self.dieAction = [CCBlink actionWithDuration:2.0 blinks:10.0];

//recover animation
self.recoverAction = [CCSequence actions:[self
animateGroupWithActionWord:@"getup" frameCount:6 delay:1.0/12.0],
[CCCallFunc actionWithTarget:self selector:@selector(idle)], nil];
```

It's the same old stuff. You create the actions in the same fashion as before for the `Hero` and `Robot` classes. The hurt and recover actions transition to the idle action when they finish, while the death action just makes the sprite blink ten times in two seconds.

## Detecting Collisions

Ready for phase two? Open **GameLayer.h** and make the following changes:

```
//change the <ActionDPadDelegate, ActionButtonDelegate> to
<ActionDPadDelegate, ActionButtonDelegate, ActionSpriteDelegate>
```

Here you just mark the layer as implementing `ActionSpriteDelegate` so it can be notified when an action sprite attacks another sprite or dies.

Then switch to **GameLayer.m** and make the following changes:

```
//add this in initHero, after self.hero = [Hero node];
_hero.delegate = self;

//add in initRobots, right after Robot *robot = [Robot node];
```

```
robot.delegate = self;

//add these methods
-(BOOL)actionSpriteDidDie:(ActionSprite *)actionSprite
{
    return NO;
}

-(BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    return NO;
}

-(BOOL)collisionBetweenAttacker:(ActionSprite *)attacker
andTarget:(ActionSprite *)target atPosition:(CGPoint *)position
{
    //first phase: check if they're on the same plane
    float planeDist = attacker.shadow.position.y -
target.shadow.position.y;

    if (fabsf(planeDist) <= kPlaneHeight)
    {
        int i, j;
        float combinedRadius = attacker.detectionRadius +
target.detectionRadius;

        //initial detection
        if (ccpDistanceSQ(attacker.position, target.position) <=
combinedRadius * combinedRadius)
        {
            int attackPointCount = attacker.attackPointCount;
            int contactPointCount = target.contactPointCount;

            ContactPoint attackPoint, contactPoint;

            //secondary detection
            for (i = 0; i < attackPointCount; i++)
            {
                attackPoint = attacker.attackPoints[i];

                for (j = 0; j < contactPointCount; j++)
                {
                    contactPoint = target.contactPoints[j];
                    combinedRadius = attackPoint.radius +
contactPoint.radius;
                }
            }
        }
    }
}
```

```

        if (ccpDistanceSQ(attackPoint.position,
contactPoint.position) <= combinedRadius * combinedRadius)
{
    //attack point collided with contact point
    position->x = attackPoint.position.x;
    position->y = attackPoint.position.y;
    return YES;
}
}
}
}
}
return NO;
}

```

This makes `GameLayer` the `delegate` for `hero` and `robot` and adds the required protocol methods, which don't do anything for now.

The collision detection method first checks if the attacker and target are on the same plane by checking if their y-coordinate distance is within the `kPlaneHeight` range.

Take a look at these three situations:



Out of the three, only the third/rightmost scenario should be considered a collision. This is why you have to check the y-position of the sprites first in this section of the code:

```

float planeDist = attacker.shadow.position.y -
target.shadow.position.y;

if (fabsf(planeDist) <= kPlaneHeight)

```

If they are too far apart, then there is no need to check if their collision shapes hit one another. Otherwise, the first two events would be considered collisions, which would make for some weird gameplay.

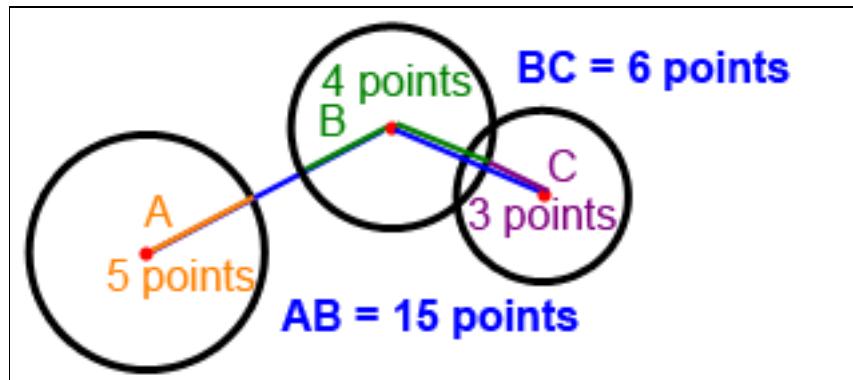
If the y-coordinate check is passed, then the method continues to the next phase – the initial detection, which consists of this section:

```
float combinedRadius = attacker.detectionRadius +
target.detectionRadius;

//initial detection
if (ccpDistanceSQ(attacker.position, target.position) <=
combinedRadius * combinedRadius)
```

It determines if the detection circles of the two sprites intersect by checking the distance of their center positions.

If their distance is less than the sum of the radii of the two circles, then they must intersect one another.



Normally you could have gone with the following instead of using `ccpDistanceSQ`:

```
if (ccpDistance(attacker.position, target.position) <=
combinedRadius)
```

This is good, and will also work. However, solving for distance requires a square root operation, which is expensive in terms of computing power. Doing it too often could result in a performance hit. Instead, you just square both sides, which is more efficient.

`ccpDistanceSQ` gets the distance of two points right before the square root operation is done, and you compare it with the squared version of `combinedRadius`.

If the detection circles intersect with one another, then it's on to the final phase – the secondary detection, as shown in this section:

```
int attackPointCount = attacker.attackPointCount;
```

```

int contactPointCount = target.contactPointCount;

ContactPoint attackPoint, contactPoint;

//secondary detection
for (i = 0; i < attackPointCount; i++)
{
    attackPoint = attacker.attackPoints[i];

    for (j = 0; j < contactPointCount; j++)
    {
        contactPoint = target.contactPoints[j];
        combinedRadius = attackPoint.radius +
contactPoint.radius;

        if (ccpDistanceSQ(attackPoint.position,
contactPoint.position) <= combinedRadius * combinedRadius)
        {
    
```

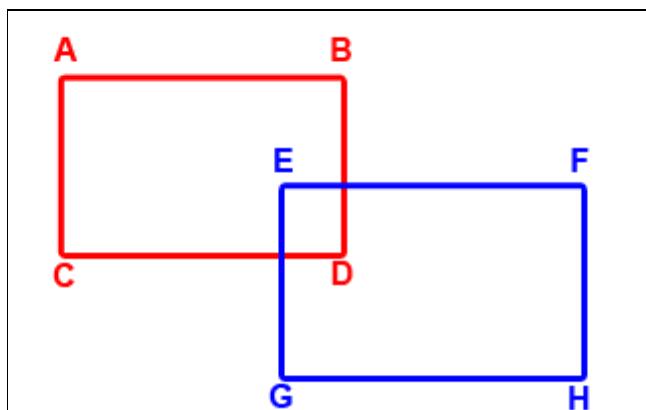
In this phase, you loop through all attack circles of the attacker and check if they intersect with any contact circles of the target, using the same distance method as before. If any one of them intersects, this piece of code is triggered:

```

//attack point collided with contact point
position->x = attackPoint.position.x;
position->y = attackPoint.position.y;
return YES;
    
```

You store the position of the attack and return `YES` to signify that a collision has occurred.

**Note:** If you're wondering why you used circles instead of rectangles to check for collisions, take a look at the following diagram:



Given the above example, if you wanted to check if the left rectangle and right rectangle intersected, you would have to check a lot of points. You would check if point E, F, G or H has an x-coordinate greater than points A and C, but less than that of B and D, and a y-coordinate greater than points D and C, but less than that of A and B, and so on.

It's too many comparisons for just two rectangles. Imagine the situation if you had more rectangles! When using circles, it will always be a comparison of only two points for each pair of circles.

You now have both collision resolution and collision detection in place. There's only one piece missing – the collision triggers.

## Triggering collisions

The attack action of an `ActionSprite` should trigger collision detection. However, you only want it to occur for the exact frame that shows the punch. If your sprite's punch animation consists of five sprite frames, but only really shows the punch at the fourth frame, you don't want the collision occurring during the first frame.

You need to detect the exact moment when the sprite's displayed frame changes to the one showing the attack image. Thankfully, there's a method for that!

When a `CCAnimation` is triggered, it just calls `ccsprite's setDisplayFrame:` every time it changes sprite frames. Your job now is to rewrite `setDisplayFrame:` so that it asks `GameLayer` to check collisions for specific frames.

Go to `Hero.m` and add this method:

```
- (void)setDisplayFrame:(CCSpriteFrame *)newFrame
{
    [super setDisplayFrame:newFrame];

    CCSpriteFrame *attackFrame = [[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"hero_attack_00_01.png"];

    if (newFrame == attackFrame)
    {
        [self.delegate actionSpriteDidAttack:self];
    }
}
```

Now switch to `Robot.m`, and add this method:

```
- (void)setDisplayFrame:(CCSpriteFrame *)newFrame
```

```
{
    [super setDisplayFrame:newFrame];

    CCSpriteFrame *attackFrame = [[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"robot_base_attack_03.png"];

    if (newFrame == attackFrame)
    {
        [self.delegate actionSpriteDidAttack:self];
    }
}
```

When `setDisplayFrame:` is triggered, you first call `setDisplayFrame:` on the super class so that the sprite still changes the displayed frame. You then check to see if the new frame is equal to the exact frame showing the sprite's punching action. If it is, then `ActionSprite` tells the `delegate`, `GameLayer`, that it just attacked.

Go back to `GameLayer.m` and replace the contents of `actionSpriteDidAttack:` with the following:

```
-(BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    // 1
    BOOL didHit = NO;
    if (actionSprite == _hero)
    {
        CGPoint attackPosition;
        Robot *robot;
        CCARRAY_FOREACH(_robots, robot)
        {
            // 2
            if (robot.actionState < kActionStateKnockedOut &&
robot.actionState != kActionStateNone)
            {
                if ([self collisionBetweenAttacker:_hero
andTarget:robot atPosition:&attackPosition])
                {
                    [robot hurtWithDamage:_hero.attackDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0.0)];
                    didHit = YES;
                }
            }
        }
        return didHit;
    }
}
```

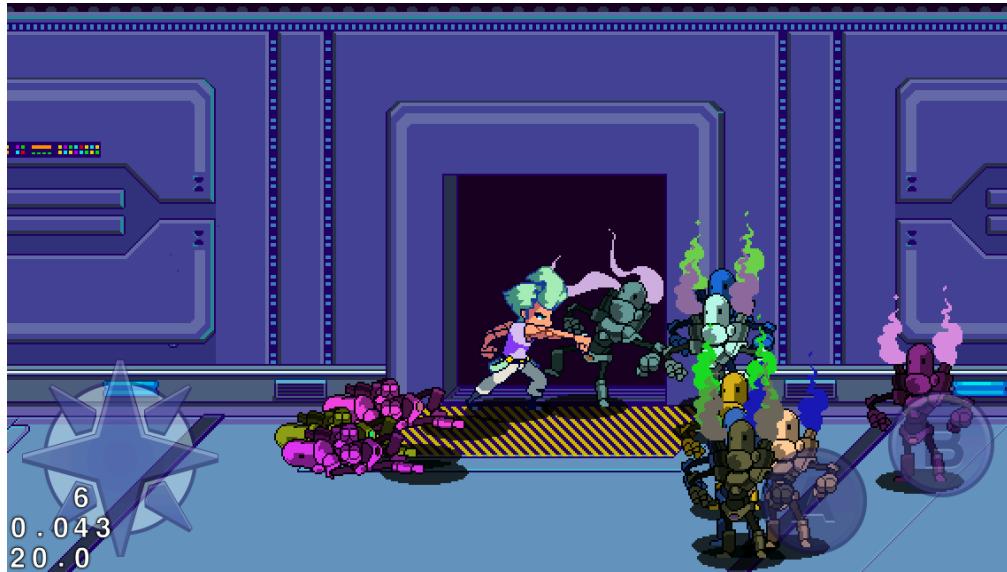
```
// 3
else
{
    if (_hero.actionState < kActionStateKnockedOut &&
    _hero.actionState != kActionStateNone)
    {
        CGPoint attackPosition;
        if ([self collisionBetweenAttacker:actionSprite
andTarget:_hero atPosition:&attackPosition])
        {
            [_hero hurtWithDamage:actionSprite.attackDamage
force:actionSprite.attackForce
direction:ccp(actionSprite.directionX, 0.0)];
            didHit = YES;
        }
    }
    return didHit;
}
```

Let's go over this section by section.

1. The new code first checks if the attacker is the hero, in which case it tests for collisions using `collisionBetweenAttacker:andTarget:atPosition:` with the hero as the attacker, and each robot as the target.
2. You pass in a `CGPoint` in which the method stores the attack position. If a collision occurs with a robot (that isn't knocked out and has a valid state), then that robot is hurt with the attack damage and direction of the hero.
3. If the attacker is not the hero, then it is automatically assumed that it is one of the enemies. The code checks for a collision between the robot attacker and the target hero, and hurts the hero if there was a collision.

Congratulations, the collision handling is now complete!

Build and run, and put some hurt on those robots!



## Knockback

Wait-a-minute... when the robots are hurt or knocked out, shouldn't their position be affected by the hero's attack force? Yes, yes they should.

To fix this, you need to put code in `GameLayer` that updates the robots and their positions. Open `GameLayer.m` and make the following changes:

```
//add inside update:, before [self updatePositions]
Robot *robot;
CCARRAY_FOREACH(_robots, robot)
{
    [robot update:delta];
}

//add inside updatePositions, after setting _hero.position
Robot *robot;
CCARRAY_FOREACH(_robots, robot)
{
    if (robot.actionState > kActionStateNone)
    {
        posY = MIN(floorHeight + (robot.centerToBottom -
robot.feetCollisionRect.size.height), MAX(robot.centerToBottom,
robot.desiredPosition.y));
        robot.groundPosition = ccp(robot.desiredPosition.x, posY);
        robot.position = ccp(robot.groundPosition.x,
robot.groundPosition.y + robot.jumpHeight);
```

```
        if (robot.actionState == kActionStateDead &&
    _hero.groundPosition.x - robot.groundPosition.x >= CENTER.x +
    robot.contentSize.width/2 * kScaleFactor)
    {
        robot.visible = NO;
        [robot reset];
    }
}
```

You update all the robots, similar to how you updated the hero before, but with slight modifications.

First, you don't restrict the x-position of the robots. The robots are allowed to go beyond the screen, which is logical since they appear all over the map.

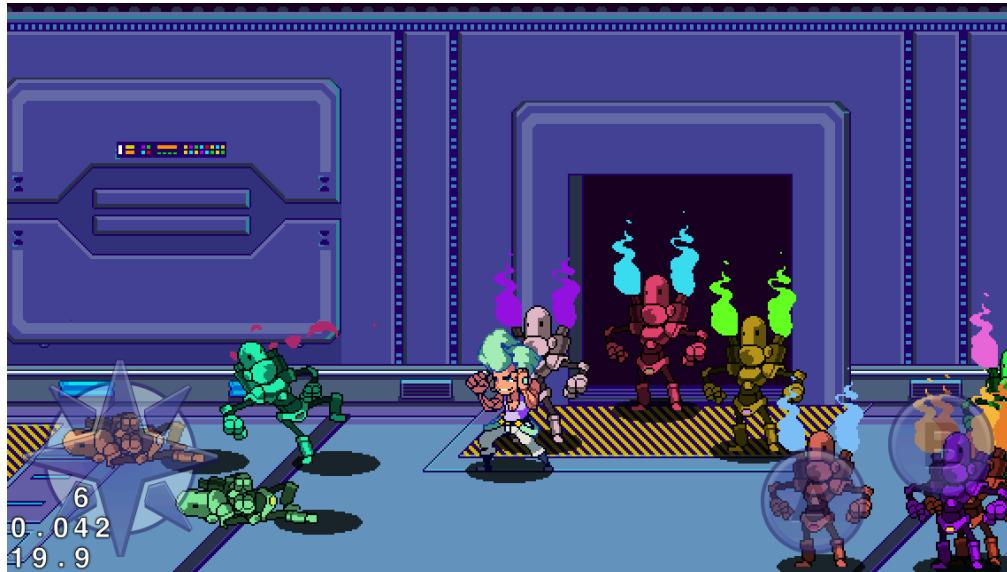
Second, when a dead robot's distance from the hero is larger than the screen's viewable area, then the robot becomes invisible and is reset.

Switch to **ActionSprite.m** and add the **reset** method:

```
-(void)reset
{
    self.actionState = kActionStateNone;
    self.position = OFFSCREEN;
    self.desiredPosition = OFFSCREEN;
    self.groundPosition = OFFSCREEN;
    self.hitPoints = _maxHitPoints;
}
```

This sends the sprite to the off-screen coordinate you defined earlier in **Defines.h**, refills the character's hit points to max and changes its action state to **kActionStateNone**.

Build and run, and push those robots around!



Congratulations – this is now starting to look like a real Beat 'Em Up Game!

Stay tuned for the next chapter, where you will learn how to make the poor robots begin to fight back!

**Challenge:** If you did the challenges so far, your hero has a completely custom look. Use the information from the “Optional Exercise” section to define your own contact and attack circles for your custom sprites.

When you’re done, see if you can modify your attack (and drawings) to be a two fisted attack – so your hero can punch enemies in front and behind him at the same time. You’ll need two attack contact points for this to work.

# Chapter 5: Brainy Bots

Currently, your robots don't have a lot of smarts – they just stand there waiting for you to beat them up.

In this chapter, you're going to change that, and add some brains into your robots – converting them into seek-and-destroy death machines out for the hero!

In addition, you're going to create a new event-based level system so you can control exactly where and when the robots will spawn in the level to challenge the player. Finally, you'll wrap things up with a dramatic entrance for your hero.

So let the butt-kicking resume!

## I, robot: decision-based AI

You punch and punch and they all fall down! But the robots never return the attack? What fun is that?

Though punching dummy robots might be satisfying, it's a lot more challenging to win a game against smart opponents. To get the spirit of competition going, each robot should move as a player does, or at least like a less-skilled player. For this to happen, the robots should be given a form of artificial intelligence.

In this section, you will fit the robots with brains. Brains to help them decide what to do in every situation. The AI that you create will be based on decisions. You will give each robot a chance to decide on a course of action at specific time intervals.

### The robot decider

In this game, there are only four things a robot can opt to do once it sees the hero:

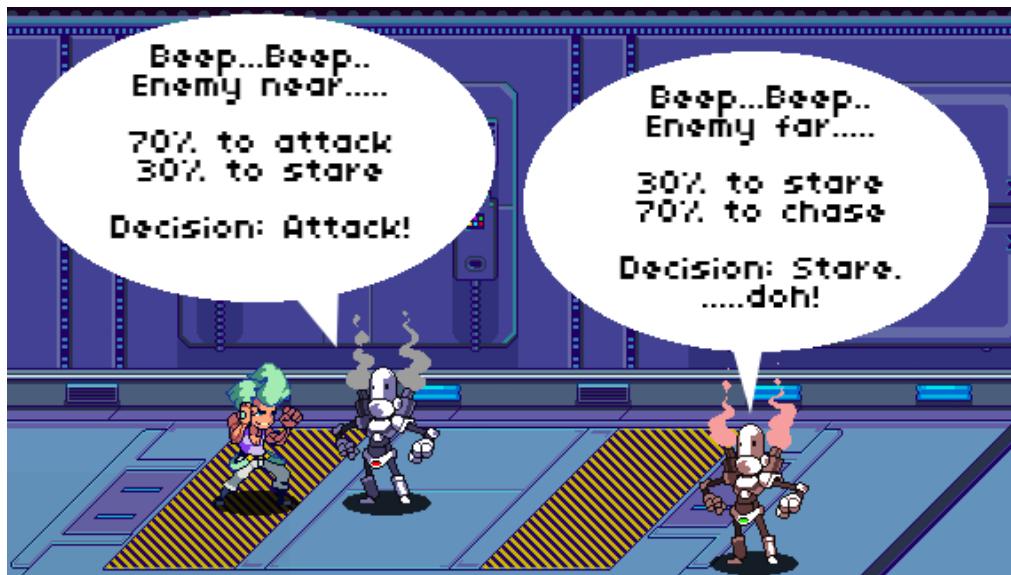
1. Stand idle.
2. Move closer to the hero to surround him.
3. Chase or move towards the hero.

#### 4. Attack the hero head-on.

For every situation, the decision depends upon different factors. If the robot can reach the hero, then it can take advantage of its proximity and attack. However, if the robot is too far and punches the air, it will look stupid. Yes, even robots feel shame. ☺

Each of the four decisions has a different probability of being picked. Further, these probabilities differ from one situation to another.

Take, for example:



The robot closer to the hero will choose to attack 70% of the time, and it does just that here. On the other hand, the farther-away robot will choose to move closer to the hero 70% of the time, but in this instance, it chose to remain idle.

In every situation, the robot weighs its decisions. The resulting decision is greatly affected by the weight of each option. The higher the percentage for an option, the more likely the robot is to choose it above the alternative.

Time to give your robots some brains!

Go to **Defines.h** and add these definitions:

```
typedef enum _AIDecision
{
    kDecisionAttack = 0,
    kDecisionStayPut,
    kDecisionChase,
    kDecisionMove
} AIDecision;
```

Next select the **PompaDroid** group in Xcode, go to **File\New\Group** and name the new group **AI**.

Then select the **AI** group, and go to **File\New\File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **NSObject** for Subclass of, click **Next** and name the new file **WeightedDecision**.

Open **WeightedDecision.h** and replace its contents with the following:

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

@interface WeightedDecision : NSObject {
    AIDecision _decision;
    int _weight;
}

@property(nonatomic, assign)AIDecision decision;
@property(nonatomic, assign)int weight;

+(id)decisionWithDecision:(AIDecision)decision
andWeight:(float)weight;
-(id)initWithDecision:(AIDecision)decision
andWeight:(float)weight;

@end
```

I'll explain this in a moment – first switch to **WeightedDecision.m** and add the implementation:

```
+ (id)decisionWithDecision:(AIDecision)decision
andWeight:(float)weight
{
    return [[self alloc] initWithDecision:decision
andWeight:weight];
}

-(id)initWithDecision:(AIDecision)decision andWeight:(float)weight
{
    if ((self = [super init]))
    {
        _decision = decision;
        _weight = weight;
    }
    return self;
}
```

**AIDecision** represents the possible actions a robot can choose. In this game there are four (idle, move, chase and attack), but you could add others.

**WeightedDecision** is a class that represents an **AIDecision** and a weight (or probability) for that decision to be chosen. There's nothing in this class except initializer methods. It merely stores the decision type and the weight.

Select the **AI** group, and go to **File\New\File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **NSObject** for Subclass of, click **Next** and name the new file **ArtificialIntelligence**.

Go to **ArtificialIntelligence.h** and replace its contents with the following:

```
#import <Foundation/Foundation.h>
#import "cocos2d.h"
#import "ActionSprite.h"
#import "WeightedDecision.h"

@interface ArtificialIntelligence : NSObject {
    WeightedDecision *_attackDecision;
    WeightedDecision *_idleDecision;
    WeightedDecision *_chaseDecision;
    WeightedDecision *_moveDecision;
}

@property(nonatomic, assign)float decisionDuration;
@property(nonatomic, weak)ActionSprite *controlledSprite;
@property(nonatomic, weak)ActionSprite *targetSprite;
@property(nonatomic, assign)AIDecision decision;
@property(nonatomic, strong)CCArray *availableDecisions;

+(id)aiWithControlledSprite:(ActionSprite *)controlledSprite
targetSprite:(ActionSprite *)targetSprite;
-(id)initWithControlledSprite:(ActionSprite *)controlledSprite
targetSprite:(ActionSprite *)targetSprite;
-(void)update:(ccTime)delta;

@end
```

First you import the necessary classes that the AI will use. Then you create four **WeightedDecisions**, one for each of the **AIDecision** types.

Additionally, you declare the following properties:

- **decisionDuration**: How long a decision lasts before a new decision is made.
- **controlledSprite**: The ActionSprite being controlled by the AI.
- **targetSprite**: The ActionSprite the AI wants to kill!

- **decision:** The current decision made by the AI.
- **availableDecisions:** An array of the four available decisions. This will make it easy to iterate over the decisions later.

Switch to **ArtificialIntelligence.m** and add these methods:

```
+ (id)aiWithControlledSprite:(ActionSprite *)controlledSprite
targetSprite:(ActionSprite *)targetSprite
{
    return [[self alloc] initWithControlledSprite:controlledSprite
targetSprite:targetSprite];
}

-(id)initWithControlledSprite:(ActionSprite *)controlledSprite
targetSprite:(ActionSprite *)targetSprite
{
    if ((self = [super init]))
    {
        self.controlledSprite = controlledSprite;
        self.targetSprite = targetSprite;
        self.availableDecisions = [CCArray arrayWithCapacity:4];

        _attackDecision = [WeightedDecision
decisionWithDecision:kDecisionAttack andWeight:0];
        _idleDecision = [WeightedDecision
decisionWithDecision:kDecisionStayPut andWeight:0];
        _chaseDecision = [WeightedDecision
decisionWithDecision:kDecisionChase andWeight:0];
        _moveDecision = [WeightedDecision
decisionWithDecision:kDecisionMove andWeight:0];

        [_availableDecisions addObject:_attackDecision];
        [_availableDecisions addObject:_idleDecision];
        [_availableDecisions addObject:_chaseDecision];
        [_availableDecisions addObject:_moveDecision];

        _decisionDuration = 0;
    }
    return self;
}
```

When an AI is created, it stores references to two **ActionSprites** – **controlledSprite** and **targetSprite**. It also creates and stores four **WeightedDecisions** using the **availableDecisions** array. These **WeightedDecisions** have zero weight by default, because there is not yet a concrete situation for the AI to determine the weights.

Lastly you set the `decisionDuration` to zero, since you want the object to immediately make a new decision when it is activated.

Still in `ArtificialIntelligence.m`, add this method:

```
-(AIDecision)decideWithAttackWeight:(int)attackWeight
idleWeight:(int)idleWeight chaseWeight:(int)chaseWeight
moveWeight:(int)moveWeight
{
    int totalWeight = attackWeight + idleWeight + chaseWeight +
moveWeight;
    _attackDecision.weight = attackWeight;
    _idleDecision.weight = idleWeight;
    _chaseDecision.weight = chaseWeight;
    _moveDecision.weight = moveWeight;

    int choice = random_range(1, totalWeight);
    int minInclusive = 1;
    int maxExclusive = minInclusive;
    int decisionWeight;

    WeightedDecision *weightedDecision;
    CCARRAY_FOREACH(_availableDecisions, weightedDecision)
    {
        decisionWeight = weightedDecision.weight;
        if (decisionWeight > 0)
        {
            maxExclusive = minInclusive + decisionWeight;

            if (choice >= minInclusive && choice < maxExclusive)
            {
                self.decision = weightedDecision.decision;
                return weightedDecision.decision;
            }
        }
        minInclusive = maxExclusive;
    }
    return -1;
}
```

This is the main decision-making function of the AI. Given a weight for each of the four `WeightedDecisions`, it will output a chosen `AIDecision`.

It goes like this:

Decision	Weight	Range (Inclusive)
attack	50	1 to 50
idle	30	51 to 80
chase	10	81 to 90
move	10	91 to 100

pick a random number from 1 to 100

result: 45  
decision: attack

result: 78  
decision: chase

The AI will select a random number between 1 and the total weight of the four decisions, and choose a decision based on the result. Each decision is assigned a range according to its weight. If the random number is within a decision's range, then it is the chosen decision.

If you use the values in the chart as an example, the selection logic will be similar to this:

```
if (choice >= 1 && choice < 51)
{
    //attack
}
else if (choice >= 51 && choice < 81)
{
    //idle
}
else if (choice >= 81 && choice < 91)
{
    //chase
}
else if (choice >= 91 && choice < 101)
{
    //move
}
```

You now have a flexible decision-maker in your hands. You can easily add more decisions if you wish.

## Action follows thought

Onto the next step – what does the AI do with each decision?

Still in **ArtificialIntelligence.m**, add this method:

```
-(void)setDecision:(AIDecision)decision
```

```

{
    _decision = decision;

    if (_decision == kDecisionAttack)
    {
        [_controlledSprite attack];
        _decisionDuration = frandom_range(0.25, 1.0);
    }
    else if (_decision == kDecisionStayPut)
    {
        [_controlledSprite idle];
        _decisionDuration = frandom_range(0.25, 1.5);
    }
    else if (_decision == kDecisionChase)
    {
        float reachDistance = _targetSprite.centerToSides +
        _controlledSprite.attackPoints[0].offset.x +
        _controlledSprite.attackPoints[0].radius;
        CGPoint reachPosition = ccp(_targetSprite.groundPosition.x
        + (random_sign * reachDistance), _targetSprite.groundPosition.y);
        CGPoint moveDirection = ccpNormalize(ccpSub(reachPosition,
        _controlledSprite.groundPosition));
        [_controlledSprite walkWithDirection:moveDirection];
        _decisionDuration = frandom_range(0.5, 1.0);
    }
    else if (_decision == kDecisionMove)
    {
        float randomX = random_sign * frandom_range(20.0 *
        kPointFactor, 100.0 * kPointFactor);
        float randomY = random_sign * frandom_range(10.0 *
        kPointFactor, 40.0 * kPointFactor);
        CGPoint randomPoint = ccp(_targetSprite.groundPosition.x +
        randomX, _targetSprite.groundPosition.y + randomY);
        CGPoint moveDirection = ccpNormalize(ccpSub(randomPoint,
        _controlledSprite.groundPosition));
        [_controlledSprite walkWithDirection:moveDirection];
        _decisionDuration = frandom_range(0.25, 0.5);
    }
}

```

`setDecision` ensures that changing the `decision` property also changes the behavior of the `controlledSprite`.

- **If the decision is to attack:**

The AI tells `controlledSprite`, the `ActionSprite` that receives the decision, to execute its attack method. The next decision will be made at a random interval between 0.25 to 1 second.

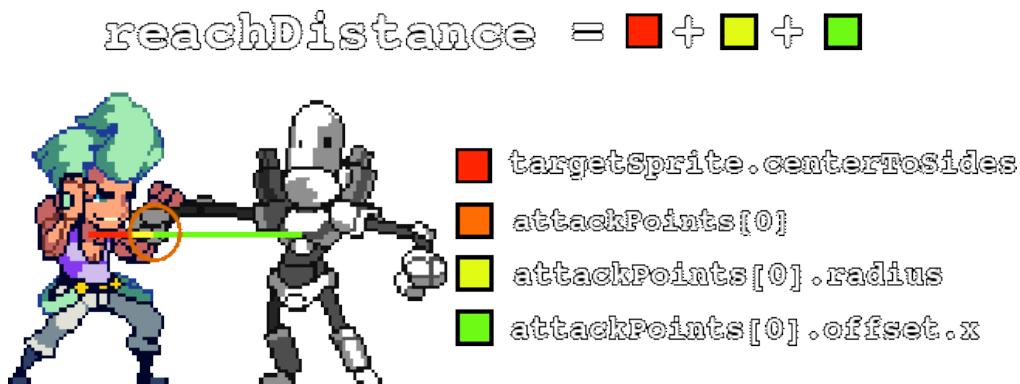
- **If the decision is to stay put:**

The AI tells the `controlledSprite` to execute its idle method and wait for a random time interval between 0.25 to 1 second.

- **If the decision is to chase:**

The AI will compute the `reachDistance`. This is the minimum x-distance that the `controlledSprite`'s attack circle needs to be in order to reach the sides of the `targetSprite`. Think of the `reachDistance` as "how close the robot needs to be to the hero in order to punch him."

To better visualize it, take a look at this example:



Once you know how close the robot needs to be, you can calculate where the robot needs to stand – the `reachPosition`. This is the position either to the right or the left of the `targetSprite` where the `controlledSprite` can reach it (you'll choose randomly between the two).

Then, it computes the `direction` (normal vector values) from the `controlledSprite`'s `groundPosition` to the `reachPosition`. Think of this as the angle between `controlledSprite` and `targetSprite` in terms of x and y, where x and y can only have values from -1 to 1. This is similar to the direction value used in moving `ActionSprite`.

The AI then tells `ActionSprite` to walk towards the `moveDirection` over a random period of 0.5 to 1 seconds. Note that since the target is also moving, the robot might not reach the target – but that's OK for this game.

- **If the decision is to move:**

The AI will pick a random position surrounding the `targetSprite`, get the direction from `controlledSprite` to this position, and ask the `controlledSprite` to move towards this position for 0.25 to 0.5 seconds.

The AI can now behave differently depending on the decision it makes. Before making decisions, it will also need to assess situations and give proper weight to each possible action.

Still in **ArtificialIntelligence.m**, add this method:

```
- (void)update:(ccTime)delta
{
    if (_targetSprite && _controlledSprite &&
    _controlledSprite.actionState > kActionStateNone)
    {
        //1
        float distanceSQ =
        ccpDistanceSQ(_controlledSprite.groundPosition,
        _targetSprite.groundPosition);
        float planeDist =
        fabsf(_controlledSprite.shadow.position.y -
        _targetSprite.shadow.position.y);

        float combinedRadius = _controlledSprite.detectionRadius +
        _targetSprite.detectionRadius;

        BOOL samePlane = NO;
        BOOL canReach = NO;
        BOOL tooFar = YES;
        BOOL canMove = NO;

        //2
        if (_controlledSprite.actionState == kActionStateWalk ||
        _controlledSprite.actionState == kActionStateIdle)
        {
            canMove = YES;
        }

        if (canMove)
        {
            //measure distances
            if (distanceSQ <= combinedRadius * combinedRadius)
            {
                tooFar = NO;

                //3
                if (fabsf(planeDist) <= kPlaneHeight)
                {
                    samePlane = YES;
                }
            }
        }
    }
}
```

```
        //check if any attack points can reach the
target's contact points
        int attackPointCount =
_controlledSprite.attackPointCount;
        int contactPointCount =
_targetSprite.contactPointCount;

        int i, j;
        ContactPoint attackPoint, contactPoint;
        for (i = 0; i < attackPointCount; i++)
{
    attackPoint =
_controlledSprite.attackPoints[i];

        for (j = 0; j < contactPointCount; j++)
{
    contactPoint =
_targetSprite.contactPoints[j];
    combinedRadius = attackPoint.radius +
contactPoint.radius;

    if
(ccpDistanceSQ(attackPoint.position, contactPoint.position) <=
combinedRadius * combinedRadius)
    {
        canReach = YES;
        break;
    }
}

        if (canReach)
{
    break;
}
}
}

//4
if (canReach && _decision == kDecisionChase)
{
    self.decision = kDecisionStayPut;
}
```

```
//5
if (_decisionDuration > 0)
{
    _decisionDuration -= delta;
}
else
{
    //6
    if (tooFar)
    {
        self.decision = [self decideWithAttackWeight:0
idleWeight:20 chaseWeight:80 moveWeight:0];
    }
    else
    {
        //7
        if (samePlane)
        {
            if (canReach)
            {
                self.decision = [self
decideWithAttackWeight:70 idleWeight:15 chaseWeight:0
moveWeight:15];
            }
            else
            {
                self.decision = [self
decideWithAttackWeight:0 idleWeight:20 chaseWeight:50
moveWeight:30];
            }
        }
        else
        {
            self.decision = [self
decideWithAttackWeight:0 idleWeight:50 chaseWeight:40
moveWeight:10];
        }
    }
}
}
```

That is one long method. Let's tackle what happens section-by-section:

1. `update`: will run constantly as long as the `controlledSprite` is alive, and it has a `targetSprite` assigned. The first part stores values to be used later. You've seen these values before – in collision handling. You store the squared distance between the controlled `ActionSprite` and the target, the plane distance between them and their combined detection radius.
2. You set `canMove` to `YES` if the `controlledSprite` is idle or walking. This means that the controlled sprite is allowed to move only when it's in either of these two states.
3. You check if the two sprites are on the same plane by comparing their plane distance with the `kPlaneHeight` constant. If they are in the same plane, you also check if `controlledSprite`'s attack circle can reach any of `targetSprite`'s contact circles. This is sort of like collision detection prediction.
4. If the `controlledSprite` is moving towards the target, the AI will make it stop as soon as the target is reachable.
5. The AI will count `decisionDuration` down to zero. Once finished, it will make a new decision.
6. If the `controlledSprite` is far from the `targetSprite`, the AI chooses between the idle and chase actions, with more preference towards chase.
7. If the two sprites are near one another, it assigns different weights for each decision based on the checks in step 3.

