

Fully updated
for Sprite Kit!



BEAT 'EM UP GAME STARTER KIT

By Allen Tan

Beat 'Em Up Game Starter Kit

Allen Tan

Copyright © 2012, 2014 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.

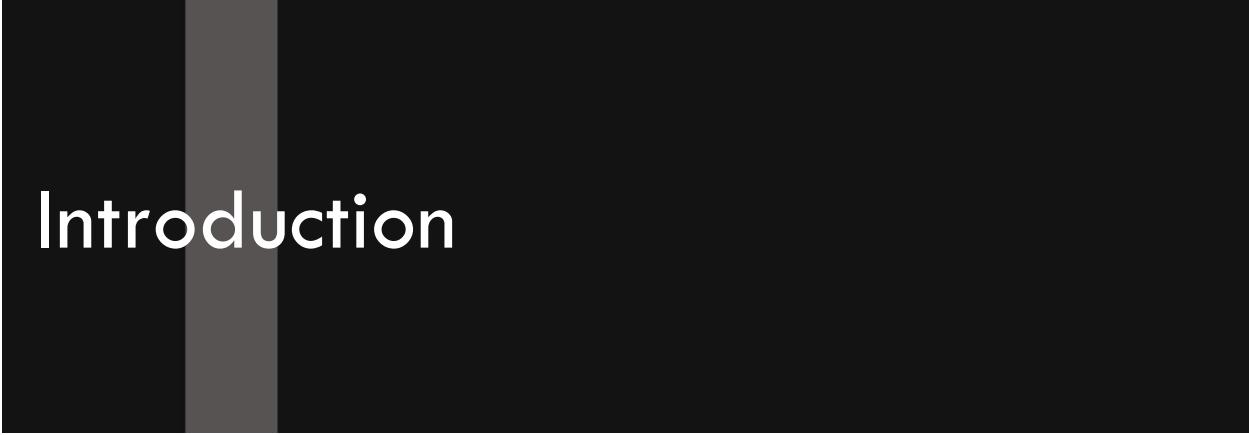
All trademarks and registered trademarks appearing in this book are the property of their respective owners.

Table of Contents

Introduction	5
Chapter 1: Getting Started.....	15
Chapter 2: Walk This Way	45
Chapter 3: Running, Jumping, and Punching	82
Chapter 4: Bring On the Droids.....	125
Chapter 5: Brainy Bots	163
Chapter 6: Power Attacks.....	199
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 hooks.

Beat 'em up games have been with us since the inception of 8-bit games, and had their share of the limelight back in the glory days of early console gaming. Lately, they aren't as common—but I believe that should change! You can't deny that beat 'em ups are great fun for many.

On the one hand, you have classic beat 'em ups that have stood the test of time. With a group of friends, one can still enjoy playing games like *Double Dragon*, *Teenage Mutant Ninja Turtles*, *Golden Axe* and *Streets of Rage*.

On the other hand, modern beat 'em ups appeal to a player's nostalgia while employing modern game technologies like player progression or a touch-based interface. *Castle Crashers* and *Scott Pilgrim vs. the World* are good examples.

If you have this kit in hand, it suggests you're no stranger to the genre and have probably heard of most, if not all of the titles referenced above—and that's why you want to make your own beat 'em up game!

That's where this starter kit comes into play. It 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 Sprite Kit.

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



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

The editorial team and I carefully crafted the chapters and source code in this book so that you'll learn about all of these features and more at a steady, logical and more importantly, a fun pace. I designed most of the things you'll create with reusability in mind so you can easily customize them for your own games.

Growing up during the glory days 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.

I sincerely hope that with the help of this starter kit, you'll soon make a knockout beat 'em up game of your own!

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 Sprite Kit but need guidance on applying their skills and knowledge to creating a beat 'em up game.

If you're a complete beginner, it's still possible to follow along with this book, because the chapters walk you through building the project in a step-by-step manner. However, to get the most from this book, I recommend first going through and completing the items listed in the Prerequisites section below.

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 on which to test your code.