The AI is now complete. Now to inject some brains into the robots!

Go to `GameLayer.h` and add this property:

```
@property(nonatomic, strong)CCArray *brains;
```

Switch to `GameLayer.m` and do the following:

```
//add to top of file
#import "ArtificialIntelligence.h"

//add to init, inside if ((self = [super init])) right after [self
initRobots];
[self initBrains];

//add this method
-(void)initBrains
{
    self.brains = [[CCArray alloc]
initWithCapacity:_robots.count];
    ArtificialIntelligence *brain;
    Robot *robot;

    CCARRAY_FOREACH(_robots, robot)
```

```
{  
    brain = [ArtificialIntelligence  
aiWithControlledSprite:robot targetSprite:_hero];  
    [_brains addObject:brain];  
}  
}  
  
//add inside update:, before Robot *robot  
ArtificialIntelligence *brain;  
CCARRAY_FOREACH(_brains, brain)  
{  
    [brain update:delta];  
}
```

You create a brain per robot and set the hero as the target. Then you simply call the AI's `update:` from `GameLayer`'s `update:` so that all the brains continuously assess the situation.

One last thing – the AI's calculations are dependent on the robot's attack circle, but if you remember from the last chapter, the robot's attack circle doesn't get set until the robot performs an attack.

You can easily fix this. Open `Robot.m` and add this line inside `init:`:

```
//add right after self.attackPoints = malloc(sizeof(ContactPoint)  
* self.attackPointCount)  
[self modifyAttackPointAtIndex:0 offset:ccp(45.0, 6.5)  
radius:10.0];
```

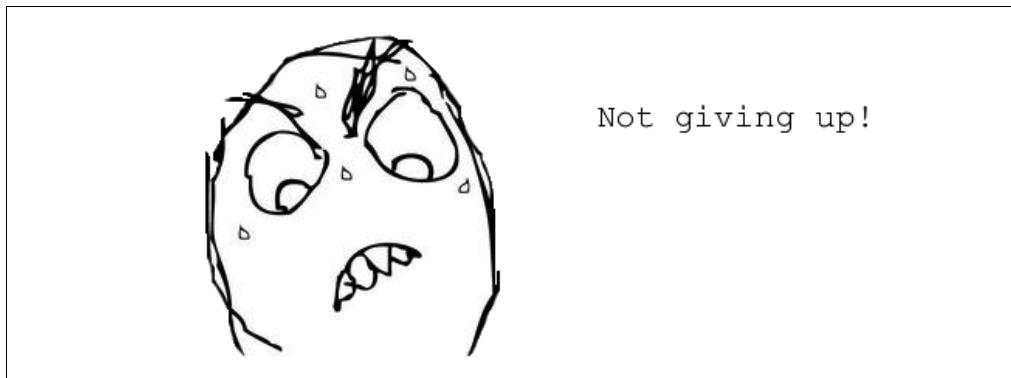
The above code positions the attack circle of the robot during initialization.

Build and run, then fight for your life!



## Your .plist event playbill

If you're like me, you probably didn't last a minute against the teeming horde of robots. 😊



It's not your fault, though – there's simply too many of them! Once the game starts, all 50 robots instantly lock on to the hero as their target and charge towards him no matter where they are.

To prevent all enemies from becoming active at once, you need a way to control the number of enemies at specific points in the game. To do this, you will have to implement game events.

Your game will have three kinds of events:

- **Free Roam Events:** These allow the player to move around the map freely.
- **Battle Events:** During these, for a fixed area, a predetermined number of robots will come and fight the player.
- **Scripted Events:** These are events when the player cannot control the hero.

This way, enemies only attack during battle events, during which the hero will have to fight everyone off before moving forward. And as you'll see in a few sections, game events are also useful for other purposes.

You will be storing events in a property list file (.plist), which are just XML files with a format that Xcode understands. A property list allows you to easily define values using common Objective-C types such as strings and numbers and store them in arrays and dictionaries.

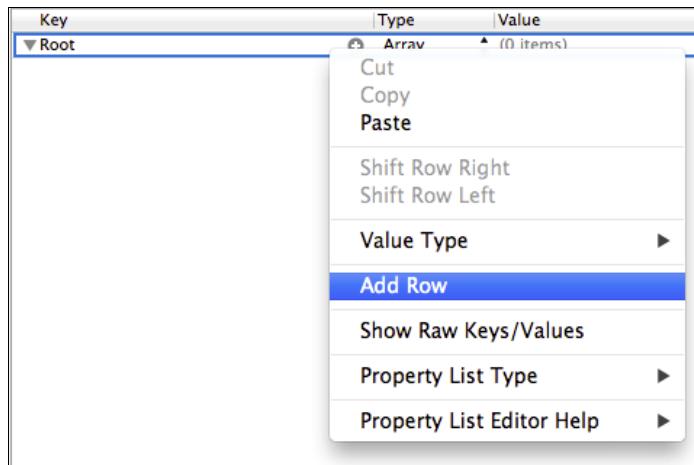
In Xcode, select the **Resources** folder, go to **File\New\File**, choose **iOS\Resource\Property List** and click **Next**. Name the file **Levels** and click **Create**.

Select **Levels.plist** and you'll see your empty property list.



Right now, there is a single root element that is a dictionary. Click on the word Dictionary under the Type column, and change this to Array.

Control-click on the Root key and select **Add Row**, as shown below:



**Note:** To add a row, you can also press the plus (+) button beside the key name, or select a row and press the Enter key.

A new row will show up with these columns:

- **Key:** This will be the string you'll use to get this row's value. Think of it as a name-based identification system. Note that string-based keys are only used for

items in dictionaries. Since your item is in an array, it is assigned a number starting from zero.

- **Type:** The data type of this row (e.g., string, number, array, dictionary).
- **Value:** The value of the entry. If the type is string, then it will contain letters. If the type is number, then it will contain numbers.

Pay close attention to the indentation and location of Item 0. In a property list, the indentation indicates the membership of a current row – or to which array or dictionary the item belongs.

Since Item 0 is indented one space away from Root and is listed below Root, it is a member of the Root array. If you add more rows with the same indentation, then they are all members of the Root array. If you change Item 0 into an array/dictionary, you can add items below it with a deeper level of indentation.

For Levels.plist, the rows under Root will be the levels in the game, where Item 0 is the first level. Since a level will have its own set of information, Item 0 needs to be able to store a set of values.

So, click on the Type column of Item 0 and change it to **Dictionary**.

Next you want to add a detail entry for the first level, so click the arrow next to the Item 0 key to expand the item, then click the plus (+) button next to the Item 0 key to create a new sub-item.

Key	Type	Value
Root	Array	(1 item)
Item 0	Dictionary	(0 items)

First each level needs to know which tile map to use. Change the Key of this new sub-item to **TileMap**, leave the Type at **String** and set the Value to **map\_level1.tmx**.

Key	Type	Value
Root	Array	(1 item)
Item 0	Dictionary	(1 item)
TileMap	String	map_level1.tmx

This will hold the filename of the **TMXTiledMap** you will load for this level.

Next this level needs a list of Battle Events. Add a new row under Item 0, change the Key to **BattleEvents** and the Type to Array.

Notice how you cannot change the value of an array. This is because an array simply contains a list of sub-items. The same is true for dictionaries.

The BattleEvents array will contain all battle events for this level.

Expand the BattleEvents item, then click the plus (+) button on the same row to create a new sub-item for BattleEvents.

For this sub-item, change the Type to **Dictionary**. Because this item is inside an array, you cannot assign a Key for it. It is set as Item 0 by default.

Key	Type	Value
▼ Root	Array	(1 item)
▼ Item 0	Dictionary	(2 items)
TileMap	String	map_level1.tmx
▼ BattleEvents	Array	(1 item)
▶ Item 0	Dictionary	(0 items)

You'll consider each row in BattleEvents to be a single battle event. Each battle event will contain:

- The tile position, in terms of column number, that triggers the event. It's like a booby-trapped column of tiles that, when stepped upon, will send hordes of enemies the hero's way.
- A list of enemies included in the event.

It's time to create the first event following the above specifications! Expand Item 0 under BattleEvents and then click the plus (+) button on the same row to create a new sub-item.

Set the Key to **Column**, change the Type to **Number** and the Value to **15**. This means that this particular event will be triggered once the hero steps onto the 15<sup>th</sup> column of tiles from the left side of the map.

Next create another new row under BattleEvent's Item 0, set the Key to **Enemies** and change the Type to Array.

Key	Type	Value
▼ Root	Array	(1 item)
▼ Item 0	Dictionary	(2 items)
TileMap	String	map_level1.tmx
▼ BattleEvents	Array	(1 item)
▼ Item 0	Dictionary	(2 items)
Column	Number	15
▶ Enemies	Array	(0 items)

This will be another array of dictionaries indicating each enemy's attributes. Expand Item 0 under Enemies and add rows as follows:

Key	Type	Value
▼ Root	Array	(1 item)
▼ Item 0	Dictionary	(2 items)
TileMap	String	map_level1.tmx
▼ BattleEvents	Array	(1 item)
▼ Item 0	Dictionary	(2 items)
Column	Number	15
▼ Enemies	Array	(1 item)
▼ Item 0	Dictionary	(4 items)
Type	Number	0
Color	Number	4
Row	Number	2
Offset	Number	-1

Each enemy item under Enemies will be a dictionary containing a list of enemy attributes. These are the attributes:

- The **Type** of enemy. A robot? A boss enemy? Or something else? This will be an ID number that will correspond to an object/character in the game.
- The **Color** of the enemy. Remember the `colorSet` attribute of the `Robot` class? You will use it here. The number 4 means that it will use `kColorRandom`.
- The **Row** of the enemy. On which row will the enemy spawn? Remember that there are only three rows from the bottom of the map that enemies (or the hero) can stand upon.
- The **Offset** of the enemy in terms of screen width. How far from the starting column of the event will the enemy spawn? A value of -1 means that it will spawn at the left edge of the screen, whereas a value of 1 means that it will spawn at the right edge of the screen.

Both the Row and Offset values make it easy to position each enemy when a battle event occurs and they save you from having to calculate `CGPoints` yourself.

At this point, you can go crazy and add as many battle events and as many enemies as you want, as long as you follow the above format.

If that's not your cup of tea, or if you want to save that effort for later, I've made a pre-edited version of `Levels.plist` for you to copy into your game, overwriting your version of the file.

The version for this section is under **Versions\Chapter5** in the resources, in case you need it. It's the enemy playbook!

## Event-based battles

Now that you have a list of battle events, you must change the game to support these events.

First go to **Defines.h** and add these definitions:

```
typedef enum _EventState
{
    kEventStateScripted = 0,
    kEventStateFreeWalk,
    kEventStateBattle,
    kEventStateEnd
} EventState;

typedef enum _EnemyType
{
    kEnemyRobot = 0,
    kEnemyBoss
} EnemyType;
```

You'll use **EventState** to differentiate between the three game events mentioned in the previous section, while you'll use **EnemyType** to differentiate between enemies. The Type key for each enemy in the property list you just created corresponds to an **EnemyType**.

Go to **GameLayer.h** and do the following:

```
//add these instance variables
int _activeEnemies;
float _viewPointOffset;
float _eventCenter;

//add these properties
@property(nonatomic, strong)CCArray *battleEvents;
@property(nonatomic, strong)NSDictionary *currentEvent;
@property(nonatomic, assign)EventState eventState;
```

Here's what the above does:

- **battleEvents**: A collection of all battle events that haven't happened yet.
- **currentEvent**: The currently active battle event. This will not be in **battleEvents** anymore.
- **eventState**: The game's current state, as per the **EventState** definition.
- **activeEnemies**: The number of enemies still alive in **currentEvent**.
- **viewPointOffset** and **eventCenter**: You'll use these later on to adjust the camera view.

Switch to **GameLayer.m** and do the following:

```

//remove this line from init
[self initTileMap:@"map_level1.tmx"];

//and replace it with this
[self loadLevel:0];

//add this method
-(void)loadLevel:(int)level
{
    NSString *levelsPlist = [[NSBundle mainBundle]
pathForResource:@"Levels" ofType:@"plist"];
    NSMutableArray *levelArray = [[NSMutableArray alloc]
initWithContentsOfFile:levelsPlist];
    NSDictionary *levelData = [[NSDictionary alloc]
initWithDictionary:levelArray[level]];

    NSString *tileMap = [levelData objectForKey:@"TileMap"];
    [self initTileMap:tileMap];

    //store the events
    _battleEvents = [CCArray arrayWithNSArray:[levelData
objectForKey:@"BattleEvents"]];
}

```

Instead of initializing the tile map based on a name provided in the code, you now initialize it based on the name provided in the property list.

In `loadLevel:`, you open `Levels.plist` and store the root array into `levelArray`. Then you get the row for the specified level from `levelArray`. This time, it's a dictionary because it contains sub-items describing the level.

One of these sub-items is the `TileMap` row. You get the value for this row and plug it into `initTileMap:` to load the map.

Then you store the `BattleEvents` into the `battleEvents` array. These correspond to the structure you set in the property list. If, for example, you loaded the first level (level 0), it would look something like this:

Key	Type	Value
▼ Root <b>levelArray</b>	Array	(1 item)
▼ Item 0 <b>levelData</b>	Dictionary	(2 items)
TileMap <b>tileMap</b>	String	map_level1.tmx
▼ BattleEvents <b>_battleEvents</b>	Array	(1 item)

Build and run! Check that the tile map shows up correctly.



If the tile map is still there, it means that `Levels.plist` loaded successfully, and `battleEvents` should be there, too.

## Bring on the horde

Now you can implement the battles!

In `GameLayer.m`, replace the current `initRobots` implementation with the following:

```
-(void)initRobots {
    int robotCount = 50;
    self.robots = [[CCArray alloc] initWithCapacity:robotCount];

    for (int i = 0; i < robotCount; i++) {
        Robot *robot = [Robot node];
        robot.delegate = self;
        [_actors addChild:robot.shadow];
        [_actors addChild:robot.smoke];
        [_actors addChild:robot];
        [_actors addChild:robot.belt];
        [_robots addObject:robot];

        robot.scale *= kScaleFactor; //scaling
    simplified
        robot.shadow.scale *= kScaleFactor;
        robot.position = OFFSCREEN; //this changed
        robot.groundPosition = robot.position;
        robot.desiredPosition = robot.position;
        robot.visible = NO; //added this line
    }
}
```

```
//this line was
removed: [robot idle];
    robot.colorSet = kColorRandom;
}
}
```

The new method has a few changes:

- **Scaling simplified:** It doesn't really matter if `scaleX` is positive or negative now since it will be set when the robots move. So you just have to make sure that the robot is scaled properly as per the device.
- **OFFSCREEN position:** When a robot is created, it is sent to the depths of nowhere, never to be seen – until it spawns, that is!
- **Visibility Off:** It's not enough that the robots are located off-screen. You also make them invisible so that Cocos2D doesn't waste time drawing them.
- **Inactive State:** You remove the line that made the robots run their idle action, leaving them inactive. This way, the AI won't be able to control the robots yet.

You sent all the robots away because you don't want all 50 of them chasing the hero all the time. Instead, you will spawn robots as you need them, based on the current battle event.

Still in `GameLayer.m`, add this method:

```
-(void)spawnEnemies:(CCArray *)enemies fromOrigin:(float)origin
{
    NSDictionary *enemyData;
    Robot *robot;

    int row, type, color;
    float offset;

    CCARRAY_FOREACH(enemies, enemyData)
    {
        row = [[enemyData objectForKey:@"Row"] floatValue];
        type = [[enemyData objectForKey:@"Type"] intValue];
        offset = [[enemyData objectForKey:@"Offset"] floatValue];

        if (type == kEnemyRobot)
        {
            color = [[enemyData objectForKey:@"Color"] intValue];

            //get an unused robot
            CCARRAY_FOREACH(_robots, robot)
            {
                if (robot.actionState == kActionStateNone)
```

```
        {
            [robot stopAllActions];
            robot.visible = NO;
            robot.groundPosition = ccp(origin + (offset *
(CENTER.x + robot.centerToSides)), robot.centerToBottom +
_tileMap.tileSize.height * row * kPointFactor);
            robot.position = robot.groundPosition;
            robot.desiredPosition = robot.groundPosition;
            [robot setColorSet:color];
            [robot idle];
            robot.visible = YES;
            break;
        }
    }
}
}
```

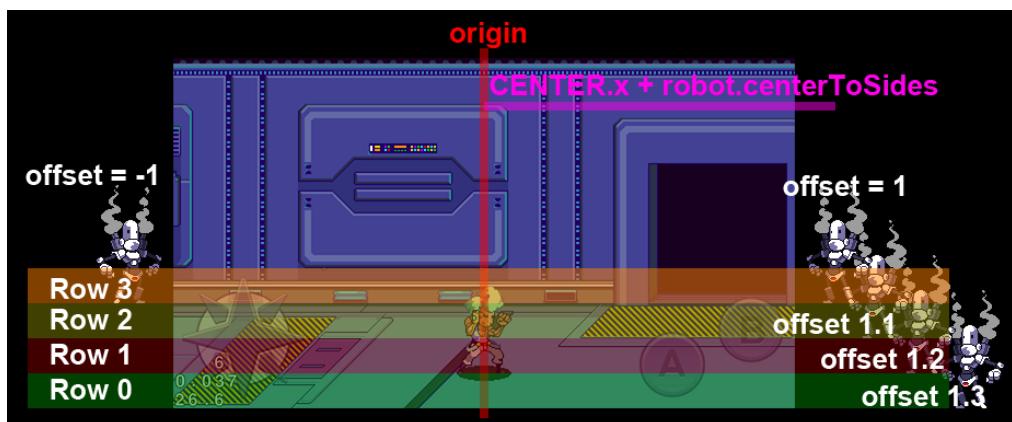
Similar to `loadLevel:`, `spawnEnemies:fromOrigin:` expects certain data to come from `Levels.plist`. It needs two values as parameters:

- **enemies:** This is an array of enemy data. It will come from the Enemies row of the battle event in the property list.
- **origin:** This is the center point of the battle event.

`spawnEnemies:fromOrigin:` cycles through all the enemy data contained in `enemies` and retrieves the Type, Color, Row and Offset of each enemy. Next, if it determines that the Type of the enemy is `kEnemyRobot` (or 0 in numerical form), it will traverse through the robots array and retrieve an unused robot.

Once it gets an inactive robot, it will change the properties of that robot based on the attributes taken from the enemy data.

The only special calculation done here is for the position. It is calculated as shown on this diagram:



Let's discuss it per coordinate:

- **The x-coordinate:**

The x-position is always measured from the origin, which is the center of the current screen. The robot's x-position is then decided by multiples of the size of half the screen and the robot's `centerToSides` attribute.

Look at the diagram: `CENTER.x + robot.centerToSides` is just enough to position the robot exactly at the start of the non-visible area of the screen on either side. Consider it as the minimum spawning distance.

You multiply the minimum value by the offset value. An offset of 1 places the robot to the right of the origin. A value of -1 positions it on the left.

- **The y-coordinate:**

The y-position is simpler. It is measured using the height of each tile and the robot's `centerToBottom` attribute. Since the robot's position is always based on the center, to position the robot's feet at exactly the bottom part of the tile, you add `centerToBottom` to the value of `row` multiplied by the tile height.

Given that the tile height is 32 points, a `row` value of 1 means the robot's y position will be exactly  $32 + \text{centerToBottom}$  points up from the bottom of the screen.

You want to call `spawnEnemies:fromOrigin:` every time the hero steps on a booby-trapped tile column. You've already stored this and the rest of the data in `battleEvents`. All that's left is to activate these events.

Still in `GameLayer.m`, do the following:

```
//add to the end of update:  
[self updateEvent];  
  
//add these methods  
-(void)updateEvent  
{  
    if (_eventState == kEventStateBattle && _activeEnemies <= 0)  
    {  
        float maxCenterX = _tileMap.mapSize.width *  
_tileMap.tileSize.width * kPointFactor - CENTER.x;  
        float cameraX = MAX(MIN(_hero.position.x, maxCenterX),  
CENTER.x);  
        _viewPointOffset = cameraX - _eventCenter;  
        _eventState = kEventStateFreeWalk;  
    }  
    else if (_eventState == kEventStateFreeWalk)  
    {  
        [self cycleEvents];  
    }  
}
```

```

        }

    -(void)cycleEvents
    {
        NSDictionary *event;
        int column;
        float tileSizeWidth = _tileMap.tileSize.width * kPointFactor;
        CCARRAY_FOREACH(_battleEvents, event)
        {
            column = [[event objectForKey:@"Column"] intValue];
            float maxCenterX = _tileMap.mapSize.width *
                _tileMap.tileSize.width * kPointFactor - CENTER.x;
            float columnPosition = column * tileSizeWidth - tileSizeWidth/2;
            _eventCenter = MAX(MIN(columnPosition, maxCenterX),
                CENTER.x);
            if (_hero.position.x >= _eventCenter) // 1
            {
                _currentEvent = event;
                _eventState = kEventStateBattle;
                CCArrray *enemyData = [CCArrray arrayWithNSArray:[event
                    objectForKey:@"Enemies"]];
                _activeEnemies = enemyData.count;
                [self spawnEnemies:enemyData fromOrigin:_eventCenter];
                [self setViewpointCenter:ccp(_eventCenter,
                    _hero.position.y)];
                break;
            }
        }

        if (_eventState == kEventStateBattle)
        {
            [_battleEvents removeObject:_currentEvent];
        }
    }
}

```

You call `updateEvent` from `update:` so that it runs for every frame. When the current event is a free-roaming event (`kEventStateFreeWalk`), you cycle through all the battle events, get the column position of each event and also calculate the `eventCenter` position, which is at the center of the screen.

If the hero walks onto (or past) the column position of an event (marked as section 1):

1. You activate the battle event by changing `eventState` to `kEventStateBattle`.

2. You retrieve the array of enemy data from the activated event, store the number of enemies in `activeEnemies` and use both the `enemyData` array and the `eventCenter` position to spawn enemies.
3. You set the viewpoint to the event's center.
4. You store the activated battle event in `currentEvent` and remove it from the `battleEvents` array.

On the other hand, when the current event is `kEventStateBattle` and there are no active enemies left, you switch the state back to `kEventStateFreeWalk` and store a value named `viewPointOffset`.

You store the intended position of the center of the camera based on where the hero is, and get the difference between that value and the center of the event. Basically it's the distance between the battle camera and the free-roaming camera.

Before you test the game again, add this method to `GameLayer.m`:

```
- (void)onEnterTransitionDidFinish
{
    [super onEnterTransitionDidFinish];
    _eventState = kEventStateFreeWalk;
}
```

This simply changes the `eventState` to `kEventStateFreeWalk` once the scene appears on the screen.

Build and run the game, and walk towards the 15<sup>th</sup> column (if you are using the scene configuration mentioned earlier) for your first battle event!



## Fighting is mandatory!

Of course, if you play for a bit, you'll realize that you can run the hero towards the end of the map and the robots aren't able to keep up. He doesn't have to fight!

Having battle events is pointless if the player can just run away from them, at least in a Beat 'Em Up Game. The simplest way to solve this problem is to restrict both the camera and the player's movement during these events.

In **GameLayer.m**, do the following:

```
//replace [self setViewpointCenter:_hero.position] in update:  
if (_eventState == kEventStateFreeWalk || _eventState ==  
kEventStateScripted)  
{  
    [self setViewpointCenter:_hero.position];  
}  
  
//replace updatePositions with the following  
-(void)updatePositions  
{  
    float mapWidth = _tileMap.mapSize.width *  
    _tileMap.tileSize.width * kPointFactor;  
    float floorHeight = 3 * _tileMap.tileSize.height *  
    kPointFactor;  
    float posX, posY;  
  
    if (_hero.actionState > kActionStateNone)  
    {  
        // 1  
        if (_eventState == kEventStateFreeWalk)  
        {  
            posX = MIN(mapWidth -  
            _hero.feetCollisionRect.size.width/2,  
            MAX(_hero.feetCollisionRect.size.width/2,  
            _hero.desiredPosition.x));  
            posY = MIN(floorHeight + (_hero.centerToBottom -  
            _hero.feetCollisionRect.size.height), MAX(_hero.centerToBottom,  
            _hero.desiredPosition.y));  
  
            _hero.groundPosition = ccp(posX, posY);  
            _hero.position = ccp(_hero.groundPosition.x,  
            _hero.groundPosition.y + _hero.jumpHeight);  
        }  
        // 2  
        else if (_eventState == kEventStateBattle)  
        {
```

```
        posX = MIN(MIN(mapWidth -
    _hero.feetCollisionRect.size.width/2, _eventCenter + CENTER.x -
    _hero.feetCollisionRect.size.width/2), MAX(_eventCenter - CENTER.x +
    _hero.feetCollisionRect.size.width/2, _hero.desiredPosition.x));
        posY = MIN(floorHeight + (_hero.centerToBottom -
    _hero.feetCollisionRect.size.height), MAX(_hero.centerToBottom,
    _hero.desiredPosition.y));
        _hero.groundPosition = ccp(posX, posY);
        _hero.position = ccp(_hero.groundPosition.x,
    _hero.groundPosition.y + _hero.jumpHeight);
    }
}

Robot *robot;
CCARRAY_FOREACH(_robots, robot)
{
    if (robot.actionState > kActionStateNone)
    {
        posY = MIN(floorHeight + (robot.centerToBottom -
    robot.feetCollisionRect.size.height), MAX(robot.centerToBottom,
    robot.desiredPosition.y));
        robot.groundPosition = ccp(robot.desiredPosition.x,
posY);
        robot.position = ccp(robot.groundPosition.x,
    robot.groundPosition.y + robot.jumpHeight);

        if (robot.actionState == kActionStateDead &&
    _hero.groundPosition.x - robot.groundPosition.x >= CENTER.x +
    robot.contentSize.width/2 * kScaleFactor)
        {
            robot.visible = NO;
            [robot reset];
        }
    }
}
```

You restrict the call to `setViewpointCenter:` to happen only on free-roam and scripted events. This way, when a battle starts, the camera won't follow the hero.

Next, you make a couple of changes to how the position of the hero is updated.

1. When the `eventState` is `kEventStateFreeWalk`, the hero's position is only clamped from the edges of the map, like before.
2. When the `eventState` is `kEventStateBattle`, then there's an additional clamping done based on the edges of the screen from the `eventCenter`.

Build and run, and you will find that the hero can't run from robots anymore. Your challenge now is to defeat the first batch of robots, walking away alive only when the robots are dead!

**Note:** Currently there is a bug affecting the hero's start position. He starts at (100, 100), but suddenly teleports to the lower-left of the screen half a second after the game loads. You'll squash this bug in the "Scripted Events" section, below.



Have you defeated the robots yet? Yes? Could you walk away after your victory? Congratulations, you are now stuck on that event! 😊

If you did manage to defeat the robot horde, then you probably found yourself unable to leave the first battle event afterwards. If you take a look back at `updateEvent`, you'll see that the condition for leaving an event is when `activeEnemies` become zero.

```
-(void)updateEvent
{
    if (_eventState == kEventStateBattle && _activeEnemies <= 0)
```

You set `activeEnemies` at the start of the battle event in `cycleEvents`, but you don't update it after that. Every time an enemy is defeated, this number should decrease.

Thanks to `ActionSprite` being delegated to `GameLayer`, you're already able to keep track of when an `ActionSprite` dies via `actionSpriteDidDie:`.

Replace the stub for `actionSpriteDidDie:` in `GameLayer.m` with the following:

```
-(BOOL)actionSpriteDidDie:(ActionSprite *)actionSprite
```

```

{
    if (actionSprite == _hero)
    {

    }
    else
    {
        _activeEnemies--;
        return YES;
    }

    return NO;
}

```

When an **ActionSprite** other than the hero dies, then the number of active enemies goes down as well.

Try it one more time. Build, run, and defeat the first wave of robots.



Does it work much better? Sort of – if you are at the center of the screen, it works fine. But if you finish a battle while the hero is close to either edge of the screen, the camera suddenly snaps back to the hero.

This simply looks ugly. Instead of snapping back, the camera should smoothly move its focus back to the hero.

This is where **viewPointOffset** comes into play. When the event switches from battle to free roam, you store the distance between the cameras of the two events. You can use this value to achieve the smooth camera effect.

Still in **GameLayer.m**, do the following:

```
//in setViewpointCenter:, change the line self.position =
viewPoint to:
self.position = ccp(viewPoint.x + _viewPointOffset, viewPoint.y);

//add this to the end of update:
if (_viewPointOffset < 0)
{
    _viewPointOffset += SCREEN.width * delta;

    if (_viewPointOffset >= 0)
    {
        _viewPointOffset = 0;
    }
}
else if (_viewPointOffset > 0)
{
    _viewPointOffset -= SCREEN.width * delta;

    if (_viewPointOffset <= 0)
    {
        _viewPointOffset = 0;
    }
}
```

**setViewPointCenter:** temporarily adjusts its focus using **viewPointOffset**. This means that when a battle ends, **setViewPointCenter:** will still focus on the **eventCenter** position. In **update:**, you gradually reduce the value of **viewPointOffset** back to zero. The rate of decrement for the **viewPointOffset** is equivalent to **SCREEN.width** per second.

The smaller **viewPointOffset** gets, the closer the camera gets to the intended position.

Build and run again. The camera should now pan smoothly back to the hero after the battle event.



There's one minor annoyance left, and it happens on the second battle event or so. Right before new robots spawn, there's a possibility that you will see a glimpse of a dead robot on the side of the screen. This is because dead robots are being reused to spawn the new robots, and if you position a dead robot at the edge of the screen, their body extends toward the visible part of the screen.

The fix for this is easy – simply reset the robot's sprites before reusing them.

Go to **Robot.m** and add this method:

```
-(void)reset
{
    [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"robot_base_idle_00.png"]];
    [_belt setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"robot_belt_idle_00.png"]];
    [_smoke setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"robot_smoke_idle_00.png"]];
    [super reset];
}
```

This just changes all the robot sprites to their idle frames when a robot is reset.

This overrides the parent class's reset method, which you're already calling in `updatePositions` when a robot is dead (and offscreen).

Test it out. There should be no more dead robot mirages!



## Scripted events

In most Beat 'Em Up Games, the hero doesn't get dropped into the scene as yours does. Instead, the player watches the hero march in through a door or simply amble in from the left side of the screen. It's all about the dramatic entrance!

This is called a scripted event. For a brief period of time, the game takes control away from the player and leaves them in the position of mere observer, whether what unfolds is for good or ill.

You can make anything happen in a scripted event – you could kill or resurrect the player, or have them destroy every robot on screen. You could even show off a new dance animation – aww, yeah!

In this section, you'll start simple and make the hero strut his stuff from one point to another without any player intervention.

First go to **ActionSprite.m** and add this method:

```
- (void)enterFrom:(CGPoint)origin to:(CGPoint)destination
{
    float diffX = fabsf(destination.x - origin.x);
    float diffY = fabsf(destination.y - origin.y);

    if (diffX > 0)
    {
        self.directionX = 1.0;
    }
    else
    {
        self.directionX = -1.0;
    }

    self.scaleX = _directionX * kScaleFactor;
```

```

    ccTime duration = MAX(diffX, diffY) / _walkSpeed;

    _actionState = kActionStateAutomated;
    [self stopAllActions];
    [self runAction:_walkAction];
    [self runAction:[CCSequence actions:[CCMoveTo
actionWithDuration:duration position:destination], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil]];
}

```

The code above will make an **ActionSprite** walk from an origin point to a destination point. First, the method gets the x and y distance from the origin to the destination, then it calculates the time required to walk there by taking the higher of the two values and dividing it by the sprite's **walkSpeed**.

The code then changes the state to **kActionStateAutomated** and executes an automated movement action using both **walkAction** and **CCMoveTo**. After moving the sprite, it automatically changes the sprite's action state to idle.

If the destination is to the right of the origin, the sprite will face right. Otherwise, it will face left.

Still in **ActionSprite.m**, make the following change to **update:**:

```

//add this else if condition
else if (_actionState == kActionStateAutomated)
{
    self.groundPosition = _position;
    _desiredPosition = _groundPosition;
}

```

**CCMoveTo** can only modify the **position** attribute of a **ccSprite**. Since **ActionSprite** is moved using **groundPosition** and **desiredPosition**, you just plug in the value of **position** into them whenever **actionState** is **kActionStateAutomated**.

Switch to **GameLayer.m** and do the following:

```

//replace onEnterTransitionDidFinish
-(void)onEnterTransitionDidFinish
{
    [super onEnterTransitionDidFinish];
    _eventState = kEventStateScripted;
    [_hero enterFrom:_hero.position to:ccp(64.0,
_hero.position.y)];
    [self performSelector:@selector(triggerEvent:)
withObject:[NSNumber numberWithInt:kEventStateFreeWalk]
afterDelay:1.2];
}

```

```

}

//add this method
-(void)triggerEvent:(NSNumber *)eventId
{
    self.eventState = [eventId intValue];
}

```

Instead of starting out with `kEventStateFreeWalk`, you now start with `kEventStateScripted`. Then you make the hero walk to a certain position. `performSelector:` will execute `triggerEvent:` and plug in `kEventStateFreeWalk` so that the event changes to a free roam event after 1.2 seconds.

Still in **GameLayer.m**, make the following changes to `initHero` (or replace the method entirely):

```

-(void)initHero
{
    self.hero = [Hero node];
    _hero.delegate = self;
    [_actors addChild:_hero.shadow];
    _hero.shadow.scale *= kScaleFactor;
    [_actors addChild:_hero];
    _hero.scale *= kScaleFactor;
    //change _hero.position = ccp(100 * kPointFactor, 100 *
kPointFactor) to
    _hero.position = ccp(-_hero.centerToSides, 80 * kPointFactor);
    //add the following two lines
    _hero.desiredPosition = _hero.position;
    _hero.groundPosition = _hero.position;
    //remove [_hero idle];
}

```

This puts the hero just beyond the left edge of the screen. The hero also no longer executes his idle action, since you want him to enter the scene first.

The last thing to do is to remove the player's control over the hero during scripted events.

Still in **GameLayer.m**, make the following changes to the indicated methods:

```

-(void)actionDPad:(ActionDPad *)actionDPad
didChangeDirectionTo:(ActionDPadDirection)direction
{
    //enclose everything in this condition
    if (_eventState != kEventStateScripted)
    {

```

```
        CGPoint directionVector = [self
vectorForDirection:direction];

        if (_runDelay > 0 && _previousDirection == direction &&
(direction == kActionDPadDirectionRight || direction ==
kActionDPadDirectionLeft))
    {
        [_hero runWithDirection:directionVector];
    }
    else if (_hero.actionState == kActionStateRun &&
abs(_previousDirection - direction) <= 1)
    {
        [_hero moveWithDirection:directionVector];
    }
    else
    {
        [_hero walkWithDirection:directionVector];
        _previousDirection = direction;
        _runDelay = 0.2;
    }
}

-(void)actionDPadTouchEnded:(ActionButton *)actionDPad
{
    //modify the if condition as shown
    if (_eventState != kEventStateScripted && (_hero.actionState
== kActionStateWalk || _hero.actionState == kActionStateRun))
    {
        [_hero idle];
    }
}

-(void)actionButtonWasPressed:(ActionButton *)actionButton
{
    //enclose everything in this condition
    if (_eventState != kEventStateScripted)
    {
        if (actionButton.tag == kTagButtonA)
        {
            [_hero attack];
        }
        else if (actionButton.tag == kTagButtonB)
        {
```

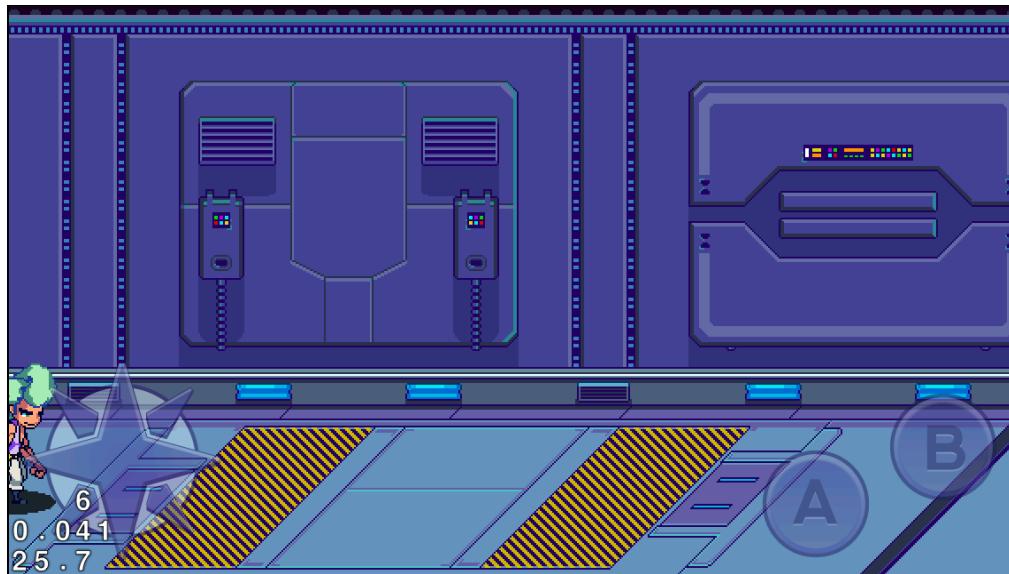
```
        CGPoint directionVector = [self
vectorForDirection:_hud.dPad.direction];
[_hero jumpRiseWithDirection:directionVector];
}
}

-(void)actionButtonIsHeld:(ActionButton *)actionButton{
}

-(void)actionButtonWasReleased:(ActionButton *)actionButton
{
    //enclose everything in this condition
    if (_eventState != kEventStateScripted)
    {
        if (actionButton.tag == kTagButtonB)
        {
            [_hero jumpCutoff];
        }
    }
}
```

Now all control inputs (D-pad and A and B buttons) will only work when the `eventState` is not `kEventStateScripted`.

Build and run, and watch in awe as the hero walks into the scene like an action star!



At this point, your game is starting to get pretty cool – you now have brainy robots, scripted events, and a flashy entrance!

This is a good spot to take a break – but stay tuned for the next chapter, where you'll add more cool attack types for your hero like a 1-2-3 punch and a running kick, as well as multiple level support!

**Challenge:** Modify the AI to have a new state “running” where the robot tries to move away from the hero if his health is low. This will help you get more practice setting up the AI behavior, and make the game more interesting too!



# 6

## Chapter 6: Power Attacks

In this chapter, you’re going to power-up your hero with some additional attacks. You’ll add a 1-2-3 punch, a jump punch, and a running kick in particular.

But with great power comes great responsibility – for punches from the robots in particular. You’ll modify the game so that if the robots punch the player enough times in a row, the player will fall to the ground.

Finally, you’ll add multiple level support into the game, and do some investigations into memory issues in the game.

Get ready for some powerful attacks!

## Complex actions

Right now, your hero can do five basic things:

1. Idle / Dance
2. Walk
3. Attack / Jab
4. Jump
5. Run

That’s not a bad variety of actions, but the hero has only one option for attacking – the attack/jab. This severely limits his ability to overcome his foes.

Fortunately, it’s an arbitrary limit. You can easily enable your hero to perform complex combination actions, such as attacking while running or jumping.

Start by adding the actions to `ActionSprite`. Go to `ActionSprite.m` and add these methods:

```
-(void)jumpAttack
```

```
{  
    if (_actionState == kActionStateJumpRise || _actionState ==  
kActionStateJumpFall)  
    {  
        _velocity = CGPointMakeZero;  
        [self stopAllActions];  
        self.actionState = kActionStateJumpAttack;  
        [self runAction:_jumpAttackAction];  
    }  
}  
  
-(void)runAttack  
{  
    if (_actionState == kActionStateRun)  
    {  
        [self stopAllActions];  
        self.actionState = kActionStateRunAttack;  
        [self runAction:_runAttackAction];  
    }  
}
```

`jumpAttack` and `runAttack` simply change the state to `kActionStateJumpAttack` and `kActionStateRunAttack`, respectively, and execute the corresponding action. In addition, `jumpAttack` zeroes out velocity so that the sprite stops in mid-air to do the attack.

Next, still in **ActionSprite.m**, do the following:

```
//add these conditions to the if statement in attack  
else if (_actionState == kActionStateJumpRise || _actionState ==  
kActionStateJumpFall)  
{  
    [self jumpAttack];  
}  
else if (_actionState == kActionStateRun)  
{  
    [self runAttack];  
}  
  
//change the first if statement in update: to this  
if (_actionState == kActionStateWalk || _actionState ==  
kActionStateRun || _actionState == kActionStateRunAttack)  
{  
    _desiredPosition = ccpAdd(_groundPosition, ccpMult(_velocity,  
delta));  
}
```

You trigger both `jumpAttack` and `runAttack` from `attack`, depending on the current `actionState` of the `ActionSprite`.

Then, to have the sprite continue moving forward when performing the run attack action, you include the `kActionStateRunAttack` state in the condition to move the `desiredPosition` of the sprite.

You might be able to guess what's next on the plate – creating animation actions for `jumpAttack` and `runAttack`!

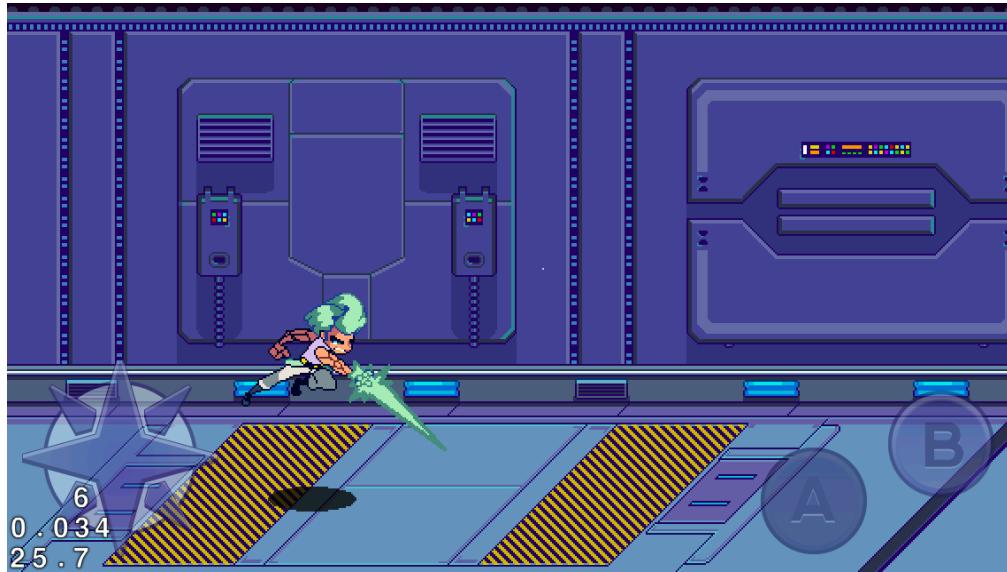
Go to `Hero.m` and do the following:

```
//add these actions to init, right after the other actions
CCAnimation *jumpAttackAnimation = [self
animationWithPrefix:@"hero_jumpattack" startFrameIdx:0
frameCount:5 delay:1.0/10.0];
self.jumpAttackAction = [CCSequence actions:[CCAnimate
actionWithAnimation:jumpAttackAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(jumpFall)], nil];

CCAnimation *runAttackAnimation = [self
animationWithPrefix:@"hero_runattack" startFrameIdx:0 frameCount:6
delay:1.0/10.0];
self.runAttackAction = [CCSequence actions:[CCAnimate
actionWithAnimation:runAttackAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];

//add these attributes to init along with the other attributes
self.jumpAttackDamage = 15.0;
self.runAttackDamage = 15.0;
```

Build, run, and try these new actions out, but for now don't use them to fight robots. See if you notice some strange behavior with the jump attack.



Both attacks work great, except that the hero can repeat the jump attack an infinite number of times, allowing him to stay suspended in the air. Give that a try while you still can if you want to have some fun, as you're about to eliminate that behavior!

To make sure that the jump attack can only be performed once per jump, add this variable to **ActionSprite.h**:

```
//add inside the curly braces of @interface
BOOL _didJumpAttack;
```

Then switch to **ActionSprite.m** and modify the following methods as per the comments:

```
- (void)jumpLand
{
    if (_actionState == kActionStateJumpFall || _actionState ==
kActionStateRecover)
    {
        _jumpHeight = 0;
        _jumpVelocity = 0;
        //add this line
        _didJumpAttack = NO;
        self.actionState = kActionStateJumpLand;
        [self runAction:_jumpLandAction];
    }
}

-(void)jumpAttack
{
```

```
//add this condition
if (!_didJumpAttack && (_actionState == kActionStateJumpRise
|| _actionState == kActionStateJumpFall))
{
    _velocity = CGPointMakeZero;
    [self stopAllActions];
    self.actionState = kActionStateJumpAttack;
    //add this line
    _didJumpAttack = YES;
    [self runAction:_jumpAttackAction];
}
}
```

An **ActionSprite** can only execute **jumpAttack** when **didJumpAttack** is set to **no**. Once the sprite performs the action, **didJumpAttack** becomes **yes** until the sprite lands on the ground again, thus ensuring that only one jump attack can be performed per jump.

Build and run again, and this time try out your newfound abilities on the robots.



Unfortunately, as you just found out, your new attacks don't work on the robots. That's because there's no collision handling code for them.

You may remember that in the previous chapter, you adjusted the attack and contact circles for the hero for both the run attack and jump attack actions. So why don't they work on the robots?

It's because triggering these actions doesn't inform **GameLayer** to check for collisions. Open **Hero.m** and replace **setDisplayFrame:** with the following:

```
-(void)setDisplayFrame:(CCSpriteFrame *)newFrame
```

```
{  
    [super setDisplayFrame:newFrame];  
  
    CCSpriteFrame *attackFrame = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_attack_00_01.png"];  
  
    //add these new frames  
    CCSpriteFrame *runAttackFrame = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_runattack_02.png"];  
    CCSpriteFrame *runAttackFrame2 = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_runattack_03.png"];  
    CCSpriteFrame *jumpAttackFrame = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_jumpattack_02.png"];  
  
    //include them in conditions  
    if (newFrame == attackFrame || newFrame == runAttackFrame ||  
newFrame == runAttackFrame2 || newFrame == jumpAttackFrame)  
    {  
        [self.delegate actionSpriteDidAttack:self];  
    }  
}
```

You reference sprite frames that show the hero in his attack pose while running and jumping, and tell the delegate that the hero is attacking when these frames are displayed.

Switch to **GameLayer.m** and modify the marked section of **actionSpriteDidAttack:** as follows:

```
//replace the code inside the curly braces of if ([self  
collisionBetweenAttacker:_hero andTarget:robot  
atPosition:&attackPosition])  
if (_hero.actionState == kActionStateJumpAttack)  
{  
    [robot knockoutWithDamage:_hero.jumpAttackDamage  
direction:ccp(_hero.directionX, 0)];  
}  
else if (_hero.actionState == kActionStateRunAttack)  
{  
    [robot knockoutWithDamage:_hero.runAttackDamage  
direction:ccp(_hero.directionX, 0)];  
}
```

```
else
{
    [robot hurtWithDamage:_hero.attackDamage
    force:_hero.attackForce direction:ccp(_hero.directionX, 0.0)];
}
didHit = YES;
```

When the hero's attack collides with a robot, you check the current state of the hero and do the appropriate collision response for each. For both attacks, the robot gets knocked out and receives damage corresponding to the attack type.

Build, run, and send robots flying!



I'm not sure if you've noticed it, but there's one bug introduced here. Now that you have the ability to knock out robots without killing them, you'll see their recover action for the first time.



After a robot gets knocked out once, they may have a different reaction to getting punched. Specifically, they could get forced back diagonally, instead of horizontally.

This is because a robot's `jumpHeight` is altered when it gets knocked out, but the `jumpHeight` is never reset when it recovers.

Go to **ActionSprite.m** and replace `recover` with the following:

```
- (void)recover
{
    if (_actionState == kActionStateKnockedOut)
    {
        self.actionState = kActionStateNone;
        _velocity = CGPointMakeZero;
        _jumpVelocity = 0;
        _jumpHeight = 0; //add this
        [self performSelector:@selector(getUp) withObject:nil
afterDelay:0.5];
    }
}
```

Now when the sprite gets up, you make sure that `jumpHeight` is set back to zero. Don't let them forget who's in charge. ☺

## I get knocked down!

The game is pretty exciting at this point, but the player might sometimes hit a scenario where they have to fight off an overwhelming crowd of robots and get pummeled to death without being able to respond. It's the stuff of which nightmares are made!



One of the reasons the hero is so helpless in that situation is because when he gets hit again and again, there is no chance of escape. While the hero is showing the hurt animation, he isn't able to move or attack, but he can continue to get hit.

To increase the hero's chances of surviving, you will have to make him weaker. Quite the contradiction, huh?



The idea is to reduce the amount of beatings the hero can take before falling down. Say when he gets punched ten times in a row, he falls to the floor. That will give him some time to recover!

Open **Hero.h** and add these variables:

```
//add inside curly braces of @interface
float _hurtTolerance;
float _recoveryRate;
float _hurtLimit;
```

These are:

- **hurtTolerance**: This is, in effect, the player's secondary hit points variable. Whenever the player is hurt, the value of `hurtTolerance` also goes down. When it reaches zero, the hero gets knocked out without dying.
- **recoveryRate**: The rate at which the hero recovers his `hurtTolerance`.
- **hurtLimit**: The maximum value of `hurtTolerance`.

Switch to **Hero.m** and do the following:

```
//add these attributes in init along with the other attributes
_recoveryRate = 5.0;
_hurtLimit = 20.0;
_hurtTolerance = _hurtLimit;

//add these methods
-(void)hurtWithDamage:(float)damage force:(float)force
direction:(CGPoint)direction
{
    [super hurtWithDamage:damage force:force direction:direction];

    if (self.actionState == kActionStateHurt)
    {
        _hurtTolerance -= damage;
        if (_hurtTolerance <= 0)
        {
            [self knockoutWithDamage:0 direction:direction];
        }
    }
}

-(void)knockoutWithDamage:(float)damage
direction:(CGPoint)direction
{
    [super knockoutWithDamage:damage direction:direction];

    if (self.actionState == kActionStateKnockedOut)
    {
        _hurtTolerance = _hurtLimit;
    }
}

-(void)update:(ccTime)delta
{
    [super update:delta];

    if (_hurtTolerance < _hurtLimit)
```

```
{  
    _hurtTolerance += _hurtLimit * delta / _recoveryRate;  
  
    if (_hurtTolerance >= _hurtLimit)  
    {  
        _hurtTolerance = _hurtLimit;  
    }  
}  
}
```

`hurtTolerance` starts at 20 points. Whenever the hero gets hurt via `hurtWithDamage:`, this number will go down. The moment `hurtTolerance` drops to or below zero, the hero gets knocked out.

In `knockOutWithDamage:`, you reset `hurtTolerance` back to `hurtLimit`. Finally, at every game loop, as long as `hurtTolerance` is less than `hurtLimit`, it will get refilled with the amount of `recoveryRate` every second.

In this case, `hurtTolerance` will increase five points per second until it reaches the limit. This way, only multiple consecutive punches in a row can knock the hero out.

All of these methods are inherited from `ActionSprite`. Since you just want to add more logic to what `ActionSprite` already does, you first call the method on the `super` class so that the parent class (`ActionSprite`) will execute its version of the method.

Build, run, and let the robots beat you up until you get knocked out. This time the hero should rise to fight again.



# 1-2-3 punch

You have two hands – the left and the right. Hold them up high so clean and bright!  
Punch them softly, 1-2-3. Strong little hands are good to see!

–Modified version of a classic Children's song

Up until now, you've been jabbing away at robots to defeat them. You need to punch a robot 20 times just to shut it down. My thumb hurts just thinking about it!

To add some variety and strength to the hero's attacks, you're going to make him do a timed three-punch combo – a chain of punches!

To do this, you're going to make a couple of changes to how the attack action is managed. Whenever an attack connects with an enemy, the hero will be given a window of opportunity to chain the attack. This window of opportunity will be very short. The chained attack can't happen too soon or too long after the previous attack.

Open **ActionSprite.h** and add these instance variables:

```
//add inside curly braces of @interface
float _actionDelay;
float _attackDelayTime;
```

Switch to **ActionSprite.m** and make changes to the existing code:

```
//add inside the if statement of idle
_actionDelay = 0.0;

//add to the end of update:, before the last curly brace
if (_actionDelay > 0)
{
    _actionDelay -= delta;

    if (_actionDelay <= 0)
    {
        _actionDelay = 0;
    }
}

//modify attack to look like this
-(void)attack
{
    if (_actionState == kActionStateIdle || _actionState ==
kActionStateWalk || (_actionState == kActionStateAttack &&
_actionDelay <= 0)) //added actionDelay as a condition
```

```
{  
    [self stopAllActions];  
    [self runAction:_attackAction];  
    self.actionState = kActionStateAttack;  
    _actionDelay = _attackDelayTime; //set actionDelay to  
the value of attackDelayTime  
}  
else if (_actionState == kActionStateJumpRise || _actionState  
== kActionStateJumpFall)  
{  
    [self jumpAttack];  
}  
else if (_actionState == kActionStateRun)  
{  
    [self runAttack];  
}  
}
```

`actionDelay` is the delay time between attack actions while `attackDelayTime` is the delay time for the first attack (the jab).

Once an `ActionSprite` performs an attack, the value of `actionDelay` changes to the value of `attackDelayTime`.

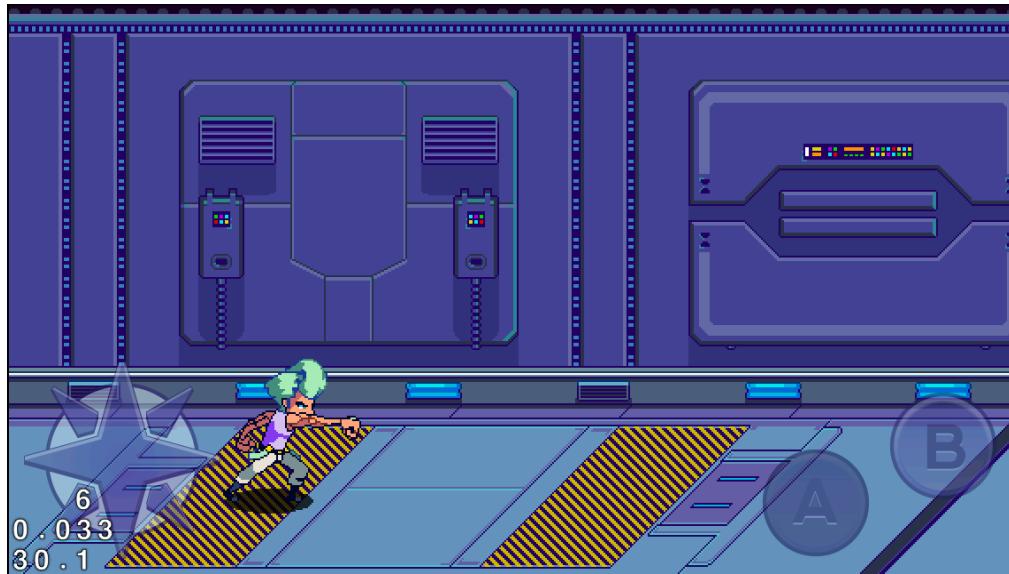
`actionDelay` constantly counts down to zero in `update::`. When it reaches zero, you allow the `ActionSprite` to do another attack.

Now go to `Hero.m` and add this inside `init`:

```
//add this attribute in init along with the other attributes  
_attackDelayTime = 0.14;
```

You set the delay time of the hero's jab attack to 0.14 seconds. This means the hero can only punch once every 0.14 seconds.

Build, run, and try punching as fast as you can. Notice that you can't punch as fast as you could before.



This delay is a serious handicap for the hero, but you're about to offset it by granting him the combo attack!

Open **Hero.h** and add the following:

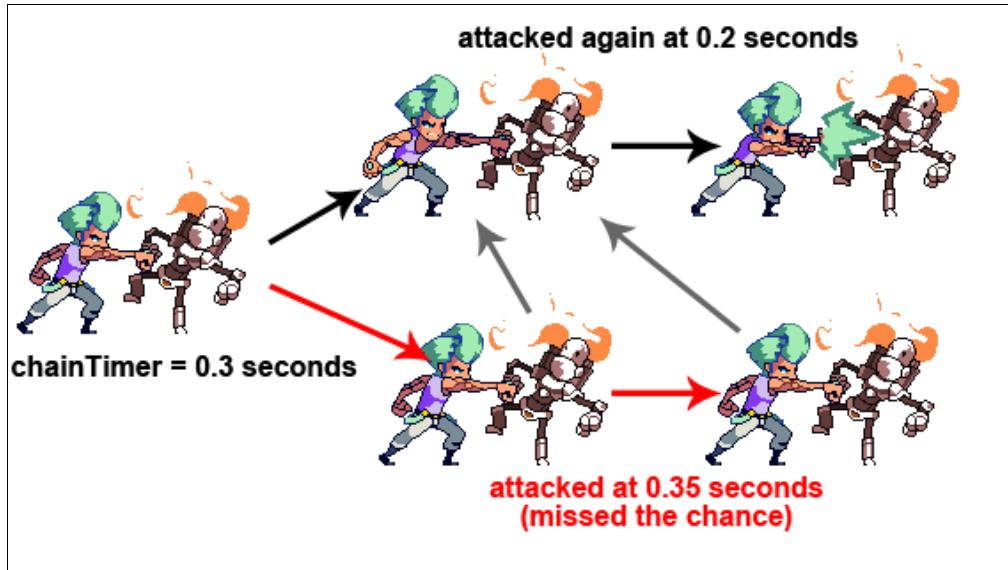
```
//add inside curly braces of @interface
float _attackTwoDelayTime;
float _attackThreeDelayTime;
float _chainTimer;

//add these properties
@property(nonatomic, strong)id attackTwoAction;
@property(nonatomic, strong)id attackThreeAction;
@property(nonatomic, assign)float attackTwoDamage;
@property(nonatomic, assign)float attackThreeDamage;
```

**attackTwoDelayTime** and **attackThreeDelayTime** are the delay times for the second and third attacks.

You'll use **chainTimer** as the time window in which the hero will be able to chain his attack.

The logic will work like this:



If the first attack connects with an enemy, there will be a 0.3-second time window in which to chain the attack. If the second attack is triggered before 0.3 seconds has elapsed, then the hero will perform the next attack in the chain.

If the second attack occurs after 0.3 seconds, however, then the hero will merely do the first attack in the chain again. Similar logic applies to the third attack – if it's done within the 0.3 second limit, the final, more powerful punch will trigger.

You will be making a couple of changes to **Hero.m**. First, do the following:

```
//add these actions in init along with the other actions
CCAnimation *attackTwoAnimation = [self
animationWithPrefix:@"hero_attack_01" startFrameIdx:0 frameCount:3
delay:1.0/12.0];
self.attackTwoAction = [CCSequence actions:[CCAnimate
actionWithAnimation:attackTwoAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];

CCAnimation *attackThreeAnimation = [self
animationWithPrefix:@"hero_attack_02" startFrameIdx:0 frameCount:5
delay:1.0/10.0];
self.attackThreeAction = [CCSequence actions:[CCAnimate
actionWithAnimation:attackThreeAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];

//add these attributes in init along with the other attributes
_attackTwoDelayTime = 0.14;
_attackThreeDelayTime = 0.45;
_chainTimer = 0;
self.attackTwoDamage = 10.0;
```

```
self.attackThreeDamage = 20.0;

//add to the beginning of update:, after [super update:delta]
if (_chainTimer > 0)
{
    _chainTimer -= delta;

    if (_chainTimer <= 0)
    {
        _chainTimer = 0;
    }
}
```

The above sets up the attack chain. The second attack has a delay of 0.14 seconds, while the third attack has a delay of 0.45 seconds. The third attack is the final attack, with nothing in the chain to follow it, so having a long delay after it is just right.

In `update:`, you make sure that `chainTimer` always counts down to zero.

Still in `Hero.m`, add this method:

```
-(void)attack
{
    if (self.actionState == kActionStateAttack && _chainTimer > 0)
    {
        _chainTimer = 0;
        [self stopAllActions];
        [self runAction:_attackTwoAction];
        self.actionState = kActionStateAttackTwo;
        [self setContactPointsForAction:self.actionState];
        _actionDelay = _attackTwoDelayTime;
    }
    else if (self.actionState == kActionStateAttackTwo &&
    _chainTimer > 0)
    {
        _chainTimer = 0;
        [self stopAllActions];
        [self runAction:_attackThreeAction];
        self.actionState = kActionStateAttackThree;
        [self setContactPointsForAction:self.actionState];
        _actionDelay = _attackThreeDelayTime;
    }
    else
    {
        [super attack];
```

```
    }  
}
```

This overrides `ActionSprite`'s `attack` method. When the hero has attacked a first time and wants to attack again, the method checks if there is still time left in the `chainTimer`.

If the conditions are passed, then the method executes the next attack in the sequence. Otherwise, the method executes the first attack again by calling `[super attack]`.

Still in `Hero.m`, replace `setDisplayFrame:` with the following (or modify it according to the comments):

```
-(void)setDisplayFrame:(CCSpriteFrame *)newFrame  
{  
    [super setDisplayFrame:newFrame];  
  
    CCSpriteFrame *attackFrame = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_attack_00_01.png"];  
  
    CCSpriteFrame *runAttackFrame = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_runattack_02.png"];  
    CCSpriteFrame *runAttackFrame2 = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_runattack_03.png"];  
    CCSpriteFrame *jumpAttackFrame = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_jumpattack_02.png"];  
  
    //add these new frames  
    CCSpriteFrame *attackFrame2 = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_attack_01_01.png"];  
    CCSpriteFrame *attackFrame3 = [[CCSpriteFrameCache  
sharedSpriteFrameCache]  
spriteFrameByName:@"hero_attack_02_02.png"];  
  
    //change the conditions  
    if (newFrame == attackFrame || newFrame == attackFrame2)  
    {  
        if ([self.delegate actionSpriteDidAttack:self])  
        {  
            _chainTimer = 0.3;  
        }  
    }  
}
```

```
        }
    }
    else if (newFrame == attackFrame3)
    {
        [self.delegate actionSpriteDidAttack:self];
    }
    else if (newFrame == runAttackFrame || newFrame ==
runAttackFrame2 || newFrame == jumpAttackFrame)
    {
        [self.delegate actionSpriteDidAttack:self];
    }
}
```

You add the sprite frames for the second and third attacks to the list of frames that will trigger the delegate notification that the sprite has attacked.

Next, you rewrite the `if-else` statement this way:

- For the first and second attacks, you notify the delegate that the sprite has attacked and you get the result of the attack from the delegate.  
`actionSpriteDidAttack:` returns `YES` if the attack connected with an enemy, and `NO` if it didn't. If the attack did connect, you set `chainTimer` to 0.3 seconds.
- Since the third attack, run attack and jump attack cannot be chained to other attacks, you simply notify the delegate to check for collisions without the need to report back the result.

Build, run, and try out your two new actions.



What's left to do? If you tried to attack a robot, then you certainly noticed that the collision resolution code behaves the same before, as if the hero were still performing a simple jab. You want this combo to knock a robot to the ground!

Remedy this quickly by going to **GameLayer.m** and modifying the following section of **actionSpriteDidAttack::**

```
//replace the code inside the curly braces of if ([self
collisionBetweenAttacker:_hero andTarget:robot
atPosition:&attackPosition])
if (_hero.actionState == kActionStateJumpAttack)
{
    [robot knockoutWithDamage:_hero.jumpAttackDamage
direction:ccp(_hero.directionX, 0)];
}
else if (_hero.actionState == kActionStateRunAttack)
{
    [robot knockoutWithDamage:_hero.runAttackDamage
direction:ccp(_hero.directionX, 0)];
}
else if (_hero.actionState == kActionStateAttackThree) //add this
{
    [robot knockoutWithDamage:_hero.attackThreeDamage
direction:ccp(_hero.directionX, 0)];
}
else if (_hero.actionState == kActionStateAttackTwo) //add this
{
    [robot hurtWithDamage:_hero.attackTwoDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0.0)];
}
else
{
    [robot hurtWithDamage:_hero.attackDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0.0)];
}
didHit = YES;
```

You add **kActionStateAttackTwo** and **kActionStateAttackThree** to the **if-else** statement. If the hero performs the second attack, the robot gets hit with **attackTwoDamage**. Then if the hero performs the third attack, the robot gets knocked out with **attackThreeDamage**.

Build, run, and unleash the 1-2-3 punch on those droids!



## Adding multi-level support

Your game is now fairly complex with several different attack modes, but you're still playing the same level as when it was a much simpler game!

You could use a change of scenery. Plus, your hero needs somewhere to go once he's beaten all the robots in the current level, right? You're about to add support for multiple levels!

Make sure that your `Levels.plist` file has definitions for at least two levels.

Optionally, you can copy `Levels.plist` from **Versions\Chapter5** or **Versions\Chapter6** of the Starter Kit resources (if you haven't done so already), since it already has three levels defined.

Using `ccscenes` in Cocos2D has at least one thing going for it – it makes it easier to reset everything. If you want to reset the game, all you have to do is replace the current `GameScene` with another instance of `GameScene`.

You can also apply this simple logic to support multiple levels. When each level is finished, just replace the current scene with a fresh copy of `GameScene` and make this new copy load a different level.

Currently, to load a level, you have this in the code:

```
[self loadLevel:0];
```

When `GameScene` creates a new `GameLayer`, it will always load the first level. You need to modify this behavior to be more dynamic.

Open `GameLayer.h` and do the following:

```
//add these properties
@property(nonatomic, assign)int totalLevels;
@property(nonatomic, assign)int currentLevel;

//add these methods
+(id)nodeWithLevel:(int)level;
-(id)initWithLevel:(int)level;
```

Now switch to **GameLayer.m** and do the following:

```
//add these methods
+(id)nodeWithLevel:(int)level
{
    return [[self alloc] initWithLevel:level];
}

-(id)initWithLevel:(int)level
{
    if ((self = [super init]))
    {
        [[CCSpriteFrameCache sharedSpriteFrameCache]
addSpriteFramesWithFile:@"sprites.plist"];
        _actors = [CCSpriteBatchNode
batchNodeWithFile:@"sprites.pvr.ccz"];
        [_actors.texture setAliasTexParameters];
        [self addChild:_actors z:-5];
        [self loadLevel:level];
        [self initHero];
        [self initRobots];
        [self initBrains];
        [self scheduleUpdate];
    }
    return self;
}

//add these to the end of loadLevel:
_totalLevels = levelArray.count;
_currentLevel = level;
```

The above allows you to specify the level you want via the new initializers instead of simply calling `node` or `init`. The rest of the code is the same as before, except that `loadLevel:` can now load any level you want.

In `loadLevel:`, you have two new variables:

- **`totalLevels`**: The number of levels defined in the property list.

- **currentLevel**: The currently-loaded level.

Now go to **GameScene.h** and add these method prototypes:

```
//add these methods
+(id)nodeWithLevel:(int)level;
-(id)initWithLevel:(int)level;
```

Switch to **GameScene.m** and add these methods:

```
+ (id)nodeWithLevel:(int)level
{
    return [[self alloc] initWithLevel:level];
}

-(id)initWithLevel:(int)level
{
    if ((self = [super init]))
    {
        GameLayer *gameLayer = [GameLayer nodeWithLevel:level];
        [self addChild:gameLayer z:0];

        HudLayer *hudLayer = [HudLayer node];
        [self addChild:hudLayer z:1];

        hudLayer.dPad.delegate = gameLayer;
        gameLayer.hud = hudLayer;

        hudLayer.buttonA.delegate = gameLayer;
        hudLayer.buttonB.delegate = gameLayer;
    }
    return self;
}
```

Similar to **GameLayer**, you now add the level number when creating a new **GameScene**. **GameScene** will just use the level number and pass it on to **GameLayer**.

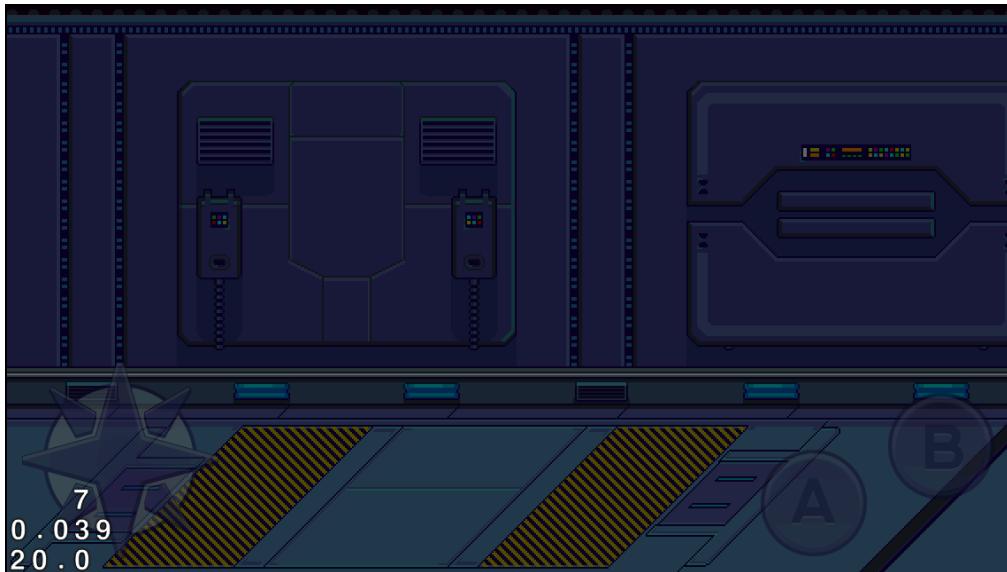
To test if your new initializers work, go to **TitleLayer.m** and change **ccTouchesBegan:withEvent:** as follows:

```
- (void)ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    //add level number 0 to GameScene creation
    [[CCDirector sharedDirector] replaceScene:[CCTransitionFade
transitionWithDuration:1.0 scene:[GameScene nodeWithLevel:0]]];
    CCSprite *start = (CCSprite *)[self getChildByTag:1];
    [start stopAllActions];
```

```
    start.visible = NO;  
}
```

This makes `GameScene` load the first level when the screen is touched in the title scene.