This starter kit assumes you have a 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 Sprite Kit, which is the library you'll be using to make the game. If you are new to Sprite Kit, I recommend you go through our free Sprite Kit Tutorial For Beginners, available here:

<http://www.raywenderlich.com/42699/spritekit-tutorial-for-beginners>

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.

Introducing the second edition

It's been a little over a year since I first wrote the Beat 'Em Up Game Starter Kit, and a lot has changed since then!

When I first wrote the starter kit, I covered making the game with a popular 2D graphics framework called Cocos2D-iPhone. However, since then Apple has released its own 2D graphics framework called Sprite Kit.

When it comes to making 2D iPhone-specific games, I believe Sprite Kit is the way of the future, so this second edition is fully ported to Sprite Kit and iOS 7 as a free update to existing customers. This is my way of saying thank you for supporting our site and everything we do at raywenderlich.com.

If you've read a previous version of the Beat 'Em Up Game Starter Kit, I think you'll enjoy this Sprite Kit update. Since Sprite Kit has a built-in texture packer and particle system generator, there are no longer any third-party tool dependencies to create the game. In addition, Sprite Kit has greatly simplified much of the code—I think you'll find this version cleaner and easier to follow than earlier versions!

Note that I've also included the old second edition of the starter kit for the Cocos2D fans out there. However, from here on out we will no longer be supporting the Cocos2D version since we're moving to Sprite Kit.

I hope you all enjoy the third edition, and thank you again for purchasing the Beat 'Em Up Game Starter Kit! ☺

Prerequisites

To use this starter kit, you need to have a Mac with Xcode 5.0 or better installed. You also need an iPhone, iPod touch or iPad on which to test your code. While you could use the Simulator, the game's controls work much better when you can actually touch the screen with two hands.

You also need to have some basic familiarity with Objective-C. If you are new to Objective-C, I recommend you check out our epic-length tutorial for complete beginners called *iOS Apprentice: Getting Started*, which you can get for free by signing up for our site's newsletter here:

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

This starter kit also assumes you have some basic familiarity with Sprite Kit, which is the framework you'll be using to make the game. If you are new to Sprite Kit, I recommend you go through our free "Sprite Kit Tutorial for Beginners" series available on raywenderlich.com:

- <http://www.raywenderlich.com/42699/spritekit-tutorial-for-beginners>

Also, you might want to check out our book called *iOS Games by Tutorials*. It covers everything you need to know about the Sprite Kit framework. Along the way, you'll create five complete games from scratch, from a zombie action game to a top-down racing game. You can find the book here:

- <http://www.raywenderlich.com/store/ios-games-by-tutorials>

That said, if you are completely new to Objective-C and Sprite Kit, you can still follow along with this starter kit because everything is presented step by step. It's just that there will be some gaps in your knowledge that the above tutorials and books will fill in.

How to use this starter kit

There are several ways you can use the Beat 'Em Up Game Starter Kit:

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

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

- A second way to use the starter kit is to go through the chapters one by one and build the beat 'em up game from scratch. This is the best way to learn the topics in this book because you'll literally write each line of the main gameplay code, one small piece at a time.

- Third, you don't necessarily have to do each chapter—for example, if you already know how to do everything in Chapter 1, you can skip straight to Chapter 2. The Beat 'Em Up Game Starter Kit includes a version of the project for each chapter that contains all of the code from the previous chapters, so you can pick up the project at any point.

Starter kit overview

In this book, you'll create a complete beat 'em up game from scratch, each chapter building upon the last. Specifically, here's what you'll do in each chapter:

1. Getting Started

Approach your game the right way. In this first chapter, you'll learn how to create a Sprite Kit project as well as useful macros to help you code effectively.

While creating the title page, you'll get your first taste of pixel art, and through the use of macros, you'll learn how to support multiple devices with the same set of art.

Then, you'll set the stage for succeeding chapters using Sprite Kit transitions and TMX 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'll first create `ActionSprite`, the base model of all characters in the game. `ActionSprite` will be no ordinary sprite. It will move, act, and animate itself in the process through the use of a state machine. You'll use this model as the basis of your very first character, the Pompadoured Hero.