Build and run, and check if the first level still loads properly.



All right, it still works!

Now you just need to increment the level number and reload `GameScene` when you want to go to the next level.

Go to `GameLayer.m` and do the following:

```
//add to the top of the file  
#import "GameScene.h"  
  
//replace updateEvent  
-(void)updateEvent  
{  
    if (_eventState == kEventStateBattle && _activeEnemies <= 0)  
    {  
        float maxCenterX = _tileMap.mapSize.width *  
        _tileMap.tileSize.width * kPointFactor - CENTER.x;  
        float cameraX = MAX(MIN(_hero.position.x, maxCenterX),  
        CENTER.x);  
        _viewPointOffset = cameraX - _eventCenter;  
        _eventState = kEventStateFreeWalk;  
    }  
    else if (_eventState == kEventStateFreeWalk)
```

```
{  
    //add this  
    if (_battleEvents.count == 0)  
    {  
        [self exitLevel];  
    }  
    else  
    {  
        [self cycleEvents];  
    }  
}  
else if (_eventState == kEventStateScripted) //add this  
{  
    float exitX = _tileMap.tileSize.width *  
    _tileMap.mapSize.width * kPointFactor + _hero.centerToSides;  
    if (_hero.position.x >= exitX)  
    {  
        _eventState = kEventStateEnd;  
        if (_currentLevel < _totalLevels - 1)  
        {  
            //next level  
            [[CCDirector sharedDirector]  
replaceScene:[CCTransitionFade transitionWithDuration:1.0  
scene:[GameScene nodeWithLevel:_currentLevel + 1]]];  
        }  
        else  
        {  
            //end game  
        }  
    }  
}  
}  
  
//add this new method  
-(void)exitLevel  
{  
    _eventState = kEventStateScripted;  
    float exitX = _tileMap.tileSize.width * _tileMap.mapSize.width  
    * kPointFactor + _hero.centerToSides;  
    [_hero enterFrom:_hero.position to:ccp(exitX,  
    _hero.position.y)];  
}
```

First you import GameScene.h so that you can access **GameScene** from within **GameLayer**.

Then you change `updateEvent` to check for a winning condition. If `eventState` is `kEventStateFreeWalk` and there are no more battle events left, then you trigger `exitLevel`. Otherwise, you cycle through the battle events as before.

`exitLevel` changes the `eventState` to `kEventStateScripted` so the player loses control of the hero. Then you make the hero walk from his current position past the right edge of the screen.

After the hero can no longer be seen, `updateEvent` changes the event to `kEventStateEnd`, signaling the end of the level. If the current level is not the last level, the scene transitions to a new instance of `GameScene` to load the next level.

If the current level is the last level, then the game should end. You'll add the code for that a little later on. ☺

For now, build and run, and see if you can make it through all the defined levels in `Levels.plist`. Exit stage right.



Ah, there's one final bug. At the end of each level, as the hero nears the edge of the map, you might see some robots disappearing right before your eyes. This visual glitch occurs because of the way you reset the robots.

Originally you made sure to reset the robots when they were half-a-screen away from the hero. This works well when the camera is always following the hero, but when the camera stops moving at the edge of the map, your trick becomes quite obvious.

To fix this, open `GameLayer.m` and modify `updatePositions` as follows:

```
//change this line
```

```
//if (robot.actionState == kActionStateDead &&
    _hero.groundPosition.x - robot.groundPosition.x >= CENTER.x +
    robot.contentSize.width/2 * kScaleFactor)
if (robot.actionState == kActionStateDead &&
    _hero.groundPosition.x - robot.groundPosition.x >= SCREEN.width +
    robot.contentSize.width/2 * kScaleFactor)
```

You simply replace `CENTER.x` with `SCREEN.width`. Now the robots are reset when they are one screen away.

## ARC and memory management

ARC lifts most of the burden of memory management from the developer. You don't have to look far for proof: it's been six chapters and you haven't had to pay much attention to how memory is being handled.

Since everything is "automatic", you wouldn't think that there would be any memory issues such as leaks, right?

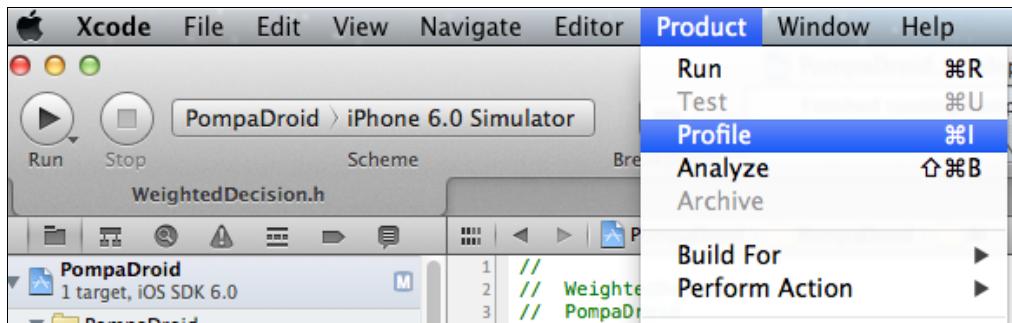
To definitively answer that question, you're going to profile your project for memory leaks.

To go through this section, you are going to have to play through an entire level. To make this go quicker, you might want to edit your `Levels.plist` to delete some of the battle events and robots so you can get through the level quickly.

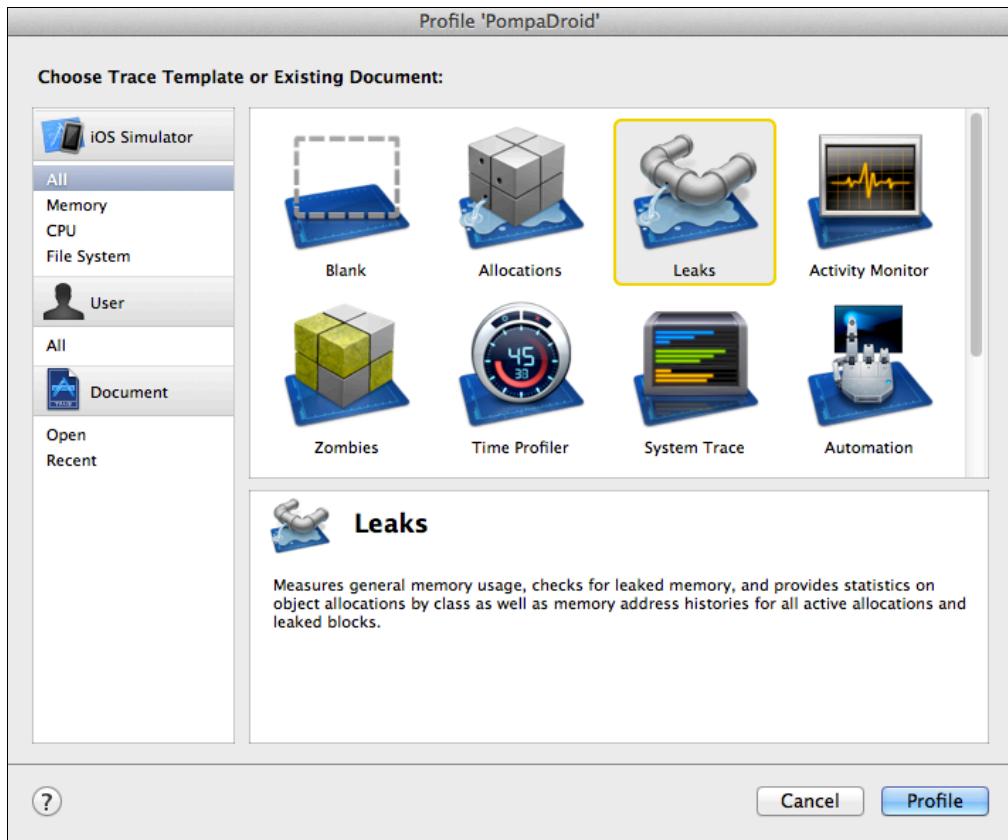
**Note:** Profiling your application just means putting its specific aspects to the test. This can mean anything from running CPU and GPU diagnostics to checking memory allocation and leaks.

## Do you have a leaky project?

Select **Product\Profile** (or use the Command+I keyboard shortcut) to run Instruments, the profiling tool.

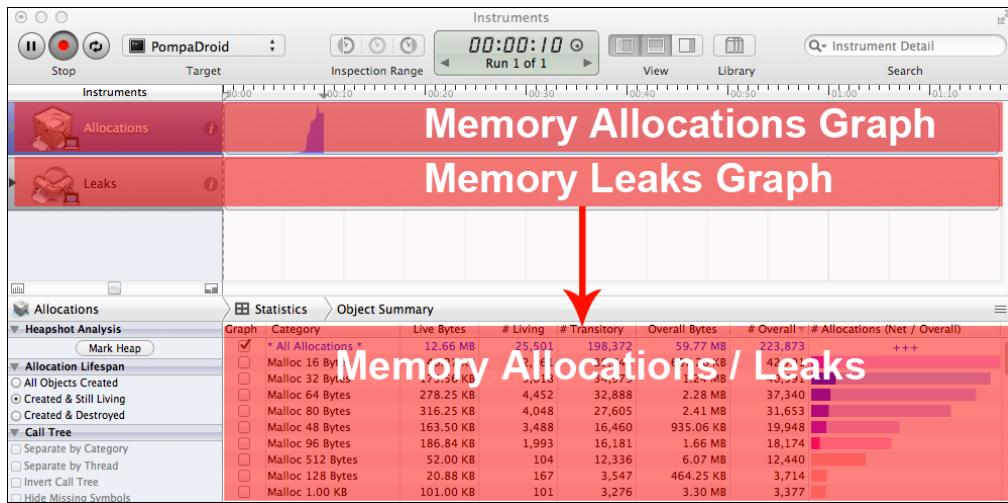


Wait a few seconds for your game to compile for the tool. When it's done, Instruments should ask you what kind of test you want to perform:



There are a lot of templates up for choice, but the one you want now is the **Leaks** test. Select it and click on the **Profile** button.

A new window will pop up showing you the Memory Allocations and Memory Leaks graphs:

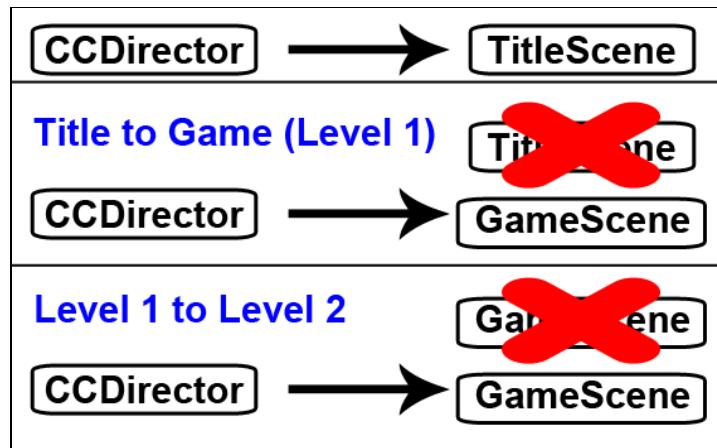


By default, the Memory Allocations Graph is selected, with the bottom panel showing all the allocations for the objects created by your code. If you click on the Memory Leaks Graph, the bottom panel will show you all your leaking objects.

**Note:** Your game might freeze from time to time while using the Leaks instrument. Don't worry – just wait it out. It happens because the Leaks instrument itself greatly reduces the performance of the game as it searches for memory leaks.

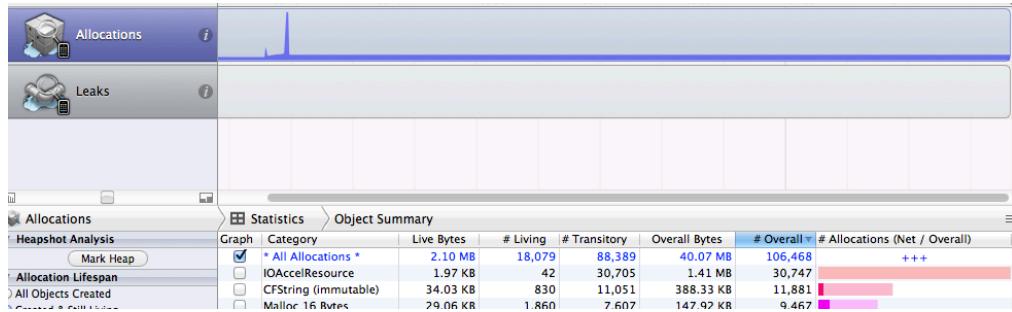
When the new window opened, you should have also seen your test device or simulator begin to run the game, showing you the title scene.

Memory leaks occur when objects aren't properly discarded/deallocated. In Cocos2D, objects are discarded in groups, depending on their `ccScene` membership. When the game transitions from one `ccScene` to another, it deallocates the old `ccScene` from memory, along with all objects inside it. Or, at least, that's the idea. ☺



Some time after you transition from `TitleScene` to `GameScene`, the game will discard `TitleScene`. If some of the objects within `TitleScene` leaked during the process, you can see it in Instruments. The same goes for switching from one `GameScene` to another.

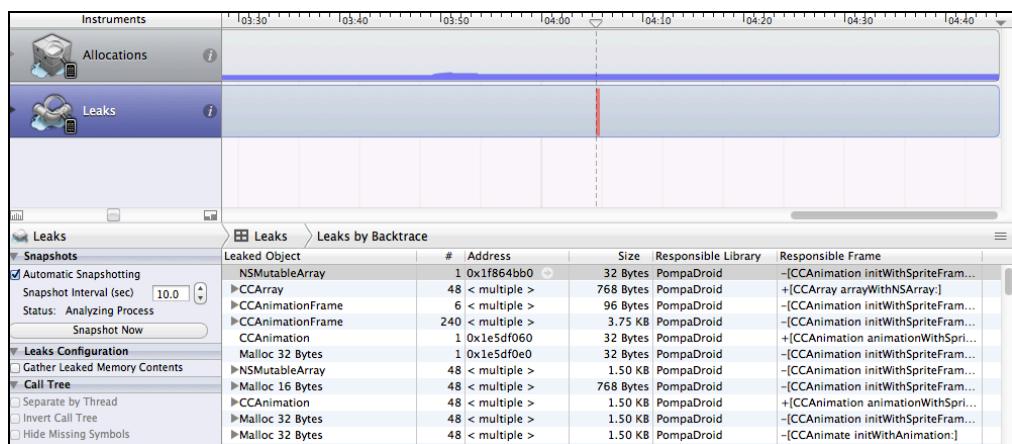
Now make the game transition from `TitleScene` to `GameScene` by tapping on the screen, and look at the Memory Leaks Graph.



It's empty, which means there are no leaking objects in `TitleScene`.

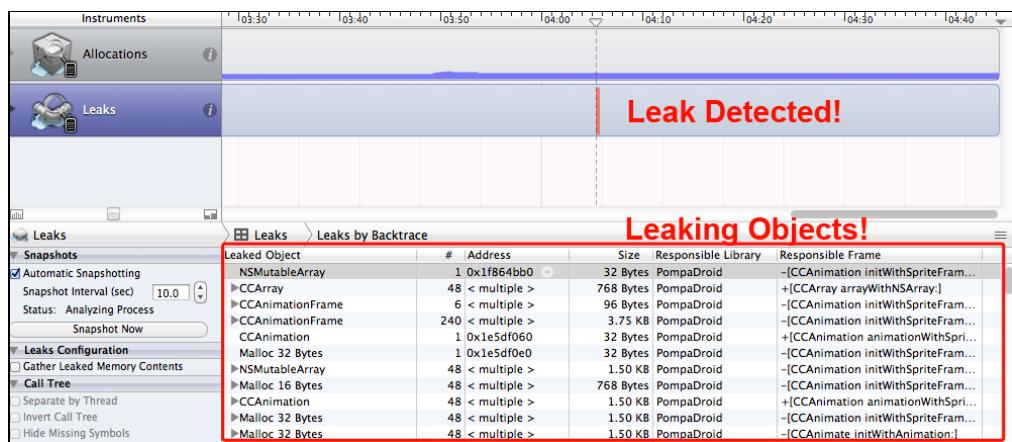
So far, so good! Now play until the end of the first level and let the game transition to the second level. After the second level loads, click on the Memory Leaks Graph again and monitor the bottom panel for objects.

Wait for at least two minutes.



If you wait long enough, you'll see something similar to the above image. There's a red mark on the Memory Leaks Graph, and a lot of objects in the bottom panel.

What does this mean?



That thin red line in the Memory Leaks Graph means there's a leak detected at that point in time, while the bottom panel shows you all the leaky culprits.

Among them are the hero, the robots, a bunch of `ccAnimations` and `ccArrays`, and probably every other object in the game given the size of the list!

But... but... but... shouldn't ARC have prevented this?



## The problem and the solution

To be fair, this wasn't all ARC's fault. Even when using ARC, developers are required to know a little bit about how memory management works so that they can adhere to the ARC rules. There are a number of reasons why the tool has reported so many leaks.

First, ARC only manages memory for Objective-C objects. This means you need to manually release any memory you allocate for non-Objective-C objects.

The arrays of attack and contact circles are one such example. To refresh your memory (pun intended!), this is how they were created:

```
self.attackPoints = malloc(sizeof(ContactPoint) *  
    self.attackPointCount);  
self.contactPoints = malloc(sizeof(ContactPoint) *  
    self.contactPointCount);
```

You created both `attackPoints` and `contactPoints` using the `malloc` command. This command is part of the C programming language, and is used to allocate memory for a collection of primitives such as `int`, `float` or `char`, or structures such as `ContactPoint`.

You reserved a portion of memory for the attack and contact point arrays using `malloc`. Since all robots and the hero have `attackPoints` and `contactPoints`, you can expect there to be a lot of leaks. Of course, these particular leaks are there only because you allocated the memory but did not explicitly release it.

There's a simple fix for this. Every class has a `dealloc` method that gets called automatically when an object is removed from memory. That's where you should place any additional instructions for manually cleaning up memory.

Open **ActionSprite.m** and add the method:

```
-(void) dealloc
{
    free(_attackPoints);
    free(_contactPoints);
}
```

The `free` command is the counterpart of `malloc`. Whereas `malloc` reserves memory, `free` clears up the reserved memory.

**Note:** Without ARC, `dealloc` plays a more important role. All the objects that need to be cleared from memory have to be released, or de-referenced, inside this method.

Recall that you actually added `dealloc` in **GameLayer** before. Here is the code once more:

```
-(void) dealloc
{
    [self unscheduleUpdate];
}
```

`unscheduleUpdate` is the counterpart of `scheduleUpdate`. The latter ensures that the `update:` method constantly runs while the program is alive, so you need to tell it to stop doing this or else you might not get back the memory allocated to perform this method.

Aside from **GameLayer**, there are two other places where you should add the `unscheduleUpdate` message – **ActionDPad** and **ActionButton**.

Open **ActionDPad.m** and **ActionButton.m**, and add this method in both files:

```
-(void) dealloc
{
    [self unscheduleUpdate];
}
```

If you like to save the best for last, this final memory issue is the most interesting one.

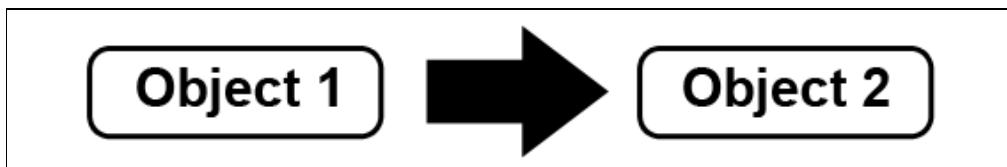
In Objective-C, an object is only cleaned up when no other object references it. This is why ARC includes the term “reference counting.” For each object, the compiler counts how many other objects still reference it. When there are no references left for an object, it gets removed from memory.

A memory leak occurs when objects aren’t deleted because they are still referenced somewhere, but the game doesn’t use them anymore, and all access to them from the current state of the program has been lost.

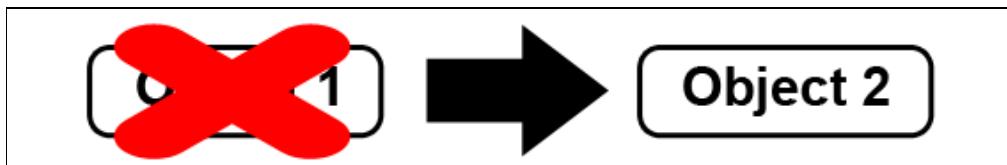
ARC’s job is to properly remove references to objects when they are not needed. However, there is one thing that confuses ARC – retain cycles.

A retain cycle occurs when two objects reference each other and it causes confusion about how to delete these objects.

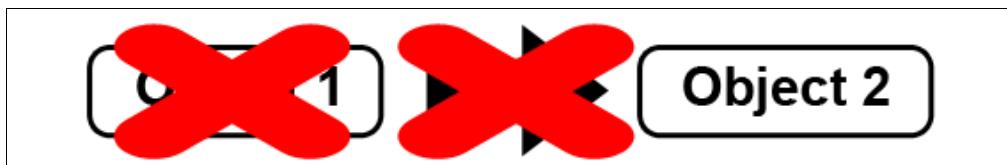
Imagine a normal situation where only one object references another:



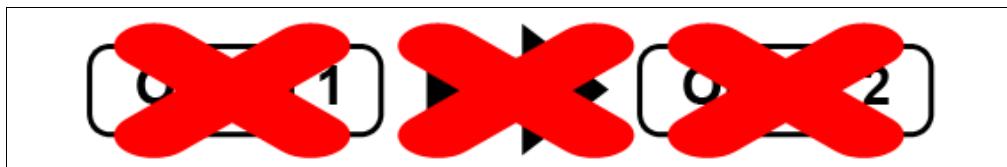
Object 1 is claiming ownership of Object 2. When nothing is claiming Object 1 anymore, then Object 1 will be removed from memory:



Because Object 1 is deleted, its reference to Object 2 is also deleted:



With the reference to Object 2 gone, there are no more objects claiming ownership of Object 2. As a result, Object 2 is also removed from memory.



In this situation, everyone is happy and no memory gets leaked.

A retain cycle situation looks like this:



Because Object 2 is referencing Object 1, Object 1 doesn't get deleted. On the other hand, Object 1 is also referencing Object 2, so Object 2 doesn't get deleted either. Both objects retain ownership of each other, and this makes them co-dependent.



When the game deletes the `ccScene` housing these two objects, the objects drift to the land of the lost, never to be accessed again – but they still occupy space in memory.

In ARC, the setter semantic, or the words inside the parenthesis of an `@property` line, dictates how an object is owned.

```
@property(nonatomic, strong)
```

The keyword `strong` indicates an owning relationship, as seen in the diagrams above.

```
@property(nonatomic, weak)
```

The keyword `weak` indicates a non-owning relationship. An object is not required to wait for non-owning references to be removed before it itself is deleted from memory. If Object 1 has a weak reference to Object 2, and there are no other references to Object 2, then Object 2 still gets deleted.

In practice, you have already been doing some references correctly. One example is the relationship between `GameLayer` and various `ActionSprites`.

Take the example of the hero. `GameLayer` has the property declared as:

```
@property(nonatomic, strong)Hero *hero;
```

As long as `GameLayer` exists, the hero object will not be released from memory. The hero also has a reference to `GameLayer` in this code in `ActionSprite`:

```
@property(nonatomic, weak)id <ActionSpriteDelegate> delegate;
```

Remember that the delegate here is `GameLayer`. This means that `GameLayer` references the hero, and the hero also references `GameLayer`.

The good thing is that the strong owning relationship only goes one way. The hero only has a weak reference to `GameLayer`, so it doesn't create a retain cycle and there's no memory issue when deciding whether or not to delete `GameLayer` from memory.

Now take a look at this line in `AnimationMember.h`:

```
@property(nonatomic, strong)CCSprite *target;
```

`AnimationMember` strongly references the target `ccsprite` when really it's just the `ccsprite` that needs to reference `AnimationMember` strongly.

Make these changes to `AnimationMember.h`:

```
//change CCSprite *_target to  
__weak CCSprite *_target;  
  
//change this property @propert(nonatomic, strong)CCSprite  
*target;  
@property(nonatomic, weak)CCSprite *target;
```

You simply change the property from `strong` to `weak`. For instance variables with properties that have `weak` semantics, you need to add the `__weak` directive to the declaration.

The next example is not as obvious because it is related to how Cocos2D operates.

In Cocos2D, when an object such as a `ccsprite` or `ActionSprite` uses the `ccCallFunc` and `ccCallBlock` actions, they keep a strong reference to that object.

When you created the hero's attack action, you used this line in `Hero.m`:

```
self.attackAction = [CCSequence actions:[CCAnimate  
actionWithAnimation:attackAnimation], [CCCallFunc  
actionWithTarget:self selector:@selector(idle)], nil];
```

Here, the `ccCallFunc` action will strongly reference `Hero`. As long as the `ccCallFunc` action `attackAction` exists, `Hero` cannot be removed from memory.

However, `attackAction` itself is also strongly referenced by `Hero` via this property:

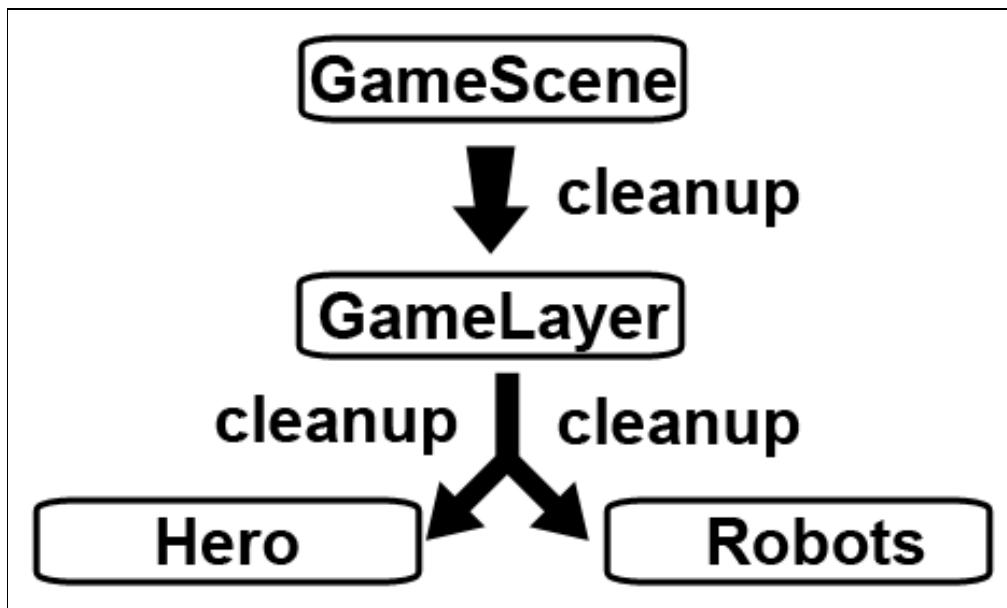
```
@property(nonatomic, strong)id attackAction;
```

Since `attackAction` is referenced strongly, it cannot be deleted, and since `Hero` is referenced by `attackAction` strongly (because of `ccCallFunc`), `Hero` cannot be deleted either. You have a retain cycle on your hands.

You could resolve this by removing the `strong` property for these actions. But if you did that, you'd always have to create the actions at the exact moment you needed them, because they wouldn't exist otherwise. If they have a `weak` property, they'll be deleted from memory right after you create them in `init`.

Another solution is to clear these actions and set them to `nil` when the `ActionSprite` is no longer used. You might be tempted to use `dealloc`, but remember that `dealloc` only gets called when removing an object from memory, which in this case won't happen.

You need a pre-step to `dealloc`. Thankfully, the `CCNode` class (the parent class of all Cocos2D objects including `CCSprite` and `ActionSprite`) has just such a method, called `cleanup`.



`cleanup` is executed right before an object is deleted, and it also executes `cleanup` on all child objects. In other words, `cleanup` is executed before `dealloc` and is the perfect place to remove `strong` references to `ActionSprite`'s actions!

Go to `ActionSprite.m` and add this method:

```
- (void)cleanup
{
    self.idleAction = nil;
    self.attackAction = nil;
```

```

    self.walkAction = nil;
    self.hurtAction = nil;
    self.knockedOutAction = nil;
    self.recoverAction = nil;
    self.runAction = nil;
    self.jumpRiseAction = nil;
    self.jumpFallAction = nil;
    self.jumpLandAction = nil;
    self.jumpAttackAction = nil;
    self.runAttackAction = nil;
    self.dieAction = nil;

    [super cleanup];
}

```

Next go to **Hero.m** and add this method:

```

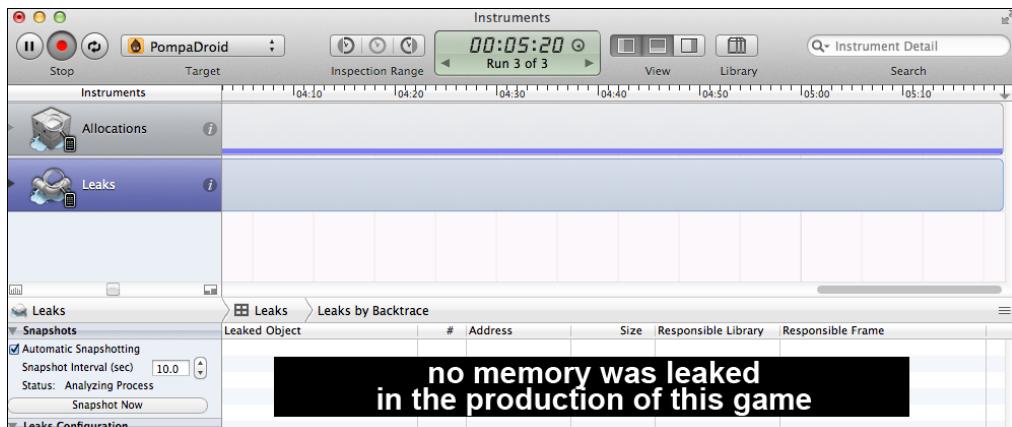
-(void)cleanup
{
    self.attackTwoAction = nil;
    self.attackThreeAction = nil;

    [super cleanup];
}

```

By setting the above actions to `nil`, you remove the `strong` reference to the `CCActions` they contained before.

That's it! Profile your game using the Leaks instrument again, and play until you get to the second level.



That's what you want to see! 😊

You've been through a lot these past six chapters. Pause and reflect – you started with nothing but an empty project to your name, and now you have a Beat 'Em Up Game with multiple levels!

As usual, you can choose to stop here and expand the game on your own, or continue on to the final two chapters to add even more content and polish.

But come on – since you're already here, you might as well go all the way to the end!

**Challenge:** Now that you have a solid basis for a Beat 'Em Up Game, try tweaking the gameplay to be more of your liking. Here are some ideas:

Make the 1-2-3 punch always trigger, even if you're not punching a robot at the moment.

For a punch to connect, the robot must be very near the hero vertically. Try relaxing this requirement a bit to be more forgiving.

To run, you have to double tap very rapidly to trigger the run, and less experienced players might miss this. Increase the time to make running easier.

Robots currently take a long time to beat up. Make them easier to kill!

Anything else you'd like to tweak for your own personal preferences. Tweaking small things like this will really help you reinforce the concepts you've learned so far!



# Chapter 7: The Droids Strike Back

In the last chapter, your hero got some spectacular new moves. You have a solid basis for a game, but you're still missing a few important things like a health display, enemy variety, and most importantly – a bad-ass boss fight!

In this chapter, you'll start by enhancing the game's HUD to make for a more player-friendly experience. A player ought to know, for example, the state of the hero's health and the amount of damage his attacks are dealing. You'll also add fireworks to the battle sequences with some explosive hit animations.

Then, in the second half of the chapter, you'll enhance the robot army by making some robots more powerful than others – and more powerful than any robot presently is. Finally, you'll create a Big Bad Boss Enemy that will await the hero in the final battle.

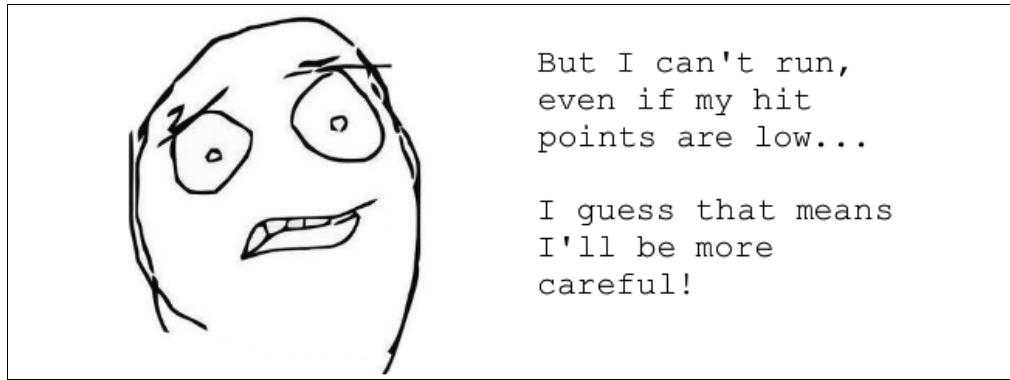
This chapter begins right where the last chapter left off. Remember that if you skipped the last chapter or missed some parts, you can start this chapter using the Chapter 6 project that's included in the resources.

Ready to find out what other challenges the hero has in store for him?

## Word to the player

If you followed the previous chapters to the letter, your HUD currently shows only the D-pad and the two buttons. There's a lot of space left, which is good, because there's more information to display that will be useful to the player.

When playing the game, it's important for the player to know how many hit points he has left. A player needs to know his odds against the robot hordes!



You will display the hero's hit points as text, but you don't want to use a plain old system font. To keep in line with the game's style, you'll use a custom bitmap font atlas that was created in GlyphDesigner.

**Note:** By default, the iPhone can use only system fonts such as Arial and Helvetica. To add new fonts, you have to add the font files to your project. By doing this, your game will be able to use those fonts as-is.

However, there are times when you want to tweak the look of a font by adding stroke outlines, shadows etc. This is where GlyphDesigner comes in. You can use it to add custom effects to any font installed on your system, and then export those fonts in a format that Cocos2D can understand.

If you wish to see how the fonts used in this game were made, [download GlyphDesigner](#) and open the GlyphProject files found in the Raw folder of your tutorial resources.

First add the font files to your project. In the **Resources** directory that came with this Starter Kit, there is a subdirectory named **Fonts**.

Once you find the **Fonts** folder, drag it into your Xcode project. In the pop-up dialog, make sure that "Copy items into destination group's folder (if needed)" is checked, "Create groups for any added folders" is selected and the **PompaDroid** target is checked, then click **Finish**.

**Note:** The drag-and-drop functionality is buggy in some versions of Xcode. If you experience build errors after adding new files, try adding the files again using the **File\Add Files to "Project Name"** option, or **Shift + Command + A**.

You should find the following files in the new Fonts folder:

- **HudFont/-hd/-ipadhd.png:** The font atlas you'll use for HUD messages. These contain images of the available letters for the bitmap font. Think of it as a sprite

sheet, but with letters instead of images. There are three sets to support different resolutions.

- **HudFont/-hd/-ipadhd.fnt:** The font file contains the mapping of each letter to the corresponding image inside the font atlas. It's similar to the PLIST file of a sprite sheet.
- **DamageFont/-hd/-ipadhd.fnt/png:** These will come in handy later in this chapter.

The hero's hit points should be displayed as part of the HUD. So naturally, you will add them to **HudLayer**. But first you need some definitions.

Go to **Defines.h** and add these:

```
#define COLOR_FULLHP ccc3(95, 255, 106)
#define COLOR_MIDHP ccc3(255, 165, 0)
#define COLOR_LOWHP ccc3(255, 50, 23)
```

These are color definitions that you will use later for the hit points. Coloring labels involves the same concept as tinting sprites, which you did in Chapter 2. Remember – since the font is white, you can simply apply a tint color to make it any color you'd like dynamically at runtime.

Open **HudLayer.h** and do the following:

```
//add this instance variable inside the curly braces of @interface
CCLabelBMFont *_hitPointsLabel;

//add this method prototype
-(void)setHitPoints:(float)newHP fromMaxHP:(float)maxHP;
```

**CCLabelBMFont** displays text using a bitmap font. **\_hitPointsLabel** will be the label containing the hero's hit points. You'll use **setHitPoints:fromMaxHP:** to change the text displayed by the label.

Switch to **HudLayer.m** and do the following:

```
//add inside init, right after [self addChild:_buttonA]
float xPadding = 10.0 * kPointFactor;
float yPadding = 18.0 * kPointFactor;
_hitPointsLabel = [CCLabelBMFont labelWithString:@"0"
fntFile:@"HudFont.fnt"];
_hitPointsLabel.anchorPoint = ccp(0.0, 0.5);
_hitPointsLabel.position = ccp(xPadding, SCREEN.height -
_hitPointsLabel.contentSize.height/2 - yPadding);
[self addChild:_hitPointsLabel];

//add this method
```

```
- (void) setHitPoints:(float) newHP fromMaxHP:(float) maxHP
{
    int wholeHP = newHP;

    [_hitPointsLabel setString:[NSString stringWithFormat:@"%@", wholeHP]];

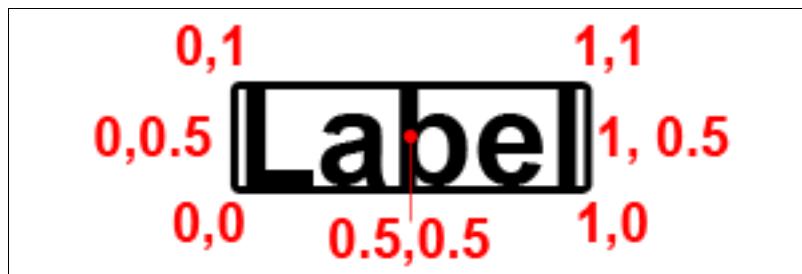
    float ratio = newHP / maxHP;

    if (ratio > 0.6)
    {
        _hitPointsLabel.color = COLOR_FULLHP;
    }
    else if (ratio > 0.2)
    {
        _hitPointsLabel.color = COLOR_MIDHP;
    }
    else
    {
        _hitPointsLabel.color = COLOR_LOWHP;
    }
}
```

The code initializes a new `cclabelBMFont` using the font atlas you added to the project. Then it positions it at the top-left corner of the screen.

The `anchorPoint` value changes the origin coordinate of the label, or which part of the label will be placed at the position you set. Its value follows OpenGL Texture coordinates.

Refer to the image below. By default, `anchorPoints` of Cocos2D objects such as `ccsprite` and `cclabelBMFont` are located at the center, or at coordinate (0.5, 0.5). By setting the `anchorPoint` to (0.0, 0.5), you anchor the label at its left side.



Think of it like left-aligning the text. No matter how long or short the text gets, its position from the left side of the screen remains the same.

In `setHitPoints`, you calculate the ratio between the current and max hit points of the hero and do the following:

- If the current hit points are more than 60% of the max, then you color the hit point label green.
- If the current hit points are between 20% and 60% of the max, then you color the label orange.
- If the current hit points are 20% or less of the max, then you make the label red.

This coloring system helps the player know the hero's health situation at a glance, without even having to comprehend the numbers.

After that, you set the label to show the numerical value of the hero's current hit points. You convert the value to an integer first to shave off the decimal places, then use `stringWithFormat:` to convert the integer into text.

Now go to **GameLayer.m** and do the following:

```
//add inside actionSpriteDidAttack:, right after [_hero
hurtWithDamage:actionSprite.attackDamage
force:actionSprite.attackForce
direction:ccp(actionSprite.directionX, 0.0)];
[_hud setHitPoints:_hero.hitPoints fromMaxHP:_hero.maxHitPoints];

//add this inside actionSpriteDidDie:, inside the curly braces of
if (actionSprite == _hero)
[_hud setHitPoints:0 fromMaxHP:_hero.maxHitPoints];

//add this method
-(void)setHud:(HudLayer *)hud
{
    _hud = hud;
    [_hud setHitPoints:_hero.hitPoints
fromMaxHP:_hero.maxHitPoints];
}
```

The above code updates the `hitPoints` label whenever the hero gets hurt. It also updates the `hitPoints` label of the `HudLayer` when a new HUD layer is set.

Build and run, and your HUD should now display the hero's health in color at the top-left corner of the screen.



The player might also find it helpful to know when the hero is allowed to walk freely. Whenever a battle ends, you can tell the player to move right by flashing a simple message.

Open **HudLayer.h** and do the following:

```
//add this instance variable inside the curly braces of @interface
CCLabelBMFont *_goLabel;

//add this method prototype
-(void)showGoMessage;
```

Switch to **HudLayer.m** and do the following:

```
//add inside init, after [self addChild:_hitPointsLabel]
_goLabel = [CCLabelBMFont labelWithString:@"[ GO>"
fntFile:@"HudFont.fnt"];
_goLabel.anchorPoint = ccp(1.0, 0.5);
_goLabel.position = ccp(SCREEN.width - xPadding, SCREEN.height -
_goLabel.contentSize.height/2 - yPadding);
_goLabel.color = COLOR_FULLHP;
_goLabel.visible = NO;
[self addChild:_goLabel];

//add this method
-(void)showGoMessage
{
    [_goLabel stopAllActions];
    [_goLabel runAction:[CCSequence actions: [CCShow action],
[CCBlink actionWithDuration:3.0 blinks:6], [CCHide action], nil]];
}
```

```
}
```

This creates `goLabel`, similar to `hitPointsLabel`. This time, the `anchorPoint` is to the right of the label, which is placed on the upper-right side of the screen. The label's `visible` property is set to `NO` so that it doesn't show up automatically when the game starts.

You'll use `showGoMessage` to make the label appear and blink six times in three seconds before hiding itself.

Switch to `GameLayer.m` and make the following changes:

```
//add this method
-(void)setEventState:(EventState)eventState
{
    _eventState = eventState;

    if (_eventState == kEventStateFreeWalk)
    {
        [_hud showGoMessage];
    }
}

// Make the following changes in updateEvent
-(void)updateEvent
{
    if (_eventState == kEventStateBattle && _activeEnemies <= 0)
    {
        float maxCenterX = _tileMap.mapSize.width *
_tileMap.tileSize.width * kPointFactor - CENTER.x;
        float cameraX = MAX(MIN(_hero.position.x, maxCenterX),
CENTER.x);
        _viewPointOffset = cameraX - _eventCenter;

        //modified this part
        if (_battleEvents.count == 0)
        {
            [self exitLevel];
        }
        else
        {
            self.eventState = kEventStateFreeWalk;
        }
    }
    else if (_eventState == kEventStateFreeWalk)
    {
```

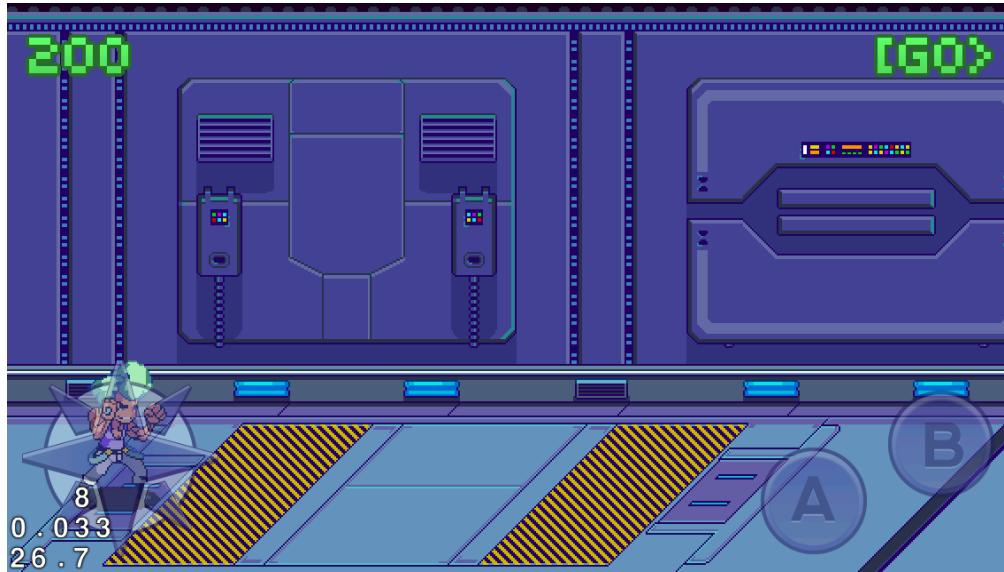
```
//modified this part
    [self cycleEvents];
}
else if (_eventState == kEventStateScripted)
{
    float exitX = _tileMap.tileSize.width *
_tileMap.mapSize.width * kPointFactor + _hero.centerToSides;
    if (_hero.position.x >= exitX)
    {
        _eventState = kEventStateEnd;
        if (_currentLevel < _totalLevels - 1)
        {
            //next level
            [[CCDirector sharedDirector]
replaceScene:[CCTransitionFade transitionWithDuration:1.0
scene:[GameScene nodeWithLevel:_currentLevel + 1]]];
        }
        else
        {
            //end game
        }
    }
}
```

The above changes to the code ensure that the GO message is shown when the `eventState` becomes `kEventStateFreeWalk`.

You've also changed the line in `updateEvent` that switches the `eventState` to `kEventStateFreeWalk`. Instead of changing the value of `eventState` directly, you use the new setter method by using `self.eventState`.

Additionally, you've moved the checker that counts if there are still any battle events left to the first `if` condition. This way, the game won't switch to a free roam event anymore when there are no more battle events left, and it also won't display the GO message.

Build and run, and the new message should flash in the top-right corner.



To spoil the player with more status indicators, you can also add a level indicator. In case the player can't count the levels or has lost track of their progress, this indicator will show the current level at the start of every level.



Open **HudLayer.h** once again and do the following:

```
//add this instance variable inside the curly braces of @interface
CCLabelBMFont *_centerLabel;

//add this method prototype
-(void)displayLevel:(int)level;
```

Switch to **HudLayer.m** and do the following:

```
//add inside init, after [self addChild:_goLabel]
_centerLabel = [CCLabelBMFont labelWithString:@"LEVEL 1"
fntFile:@"HudFont.fnt"];
_centerLabel.color = COLOR_MIDHP;
_centerLabel.visible = NO;
```

```
[self addChild:_centerLabel];

//add this method
-(void)displayLevel:(int)level
{
    [_centerLabel setString:[NSString stringWithFormat:@"LEVEL %d", level]];
    [_centerLabel runAction:[CCSequence actions: [CCPlace
actionWithPosition:ccp(SCREEN.width +
_centerLabel.contentSize.width/2, CENTER.y)], [CCShow action],
[CCMoveTo actionWithDuration:0.2 position:CENTER], [CCDelayTime
actionWithDuration:1.0], [CCMoveTo actionWithDuration:0.2
position:ccp(-_centerLabel.contentSize.width/2, CENTER.y)],
[CCHide action], nil]]];
}
```

The above code creates a new label named `centerLabel`, the text of which can be changed by `displayLevel` to show the current level. The label then slides in from the right side of the screen, stays in the middle for a short while and slides out to the left.

Now go to **GameLayer.m** and do this in `onEnterTransitionDidFinish`:

```
//add this line after [super onEnterTransitionDidFinish]
[_hud displayLevel:_currentLevel + 1];
```

This makes the game show the current level right after `GameLayer` appears. You have to add one to the value of `currentLevel` because it starts counting levels from zero.

Build and run, and the level text should appear at the beginning of each level.

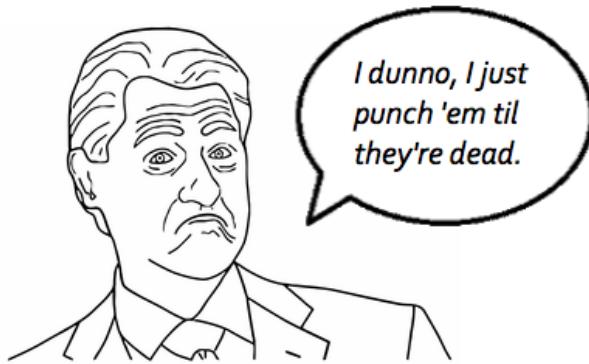


Now your players should feel very well informed. ☺

## What's your damage?

Now your game screen provides a lot of information at a glance. But there's another bone you can throw to the player that will make the game a lot more engaging.

With all the beat-downs going on, and all the hero's awesome moves, it's hard to tell which attack is the most effective. As the developer, you have knowledge of all the numbers in the game, including how much damage each attack deals. But the average player has no way of knowing, for example, that the second punch is stronger than the first punch.



One obvious solution is to display the damage numbers for every attack as it occurs, so long as the attack connects with an enemy. That's what you'll do next.

Select the **PompaDroid** group in Xcode, go to **File\New\Group** and name the new group **Effects**.

Next, select the **Effects** group and go to **File\New\ File**. Choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **CCLabelBMFont** for Subclass of, click **Next** and name the new file **DamageNumber**.

Open **DamageNumber.h** and do the following:

```
//add this property
@property(nonatomic, strong)id damageAction;

//add this method prototype
-(void)showWithValue:(int)value fromOrigin:(CGPoint)origin;
```

This defines a property you will use to store the action that moves the number a bit after it appears, and a helper method to present a damage number when the hero hits a robot.

Switch to **DamageNumber.m** and add these methods:

```
-(id)init
{
    if ((self = [super initWithString:@"0"
fntFile:@"DamageFont.fnt"]))
    {
        self.damageAction = [CCSequence actions:[CCShow action],
[CCMoveBy actionWithDuration:0.6 position:ccp(0.0, 40.0 *
kPointFactor)], [CCHide action], nil];
    }
    return self;
}
-(void)showWithValue:(int)value fromOrigin:(CGPoint)origin
{
    self.string = [NSString stringWithFormat:@"%d", value];
    self.position = origin;
    [self stopAllActions];
    [self runAction:_damageAction];
}
```

This creates a subclass of **CCLabelBMFont**, which does the following:

- It uses the atlas font DamageFont.fnt. You added this font along with HudFont.fnt in the previous section.
- It stores an action named **damageAction**. This action makes the label appear, and move up on the screen by a few coordinates before disappearing again.
- **showWithValue:fromOrigin:** changes the label's text and makes it appear at a certain position before executing **damageAction**.

Whenever an attack connects with an enemy, a **DamageNumber** should appear with the damage value of that attack.

Go to **GameLayer.h** and add this property:

```
@property(nonatomic, strong)CCArray *damageNumbers;
```

Switch to **GameLayer.m** and do the following:

```
//add to top of file
#import "DamageNumber.h"

//add to initWithLevel:, right after [self initBrains]
[self initEffects];

//add these methods
-(void)initEffects
```

```

{
    int i;

    self.damageNumbers = [CCArray arrayWithCapacity:20];
    DamageNumber *number;

    for (i = 0; i < 20; i++)
    {
        number = [DamageNumber node];
        number.visible = NO;
        number.position = OFFSCREEN;
        [self addChild:number];
        [_damageNumbers addObject:number];
    }
}

-(DamageNumber *)getDamageNumber
{
    DamageNumber *number;
    CCARRAY_FOREACH(_damageNumbers, number)
    {
        if (number.numberOfRunningActions == 0)
        {
            return number;
        }
    }
    return number;
}

```

The changes to `initWithLevel:` create 20 `DamageNumbers`, makes them invisible, sets them somewhere offscreen and stores them in an array. `getDamageNumber` retrieves any available `DamageNumber`.

This is similar to how you fetch robots. To make things work faster and to save you from having to create objects on the fly, you cache a number of objects and reuse them whenever possible. When a `DamageNumber` is not running any action (specifically, not running `damageAction`), then it is available for use.

Next, still in `GameLayer.m`, add these to `actionSpriteDidAttack:` (existing code has dark highlights):

```

-(BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    BOOL didHit = NO;
    if (actionSprite == _hero)
    {

```

```
CGPoint attackPosition;
Robot *robot;
CCARRAY_FOREACH(_robots, robot)
{
    if (robot.actionState < kActionStateKnockedOut &&
robot.actionState != kActionStateNone)
    {
        if ([self collisionBetweenAttacker:_hero
andTarget:robot atPosition:&attackPosition])
        {
            //add this
            DamageNumber *damageNumber = [self
getDamageNumber];

            if (_hero.actionState ==
kActionStateJumpAttack)
            {
                [robot
knockoutWithDamage:_hero.jumpAttackDamage
direction:ccp(_hero.directionX, 0)];
                //add this
                [damageNumber
showWithValue:_hero.jumpAttackDamage fromOrigin:robot.position];
            }
            else if (_hero.actionState ==
kActionStateRunAttack)
            {
                [robot
knockoutWithDamage:_hero.runAttackDamage
direction:ccp(_hero.directionX, 0)];
                //add this
                [damageNumber
showWithValue:_hero.runAttackDamage fromOrigin:robot.position];
            }
            else if (_hero.actionState ==
kActionStateAttackThree)
            {
                [robot
knockoutWithDamage:_hero.attackThreeDamage
direction:ccp(_hero.directionX, 0)];
                //add this
                [damageNumber
showWithValue:_hero.attackThreeDamage fromOrigin:robot.position];
            }
        }
    }
}
```

```
        else if (_hero.actionState ==  
kActionStateAttackTwo)  
    {  
        [robot  
hurtWithDamage:_hero.attackTwoDamage force:_hero.attackForce  
direction:ccp(_hero.directionX, 0.0)];  
        //add this  
        [damageNumber  
showWithValue:_hero.attackTwoDamage fromOrigin:robot.position];  
    }  
    else  
    {  
        [robot hurtWithDamage:_hero.attackDamage  
force:_hero.attackForce direction:ccp(_hero.directionX, 0.0)];  
        //add this  
        [damageNumber  
showWithValue:_hero.attackDamage fromOrigin:robot.position];  
    }  
    didHit = YES;  
}  
}  
}  
}  
return didHit;  
}  
else  
{  
    if (_hero.actionState < kActionStateKnockedOut &&  
_hero.actionState != kActionStateNone)  
    {  
        CGPoint attackPosition;  
        if ([self collisionBetweenAttacker:actionSprite  
andTarget:_hero atPosition:&attackPosition])  
        {  
            [_hero hurtWithDamage:actionSprite.attackDamage  
force:actionSprite.attackForce  
direction:ccp(actionSprite.directionX, 0.0)];  
            [_hud setHitPoints:_hero.hitPoints  
fromMaxHP:_hero.maxHitPoints];  
            didHit = YES;  
  
            //add these  
            DamageNumber *damageNumber = [self  
getDamageNumber];  
        }  
    }  
}
```

```
        [damageNumber  
showWithValue:actionSprite.attackDamage  
fromOrigin:_hero.position];  
    }  
}  
  
return didHit;  
}
```

Take note of the comments in the code above. The lines after the comments are the ones that need to be added.

Whenever an attack connects, from either a robot or the hero, the code fetches an unused `DamageNumber`, gives it the appropriate damage value based on the attack and makes it appear from the position of the attack's recipient.

Build and run, and do the 5-10-20 punch! You'll see what I mean. ☺



## Hands on fire!

If there's one thing that can express power more than numbers, it's explosions! So that the fights look even more spectacular, you're going to add explosion animations when attacks connect.

This will also help "sell" the battles, because without any hit animations, they can look somewhat choreographed. And you want to convey life-or-death scenarios, not dance routines.

Select the **Effects** group, go to **File\New\File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **CCSprite** for Subclass of, click **Next** and name the new file **HitEffect**.

Open **HitEffect.h** and do the following:

```
//add this property
@property(nonatomic, strong)id effectAction;

//add this method prototype
-(void)showEffectAtPosition:(CGPoint)position;
```

Similar to how you implemented `DamageNumber`, here you create a property to store the action to display the effect, and a helper method to show the effect at a given position.

Switch to **HitEffect.m** and add these methods:

```
-(id)init
{
    if ((self = [super
initWithSpriteFrameName:@"hiteffect_00.png"]))
    {
        CCArray *frames = [CCArray arrayWithCapacity:6];
        int i;
        CCSpriteFrame *frame;
        for (i = 0; i < 6; i++)
        {
            frame = [[CCSpriteFrameCache sharedSpriteFrameCache]
spriteFrameByName:[NSString
stringWithFormat:@"hiteffect_%02d.png", i]];
            [frames addObject:frame];
        }

        self.effectAction = [CCSequence actions:[CCShow action],
[CCAnimate actionWithAnimation:[CCAnimation
animationWithSpriteFrames:[frames getNSArray] delay:1.0/12.0]],
[CCHide action], nil];
    }
    return self;
}

-(void)showEffectAtPosition:(CGPoint)position
{
    [self stopAllActions];
    self.position = position;
    [self runAction:_effectAction];
```

```
}
```

To create `HitEffect`, you mix some concepts from both `DamageNumber` and `ActionSprite`. Like `ActionSprite`, `HitEffect` is a subclass of `ccsprite`, and is created with an image found in the sprite sheet you added in Chapter 1.

Then, similar to `DamageNumber`, it has a stored action, `effectAction`, to make it appear and disappear. But unlike `DamageNumber`, `HitEffect` will animate using sprite frames in between appearing and disappearing.

`showEffectAtPosition`: simply places the effect at a specified position before executing `effectAction`.

To put `HitEffect` to use, you'll retrace some familiar steps.

Go to `GameLayer.h` and add this property:

```
@property(nonatomic, strong)CCArray *hitEffects;
```

Switch to `GameLayer.m` and do the following:

```
//add to top of file
#import "HitEffect.h"

//add inside initEffects, after the first for loop
self.hitEffects = [CCArray arrayWithCapacity:20];
HitEffect *effect;

for (i = 0; i < 20; i++)
{
    effect = [HitEffect node];
    effect.visible = NO;
    effect.scale *= kScaleFactor;
    effect.position = OFFSCREEN;
    [_actors addChild:effect];
    [_hitEffects addObject:effect];
}

//add this method
-(HitEffect *)getHitEffect
{
    HitEffect *effect;
    CCARRAY_FOREACH(_hitEffects, effect)
    {
        if (effect.numberOfRunningActions == 0)
        {
            return effect;
        }
    }
}
```

```
    }
    return effect;
}
```

This initializes 20 `HitEffects`, scales them to the device, makes them disappear and stores them in an array. This time, instead of adding them directly to `GameLayer`, you add them as children of the batch node.

`getHitEffect` works the same as `getDamageNumber`. It retrieves an unused `hitEffect`.

Still in `GameLayer.m`, modify `actionSpriteDidAttack:` as follows (existing code has dark highlights):

```
- (BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    BOOL didHit = NO;
    if (actionSprite == _hero)
    {
        CGPoint attackPosition;
        Robot *robot;
        CCARRAY_FOREACH(_robots, robot)
        {
            if (robot.actionState < kActionStateKnockedOut &&
robot.actionState != kActionStateNone)
            {
                if ([self collisionBetweenAttacker:_hero
andTarget:robot atPosition:&attackPosition])
                {
                    //add this
                    BOOL showEffect = YES;

                    DamageNumber *damageNumber = [self
getDamageNumber];

                    if (_hero.actionState ==
kActionStateJumpAttack)
                    {
                        [robot
knockoutWithDamage:_hero.jumpAttackDamage
direction:ccp(_hero.directionX, 0)];
                        [damageNumber
showWithValue:_hero.jumpAttackDamage fromOrigin:robot.position];
                        //add this
                        showEffect = NO;
                    }
                    else if (_hero.actionState ==
kActionStateRunAttack)
```

```
        {
            [robot
knockoutWithDamage:_hero.runAttackDamage
direction:ccp(_hero.directionX, 0)];
                [damageNumber
showWithValue:_hero.runAttackDamage fromOrigin:robot.position];
            }
            else if (_hero.actionState ==
kActionStateAttackThree)
            {
                [robot
knockoutWithDamage:_hero.attackThreeDamage
direction:ccp(_hero.directionX, 0)];
                [damageNumber
showWithValue:_hero.attackThreeDamage fromOrigin:robot.position];
                //add this
                showEffect = NO;
            }
            else if (_hero.actionState ==
kActionStateAttackTwo)
            {
                [robot
hurtWithDamage:_hero.attackTwoDamage force:_hero.attackForce
direction:ccp(_hero.directionX, 0.0)];
                [damageNumber
showWithValue:_hero.attackTwoDamage fromOrigin:robot.position];
            }
            else
            {
                [robot hurtWithDamage:_hero.attackDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0.0)];
                [damageNumber
showWithValue:_hero.attackDamage fromOrigin:robot.position];
            }
            didHit = YES;

        //add this if statement and its contents
        if (showEffect)
        {
            HitEffect *hitEffect = [self
getHitEffect];
            [_actors reorderChild:hitEffect
z:MAX(robot.zOrder, _hero.zOrder) + 1];
            [hitEffect
showEffectAtPosition:attackPosition];
        }
    }
}
```

```

        }

    }

    return didHit;
}

else
{
    if (_hero.actionState < kActionStateKnockedOut &&
_hero.actionState != kActionStateNone)
    {
        CGPoint attackPosition;
        if ([self collisionBetweenAttacker:actionSprite
andTarget:_hero atPosition:&attackPosition])
        {
            [_hero hurtWithDamage:actionSprite.attackDamage
force:actionSprite.attackForce
direction:ccp(actionSprite.directionX, 0.0)];
            [_hud setHitPoints:_hero.hitPoints
fromMaxHP:_hero.maxHitPoints];
            didHit = YES;

            DamageNumber *damageNumber = [self
getDamageNumber];
            [damageNumber
showWithValue:actionSprite.attackDamage
fromOrigin:_hero.position];

            //add these
            HitEffect *hitEffect = [self getHitEffect];
            [_actors reorderChild:hitEffect
z:MAX(actionSprite.zOrder, _hero.zOrder) + 1];
            [hitEffect showEffectAtPosition:attackPosition];
        }
    }
}

return didHit;
}

```

Once again, only add the lines after the comments – the whole rewritten method is provided for clarity.

This is very similar to retrieving and showing **DamageNumbers**, save for a few differences:

- You show a **HitEffect** only in the following scenarios:

- The hero attacks a robot using the first two punches of a combo, or a running attack. You don't show the effect for the hero's other attacks because these attacks already trigger a green beam effect from the hero's hands.
- A robot attacks the hero.
- A `HitEffect` needs to appear in the correct z-order. It would be weird if the hero punched a robot behind another robot, and the explosion came out in front of the second robot. You get the higher z-order between the hero and the robot, and make the `HitEffect` appear one z-order higher.
- A `HitEffect` appears at the center of the relevant attack circle.

Build and run, and cause mayhem. Boom, boom. POW!



## The robot ranks

There are four kinds of robots in the game – the bronze bot, silver bot, gold bot, and the randomly colored psychedelic bot. Each looks different onscreen according to their respective coloring. But at the moment, that's the only difference between them.

It would make for more exciting gameplay if each robot also had different attributes – and if some were stronger than others.

One way to go about it – and this is the object-oriented way – is to make subclasses of the `Robot` class and change the attributes from there.

For your current `Robot` implementation, though, there's an easier way. When you change the `colorSet` of a robot, you're already halfway to modifying its properties. You can simply add more value updates there and it should work perfectly.

Open `Robot.m` and do the following to `setColorSet::`:

```
//add inside curly braces of if (colorSet == kColorLess)
self.maxHitPoints = 50.0;
self.attackDamage = 2;

//add inside curly braces of if (colorSet == kColorCopper)
self.maxHitPoints = 100.0;
self.attackDamage = 4;

//add inside curly braces of if (colorSet == kColorSilver)
self.maxHitPoints = 125.0;
self.attackDamage = 5;

//add inside curly braces of if (colorSet == kColorGold)
self.maxHitPoints = 150.0;
self.attackDamage = 6;

//add inside curly braces of if (colorSet == kColorRandom)
self.maxHitPoints = random_range(100, 250);
self.attackDamage = random_range(4, 10);

//add before the last curly brace
self.hitPoints = self.maxHitPoints;
```

As the color changes from bronze to silver and then to gold, the `hitPoints` and `attackDamage` of the robot increase. When the color is random (the psychedelic bot), `hitPoints` and `attackDamage` are also random.

That's it – quick and easy. ☺

Build and run, and try to defeat the stronger robots.



# The big boss man

Now the game has some real complexity – the player meets a variety of enemies with different strength levels, and has a choice of attacks to deploy against them, guided by a full-featured HUD.

And yet, the ending of the game is anti-climactic. There's no antagonist for your protagonist to defeat. No boss villain to your hero.

In this section, you're going to add the final boss fight. This boss will be a worthy adversary to the hero – even matching the hero's fancy hairdo with his own!



## A class of his own

First up, create the **Boss** class. This should be a piece of cake if you still remember how you created the **Hero** and **Robot** classes.

Select the **Characters** group, go to **File\New\File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **ActionSprite** for Subclass of, click **Next** and name the new file **Boss**.

Open **Boss.h** and add this to the top:

```
#import "ActionSprite.h"
```

Switch to **Boss.m** and add these methods:

```
-(id)init
{
    if ((self = [super
initWithSpriteFrameName:@"boss_idle_00.png"]))
    {
        self.shadow = [CCSprite
spriteWithSpriteFrameName:@"shadow_character.png"];
        self.shadow.opacity = 190;
```

```
//idle animation
CCAnimation *idleAnimation = [self
animationWithPrefix:@"boss_idle" startFrameIdx:0 frameCount:5
delay:1.0/10.0];
    self.idleAction = [CCRepeatForever
actionWithAction:[CCAnimate actionWithAnimation:idleAnimation]];

//attack animation
CCAnimation *attackAnimation = [self
animationWithPrefix:@"boss_attack" startFrameIdx:0 frameCount:5
delay:1.0/8.0];

    self.attackAction = [CCSequence actions:[CCAnimate
actionWithAnimation:attackAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];

//walk animation
CCAnimation *walkAnimation = [self
animationWithPrefix:@"boss_walk" startFrameIdx:0 frameCount:6
delay:1.0/8.0];

    self.walkAction = [CCRepeatForever
actionWithAction:[CCAnimate actionWithAnimation:walkAnimation]];

//hurt animation
CCAnimation *hurtAnimation = [self
animationWithPrefix:@"boss_hurt" startFrameIdx:0 frameCount:3
delay:1.0/12.0];

    self.hurtAction = [CCSequence actions:[CCAnimate
actionWithAnimation:hurtAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];

//knocked out animation
CCAnimation *knockedOutAnimation = [self
animationWithPrefix:@"boss_knockout" startFrameIdx:0 frameCount:4
delay:1.0/12.0];

    self.knockedOutAction = [CCAnimate
actionWithAnimation:knockedOutAnimation];

//die action
    self.dieAction = [CCBlink actionWithDuration:2.0
blinks:10.0];
```

```
//recover animation
CCAnimation *recoverAnimation = [self
animationWithPrefix:@"boss_getup" startFrameIdx:0 frameCount:6
delay:1.0/12.0];

    self.recoverAction = [CCSequence actions:[CCAnimate
actionWithAnimation:recoverAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];

    self.walkSpeed = 60 * kPointFactor;
    self.runSpeed = 120 * kPointFactor;
    self.directionX = 1.0;
    self.centerToBottom = 39.0 * kPointFactor;
    self.centerToSides = 42.0 * kPointFactor;

    self.detectionRadius = 90.0 * kPointFactor;

    self.contactPointCount = 4;
    self.contactPoints = malloc(sizeof(ContactPoint) *
self.contactPointCount);
    self.attackPointCount = 1;
    self.attackPoints = malloc(sizeof(ContactPoint) *
self.attackPointCount);
    [self modifyAttackPointAtIndex:0 offset:ccp(65.0, 42.0)
radius:23.7];

    self.maxHitPoints = 500.0;
    self.hitPoints = self.maxHitPoints;
    self.attackDamage = 15.0;
    self.attackForce = 2.0 * kPointFactor;
}
return self;
}

-(void)setContactPointsForAction:(ActionState)actionState
{
    if (actionState == kActionStateIdle)
    {
        [self modifyContactPointAtIndex:0 offset:ccp(7.0, 36.0)
radius:23.0];
        [self modifyContactPointAtIndex:1 offset:ccp(-11.0, 17.0)
radius:23.5];
        [self modifyContactPointAtIndex:2 offset:ccp(-2.0, -20.0)
radius:23.0];
    }
}
```

```
[self modifyContactPointAtIndex:3 offset:ccp(24.0, 9.0)
radius:18.0];
}
else if (actionState == kActionStateWalk)
{
    [self modifyContactPointAtIndex:0 offset:ccp(6.0, 41.0)
radius:22.0];
    [self modifyContactPointAtIndex:1 offset:ccp(-5.0, 16.0)
radius:26.0];
    [self modifyContactPointAtIndex:2 offset:ccp(1.0, -11.0)
radius:17.0];
    [self modifyContactPointAtIndex:3 offset:ccp(-13.0, -25.0)
radius:10.0];
}
else if (actionState == kActionStateAttack)
{
    [self modifyContactPointAtIndex:0 offset:ccp(20.0, 38.0)
radius:22.0];
    [self modifyContactPointAtIndex:1 offset:ccp(-8.0, 7.0)
radius:27.3];
    [self modifyContactPointAtIndex:2 offset:ccp(49.0, 18.0)
radius:19.0];
    [self modifyContactPointAtIndex:3 offset:ccp(12.0, -8.0)
radius:31.0];
    [self modifyAttackPointAtIndex:0 offset:ccp(65.0, 42.0)
radius:23.7];
}
}

-(void)setDisplayFrame:(CCSpriteFrame *)newFrame
{
    [super setDisplayFrame:newFrame];

    CCSpriteFrame *attackFrame = [[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"boss_attack_01.png"];
    CCSpriteFrame *attackFrame2 = [[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"boss_attack_02.png"];

    if (newFrame == attackFrame || newFrame == attackFrame2)
    {
        [self.delegate actionSpriteDidAttack:self];
    }
}
```

This is a lot of code, but it's all stuff you've done before – so I'll just discuss the highlights.

As with the hero, you create the boss using the corresponding sprite frames and animations in the sprite sheet and set his attributes. The boss enemy moves slower than the others, but hits harder and endures more damage.