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

3. Running, Jumping and Punching

This chapter expands on the previous one with new actions for the hero. You'll begin by learning to simulate a directional double-tap gesture to make the hero run.

Building on your virtual control scheme, you'll create and add two custom buttons to make the hero perform other actions, such as jabbing. Then you'll give the game a 3-D feel using shadows and jumping, and also begin to build an army of robots to provide some much-needed drama.

These robots are the most interesting part of the chapter. Unlike the hero, who will have a single image to represent him, you'll assemble the robot's appearance piece by piece so you can easily make many colored variations of this robot.

As a part of this process, you'll learn how to animate a complex sprite by creating grouped animations and actions. Finally, to spice things up, you'll learn and apply the sprite tinting technique to achieve a seemingly unlimited number of color combinations for the robots.

4. Bring on the Droids

Your game can't be considered a beat 'em up without actual fist-throwing fights. On the way to implementing these fights, you'll learn to create your own collision detection system by marking various areas of your ActionSprites with circles. To aid you in your task, you'll also learn how to programmatically show these collision shapes through debug drawing.

With the circles in place, you'll 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'll work on artificial intelligence for the robots, implementing a system of weighted decisions that will determine each robot's action. This time, the robots won't stand idly by while the hero pummels them into junk—they will aggressively fight back.

To control the chaos that ensues, you'll enable your game to switch between scripted events, battle events and free-roaming events. You'll also learn to control and define enemy spawning through the use of property lists. This technique will pave the way for you to add more levels in the next chapter.

6. Power Attacks

Naturally, you'll want to trick out your hero with additional moves. In this chapter, you'll dig deep and learn to extend the current actions system by giving the hero the ability to do a chained attack with a 1-2-3 punch. You'll also give him combination actions, such as a jump punch and a running kick.

Afterward, you'll expand your use of property lists by adding support for multiple levels. Lastly, you'll learn a very important lesson about ARC and memory management when you experience memory issues firsthand. Be sure not to miss this important section!

7. The Droids Strike Back

You'll begin the chapter by creating HUD indicators to keep the player informed of such things as the hero's remaining hit points and when to move forward on the tile map. Then, you'll polish the game to a shine with floating damage numbers and explosive hit animations for each attack.

Finally, to balance the hero's newfound abilities from the previous chapter, you'll create stronger robots and introduce the hero's most formidable enemy, an ape-like boss with a high-top fade.

8. Final Encounter

It's the homestretch! In this chapter, you'll add the final pieces: power gauntlets that the hero can equip to rejig the odds in his favor, and destructible trashcans located across the map in which you'll stash these gauntlets for the hero to find.

To bring the game closer to a production-ready level, you'll add a scripted ending event and top everything off with gratuitous music and sound effects. By the end of this chapter, you'll have a complete 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 Beat 'Em Up Game Starter Kit here:

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

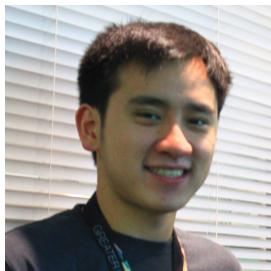
This is a great place to ask any questions you have about the book or about making beat 'em up games in general, and to submit any errata you may find.

Acknowledgements

I would like to thank several people for their assistance in 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 moral support and patience during those busy, work-focused days.
- And most importantly, **the readers of raywenderlich.com and you!** Thank you so much for reading our site and purchasing this starter kit. Your continued readership and support is what makes this all possible.

About the author



Allen Tan is the 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



Pietro Rea is a software developer specializing in mobile app development. He's worked on consumer facing iOS applications for AOL and Amazon and loves all things Apple. He lives in New York City with his wife and their Dachshund puppy. You can find him on Twitter as [@pietrorea](#).