`setContactPointsForAction:` adjusts the contact circles and attack circles to their proper positions for each relevant action, and `setDisplayFrame:` informs the `delegate` to check collisions for the two specific attack frames.

One thing that will make the boss different, and harder, than an enemy robot is his toughness. When regular attacks hit the boss, he shouldn't flinch except when he's already fairly weak.

Still in `Boss.m`, add this method:

```
- (void)hurtWithDamage:(float)damage force:(float)force
direction:(CGPoint)direction
{
    if (self.actionState > kActionStateNone && self.actionState <
kActionStateKnockedOut)
    {
        float ratio = self.hitPoints / self.maxHitPoints;

        if (ratio <= 0.1)
        {
            [self stopAllActions];
            [self runAction:self.hurtAction];
            self.actionState = kActionStateHurt;
        }

        self.hitPoints -= damage;
        self.desiredPosition = ccp(self.position.x + direction.x *
force, self.position.y);

        if (self.hitPoints <= 0)
        {
            [self knockoutWithDamage:0 direction:direction];
        }
    }
}
```

This rewrites `hurtWithDamage:` for the boss. He will only show his hurt animation and switch to a hurt state if his `hitPoints` are only 10% of his `maxHitPoints`. Otherwise, his life will decrease, but he will still continue whatever action he was already doing. It will feel like fighting the Hulk!

## Where bosses lurk

With the `Boss` class completed, you can now add a boss enemy to the game. Since you implemented multiple levels using `Levels.plist`, you need to add information for the boss enemy there, too.

But before that, go to `Defines.h` and add this definition:

```
typedef enum _BossType
{
    kBossNone = 0,
    kBossMohawk
} BossType;
```

Then open `Levels.plist` and for each level listed, add a new row right after the `TileMap` row, change the **Key** to **BossType**, and set the **Type** to **Number**.

Key	Type	Value
▼ Root	Array	(3 items)
▼ Item 0	Dictionary	(3 items)
TileMap	String	map_level1.tmx
BossType	Number	0
► BattleEvents	Array	(6 items)
▼ Item 1	Dictionary	(3 items)
TileMap	String	map_level1.tmx
BossType	Number	0
► BattleEvents	Array	(8 items)
▼ Item 2	Dictionary	(3 items)
TileMap	String	map_level1.tmx
BossType	Number	0
► BattleEvents	Array	(8 items)

On the last level, in this case **Item 2**, change the value of **BossType** to **1**.

▼ Item 2	Dictionary	(3 items)
TileMap	String	map_level1.tmx
BossType	Number	1
► BattleEvents	Array	(8 items)

The value of the `BossType` row in `Levels.plist` corresponds to the `BossType` enumeration you created. If the value is 0, or `kBossNone`, then it means that that level doesn't have a boss. If the value is other than 0, then that level will have a boss of that type.

This means the first two levels won't have a boss – unless you decide to add them on your own – while the third level will have the Mohawk Boss you just created.

Now you need to specify the battle event in which the boss enemy will spawn. You'll add the boss in the natural place – the last battle event of the last level of the game.

Expand **Item 2** under **Root**, then expand **BattleEvents**, the last item of **BattleEvents**, and the **Enemies** of that item, and finally the first item under **Enemies**, like so:

Key	Type	Value
▼ Root	Array	(3 items)
▶ Item 0	Dictionary	(3 items)
▶ Item 1	Dictionary	(3 items)
▼ Item 2	Dictionary	(3 items)
TileMap	String	map_level1.tmx
BossType	Number	1
▼ BattleEvents	Array	(8 items)
▶ Item 0	Dictionary	(2 items)
▶ Item 1	Dictionary	(2 items)
▶ Item 2	Dictionary	(2 items)
▶ Item 3	Dictionary	(2 items)
▶ Item 4	Dictionary	(2 items)
▶ Item 5	Dictionary	(2 items)
▶ Item 6	Dictionary	(2 items)
▼ Item 7	Dictionary	(2 items)
Column	Number	95
▼ Enemies	Array	(6 items)
▼ Item 0	Dictionary	(4 items)
Type	Number	0
Color	Number	4
Row	Number	2
Offset	Number	-1

Change the value of **Type** and **Offset** to **1**. Select the **Color** row, and click on the minus sign to delete that row. You should end up with this:

▼ Item 0	Dictionary	(3 items)
Type	Number	1
Row	Number	2
Offset	Number	1

Remember that the value of the Type row corresponds to the `enemyType` enumeration defined in `Defines.h`. By changing it to 1, you've changed the

EnemyType from `kEnemyRobot` to `kEnemyBoss`. You removed the Color row because the `Boss` class doesn't support color tinting like the `Robot` class does.

**Note:** If you haven't been using the same Levels.plist as the one mentioned above, and would like your Levels.plist to be in sync with this chapter, you can copy the Levels.plist from **Versions\Chapter7** in your Starter Kit to your project directory and replace your own version. This copy already has the above changes implemented.

Next up: spawning like a boss.

Go to **GameLayer.h** and do the following:

```
//add to top of file
#import "Boss.h"

//add this property
@property(nonatomic, strong) Boss *boss;
```

Switch to **GameLayer.m** and do the following:

```
//add this to loadLevel:, after _currentLevel = level
NSInteger boss = [[levelData objectForKey:@"BossType"] intValue];
[self initBossWithType:boss];

//add this method
-(void)initBossWithType:(BossType)type
{
    if (type == kBossMohawk)
    {
        self.boss = [Boss node];
        _boss.delegate = self;
        [_actors addChild:_boss.shadow];
        _boss.shadow.scale *= kScaleFactor;
        [_actors addChild:_boss];
        _boss.scale *= kScaleFactor;
        _boss.visible = NO;
        _boss.position = OFFSCREEN;
        _boss.groundPosition = OFFSCREEN;
        _boss.desiredPosition = OFFSCREEN;
    }
}
```

In `loadLevel:`, you take the value of `BossType` for the current level and pass it to `initBossWithType:..`. If the `BossType` is `kBossMohawk`, then a new boss is created.

If you want to create different subclasses of `Boss` in the future, just add new `BossType` definitions and then, in this method, add additional boss creation code to handle the different types.

**Note:** If you want to see how the attack and contact circles of the boss enemy are positioned, you can turn on debug drawing. Just set the value of `DRAW_DEBUG_SHAPES` in `Defines.h` to 1 and include the boss object in `GameLayer's draw` method.

For reference, the finished project in **Versions\Chapter7** of your Starter Kit already has the boss, along with all the other objects you will be adding later, included in debug drawing.

## Bringing the boss to life

To finish adding support for the boss in to the game, you still need to do the following:

1. Spawn the boss enemy in `spawnEnemies::`.
2. Update the boss logic in `update::`.
3. Update the boss's position in `updatePositions`.
4. Dynamically change the z-order of the boss in `reorderActors`.
5. Implement collision handling in `actionSpriteDidAttack::`.
6. Give the boss some brains in `initBrains`.

Take it one step at a time! You'll do everything in `GameLayer.m`.

First, do this in `spawnEnemies:fromOrigin::`:

```
//add this as the else if statement of if (type == kEnemyRobot)
else if (type == kEnemyBoss)
{
    _boss.groundPosition = ccp(origin + (offset * (CENTER.x +
_boss.centerToSides)), _boss.centerToBottom +
_tileMap.tileSize.height * row * kPointFactor);
    _boss.position = _boss.groundPosition;
    _boss.desiredPosition = _boss.groundPosition;
    [_boss idle];
    _boss.visible = YES;
}
```

When the type is `kEnemyBoss`, the method spawns the boss in the proper location based on `Levels.plist`.

Next, do the following:

```

//add this to update:, right after [_hero update:delta]
if (_boss)
{
    [_boss update:delta];
}

//add this to updatePositions, before the last curly brace
if (_boss && _boss.actionState > kActionStateNone)
{
    posY = MIN(floorHeight + (_boss.centerToBottom -
_boss.feetCollisionRect.size.height), MAX(_boss.centerToBottom,
_boss.desiredPosition.y));
    _boss.groundPosition = ccp(_boss.desiredPosition.x, posY);
    _boss.position = ccp(_boss.groundPosition.x,
_boss.groundPosition.y + _boss.jumpHeight);
}

//add this to reorderActors, before the last curly brace
if (_boss)
{
    spriteZ = [self getZFromYPosition:_boss.groundPosition.y];
    [_actors reorderChild:_boss.shadow z:spriteZ];
    [_actors reorderChild:_boss z:spriteZ];
}

```

These are steps 2, 3, and 4 – a triple smash! Everything you did to update the hero's position, you just did for the boss's position, but only if the boss exists. There's nothing else new here that you haven't encountered before.

Next, to implement collision handling, modify `actionSpriteDidAttack:` as follows (existing code is highlighted):

```

-(BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    BOOL didHit = NO;
    if (actionSprite == _hero)
    {
        CGPoint attackPosition;
        Robot *robot;
        CCARRAY_FOREACH(_robots, robot)
        {
            if (robot.actionState < kActionStateKnockedOut &&
robot.actionState != kActionStateNone)
            {
                if ([self collisionBetweenAttacker:_hero
andTarget:robot atPosition:&attackPosition])

```

```
        {
            BOOL showEffect = YES;

            DamageNumber *damageNumber = [self
getDamageNumber];

                if (_hero.actionState ==
kActionStateJumpAttack)
                {
                    [robot
knockoutWithDamage:_hero.jumpAttackDamage
direction:ccp(_hero.directionX, 0)];
                    [damageNumber
showWithValue:_hero.jumpAttackDamage fromOrigin:robot.position];
                    showEffect = NO;
                }
                else if (_hero.actionState ==
kActionStateRunAttack)
                {
                    [robot
knockoutWithDamage:_hero.runAttackDamage
direction:ccp(_hero.directionX, 0)];
                    [damageNumber
showWithValue:_hero.runAttackDamage fromOrigin:robot.position];
                }
                else if (_hero.actionState ==
kActionStateAttackThree)
                {
                    [robot
knockoutWithDamage:_hero.attackThreeDamage
direction:ccp(_hero.directionX, 0)];
                    [damageNumber
showWithValue:_hero.attackThreeDamage fromOrigin:robot.position];
                    showEffect = NO;
                }
                else if (_hero.actionState ==
kActionStateAttackTwo)
                {
                    [robot
hurtWithDamage:_hero.attackTwoDamage force:_hero.attackForce
direction:ccp(_hero.directionX, 0.0)];
                    [damageNumber
showWithValue:_hero.attackTwoDamage fromOrigin:robot.position];
                }
            else
```

```
        {
            [_robot hurtWithDamage:_hero.attackDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0.0)];
            [damageNumber
showWithValue:_hero.attackDamage fromOrigin:_robot.position];
        }
        didHit = YES;

        if (showEffect)
        {
            HitEffect *hitEffect = [self
getHitEffect];
            [_actors reorderChild:hitEffect
z:MAX(robot.zOrder, _hero.zOrder) + 1];
            [hitEffect
showEffectAtPosition:attackPosition];
        }
    }
}

//add this new if block
if (_boss && _boss.actionState < kActionStateKnockedOut &&
_boss.actionState != kActionStateNone)
{
    if ([self collisionBetweenAttacker:_hero
andTarget:_boss atPosition:&attackPosition])
    {
        BOOL showEffect = YES;
        DamageNumber *damageNumber = [self
getDamageNumber];

        if (_hero.actionState == kActionStateJumpAttack)
        {
            [_boss hurtWithDamage:_hero.jumpAttackDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0)];
            [damageNumber
showWithValue:_hero.jumpAttackDamage fromOrigin:_boss.position];
            showEffect = NO;
        }
        else if (_hero.actionState ==
kActionStateRunAttack)
        {
            [_boss hurtWithDamage:_hero.runAttackDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0)];
        }
    }
}
```

```
[damageNumber
showWithValue:_hero.runAttackDamage fromOrigin:_boss.position];
}
else if (_hero.actionState ==
kActionStateAttackThree)
{
[_boss hurtWithDamage:_hero.attackThreeDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0)];
[damageNumber
showWithValue:_hero.attackThreeDamage fromOrigin:_boss.position];
showEffect = NO;
}
else if (_hero.actionState ==
kActionStateAttackTwo)
{
[_boss hurtWithDamage:_hero.attackTwoDamage
force:_hero.attackForce/2 direction:ccp(_hero.directionX, 0.0)];
[damageNumber
showWithValue:_hero.attackTwoDamage fromOrigin:_boss.position];
}
else
{
[_boss hurtWithDamage:_hero.attackDamage
force:0 direction:ccp(_hero.directionX, 0.0)];
[damageNumber showWithValue:_hero.attackDamage
fromOrigin:_boss.position];
}
didHit = YES;

if (showEffect)
{
HitEffect *hitEffect = [self getHitEffect];
[_actors reorderChild:hitEffect
z:MAX(_boss.zOrder, _hero.zOrder) + 1];
[hitEffect
showEffectAtPosition:attackPosition];
}
}

return didHit;
}
else if (actionSprite == _boss) //add this else if statement
{
```

```
        if (_hero.actionState < kActionStateKnockedOut &&
    _hero.actionState != kActionStateNone)
    {
        CGPoint attackPosition;
        if ([self collisionBetweenAttacker:_boss
andTarget:_hero atPosition:&attackPosition])
        {
            [_hero knockoutWithDamage:_boss.attackDamage
direction:ccp(actionSprite.directionX, 0.0)];
            [_hud setHitPoints:_hero.hitPoints
fromMaxHP:_hero.maxHitPoints];
            didHit = YES;

            DamageNumber *damageNumber = [self
getDamageNumber];
            [damageNumber showWithValue:_boss.attackDamage
fromOrigin:_hero.position];

            HitEffect *hitEffect = [self getHitEffect];
            [_actors reorderChild:hitEffect
z:MAX(_boss.zOrder, _hero.zOrder) + 1];
            [hitEffect showEffectAtPosition:attackPosition];
        }
    }
}
else
{
    if (_hero.actionState < kActionStateKnockedOut &&
    _hero.actionState != kActionStateNone)
    {
        CGPoint attackPosition;
        if ([self collisionBetweenAttacker:actionSprite
andTarget:_hero atPosition:&attackPosition])
        {
            [_hero hurtWithDamage:actionSprite.attackDamage
force:actionSprite.attackForce
direction:ccp(actionSprite.directionX, 0.0)];
            [_hud setHitPoints:_hero.hitPoints
fromMaxHP:_hero.maxHitPoints];
            didHit = YES;

            DamageNumber *damageNumber = [self
getDamageNumber];
    }
}
```

```

        [damageNumber
showWithValue:actionSprite.attackDamage
fromOrigin:_hero.position];

        HitEffect *hitEffect = [self getHitEffect];
        [_actors reorderChild:hitEffect
z:MAX(actionSprite.zOrder, _hero.zOrder) + 1];
        [hitEffect showEffectAtPosition:attackPosition];
    }
}

return didHit;
}

```

The code above includes the whole method, but only the parts with comments are new – specifically the part that checks collisions between the hero and the boss.

When the hero attacks the boss, the boss gets hurt with the appropriate damage. The boss's advantage is that he never gets knocked out from a direct attack – he only gets hurt. It's up to the boss's `hurtWithDamage:` method to decide if he gets knocked out or not.

The knockback force also varies per attack type. The stronger attacks to the boss generate a normal `attackForce`. The second punch generates half the normal force, while the regular jab has no force. This way, the boss won't get pushed around much – which is fitting for his size and strength.

On the other hand, when the boss punches the hero, the hero instantly gets knocked out. You can already imagine how hard it will be to beat this guy.

## CHALLENGE ACCEPTED



Now that the boss has brawns, it's time to add some brains.

Replace the contents of `initBrains` (still in `GameLayer.m`, of course):

```

-(void)initBrains
{

```

```
self.brains = [[CCArray alloc] initWithCapacity:_robots.count
+ 1];

    ArtificialIntelligence *brain = [ArtificialIntelligence
aiWithControlledSprite:_boss targetSprite:_hero];
[_brains addObject:brain];

Robot *robot;

CCARRAY_FOREACH(_robots, robot)
{
    brain = [ArtificialIntelligence
aiWithControlledSprite:robot targetSprite:_hero];
[_brains addObject:brain];
}
}
```

The brains array now has one more brain in it – the boss's brain. You set the boss as the `controlledSprite` and the hero as the `targetSprite` of this brain. The rest is the same as before.

**Optional:** To differentiate the boss further, you could make a subclass of `ArtificialIntelligence` and make this new AI more aggressive. The attack and chase decisions should have more weight than the idle and move decisions.

Okay, are you ready to fight this giant? To see the boss right away, you can modify your `Levels.plist` to put the boss at the beginning of the first level if you'd like.

Build and run. And good luck!



Wow, talk about a challenge! The game instantly became too difficult to finish!

I don't know about you, but I even had a tough time beating the gold robots to reach the boss, and having seen the screenshot above, you probably know what happened next.

One way to make the game a little easier, and yet keep it challenging, is to give the hero a way to match the strength of his enemies temporarily. In some Beat 'Em Ups, you can pick up various items and use them as weapons.

If that thought gets you excited, and you can't wait to deal one back to the boss man, you'll love the next and last chapter. In it, you'll equip the hero with what he needs to succeed and add some finishing touches to the game.

**Challenge:** In this chapter, you used a `CCLabelBMFont` to display the "Go" text that appears to encourage the user to keep moving. See if you can modify the code to replace the "Go" text with an image of your own creation – maybe an arrow pointing forward for example.

# 8

## Chapter 8: Final Encounter

Welcome to the last chapter of the Beat 'Em Up Starter Kit! Everything you did in the previous chapters has been leading up to this point – the final encounter!

In this chapter, you will be adding the final pieces of content to the game –the hero's gauntlet weapon and a destructible trashcan. Then you'll polish things up with end game scenarios, sound effects and music, all contributing to what makes a game production-ready.

As always, you're going to pick up from where you left off in the last chapter. If you skipped it or missed some parts, you can still continue on with the Chapter 7 project that is included in the resources.

Let's proceed – your game will be finished in no time!

### Power gauntlets!

Right now the robot forces are on the rise. They've graduated into ranks and chosen a powerful mohawked leader. It's tough for a hero to survive out there – he needs help.

To restore the hero's fighting chance, you'll grant him a tailor-made weapon – the power gauntlets.



In creating the gaunlets, you'll develop a weapons template that you can use to add other sorts of weapons to the game. Fancy a straight pipe or a blade? All you'll need are the sprites!

## The weapon template

A weapon works similar to `ActionSprite`, but it requires some special handling to handle equipping the weapon.

Let's try this out. First go to **Defines.h** and add this definition:

```
typedef enum _WeaponState
{
    kWeaponStateNone = 0,
    kWeaponStateEquipped,
    kWeaponStateUnequipped,
    kWeaponStateDestroyed
} WeaponState;
```

These are the different states a weapon can have. The names speak for themselves.

Select the **PompaDroid** group in Xcode, go to **File\New\Group** and name the new group **Weapons**.

Next select the **Weapons** group and go to **File\New\File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **CCSprite** for Subclass of, click **Next** and name the new file **Weapon**.

Open **Weapon.h** and replace its contents with the following:

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

@class Weapon;

@protocol WeaponDelegate <NSObject>

-(void)weaponDidReachLimit:(Weapon *)weapon;

@end

@interface Weapon : CCSprite {
}

//delegate
@property(nonatomic, weak)id <WeaponDelegate> delegate;
```

```

@property(nonatomic, strong)CCSprite *shadow;
@property(nonatomic, strong)AnimationMember *attack;
@property(nonatomic, strong)AnimationMember *attackTwo;
@property(nonatomic, strong)AnimationMember *attackThree;
@property(nonatomic, strong)AnimationMember *idle;
@property(nonatomic, strong)AnimationMember *walk;
@property(nonatomic, strong)id droppedAction;
@property(nonatomic, strong)id destroyedAction;
@property(nonatomic, assign)float damageBonus;
@property(nonatomic, assign)int limit;
@property(nonatomic, assign)float jumpVelocity;
@property(nonatomic, assign)CGPoint velocity;
@property(nonatomic, assign)float jumpHeight;
@property(nonatomic, assign)CGPoint groundPosition;
@property(nonatomic, assign)WeaponState weaponState;
@property(nonatomic, assign)float centerToBottom;
@property(nonatomic, assign)float detectionRadius;

-(CCAnimation *)animationWithPrefix:(NSString *)prefix
startFrameIdx:(NSUInteger)startFrameIdx
frameCount:(NSUInteger)frameCount delay:(float)delay;
-(AnimationMember *)animationMemberWithPrefix:(NSString *)prefix
startFrameIdx:(NSUInteger)startFrameIdx
frameCount:(NSUInteger)frameCount delay:(float)delay
target:(id)target;
-(void)used;
-(void)pickedUp;
-(void)droppedFrom:(float)height to:(CGPoint)destination;
-(void)reset;

@end

```

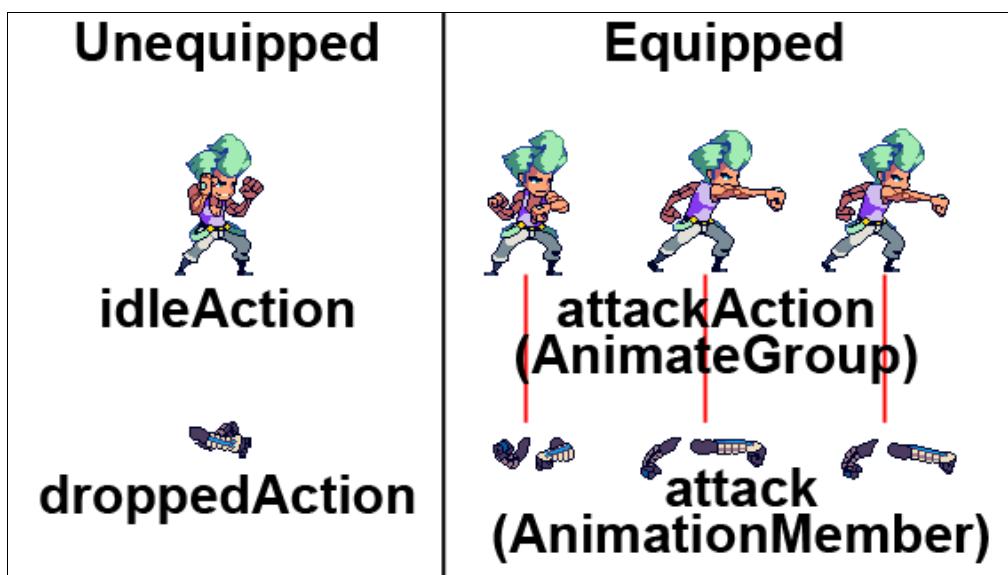
This is the base structure of a weapon. First consider the similarities to **ActionSprite**:

- Weapon is also a subclass of **ccsprite** because it must display an image.
- It has a delegate – the **GameLayer**.
- Both **ActionSprite** and Weapon have shadow sprites.
- Both have actions and animations.
- Both have movement values – **jumpVelocity**, **velocity**, **jumpHeight** and **groundPosition**.
- Both have measurements – **centerToBottom**, **detectionRadius**.

Now consider the differences:

- Instead of `actionState`, Weapon has a `weaponState`.
- Instead of `attackDamage`, Weapon has a `damageBonus`. This will be the damage boost received by the hero when he is equipped with the weapon.
- Weapon has a `limit`. A weapon can only be used to hit the enemy a number of times before it breaks.
- Weapon only has two actions that it controls:
  - `droppedAction`: Animation for when an equipped weapon is removed by the hero.
  - `destroyedAction`: What happens when the weapon has reached its `limit`.
- Weapon has `AnimationMembers`, or animations that it doesn't control. You already encountered `AnimationMembers` in the `Robot` class, and you know that it needs a parent `AnimateGroup` action to follow.

When unequipped, the weapon will decide its own animation, but when it is equipped, the hero's actions will control the weapon's animations:



Switch to `Weapon.m` and add these methods:

```
- (CCAnimation *)animationWithPrefix:(NSString *)prefix
startFrameIdx:(NSUInteger)startFrameIdx
frameCount:(NSUInteger)frameCount delay:(float)delay
{
    int idxCount = frameCount + startFrameIdx;
    CCArray *frames = [CCArray arrayWithCapacity:frameCount];
    int i;
    CCSpriteFrame *frame;
    for (i = startFrameIdx; i < idxCount; i++)
```

```

    {
        frame = [[CCSpriteFrameCache sharedSpriteFrameCache]
spriteFrameByName:[NSString stringWithFormat:@"%@_%02d.png",
prefix, i]];
        [frames addObject:frame];
    }

    return [CCAnimation animationWithSpriteFrames:[frames
getNSArray] delay:delay];
}

-(AnimationMember *)animationMemberWithPrefix:(NSString *)prefix
startFrameIdx:(NSUInteger)startFrameIdx
frameCount:(NSUInteger)frameCount delay:(float)delay
target:(id)target
{
    CCAnimation *animation = [self animationWithPrefix:prefix
startFrameIdx:startFrameIdx frameCount:frameCount delay:delay];
    return [AnimationMember memberWithAnimation:animation
target:target];
}

```

These are helper methods for creating `AnimationMembers`. It's similar to the helper methods you wrote for `ActionSprite` before.

Still in `Weapon.m`, add these methods:

```

-(void)reset
{
    self.visible = NO;
    _shadow.visible = NO;
    _weaponState = kWeaponStateNone;
    self.velocity = CGPointMakeZero;
    self.jumpVelocity = 0;
}

-(void)setVisible:(BOOL)visible
{
    [super setVisible:visible];
    self.shadow.visible = visible;
}

-(void)setGroundPosition:(CGPoint)groundPosition
{
    _groundPosition = groundPosition;
}

```

```
    _shadow.position = ccp(groundPosition.x, groundPosition.y -  
    _centerToBottom);  
}
```

`reset` makes the weapon invisible and changes all relevant values back to their default values. `setVisible:` and `setGroundPosition:` are overridden property methods so that the `shadow` always follows the weapon.

Here is the last batch of methods to add to **Weapon.m**:

```
-(void)used  
{  
    self.limit--;  
  
    if (_limit <= 0)  
    {  
        [_delegate weaponDidReachLimit:self];  
        self.weaponState = kWeaponStateDestroyed;  
        [self runAction:_destroyedAction];  
    }  
}  
  
-(void)pickedUp  
{  
    self.weaponState = kWeaponStateEquipped;  
    self.shadow.visible = NO;  
}  
  
-(void)droppedFrom:(float)height to:(CGPoint)destination  
{  
    _jumpVelocity = kJumpCutoff;  
    _jumpHeight = height;  
    self.groundPosition = destination;  
    self.weaponState = kWeaponStateUnequipped;  
    self.shadow.visible = YES;  
    [self runAction:_droppedAction];  
}  
  
-(void)update:(ccTime)delta  
{  
    if (_weaponState > kWeaponStateEquipped)  
    {  
        self.groundPosition = ccpAdd(self.groundPosition,  
        ccpMult(_velocity, delta));  
        _jumpVelocity -= kGravity * delta;  
        _jumpHeight += _jumpVelocity * delta;  
    }  
}
```

```

        if (_jumpHeight < 0)
    {
        _velocity = CGPointMakeZero;
        _jumpVelocity = 0;
        _jumpHeight = 0;
    }
}
}

```

Here are quick explanations of each of the above methods:

- **used**: Whenever the weapon hits an enemy, the weapon's `limit` value decreases. When `limit` reaches zero, the weapon performs `destroyedAction`, changes its state to `kWeaponStateDestroyed` and informs the delegate that it has been used up.
- **pickedUp**: This simply changes the state to `kWeaponStateEquipped` and hides the shadow.
- **droppedFrom:to**: When the hero voluntarily drops the weapon, it jumps from his hands. You make the weapon jump from a specified starting height by setting the minimum `jumpVelocity` and setting its `groundPosition` to where it should land. You execute `droppedAction` so that the weapon changes its image back to the default image, and the `shadow` becomes visible again.
- **update**: When the weapon is unequipped, it will fall back to the ground. The velocity, jump and position calculations are the same as `ActionSprite`.

There are still a lot of things to be done. But before anything else, you should create your first weapon.

## The gauntlets of power

Select the **Weapons** group, go to **File\New\File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **Weapon** for Subclass of, click **Next** and name the new file **Gauntlets**.

Open **Gauntlets.h** and add this to the top of the file:

```
#import "Weapon.h"
```

Switch to **Gauntlets.m** and add these methods:

```

-(id)init
{
    if ((self = [super
initWithSpriteFrameName:@"weapon_unequipped.png"]))
    {

```

```
        self.attack = [self
animationMemberWithPrefix:@"weapon_attack_00" startFrameIdx:0
frameCount:3 delay:1.0/15.0 target:self];
        self.attackTwo = [self
animationMemberWithPrefix:@"weapon_attack_01" startFrameIdx:0
frameCount:3 delay:1.0/12.0 target:self];
        self.attackThree = [self
animationMemberWithPrefix:@"weapon_attack_02" startFrameIdx:0
frameCount:5 delay:1.0/10.0 target:self];
        self.idle = [self animationMemberWithPrefix:@"weapon_idle"
startFrameIdx:0 frameCount:6 delay:1.0/12.0 target:self];
        self.walk = [self animationMemberWithPrefix:@"weapon_walk"
startFrameIdx:0 frameCount:8 delay:1.0/12.0 target:self];

        CCSpriteFrame *dropFrame = [[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"weapon_unequipped.png"];
        CCAnimation *dropAnimation = [CCAnimation
animationWithSpriteFrames:@[dropFrame] delay:1/12.0];
        self.droppedAction = [CCAnimate
actionWithAnimation:dropAnimation];
        self.destroyedAction = [CCSequence actions:[CCBlink
actionWithDuration:2.0 blinks:5], [CCCallFunc
actionWithTarget:self selector:@selector(reset)],nil];
        self.damageBonus = 20.0;
        self.centerToBottom = 5.0 * kPointFactor;

        self.shadow = [CCSprite
spriteWithSpriteFrameName:@"shadow_weapon.png"];
        self.shadow.opacity = 190;
        self.detectionRadius = 10.0 * kPointFactor;

        [self reset];
    }
    return self;
}

-(void)reset
{
    [super reset];
    self.limit = 20;
}

-(void)cleanup
{
```

```

    self.attack = nil;
    self.attackTwo = nil;
    self.attackThree = nil;
    self.idle = nil;
    self.walk = nil;
    self.droppedAction = nil;
    self.destroyedAction = nil;
    [super cleanup];
}

```

You create the gauntlets with an unequipped image. Then you create **AnimationMembers** for five actions – **attack**, **attackTwo**, **attackThree**, **idle** and **walk**.

The gauntlets will only work for these five actions. When the hero is equipped with the gauntlets, he won't be able to run, jump or get hurt without dropping the gauntlets first.

Next you create the **droppedAction**, which just changes the image back to the unequipped image. The **destroyedAction** makes the gauntlets blink before resetting them.

You create the shadow as before and set measurement values. You'll use these measurement values to check if the hero is close enough to a gauntlet to pick it up.

The gauntlets give a +20 damage bonus and can be used 20 times. Last, you add **cleanup** to make sure that there are no retain cycles that can cause memory leaks.

Before going any further, give the gauntlets a test in their unequipped state.

Go to **GameLayer.h** and add this property:

```
@property(nonatomic, strong)CCArray *weapons;
```

Switch to **GameLayer.m** and do the following:

```

//add to top of file
#import "Gauntlets.h"

//add inside initWithLevel:, after [self initRobots]
[self initWeapons];

//add this method
-(void)initWeapons
{
    int i;
    self.weapons = [CCArray arrayWithCapacity:3];
    Weapon *weapon;
}

```

```
for (i = 0; i < 3; i++)
{
    weapon = [Gauntlets node];
    weapon.visible = YES;
    weapon.shadow.scale *= kScaleFactor;
    weapon.scale *= kScaleFactor;
    float maxX = _tileMap.mapSize.width *
_tileMap.tileSize.width * kPointFactor;
    float minY = weapon.centerToBottom;
    float maxY = 3 * _tileMap.tileSize.height * kPointFactor +
weapon.centerToBottom;
    weapon.weaponState = kWeaponStateUnequipped;
    weapon.groundPosition = ccp(frandom_range(0, maxX),
frandom_range(minY, maxY));
    [_actors addChild:weapon.shadow];
    [_actors addChild:weapon];
    [_weapons addObject:weapon];
}
}

//add this to update:, after [_hero update:delta]
Weapon *weapon;
CCARRAY_FOREACH(_weapons, weapon)
{
    [weapon update:delta];
}

//add this to updatePositions, before the last curly brace
Weapon *weapon;
CCARRAY_FOREACH(_weapons, weapon)
{
    if (weapon.weaponState > kWeaponStateEquipped)
    {
        weapon.position = ccp(weapon.groundPosition.x,
weapon.groundPosition.y + weapon.jumpHeight);
    }
}

//add this to reorderActors, before the last curly brace
Weapon *weapon;
CCARRAY_FOREACH(_weapons, weapon)
{
    if (weapon.weaponState != kWeaponStateEquipped)
    {
```

```
        spriteZ = [self
getZFromYPosition:weapon.groundPosition.y];
[_actors reorderChild:weapon.shadow z:spriteZ];
[_actors reorderChild:weapon z:spriteZ];
}
}
```

The above creates three gauntlets and positions them randomly across the map. When a weapon is unequipped, you update its position and z-order separately in `updatePositions` and `reorderActors`.

Build, run, and search for the gauntlets.



Found one? Great! You can't pick it up yet, though. For that to work, the hero needs to be fitted with weapon-wielding capabilities. But before you do that, notice that the z-ordering doesn't work quite as expected with the weapons on the ground.

When the hero is below the gauntlets, it may happen that the gauntlets are still drawn on top. This is because you've been using the `groundPosition` of each object to dictate its z-order, and the `groundPosition` is located at the center of the sprite.

This worked well up until now because you've been using similarly-sized sprites – the hero, the boss and the robots. All their `groundPositions` are approximately the same height from the ground. This time, though, the weapon's `groundPosition` is much closer to the ground.

What you really need to use is the weapon's position on the ground itself, not the center of the sprite. In other words, you need the weapon's position on the plane, or where it's actually sitting. Guess what? You already have access to this position – the shadow's position.

Still in **GameLayer.m**, do the following in **reorderActors**:

```
//replace this:  
NSInteger spriteZ = [self  
getZFromYPosition:_hero.groundPosition.y];  
//with this:  
NSInteger spriteZ = [self  
getZFromYPosition:_hero.shadow.position.y];  
  
//replace this:  
spriteZ = [self getZFromYPosition:robot.groundPosition.y];  
//with this:  
spriteZ = [self getZFromYPosition:robot.shadow.position.y];  
  
//replace this:  
spriteZ = [self getZFromYPosition:_boss.groundPosition.y];  
//with this:  
spriteZ = [self getZFromYPosition:_boss.shadow.position.y];  
  
//replace this:  
spriteZ = [self getZFromYPosition:weapon.groundPosition.y];  
//with this:  
spriteZ = [self getZFromYPosition:weapon.shadow.position.y];
```

Instead of using a sprite's **groundPosition** to get **spriteZ**, you now use its shadow's position.

That should fix the z-ordering issue. No need to build and run for now since it's a very small fix.

## Equipping the hero

I bet you're eager to put those gauntlets in the hero's hands so that you can crash them into the faces of some robots.

Open **Hero.h** and do the following:

```
//add to top of file  
#import "Weapon.h"  
#import "AnimateGroup.h"  
  
//add the protocol to the @interface statement  
@interface Hero : ActionSprite <WeaponDelegate> {  
  
//add these instance variables inside @interface  
AnimateGroup *_attackGroup;  
AnimateGroup *_attackTwoGroup;
```

```
AnimateGroup *_attackThreeGroup;
AnimateGroup *_idleGroup;
AnimateGroup *_walkGroup;

//add this property
@property(nonatomic, weak)Weapon *weapon;

//add these methods
-(void)dropWeapon;
-(BOOL)pickUpWeapon:(Weapon *)weapon;
```

This makes the hero a delegate of the `Weapon` class, meaning that the hero can now receive messages from weapon objects.

It then creates five `AnimateGroups` for the hero. These will replace the `CCAnimate` actions inside the attack, idle and walk actions. It also adds a weak reference to a weapon, and some methods to drop and pick up weapons.

Switch to `Hero.m` and do the following inside `init`:

```
//replace this line:
self.idleAction = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:idleAnimation]];
//with these lines:
_idleGroup = [AnimateGroup actionWithAnimation:idleAnimation
memberCount:1];
self.idleAction = [CCRepeatForever actionWithAction:_idleGroup];

//replace this line:
self.walkAction = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:walkAnimation]];
//with these lines:
_walkGroup = [AnimateGroup actionWithAnimation:walkAnimation
memberCount:1];
self.walkAction = [CCRepeatForever actionWithAction:_walkGroup];

//replace this line:
self.attackAction = [CCSequence actions:[CCAnimate
actionWithAnimation:attackAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];
//with these lines:
_attackGroup = [AnimateGroup actionWithAnimation:attackAnimation
memberCount:1];
self.attackAction = [CCSequence actions:_attackGroup, [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];
```

```
//replace this line:
self.attackTwoAction = [CCSequence actions:[CCAnimate
actionWithAnimation:attackTwoAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];
//with these lines:
_attackTwoGroup = [AnimateGroup
actionWithAnimation:attackTwoAnimation memberCount:1];
self.attackTwoAction = [CCSequence actions:_attackTwoGroup,
[CCCallFunc actionWithTarget:self selector:@selector(idle)], nil];

//replace this line:
self.attackThreeAction = [CCSequence actions:[CCAnimate
actionWithAnimation:attackThreeAnimation], [CCCallFunc
actionWithTarget:self selector:@selector(idle)], nil];
//with these lines:
_attackThreeGroup = [AnimateGroup
actionWithAnimation:attackThreeAnimation memberCount:1];
self.attackThreeAction = [CCSequence actions:_attackThreeGroup,
[CCCallFunc actionWithTarget:self selector:@selector(idle)], nil];
```

The above replaces all `ccAnimate` actions with a `ccAnimateGroup` action with capacity for one member animation each. The member animation will be the weapon's `AnimationMember` later, when a weapon is equipped.

You also store each `ccAnimateGroup` action in its own instance variable, because you will need direct access to them later.

Next implement the two new methods in `Hero.m`:

```
-(BOOL)pickUpWeapon:(Weapon *)weapon
{
    if (self.actionState == kActionStateIdle)
    {
        [self stopAllActions];
        [weapon pickedUp];
        [self setLandingDisplayFrame];
        [self performSelector:@selector(setWeapon:)
withObject:weapon afterDelay:1.0/12.0];
        return YES;
    }
    return NO;
}

-(void)dropWeapon
{
    Weapon *weapon = _weapon;
```

```

    self.weapon = nil;
    [weapon droppedFrom:(self.groundPosition.y -
    self.shadow.position.y) to:self.shadow.position];
}

}

```

The hero can only pick up a weapon when in the idle state. Once he does, he animates to the landing frame (because the landing frame makes him crouch down), and `setWeapon` is called after 1/12<sup>th</sup> of a second. It's like saying `self.weapon = weapon`.

The weapon also works in tandem with the hero during the pick-up period by executing its own `pickedUp` method.

When the hero discards the weapon in `dropWeapon`, you set the weapon to `nil` and inform it that it was dropped.

The position for `droppedFrom:` is calculated like this:



To make the weapon start at the center of the hero, you take the difference of the two position values and use that as the starting height of the drop. The landing position is equal to the shadow's position because that's where the hero should be standing.

It's not enough to set the hero's weapon using the default setter method. When you set a weapon, you need to connect the weapon's `AnimationMembers` to the hero's `AnimateGroup` actions.

To do that, add these methods to `Hero.m`:

```

-(void)removeAllAnimationMembers
{
    [_attackGroup.members removeAllObjects];
    [_attackTwoGroup.members removeAllObjects];
    [_attackThreeGroup.members removeAllObjects];
    [_idleGroup.members removeAllObjects];
    [_walkGroup.members removeAllObjects];
}

```

```

-(void)setWeapon:(Weapon *)weapon
{
    [self stopAllActions];

    // 1
    if (_weapon)
    {
        [self removeAllAnimationMembers];
    }

    _weapon = weapon;

    // 2
    if (_weapon)
    {
        _weapon.delegate = self;
        _weapon.scaleX = self.scaleX;
        [_attackGroup.members addObject:_weapon.attack];
        [_attackTwoGroup.members addObject:_weapon.attackTwo];
        [_attackThreeGroup.members addObject:_weapon.attackThree];
        [_idleGroup.members addObject:_weapon.idle];
        [_walkGroup.members addObject:_weapon.walk];
    }

    // 3
    [self runAction:self.idleAction];
    self.velocity = CGPointMakeZero;
    self.actionState = kActionStateIdle;
    _actionDelay = 0.0;
}

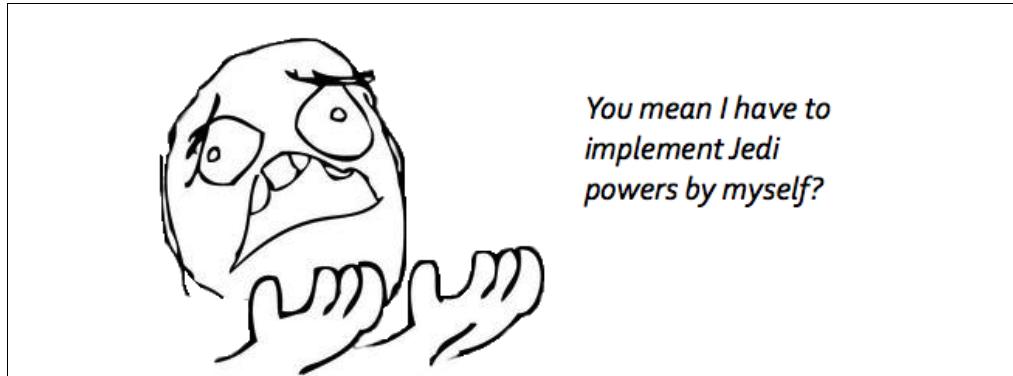
```

When the hero is assigned a new weapon, the following happens:

1. If the hero has an existing assigned weapon, then he removes all **AnimationMembers** of the old weapon from his **AnimateGroup** actions before replacing it with the new weapon.
2. If the new weapon is not **nil** (you use **self.weapon = nil** in **dropWeapon**), then:
  - a. The hero becomes the delegate of the weapon.
  - b. The weapon's **scaleX** follows the hero's so that they face the same direction.
  - c. The weapon's **AnimationMembers** are made members of the hero's corresponding **AnimateGroup** actions.
3. The hero will go back to his idle state, with or without a new weapon.

You're not finished with the hero yet! For now, though, it would be good to test if the hero is able to pick up a weapon. To do that, you need to write the collision

code between the hero and the weapon, such that the hero only picks up a weapon if it is right next to him.



Go to **GameLayer.m** and add the following method:

```
-(BOOL)collisionBetweenPlayer:(ActionSprite *)player
andWeapon:(Weapon *)weapon
{
    // 1: check if they're on the same plane
    float planeDist = player.shadow.position.y -
    weapon.shadow.position.y;

    if (fabsf(planeDist) <= kPlaneHeight)
    {
        float combinedRadius = player.detectionRadius +
        weapon.detectionRadius;
        int i;

        // 2: initial detection
        if (ccpDistanceSQ(player.position, weapon.position) <=
        combinedRadius * combinedRadius)
        {
            int contactPointCount = player.contactPointCount;
            ContactPoint contactPoint;

            // 3: secondary detection
            for (i = 0; i < contactPointCount; i++)
            {
                contactPoint = player.contactPoints[i];
                combinedRadius = contactPoint.radius +
                weapon.detectionRadius;

                if (ccpDistanceSQ(contactPoint.position,
                weapon.position) <= combinedRadius * combinedRadius)
                {

```

```
        return YES;
    }
}
}
return NO;
}
```

This collision detection method is similar to the one for the attacker and target - `collisionBetweenAttacker:andTarget:atPosition:`.

You have three phases:

1. Check if the two objects are on the same plane using `kPlaneHeight`.
  2. Check if the detection circles of both objects intersect.
  3. This is the phase that makes this collision method different from the others you've seen so far in this Starter Kit. Since the weapon's shape is simple enough that its detection circle can also serve as its contact circle, you just check if any of the hero's contact circles intersect with the weapon's detection circle.

If the player and the weapon pass all three tests, then it means the player is standing directly above the weapon and will be able to pick it up.

Still in **GameLayer.m**, modify `actionButtonWasPressed:` as follows (unchanged code is highlighted):

```
- (void)actionButtonWasPressed:(ActionButton *)actionButton
{
    if (_eventState != kEventStateScripted)
    {
        if (actionButton.tag == kTagButtonA)
        {
            //replace the contents of this if statement
            BOOL pickedUpWeapon = NO;
            if (!_hero.weapon)
            {
                //check collision for all weapons
                Weapon *weapon;
                CCARRAY_FOREACH(_weapons, weapon)
                {
                    if (weapon.weaponState ==
kWeaponStateUnequipped)
                    {
                        if ([self collisionBetweenPlayer:_hero
andWeapon:weapon])
                    }
                }
            }
        }
    }
}
```

```
        pickedUpWeapon = [_hero
pickUpWeapon:weapon];
[_actors reorderChild:weapon
z:_hero.zOrder + 1];
break;
}
}
}

if (!pickedUpWeapon)
{
[_hero attack];
}
}

else if (actionButton.tag == kTagButtonB)
{
//replace the contents of this else if statement
if (_hero.weapon)
{
[_hero dropWeapon];
}
else
{
CGPoint directionVector = [self
vectorForDirection:_hud.dPad.direction];
[_hero jumpRiseWithDirection:directionVector];
}
}
}
```

When the player presses the A button, you check if the hero is standing on any unequipped weapons. If he is, then he picks up the weapon; if not, then he simply attacks. When the player presses the B button, the hero either drops his equipped weapon or jumps.

Build and run, and try to pick up and drop a weapon:



The gauntlets work perfectly... that is, if you want to punch an enemy in the feet! I've heard of sweeping kicks, but that's not what you had in mind. ☺

You'll also notice the gauntlets don't follow the hero properly. The weapon implementation isn't complete yet, but at least you know that picking up and dropping a weapon works as intended. You just have to make some adjustments to the weapon's position.

## In the hands of the hero

To make the weapon follow the hero, go to **Hero.m** and add these methods:

```
-(void)setPosition:(CGPoint)position
{
    [super setPosition:position];

    if (_weapon)
    {
        _weapon.position = position;
    }
}

-(void)setScaleX:(float)scaleX
{
    [super setScaleX:scaleX];

    if (_weapon)
    {
        _weapon.scaleX = scaleX;
    }
}
```

```
}

-(void)setScaleY:(float)scaleY
{
    [super setScaleY:scaleY];

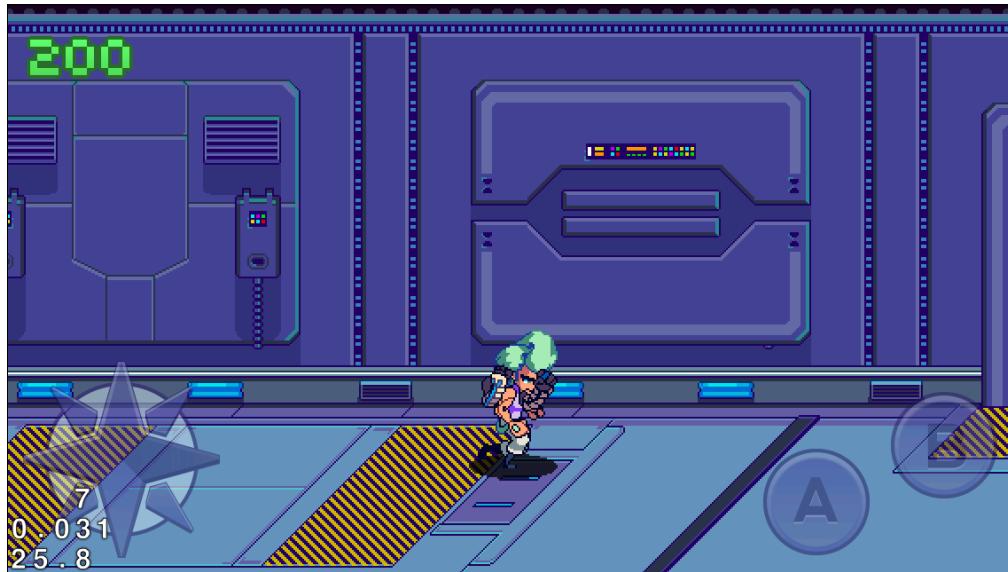
    if (_weapon)
    {
        _weapon.scaleY = scaleY;
    }
}

-(void)setScale:(float)scale
{
    [super setScale:scale];

    if (_weapon)
    {
        _weapon.scale = scale;
    }
}
```

Now when the hero has a weapon equipped, the weapon follows the hero. If the hero looks left or right using `setScale:`, then the weapon faces the same direction.

Build and run, and try picking up a weapon again.



That's a little more like it... but there are still three problems with the weapon animations.

First, when the hero runs, the gauntlets just float by his side because the gauntlets don't have an `AnimationMember` that corresponds to the running action.

Second, when the hero gets hurt, the gauntlets don't follow with the animation.

Third, when the hero walks toward the bottom of the map, the z-order for the gauntlets and the hero sometimes goes wrong.

It's tinkering time! Do the following in `GameLayer.m` (existing code is highlighted):

```
//add " !_hero.weapon &&" in the inner if statement of
actionDPad:didChangeDirectionTo:
-(void)actionDPad:(ActionDPad *)actionDPad
didChangeDirectionTo:(ActionDPadDirection)direction
{
    if (_eventState != kEventStateScripted)
    {
        CGPoint directionVector = [self
vectorForDirection:direction];

        //add " !_hero.weapon &&" to this if statement
        if (!_hero.weapon && _runDelay > 0 && _previousDirection
== direction && (direction == kActionDPadDirectionRight ||
direction == kActionDPadDirectionLeft))
        {
            [_hero runWithDirection:directionVector];
        }
        else if (_hero.actionState == kActionStateRun &&
abs(_previousDirection - direction) <= 1)
        {
            [_hero moveWithDirection:directionVector];
        }
        else
        {
            [_hero walkWithDirection:directionVector];
            _previousDirection = direction;
            _runDelay = 0.2;
        }
    }
}

//add this to reorderActors, right after [_actors
reorderChild:_hero z:spriteZ]
if (_hero.weapon)
{
    [_actors reorderChild:_hero.weapon z:spriteZ];
```

```
}
```

Then in `Hero.m`, do the following:

```
//add this at the beginning (after the call to super) of
hurtWithDamage:force:direction:
if (_weapon)
{
    [self dropWeapon];
}

//add this at the beginning (after the call to super) of
knockoutWithDamage:direction:
if (_weapon)
{
    [self dropWeapon];
```

}In `actionDPad:didChangeDirectionTo:`, you only allow the hero to run if he doesn't have a weapon equipped. Then, when the hero gets hurt or knocked out, he simply drops the weapon he's currently holding.

Also, the weapon's z-order now follows the hero's z-order in `reorderActors`. Since the weapon was reordered after the hero, it now always appears on top of the hero even if they have the same z-order value.

Build and run, and give your enemies a taste of those gauntlets.



Did you think you were finished with this part? Remember that you want the gauntlets to give the hero a damage bonus, and that's something they don't yet do. Worse, the gauntlets limit the hero's movements quite a bit, which is a bummer.

You can fix these issues. You can do it!



Open **Hero.m** and add these methods:

```
-(float)attackDamage
{
    if (_weapon)
    {
        return [super attackDamage] + _weapon.damageBonus;
    }

    return [super attackDamage];
}

-(float)attackTwoDamage
{
    if (_weapon)
    {
        return _attackTwoDamage + _weapon.damageBonus;
    }

    return _attackTwoDamage;
}

-(float)attackThreeDamage
{
    if (_weapon)
    {
        return _attackThreeDamage + _weapon.damageBonus;
    }

    return _attackThreeDamage;
}
```

When `GameLayer` asks the hero for his damage output, the hero gives the augmented damage value whenever he has a weapon equipped. He does this by adding the `damageBonus` property of the weapon to his attack damage.

If he doesn't have a weapon, then he just gives his normal attack damage.

Still in `Hero.m`, do the following (existing code is highlighted):

```
//modify setDisplayFrame: (unchanged code is highlighted)
-(void)setDisplayFrame:(CCSpriteFrame *)newFrame
{
    [super setDisplayFrame:newFrame];

    CCSpriteFrame *attackFrame = [[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"hero_attack_00_01.png"];
    CCSpriteFrame *runAttackFrame = [[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"hero_runattack_02.png"];
    CCSpriteFrame *runAttackFrame2 = [[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"hero_runattack_03.png"];
    CCSpriteFrame *jumpAttackFrame = [[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"hero_jumpattack_02.png"];
    CCSpriteFrame *attackFrame2 = [[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"hero_attack_01_01.png"];
    CCSpriteFrame *attackFrame3 = [[CCSpriteFrameCache
sharedSpriteFrameCache]
spriteFrameByName:@"hero_attack_02_02.png"];

    if (newFrame == attackFrame || newFrame == attackFrame2)
    {
        if ([self.delegate actionSpriteDidAttack:self])
        {
            _chainTimer = 0.3;

            //add this if statement
            if (_weapon)
            {
                [_weapon used];
            }
        }
    }
    else if (newFrame == attackFrame3)
```

```
{  
    //replace the contents of this else if statement  
    if ([self.delegate actionSpriteDidAttack:self])  
    {  
        if (_weapon)  
        {  
            [_weapon used];  
        }  
    }  
    }  
    else if (newFrame == runAttackFrame || newFrame ==  
runAttackFrame2 || newFrame == jumpAttackFrame)  
    {  
        [self.delegate actionSpriteDidAttack:self];  
    }  
}  
  
//add this method  
-(void)weaponDidReachLimit:(Weapon *)weapon  
{  
    [self dropWeapon];  
}
```

Whenever the hero delivers a successful attack, the weapon gets “used.” Remember that the `used` method in `Weapon.m` simply decreases the `limit` value of the weapon.

When `limit` reaches zero, the weapon destroys itself and tells its delegate, the hero, to execute `weaponDidReachLimit:`.

Then, in `weaponDidReachLimit:`, the hero just drops the weapon.

That’s it. Build, run, and finally have fun kicking metallic ass with the gauntlets!



## Taking out the trash (cans)

Oh look, there's a pair of gauntlets on the floor! Score!

That's probably the feeling you get every time you play the game right now, since the game randomly places the gauntlets all over the map. There may be times when designing a level when you want to have more control over where your weapons appear.

Here's an idea: why not add some new objects to the map which contain the gauntlets. It looks strange to have them sitting on the floor, anyway. So instead, you'll put the gauntlets inside a trashcan. That's less weird, right? ☺

### The MapObjects template

Begin by creating a template map object. This will make it easy for you to create your own custom objects in the future.

You've done most of these things before, so I'll just breeze through them and focus on discussing newer concepts only.

First, go to **Defines.h** and add this:

```
typedef enum _ObjectState
{
    kObjectStateActive,
    kObjectStateDestroyed
} ObjectState;
```

Then select the **PompaDroid** group in Xcode, go to **File\New\Group** and name the new group **MapObjects**.

Next select the **MapObjects** group, go to **File\New\File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **CCSprite** for Subclass of, click **Next** and name the new file **MapObject**.

Open **MapObject.h** and replace its contents with the following:

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

@interface MapObject : CCSprite {

}

@property(nonatomic, assign)float detectionRadius;
@property(nonatomic, assign)ContactPoint *contactPoints;
@property(nonatomic, assign)int contactPointCount;
@property(nonatomic, assign)ObjectState objectState;

-(CGRect)collisionRect;
-(void)modifyContactPointAtIndex:(const NSUInteger)pointIndex
offset:(const CGPoint)offset radius:(const float)radius;
-(void)destroyed;

@end
```

**MapObject** is a much simpler version of **ActionSprite** – it can't move or have any actions. It just has the following:

- **detectionRadius, contactPoints, contactPointCount**: The same old collision detection variables and helper methods you use for detecting if one object collides with another. **modifyContactPointAtIndex** helps adjust these points.
- **objectState**: The state of the object.
- **collisionRect**: You'll use this to detect collisions.
- **destroyed**: What happens when the object is destroyed.

Switch to **MapObject.m** and add these methods:

```
-(void)destroyed
{
    self.objectState = kObjectStateDestroyed;
}

-(CGRect)collisionRect
{
    return CGRectZero;
}
```

```
-(void) dealloc
{
    free(_contactPoints);
}

-(void) setPosition:(CGPoint) position
{
    [super setPosition:position];
    [self transformPoints];
}

-(void) transformPoints
{
    int i;
    for (i = 0; i < _contactPointCount; i++)
    {
        _contactPoints[i].position = ccpAdd(_position,
ccp(_contactPoints[i].offset.x, _contactPoints[i].offset.y));
    }
}

-(void) modifyContactPointAtIndex:(const NSUInteger) pointIndex
offset:(const CGPoint) offset radius:(const float) radius
{
    ContactPoint *contactPoint = &_contactPoints[pointIndex];
    [self modifyPoint:contactPoint offset:offset radius:radius];
}

-(void) modifyPoint:(ContactPoint *) point offset:(const
CGPoint) offset radius:(const float) radius
{
    point->offset = ccpMult(offset, kPointFactor);
    point->radius = radius * kPointFactor;
    point->position = ccpAdd(_position, point->offset);
}
```

Most of these methods are counterparts of methods from `ActionSprite`, except for:

- **destroyed**: This simply changes the `objectState` to `kObjectStateDestroyed`.
- **collisionRect**: Returns an empty rectangle. The `collisionRect` should be specific per object, so you will have to implement this in every `MapObject` subclass.

The template is now ready for use. You can proceed to make your very first map object.

## Your sanitation plan

Select the **MapObjects** group, go to **File\New\File**, choose the **iOS\cocos2D v2.x\CCNode class** template and click **Next**. Enter **MapObject** for Subclass of, click **Next** and name the new file **TrashCan**.

Go to **TrashCan.h** and add this to the top of the file:

```
#import "MapObject.h"
```

Switch to **TrashCan.m** and add these methods:

```
-(id)init
{
    if ((self = [super initWithSpriteFrameName:@"trashcan.png"]))
    {
        self.objectState = kObjectStateActive;
        self.detectionRadius = 57.0 * kPointFactor;
        self.contactPointCount = 5;
        self.contactPoints = malloc(sizeof(ContactPoint) *
self.contactPointCount);

        [self modifyContactPointAtIndex:0 offset:ccp(0.0, 2.0)
radius:33.0];
        [self modifyContactPointAtIndex:1 offset:ccp(0.0, -15.0)
radius:33.0];
        [self modifyContactPointAtIndex:2 offset:ccp(0.0, 26.0)
radius:17.0];
        [self modifyContactPointAtIndex:3 offset:ccp(19.0, 29.0)
radius:16.0];
        [self modifyContactPointAtIndex:4 offset:ccp(-23.0, -38.0)
radius:10.0];
    }
    return self;
}

-(void)destroyed
{
    CCSpriteFrame *destroyedFrame = [[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"trashcan_hit.png"];
    [self setDisplayFrame:destroyedFrame];
    [super destroyed];
}

-(CGRect)collisionRect
{
```

```
    return CGRectMake(self.position.x - self.contentSize.width/2 *  
kScaleFactor, self.position.y - self.contentSize.height/2 *  
kScaleFactor, 64 * kPointFactor, 32 * kPointFactor);  
}
```

In `init`, you create a `ccsprite` using the `trashcan.png` image found in the sprite sheet. Then you create the detection and contact circles for the sprite with the same measurement technique as before (using PhysicsEditor).

You measure the `collisionRect` the same way, just using a rectangle instead of a circle. The rectangle represents the isometric collision area of the trashcan. In `ActionSprite`, this is `feetCollisionRect`. Here, you use the `ContentSize` property instead of measurement values such as `centerToSides` and `centerToBottom` because the trashcan sprite mostly covers the whole area of the sprite.

When the trashcan is destroyed, the displayed image switches to `trashcan_hit.png`.

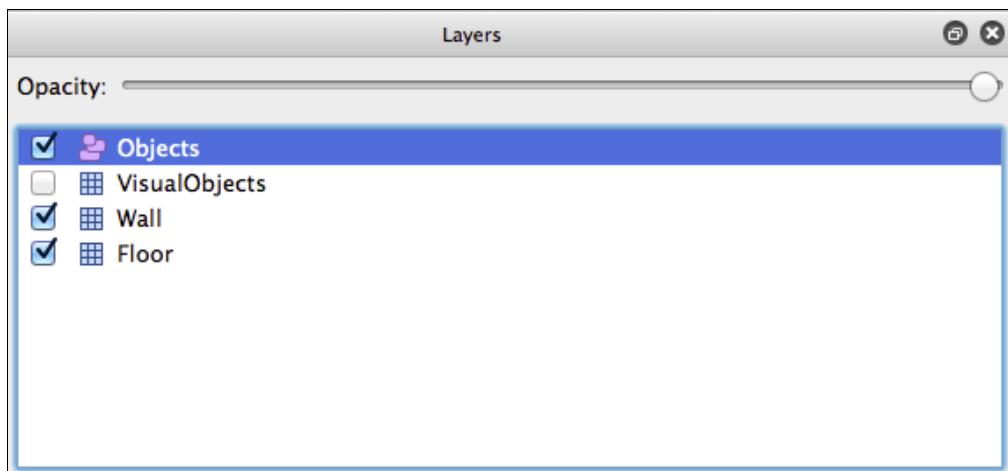
Now that you've finished the `TrashCan` class, you can put some trashcans on the map. You could define the position of each map object in `Levels.plist`, just as you placed the enemies and battle events on the map.

However, for this starter kit, you'll position the trashcans visually, using the tile map itself. This is nice and easy, and is a good learning experience!

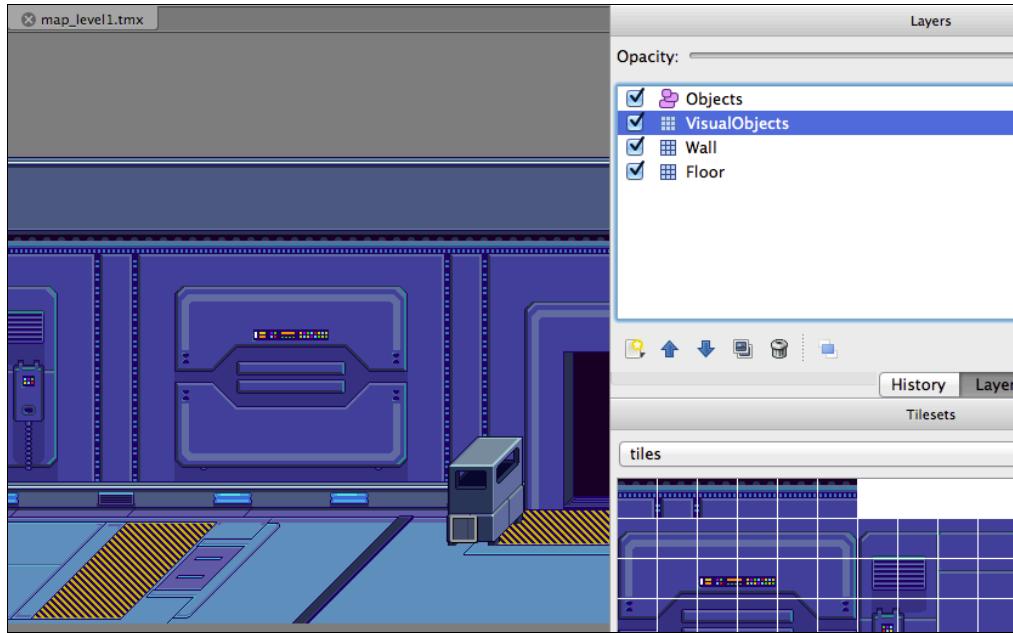
Lucky for you, the tile map you've been using, `map_level1.tmx`, already has the trashcans positioned properly. To see exactly how it was done, take a look at the file.

Run the [Tiled Map Editor](#), go to **File\Open** and navigate to the `map_level1.tmx` file in the **Resources\Images** folder of your project directory.

Once it's open, take a look at the **Layers** panel on the upper-right. Notice a layer named **VisualObjects** whose checkbox is un-ticked.



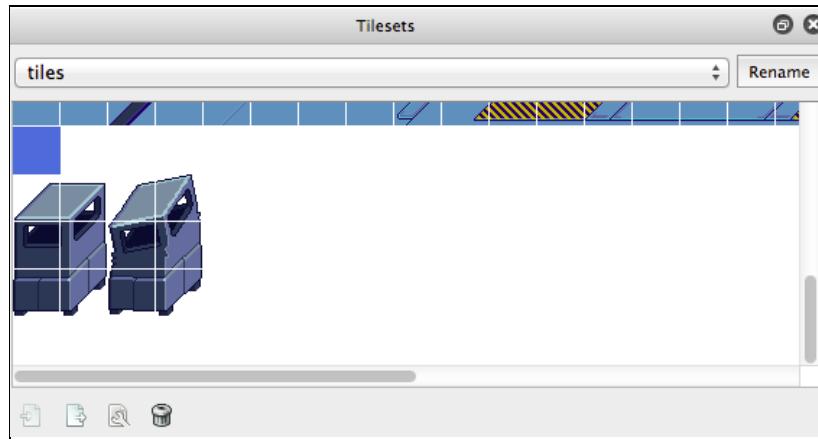
Check the **VisualObjects** checkbox. Once you've done this, take a look at the tile map to the left and you will see some trashcans appear.



If you leave the **VisualObjects** layer visible and save the file, your game will have trashcans in it. However, there's a reason why the **VisualObjects** layer was not set to visible by default.

The **VisualObjects** layer is nothing more than a layer used to help determine the positions of the trashcans in the map, and is not meant to be used to show the actual trashcans in the game.

When a trashcan is placed in the editor, it is done so using tiles. This means that each trashcan you see in the level is not a whole object, but rather, six tiles joined together, as you can see in the **Tilesets** panel on the lower-right.

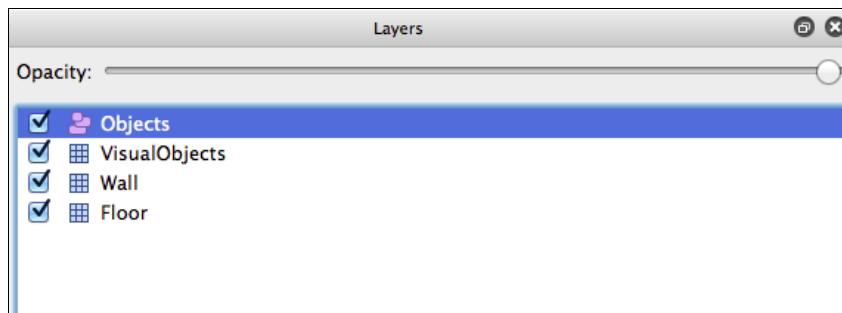


You need a trashcan to be treated as single object, a **trashcan** object, and it needs to animate as one object as well. If your trashcan were made up of six tiles, you'd need to replace all six tiles just to animate it by one frame.

A better way is to just use the editor to mark the position of each trashcan. Zoom in on one trashcan in the editor, and you will notice a hollow gray box on its lower left corner.



This is the actual marker for the trashcan. Whereas the VisualObjects layer visually represents the trashcans on the map, this marker pinpoints the exact tile location of the trashcan, and is part of the Objects layer.

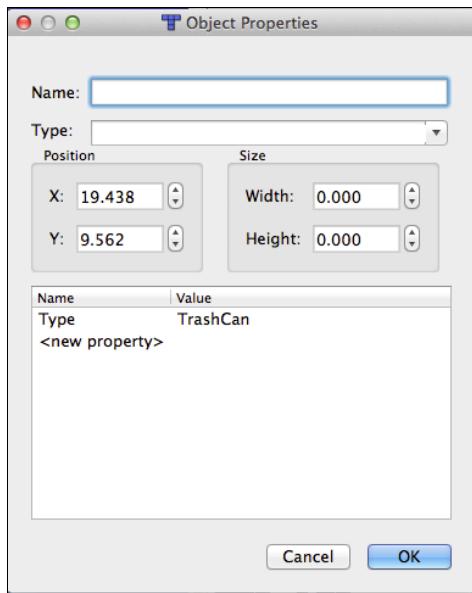


The icon beside the Objects layer is different from the other layers because it's not a Tile Layer, or a layer used to place visual tiles – rather, it's an Object Layer, or a layer used to place markers with properties.

Control+Click (or right-click) on one of the object markers on the map and select **Object Properties**.



After you select Object Properties, a new window should appear showing you something like this:



Here you can see the properties of the selected object marker. In Cocos2D, you can access these properties as if they were in a dictionary by using text-based identifiers.

You have the default properties – name, type, position and size, as well as custom properties that you can define in the bottom panel.

In this case, a property named **Type** has been pre-defined for you with a value of **TrashCan** to indicate that this marker is used to position a trashcan.

**Note:** Since there's only one property, it's actually not necessary to define a custom property like this, and it even uses the same name (Type) as one of the default properties.

However, for the sake of example, the above shows you how you can possibly use the property window.

Remember this defined property and its location on the map, as it will be very important shortly.

## Creating trashcan objects on the map

You have everything you need from the map now, so close the editor without saving. Make sure you leave the VisualObjects layer unchecked so that it doesn't show up in the game.