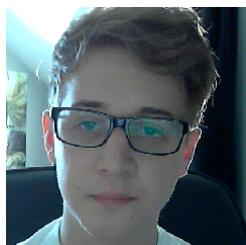


Bradley C. Phillips is an independent editor who splits his time between New York City and the northwestern Catskill Mountains. He has worked as a journalist, an EMT and a private investigator. For the present he is deeply involved in renovating a cabin and driving a monster truck. If you're in need of a skilled and experienced copywriter, copyeditor or proofreader, you can reach him at bradley.c.phillips@gmail.com.



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 many tutorials about iOS development available at <http://www.raywenderlich.com>.

About the artist



Hunter Russell is a pixel artist with seven years of experience. He has worked with many types of game developers and would love to help out with your next project. Have a look at his portfolio at <http://hunter.digitalhaven-ent.net> and his Tumblr at <http://professionalmanlyguy69.tumblr.com>, or contact him via email at hunter@digitalhaven-ent.net.

Chapter 1: Getting Started

This first chapter will get you well on your way to making your own beat 'em up game. 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 goes a lot deeper than the original tutorial and also contains major differences in approach. Even if you've completed the tutorial, you should still go through this chapter from the beginning!

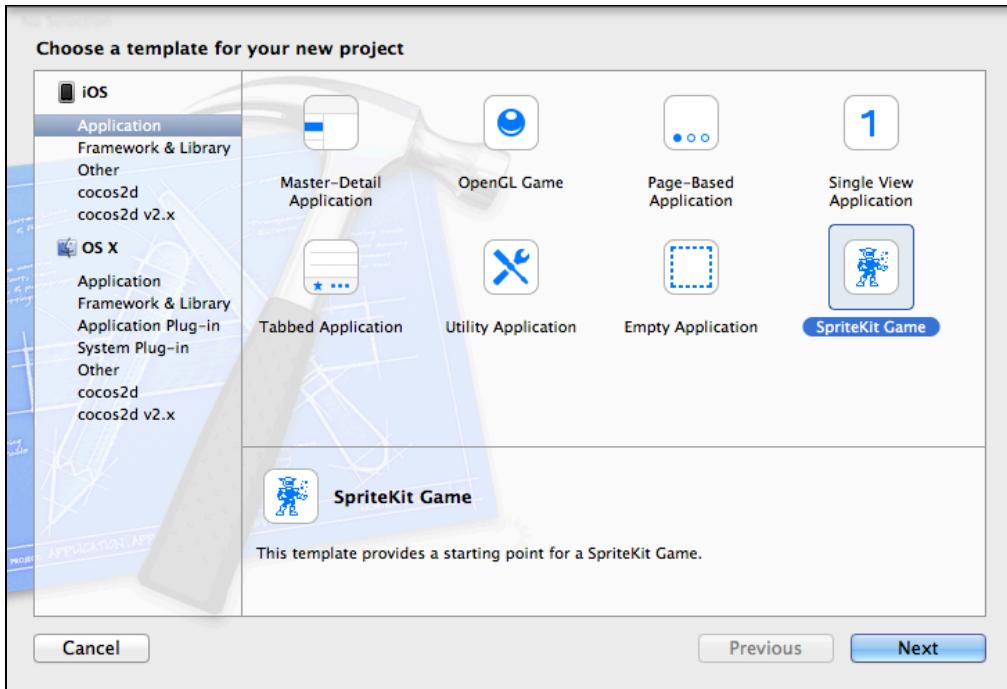
What are you waiting for? Whenever you're ready... get set, go! ☺

Note: If you already know how to create a Sprite Kit project, you might want to skip ahead to the section called "Creating a title scene" to get straight to the action. We'll have a starter project ready for you there.