Back in Xcode, open **GameLayer.h** and add this property:

```
@property(nonatomic, strong)CCArray *mapObjects;
```

Switch to **GameLayer.m** and do the following:

```
//add to top of file
#import "TrashCan.h"

//add to initWithLevel:, right after [self initEffects]
[self initMapObjects];

//add these methods
-(void)initMapObjects
{
    CCTMXObjectGroup *objectGroup = [_tileMap
objectGroupNamed:@"Objects"];
    self.mapObjects = [CCArray
arrayWithCapacity:objectGroup.objects.count];

    NSMutableDictionary *object;
    NSString *type;
    CGPoint position, coord, origin;

    for (object in [objectGroup objects])
    {
        type = [object valueForKey:@"Type"];

        if (type && [type compare:@"TrashCan" == NSOrderedSame])
        {
            position = ccp([[object valueForKey:@"x"] floatValue],
[[object valueForKey:@"y"] floatValue]);
            coord = [self tileCoordForPosition:position];
            origin = [self tilePositionForCoord:coord
anchorPoint:ccp(0.0, 0.0)];

            TrashCan *trashCan = [TrashCan node];
            trashCan.scale *= kScaleFactor;
            CGSize scaledSize =
CGSizeMake(trashCan.contentSize.width * kScaleFactor,
trashCan.contentSize.height * kScaleFactor);
            CGPoint actualOrigin = ccpMult(origin, kPointFactor);
            trashCan.position = ccp(actualOrigin.x +
scaledSize.width * trashCan.anchorPoint.x, actualOrigin.y +
scaledSize.height * trashCan.anchorPoint.y);
            [_actors addChild:trashCan];
            [_mapObjects addObject:trashCan];
        }
    }
}
```

```
        }
    }

-(CGPoint)tileCoordForPosition:(CGPoint)position
{
    float tileSizeWidth = _tileMap.tileSize.width;
    float tileSizeHeight = _tileMap.tileSize.height;
    float levelHeight = _tileMap.mapSize.height * tileSizeHeight;

    float x = floor(position.x / tileSizeWidth);
    float y = floor((levelHeight - position.y) / tileSizeHeight);
    return ccp(x, y);
}

-(CGPoint)tilePositionForCoord:(CGPoint)coord
anchorPoint:(CGPoint)anchorPoint
{
    float w = _tileMap.tileSize.width;
        float h = _tileMap.tileSize.height;
    return ccp((coord.x * w) + (w * anchorPoint.x),
    ((_tileMap.mapSize.height - coord.y - 1) * h) + (h *
    anchorPoint.y));
}
```

When the map is loaded, you check for all objects on the Objects layer of the tile map. If the object has a Type value of TrashCan, you use the properties of that object to create a `TrashCan` object.

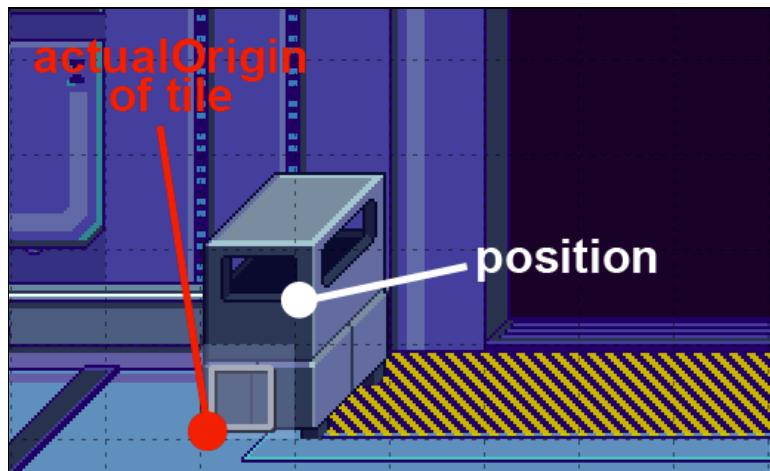
First you get the position from the x and y properties. Then you use a helper method, `tileCoordForPosition:`, to get the tile coordinates of the object's position.

Each tile has a coordinate, starting from (0, 0) for the upper-leftmost tile on the map. As you traverse right, the x-coordinate increases, and as you traverse downwards, the y-coordinate increases.



The y-coordinate is opposite to what you're used to with Cocos2D, where the (0, 0) coordinate is on the lower-left and y increases going upward.

Next, given the x and y tile coordinates, you use another helper method, `tilePositionForCoord:`, to retrieve the position of the origin of the tile in Cocos2D's coordinate system. The origin of the tile is relative to the anchor point, which in this case is (0, 0), the position of the lower-left corner of that tile. You multiply the appropriate point-scaling factor to get the `actualOrigin`.



Finally, you position the `TrashCan` object relative to that origin. Remember that the default `anchorPoint` of a `ccSprite` is (0.5, 0.5), or at the center of the sprite, so you add half the width and half the height of the sprite to the origin to get the position.

Build and run to see some trashcans scattered across the map.



## Up close and personal with a trashcan

The trashcans are in a ghostly state at the moment – meaning that the hero can pass right through them. Their z-ordering might also be a little funky.

First up, let's fix the movement collisions. You've encountered movement collisions between the hero and the map boundaries before, but this case is a bit different.

You were able to easily limit the hero's movement relative to the map because when it comes to collisions with the map boundaries, you are always sure of one thing – the direction from which the hero is coming.

When he collides with the upper boundary of the map, he must be coming from below, and when he collides with the left boundary of the map, he must be coming from the right, and so on.

For object collisions, it isn't as straightforward, so the first thing to determine is the relative position of the two objects.

Add this method to **GameLayer.m**:

```
-(void)objectCollisionsForSprite:(ActionSprite *)sprite
{
    MapObject *mapObject;

    CCARRAY_FOREACH(_mapObjects, mapObject)
    {
        if (CGRectIntersectsRect(sprite.feetCollisionRect,
                               mapObject.collisionRect))
        {
            float x = sprite.desiredPosition.x;
            float y = sprite.desiredPosition.y;
```

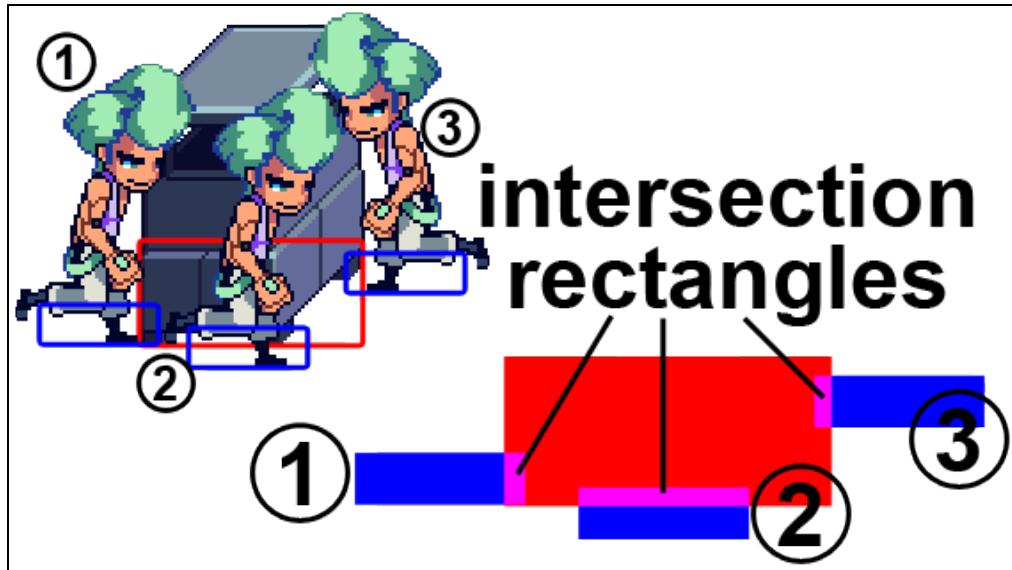
```
    CGRect intersection =
CGRectIntersection(sprite.feetCollisionRect,
mapObject.collisionRect);

        if (intersection.size.width >
intersection.size.height)
        {
            if (sprite.groundPosition.y <
mapObject.position.y)
            {
                y = sprite.desiredPosition.y -
intersection.size.height;
            }
            else
            {
                y = sprite.desiredPosition.y +
intersection.size.height;
            }
        }
        else
        {
            if (sprite.groundPosition.x <
mapObject.position.x)
            {
                x = sprite.desiredPosition.x -
intersection.size.width;
            }
            else
            {
                x = sprite.desiredPosition.x +
intersection.size.width;
            }
        }

        sprite.desiredPosition = ccp(x, y);
    }
}
}
```

This method tests if an `ActionSprite`'s `feetCollisionRect` intersects with a `MapObject`'s `collisionRect`. Then it forms a rectangle from the intersection. With this new rectangle, you can figure out from which side of the `MapObject` the `ActionSprite` came, and how to resolve the collision.

Consider the following three scenarios:



1. The hero walks right and hits the left side of the trashcan. The rectangle formed by the intersection has a longer height than width.
2. The hero walks up and hits the bottom edge of the trashcan. The rectangle formed by the intersection has a longer width than height.
3. The hero walks left and hits the right side of the trashcan. The rectangle formed by the intersection has a longer height than width.

If you backtrack a bit, by comparing the width and height of the new rectangle, you can determine the kind of collision you need to handle.

In the first and third scenarios, it is a horizontal collision because the height of the new rectangle is longer than its width. To resolve this collision, you just need to push the hero back or forward by exactly the width of the intersection rectangle.

The second scenario is a vertical collision, and you just need to push the hero down by exactly the height of the intersection rectangle.

That's precisely what happens in `objectCollisionsForSprite:`. You take the desired position of the sprite and adjust it based on the kind of collision that occurred.

Now all you need to do is to make use of this collision method.

In `GameLayer.m`, modify `updatePositions` as follows:

```
//add inside if (_hero.actionState > kActionStateNone), right
after the {
[self objectCollisionsForSprite:_hero];

//add inside if (robot.actionState > kActionStateNone), right
after the {
[self objectCollisionsForSprite:robot];
```

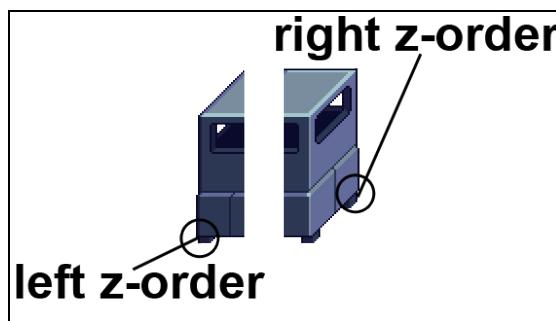
```
//add inside if (_boss && _boss.actionState > kActionStateNone),  
right after the {  
[self objectCollisionsForSprite:_boss];
```

Build, run, and try colliding with the trashcan.



Because of the perspective of the trashcan, getting it to draw with the proper z-order is going to be tricky. Unlike the `ActionSprite` and `Weapon` classes, `TrashCan` doesn't have a single point on the ground you can consider as its position. You could even say that the left and right sides of the trashcan need different z-orders.

In fact, splitting the image in half is a good way of handling z-order for the trashcan.



The lower-left point of the trashcan's `collisionRect` will have a different z-order than the upper-right point of the rectangle. Let's call these the left and right z-order points.

When a sprite bumps with the left side of the trashcan and moves above the left z-order point, it will appear behind the trashcan. But when the sprite bumps the right side of the trashcan, but is below the right z-order point, it will be drawn in front of the trashcan.

That's the ideal behavior. However, implementing this requires more steps from you. You'd have to cut the sprite in two and change the structure of `MapObject` to handle two images.

So for now, you have my permission to settle for a "good enough" solution: just use the point in between the left and right z-order points.

Still in `GameLayer.m`, modify `reorderActors`:

```
//add at the end of the method, before the last }
MapObject *object;
CCARRAY_FOREACH(_mapObjects, object)
{
    spriteZ = [self
getZFromYPosition:object.collisionRect.origin.y +
object.collisionRect.size.height/2];
    [_actors reorderChild:object z:spriteZ];
}
```

You use the center of the `collisionRect` as the basis of the object's z-order.

Build, run, and touch the trashcan again. It's clean. Don't worry.



## Stashing the weapons, trashing the cans

With the trashcans properly in place, you can now hide weapons inside them. When the hero punches the trashcan, it should get destroyed, revealing a pair of gauntlets in the process.

As always, the collision detection method comes first.

Still in `GameLayer.m`, add this method:

```
-(BOOL)collisionBetweenAttacker:(ActionSprite *)attacker
andObject:(MapObject *)object atPosition:(CGPoint *)position
{
    //first phase: check if they're on the same plane
    float objectBottom = object.collisionRect.origin.y;
    float objectTop = objectBottom +
    object.collisionRect.size.height;
    float attackerBottom = attacker.feetCollisionRect.origin.y;
    float attackerTop = attackerBottom +
    attacker.feetCollisionRect.size.height;

    if ((attackerBottom > objectBottom && attackerBottom <
    objectTop) || (attackerTop > objectBottom && attackerTop <
    objectTop))
    {
        int i, j;
        float combinedRadius = attacker.detectionRadius +
        object.detectionRadius;

        //initial detection
        if (ccpDistanceSQ(attacker.position, object.position) <=
        combinedRadius * combinedRadius)
        {
            int attackPointCount = attacker.attackPointCount;
            int contactPointCount = object.contactPointCount;

            ContactPoint attackPoint, contactPoint;

            //secondary detection
            for (i = 0; i < attackPointCount; i++)
            {
                attackPoint = attacker.attackPoints[i];

                for (j = 0; j < contactPointCount; j++)
                {
                    contactPoint = object.contactPoints[j];
                    combinedRadius = attackPoint.radius +
                    contactPoint.radius;

                    if (ccpDistanceSQ(attackPoint.position,
                    contactPoint.position) <= combinedRadius * combinedRadius)
                    {
                        //attack point collided with contact point
                        position->x = attackPoint.position.x;
                        position->y = attackPoint.position.y;
                    }
                }
            }
        }
    }
}
```

```
        return YES;
    }
}
}
}
}
return NO;
}
```

This is very similar to `collisionBetweenAttacker:andTarget:atPosition:`. The only difference is the first phase – checking if the two objects are on the same plane.

Previously, you used `kPlaneHeight` and the simple distance between the y-axis positions of the two sprites to determine if they are standing on the same plane. This time, since the trashcan occupies a much larger ground space, you just make sure that the `feetCollisionRect` of the sprite is within the `collisionRect` of the object.

To make the weapons appear from out of the trashcans, do the following in **GameLayer.m**:

```
//add this method
-(Weapon *)getWeapon
{
    Weapon *weapon;
    CCARRAY_FOREACH(_weapons, weapon)
    {
        if (weapon.weaponState == kWeaponStateNone)
        {
            return weapon;
        }
    }
    return NULL;
}

//replace the contents of initWeapons
-(void)initWeapons
{
    int i;
    self.weapons = [CCArray arrayWithCapacity:3];
    Weapon *weapon;

    for (i = 0; i < 3; i++)
    {
        weapon = [Gauntlets node];
        weapon.visible = NO;
    }
}
```

```

        weapon.shadow.scale *= kScaleFactor;
        weapon.scale *= kScaleFactor;
        weapon.groundPosition = OFFSCREEN;
        [_actors addChild:weapon.shadow];
        [_actors addChild:weapon];
        [_weapons addObject:weapon];
    }

}

//add this to actionSpriteDidAttack:, inside if (actionSprite ==
//_hero), right before return didHit;
MapObject *mapObject;
CCARRAY_FOREACH(_mapObjects, mapObject)
{
    if ([self collisionBetweenAttacker:_hero andObject:mapObject
atPosition:&attackPosition])
    {
        HitEffect *hitEffect = [self getHitEffect];
        [_actors reorderChild:hitEffect z:MAX(mapObject.zOrder,
_hero.zOrder) + 1];
        [hitEffect showEffectAtPosition:attackPosition];

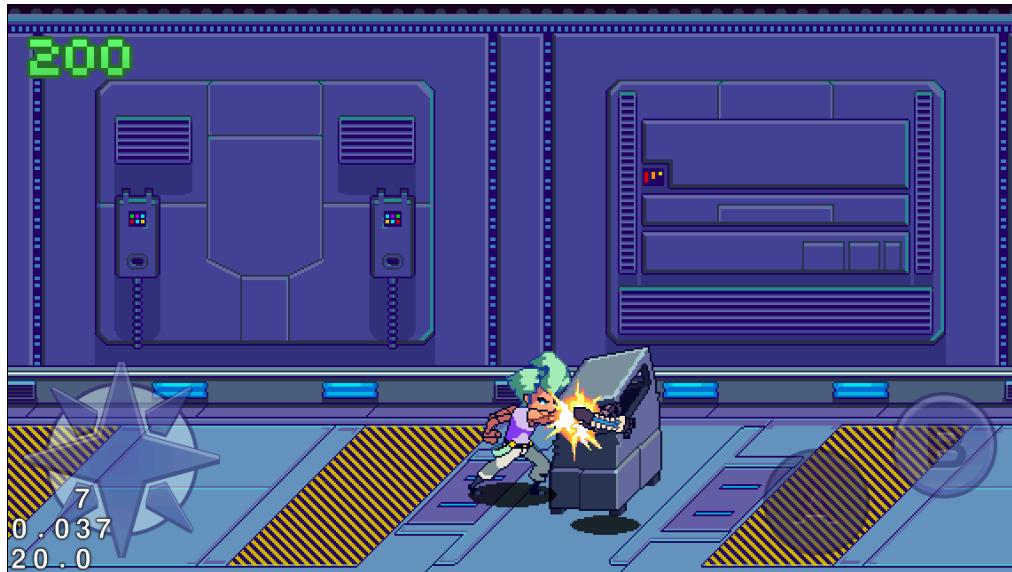
        if (mapObject.objectState != kObjectStateDestroyed)
        {
            [mapObject destroyed];
            Weapon *weapon = [self getWeapon];
            [weapon droppedFrom:mapObject.contentSize.height/2 *
kScaleFactor to:ccp(mapObject.position.x, mapObject.position.y -
mapObject.contentSize.height/2 * kScaleFactor)];
            weapon.visible = YES;
        }
    }
}

```

`getWeapon` retrieves an unused weapon, similar to `getHitEffect`. Then, in `initWeapons`, you make sure that all weapons are invisible and unused.

Finally, you check for collisions between the hero's attack and map objects in `actionSpriteDidAttack:`. When a collision occurs, you show a hit effect, and if the punched map object isn't destroyed yet, then you destroy it and drop a weapon from its center.

Build, run, and scavenge for a pair of gauntlets!



## Ending scenarios

If you have managed to get through the entire game and beat the final boss you might have thought to yourself, "Woohoo! ... But now what?"

A complete game requires both winning and losing conditions. Currently, you can beat all the levels and nothing will happen, which is quite disappointing!

You want the game to end noticeably when either all the enemies are wiped out, or the hero dies.

To keep things simple, when the game ends, you will just show a message informing the player that they've won or lost. Then you'll transition back to the **TitleScene**.

Go to **HudLayer.h** and add this method prototype:

```
- (void)showMessage:(NSString *)message color:(ccColor3B)color;
```

Switch to **HudLayer.m** and add the method implementation:

```
- (void)showMessage:(NSString *)message color:(ccColor3B)color
{
    _centerLabel.color = color;
    [_centerLabel setString:message];
    [_centerLabel runAction:[CCSequence actions: [CCPlace
actionWithPosition:ccp(SCREEN.width +
_centerLabel.contentSize.width/2, CENTER.y)], [CCShow action],
[CCMoveTo actionWithDuration:0.2 position:CENTER], [CCDelayTime
actionWithDuration:1.0], [CCMoveTo actionWithDuration:0.2
```

```
position:ccp(-_centerLabel.contentSize.width/2, CENTER.y)],  
[CCHide action], nil]);  
}
```

This uses the previously created `centerLabel` to show whatever message you want, in a color you specify.

Open `GameLayer.m` and do the following:

```
//add to top of file  
#import "TitleScene.h"  
  
//add this to actionSpriteDidDie:, inside if (actionSprite ==  
_hero), right after [_hud setHitPoints:0  
fromMaxHP:_hero.maxHitPoints]  
[_hud showMessage:@"GAME OVER" color:COLOR_LOWHP];  
[self runAction:[CCSequence actions:[CCDelayTime  
actionWithDuration:2.0], [CCCallBlock actionWithBlock:^(void){  
    [[CCDirector sharedDirector] replaceScene:[CCTransitionFade  
transitionWithDuration:1.0 scene:[TitleScene node]]];  
}], nil];  
  
//add this to the end of updateEvent, right after the //end game  
comment  
[_hud showMessage:@"YOU WIN" color:COLOR_FULLHP];  
[self runAction:[CCSequence actions:[CCDelayTime  
actionWithDuration:2.0], [CCCallBlock actionWithBlock:^(void){  
    [[CCDirector sharedDirector] replaceScene:[CCTransitionFade  
transitionWithDuration:1.0 scene:[TitleScene node]]];  
}], nil]];
```

When the hero dies, the HUD will show a message saying “GAME OVER” and change the scene back to the `TitleScene` after two seconds. Likewise, when the hero wins on the last level, the HUD will show a “YOU WIN” message, and also transition to the `TitleScene` afterwards.

It’s a simple end to a simple game – though if you wanted, it would not be too complicated to create another scripted event for a winning or losing scenario, or allow the vanquished player to begin again at the start of the highest level they reached, or show a neat story sequence!

Build, run, and beat the game... or die trying.



## Gratuitous music and sound effects

The game now plays pretty nicely, but socking it to a robot isn't as satisfying as it could be – there's no audio feedback! Some background music wouldn't hurt, either.

For this game, you'll be using background music composed by [Kevin MacLeod of Incompetech](#), as well as some 8-bit sound effects that I either created using the neat [bfxr](#) utility or downloaded from [freesound](#).

You can add music and sound effects to the game in just a few simple steps.

Drag the **Sounds** folder from the **Resources** folder of your Starter Kit into the **Resources** group of your project. Make sure that “Copy items into destination group’s folder” is checked, that “Create groups for any added folders” is selected and that the **PompaDroid** target is selected.

Open **TitleLayer.m** and do the following:

```
//add to top of file
#import "SimpleAudioEngine.h"

//add to init, inside curly braces of the if statement
[[SimpleAudioEngine sharedEngine] preloadEffect:@"hit0.caf"];
[[SimpleAudioEngine sharedEngine] preloadEffect:@"hit1.caf"];
[[SimpleAudioEngine sharedEngine] preloadEffect:@"herodeath.caf"];
[[SimpleAudioEngine sharedEngine]
preloadEffect:@"enemydeath.caf"];
[[SimpleAudioEngine sharedEngine] preloadEffect:@"blip.caf"];
```

```
[[SimpleAudioEngine sharedEngine]
playBackgroundMusic:@"latin_industries.aifc"];
[[SimpleAudioEngine sharedEngine] setBackgroundMusicVolume:0.5];

//add to the beginning of ccTouchesBegan:
[[SimpleAudioEngine sharedEngine] playEffect:@"blip.caf"];
```

When the `TitleLayer` is created, you preload some sound effects and play the background music at half the volume, because the music was recorded at a louder level than the sound effects. Once the player touches the screen, you play the blip sound effect.

Switch to `GameLayer.m` and do the following (existing code is highlighted):

```
//add to top of file
#import "SimpleAudioEngine.h"

//Modify actionSpriteDidAttack: (existing code is highlighted)
-(BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    BOOL didHit = NO;
    if (actionSprite == _hero)
    {
        CGPoint attackPosition;
        Robot *robot;
        CCARRAY_FOREACH(_robots, robot)
        {
            if (robot.actionState < kActionStateKnockedOut &&
robot.actionState != kActionStateNone)
            {
                if ([self collisionBetweenAttacker:_hero
andTarget:robot atPosition:&attackPosition])
                {
                    BOOL showEffect = YES;

                    DamageNumber *damageNumber = [self
getDamageNumber];

                    if (_hero.actionState ==
kActionStateJumpAttack)
                    {
                        //add this
                        [[SimpleAudioEngine sharedEngine]
playEffect:@"hit1.caf"];
                    }
                }
            }
        }
    }
}
```

```
[robot
knockoutWithDamage:_hero.jumpAttackDamage
direction:ccp(_hero.directionX, 0)];
[damageNumber
showWithValue:_hero.jumpAttackDamage fromOrigin:robot.position];
showEffect = NO;
}
else if (_hero.actionState ==
kActionStateRunAttack)
{
    //add this
    [[SimpleAudioEngine sharedEngine]
playEffect:@"hit0.caf"];
[robot
knockoutWithDamage:_hero.runAttackDamage
direction:ccp(_hero.directionX, 0)];
[damageNumber
showWithValue:_hero.runAttackDamage fromOrigin:robot.position];
}
else if (_hero.actionState ==
kActionStateAttackThree)
{
    //add this
    [[SimpleAudioEngine sharedEngine]
playEffect:@"hit1.caf"];
[robot
knockoutWithDamage:_hero.attackThreeDamage
direction:ccp(_hero.directionX, 0)];
[damageNumber
showWithValue:_hero.attackThreeDamage fromOrigin:robot.position];
showEffect = NO;
}
else if (_hero.actionState ==
kActionStateAttackTwo)
{
    //add this
    [[SimpleAudioEngine sharedEngine]
playEffect:@"hit0.caf"];
[robot
hurtWithDamage:_hero.attackTwoDamage force:_hero.attackForce
direction:ccp(_hero.directionX, 0.0)];
[damageNumber
showWithValue:_hero.attackTwoDamage fromOrigin:robot.position];
}
else
```

```
        {
            //add this
            [[SimpleAudioEngine sharedEngine]
playEffect:@"hit0.caf"];
            [_boss hurtWithDamage:_hero.attackDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0.0)];
            [damageNumber
showWithValue:_hero.attackDamage fromOrigin:robot.position];
        }
        didHit = YES;

        if (showEffect)
        {
            HitEffect *hitEffect = [self
getHitEffect];
            [_actors reorderChild:hitEffect
z:MAX(robot.zOrder, _hero.zOrder) + 1];
            [hitEffect
showEffectAtPosition:attackPosition];
        }
    }
}

if (_boss && _boss.actionState < kActionStateKnockedOut &&
_boss.actionState != kActionStateNone)
{
    if ([self collisionBetweenAttacker:_hero
andTarget:_boss atPosition:&attackPosition])
    {
        BOOL showEffect = YES;
        DamageNumber *damageNumber = [self
getDamageNumber];

        if (_hero.actionState == kActionStateJumpAttack)
        {
            //add this
            [[SimpleAudioEngine sharedEngine]
playEffect:@"hit1.caf"];
            [_boss hurtWithDamage:_hero.jumpAttackDamage
force:_hero.attackForce direction:ccp(_hero.directionX, 0)];
            [damageNumber
showWithValue:_hero.jumpAttackDamage fromOrigin:_boss.position];
            showEffect = NO;
        }
    }
}
```

```
        else if (_hero.actionState ==  
kActionStateRunAttack)  
    {  
        //add this  
        [[SimpleAudioEngine sharedEngine]  
playEffect:@"hit0.caf"];  
        [_boss hurtWithDamage:_hero.runAttackDamage  
force:_hero.attackForce direction:ccp(_hero.directionX, 0)];  
        [damageNumber  
showWithValue:_hero.runAttackDamage fromOrigin:_boss.position];  
    }  
    else if (_hero.actionState ==  
kActionStateAttackThree)  
    {  
        //add this  
        [[SimpleAudioEngine sharedEngine]  
playEffect:@"hit1.caf"];  
        [_boss hurtWithDamage:_hero.attackThreeDamage  
force:_hero.attackForce direction:ccp(_hero.directionX, 0)];  
        [damageNumber  
showWithValue:_hero.attackThreeDamage fromOrigin:_boss.position];  
        showEffect = NO;  
    }  
    else if (_hero.actionState ==  
kActionStateAttackTwo)  
    {  
        //add this  
        [[SimpleAudioEngine sharedEngine]  
playEffect:@"hit0.caf"];  
        [_boss hurtWithDamage:_hero.attackTwoDamage  
force:_hero.attackForce/2 direction:ccp(_hero.directionX, 0.0)];  
        [damageNumber  
showWithValue:_hero.attackTwoDamage fromOrigin:_boss.position];  
    }  
    else  
    {  
        //add this  
        [[SimpleAudioEngine sharedEngine]  
playEffect:@"hit0.caf"];  
        [_boss hurtWithDamage:_hero.attackDamage  
force:0 direction:ccp(_hero.directionX, 0.0)];  
        [damageNumber showWithValue:_hero.attackDamage  
fromOrigin:_boss.position];  
    }  
    didHit = YES;
```

```
        if (showEffect)
        {
            HitEffect *hitEffect = [self getHitEffect];
            [_actors reorderChild:hitEffect
z:MAX(_boss.zOrder, _hero.zOrder) + 1];
            [hitEffect
showEffectAtPosition:attackPosition];
        }
    }
}

MapObject *mapObject;
CCARRAY_FOREACH(_mapObjects, mapObject)
{
    if ([self collisionBetweenAttacker:_hero
andObject:mapObject atPosition:&attackPosition])
    {
        HitEffect *hitEffect = [self getHitEffect];
        [_actors reorderChild:hitEffect
z:MAX(mapObject.zOrder, _hero.zOrder) + 1];
        [hitEffect showEffectAtPosition:attackPosition];

        if (mapObject.objectState != kObjectStateDestroyed)
        {
            //add this
            [[SimpleAudioEngine sharedEngine]
playEffect:@"hit1.caf"];
            [mapObject destroyed];
            Weapon *weapon = [self getWeapon];
            [weapon
droppedFrom:mapObject.contentSize.height/2 * kScaleFactor
to:ccp(mapObject.position.x, mapObject.position.y -
mapObject.contentSize.height/2 * kScaleFactor)];
            weapon.visible = YES;
        }
        else//add this
        {
            [[SimpleAudioEngine sharedEngine]
playEffect:@"hit0.caf"];
        }
    }
}
```

```
        return didHit;
    }
    else if (actionSprite == _boss)
    {
        if (_hero.actionState < kActionStateKnockedOut &&
        _hero.actionState != kActionStateNone)
        {
            CGPoint attackPosition;
            if ([self collisionBetweenAttacker:_boss
            andTarget:_hero atPosition:&attackPosition])
            {
                //add this
                [[SimpleAudioEngine sharedEngine]
                playEffect:@"hit1.caf"];

                [_hero knockoutWithDamage:_boss.attackDamage
                direction:ccp(actionSprite.directionX, 0.0)];
                [_hud setHitPoints:_hero.hitPoints
                fromMaxHP:_hero.maxHitPoints];
                didHit = YES;
            }

            DamageNumber *damageNumber = [self
            getDamageNumber];
            [damageNumber showWithValue:_boss.attackDamage
            fromOrigin:_hero.position];

            HitEffect *hitEffect = [self getHitEffect];
            [_actors reorderChild:hitEffect
            z:MAX(_boss.zOrder, _hero.zOrder) + 1];
            [hitEffect showEffectAtPosition:attackPosition];
        }
    }
    else
    {
        if (_hero.actionState < kActionStateKnockedOut &&
        _hero.actionState != kActionStateNone)
        {
            CGPoint attackPosition;
            if ([self collisionBetweenAttacker:actionSprite
            andTarget:_hero atPosition:&attackPosition])
            {
                //add this
                [[SimpleAudioEngine sharedEngine]
                playEffect:@"hit0.caf"];
            }
        }
    }
}
```

```

        [_hero hurtWithDamage:actionSprite.attackDamage
force:actionSprite.attackForce
direction:ccp(actionSprite.directionX, 0.0)];
        [_hud setHitPoints:_hero.hitPoints
fromMaxHP:_hero.maxHitPoints];
        didHit = YES;

        DamageNumber *damageNumber = [self
getDamageNumber];
        [damageNumber
showWithValue:actionSprite.attackDamage
fromOrigin:_hero.position];

        HitEffect *hitEffect = [self getHitEffect];
        [_actors reorderChild:hitEffect
z:MAX(actionSprite.zOrder, _hero.zOrder) + 1];
        [hitEffect showEffectAtPosition:attackPosition];
    }

}

return didHit;
}

```

All you did here was add some lines in the appropriate spots to play some sound effects.

Switch to **Hero.m** and do the following:

```

//add to top of file
#import "SimpleAudioEngine.h"

//add inside knockoutWithDamage:direction:, inside if
(self.actionState == kActionStateKnockedOut)
if (self.hitPoints <= 0)
{
    [[SimpleAudioEngine sharedEngine]
playEffect:@"herodeath.caf"];
}

```

Finally, do the following for both **Robot.m** and **Boss.m**:

```

//add to top of file
#import "SimpleAudioEngine.h"

```

```
//add this method
-(void)knockoutWithDamage:(float)damage
direction:(CGPoint)direction
{
    [super knockoutWithDamage:damage direction:direction];
    if (self.actionState == kActionStateKnockedOut &&
self.hitPoints <= 0)
    {
        [[SimpleAudioEngine sharedEngine]
playEffect:@"enemydeath.caf"];
    }
}
```

The above simply plugs in the appropriate sound effects for the death and hit events.

You're done! Build, run, and play until you drop!

## Finishing touches

I'm sure you can think of much more you want to add to your game, but once you've finished and your game is ready for the App Store, there are a few remaining things you should do.

First, you need to turn off Cocos2D's diagnostics found in the lower-left corner of the game screen.

To do this, open **AppDelegate.m** and look for this line:

```
[director_ setDisplayStats:YES];
```

Change it to this:

```
[director_ setDisplayStats:NO];
```

Next, Cocos2D has a switch that allows it to reduce any unnecessary OpenGL calls, giving your app a small boost in performance. You should enable this to get the most performance out of your app.

In the project navigator, go to **PompaDroid\libs\cocos2d\** and open **ccConfig.h**. Search for this line:

```
#define CC_ENABLE_GL_STATE_CACHE 0
```

Change it to this:

```
#define CC_ENABLE_GL_STATE_CACHE 1
```

And that's it! You now have a complete beat-em-up game – feel free to tweak or extend from here, or build your own game!

## Where to go from here?

Congratulations! You've completed your very first Beat 'Em Up Game!

It's been a long journey, but you made it.

Perhaps the best thing about what you've done is that you started with template classes for almost everything you created.

This means that, if you wish, you can add a lot to the game just by customizing your classes or creating new objects or characters based on the templates.

Here are some ideas just for starters:

- Create a varying decision-making AI using the `ArtificialIntelligence` class. All it takes is playing around with the weights of each possible action for a given situation.
- Create more levels in `Levels.plist`, and more maps using the Tiled Map Editor.
- Create more of everything else:
  - For characters and enemies, you have the `ActionSprite` class template.
  - For weapons, you have the `Weapons` class template.
  - For hit effects, you have `HitEffect` and `DamageNumber` classes.
  - For grouped actions, you have `AnimationMember` and `AnimateGroup`.
  - For map objects, you have the `MapObject` class template.

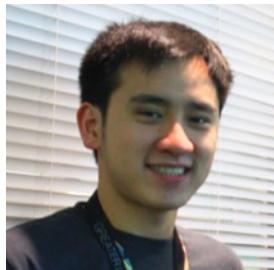
And of course, there are optional things you may want to add to improve the game:

- Create more actions – e.g., a double-tap up or down to make the hero jump tiles quickly.
- Create character and level selection screens.
- Improve the HUD by adding character portraits.
- Give the player multiple lives and/or access to objects that restore hit points.
- Better music and sound effects.

That's it. I wish you the best of luck in making your own game!

**Final Challenge:** Make your own beat-em-up game using some of the techniques you learned in this Starter Kit! :]

# Thank You!



I hope you enjoyed the Beat 'Em Up Game Starter Kit and had fun making this game. I can't thank you enough for your continued support of [raywenderlich.com](http://raywenderlich.com) and everything our team works on there.

I appreciate each and every one of you for taking time to try out the Beat 'Em Up Game Starter Kit. If you have an extra second, I would really love to hear what you thought of this Starter Kit!

Please leave a comment on the official private forums for the Beat 'Em Up Game Starter Kit at [www.raywenderlich.com/forums](http://www.raywenderlich.com/forums). If you do not have access to the forums, you can sign up here:

<http://www.raywenderlich.com/forum-signup>

Or if you'd rather reach me privately, please don't hesitate to shoot me an email. Although sometimes it takes me a while to respond, I do read each and every email, so please drop me a note!

Please stay in touch, and I look forward to checking out your Beat 'Em Up games!

Allen Tan

[allen@whitewidget.com](mailto:allen@whitewidget.com)