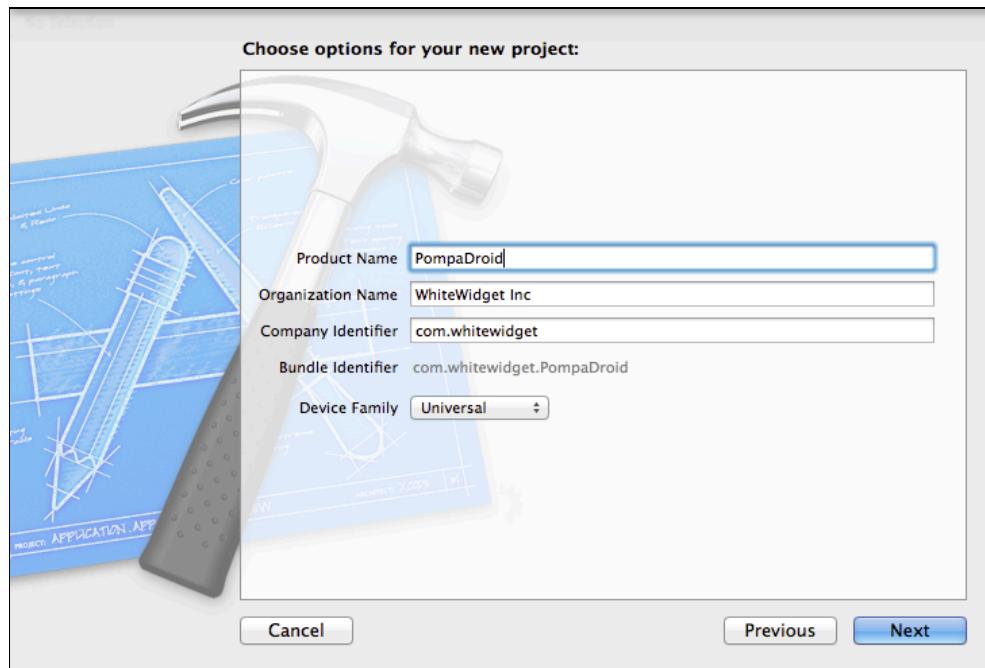
But if you're the type who likes to start from scratch to make sure you understand every single step, keep reading!

Creating a Sprite Kit project

Start up Xcode, go to **File\New\Project...** (Shift + Command + N) and choose the **iOS\Application\SpriteKit Game** 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” Sprite Kit template project. To see what it looks like, just build and run the project—in the left portion of your Xcode toolbar, choose iPhone Simulator from the Scheme dropdown and click the Run button. For now, use the iPhone Retina (3.5-inch) simulator as shown in the example below.



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



The numbers in the lower right are Sprite Kit’s diagnostic/statistic numbers. From left to right, these are:

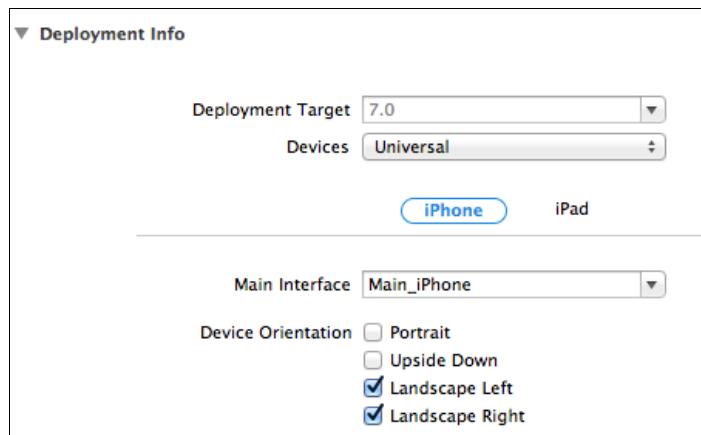
- **Node Count:** The number of Sprite Kit-related objects, called nodes, added to the current scene.
- **Frames Per Second (FPS):** Maxed at 60, this is the rate at which the display is refreshed every second. Higher means a faster refresh rate.

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

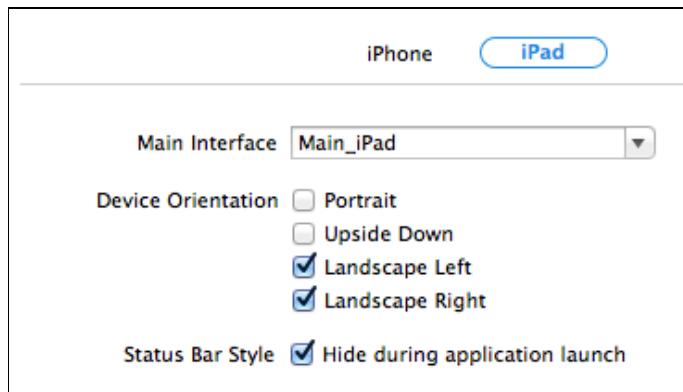
Sprite Kit supports both Portrait and Landscape orientations and projects begin with the former by default. For this beat 'em up game, you will only support the Landscape orientation.

In the Project Navigator, click on the project file and your center panel should show the General page. Make sure **PompaDroid** is highlighted under TARGETS and look under the **Deployment Info** section.

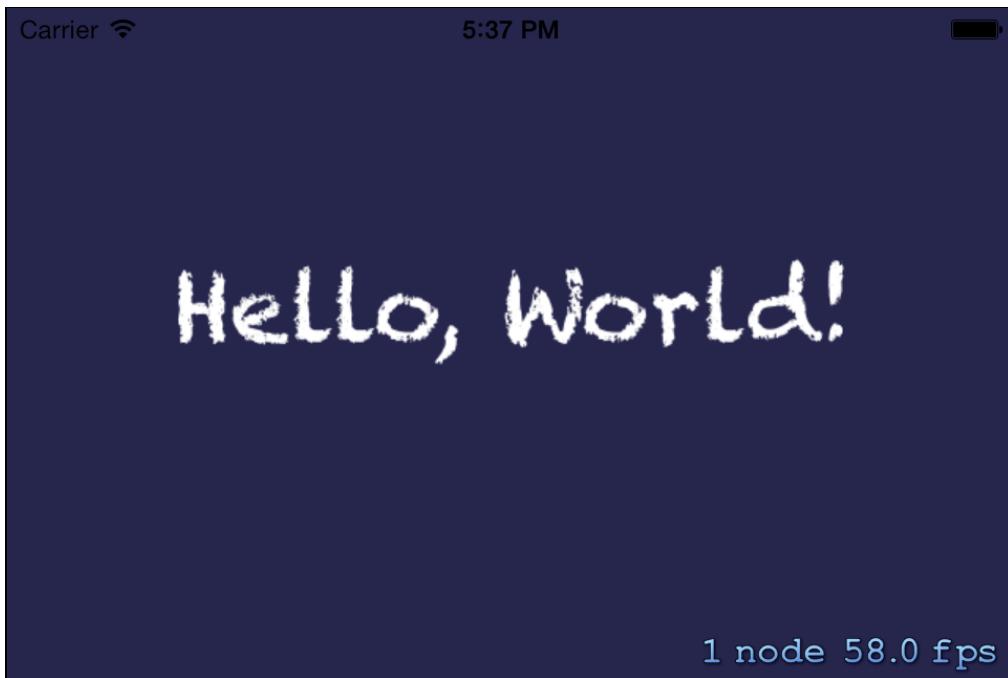
Select iPhone and unselect the checkbox next to Portrait. Your Deployment Info configuration should now look like this:



Now click on iPad and unselect the checkboxes next to both Portrait and Upside Down.

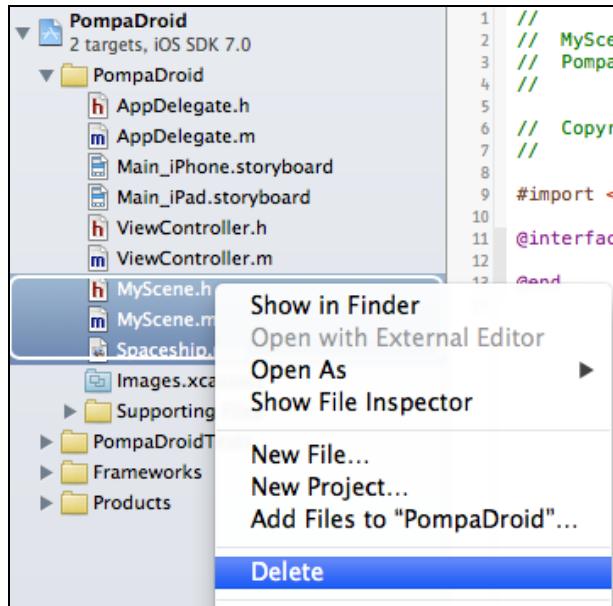


Build and run, and the game will begin in landscape mode.



The game now has the orientation you want, but there's still clutter left from the template. There's no room for a "Hello, World" message in the beat 'em up game you're going to build, so you have to clean up that screen.

The code responsible for the "Hello, World!" message is in **MyScene.h** and **MyScene.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. Do this for **Spaceship.png** as well - this is a test image that was added to the project for you automatically when you selected the Sprite Kit template, and you don't need it anymore.



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

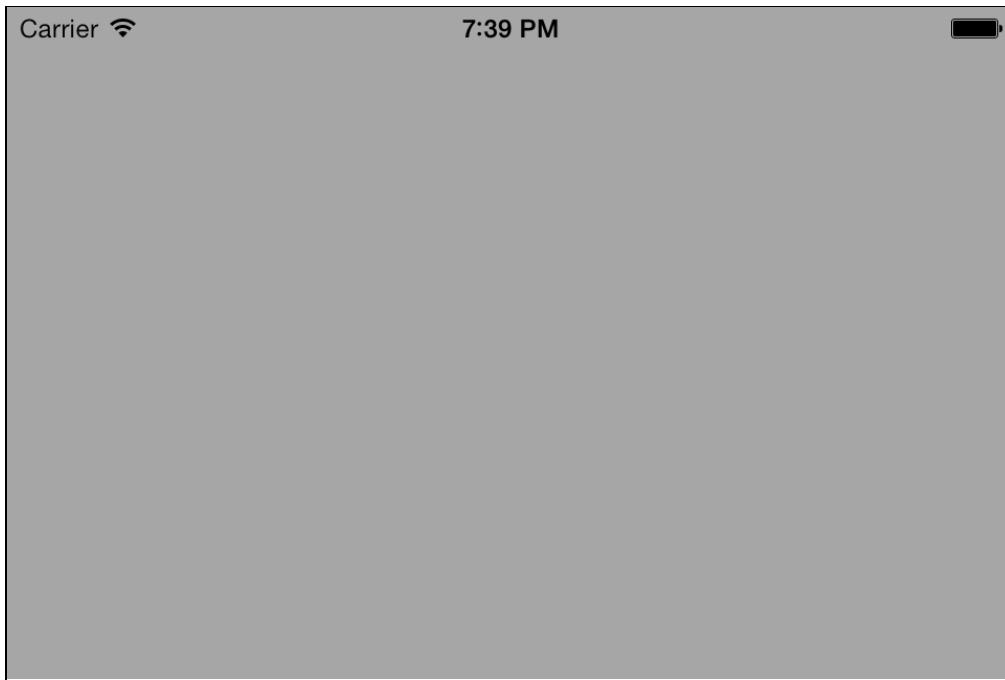
```
#import "MyScene.h"
```

Also remove these lines in `viewDidLoad`:

```
// Configure the view.  
SKView * skView = (SKView *)self.view;  
skView.showsFPS = YES;  
skView.showsNodeCount = YES;  
  
// Create and configure the scene.  
SKScene * scene = [MyScene sceneWithSize:skView.bounds.size];  
scene.scaleMode = SKSceneScaleModeAspectFill;  
  
// Present the scene.  
[skView presentScene:scene];
```

These lines presented the test scene created for you by the template. Since you removed the test scene, you need to delete these lines too. You'll be adding your own scene and code to present that soon.

Build and run to make sure everything still works. You should have a blank screen with only the status bar visible on top:



You're almost done preparing your blank canvas! The last step is to remove that status bar.

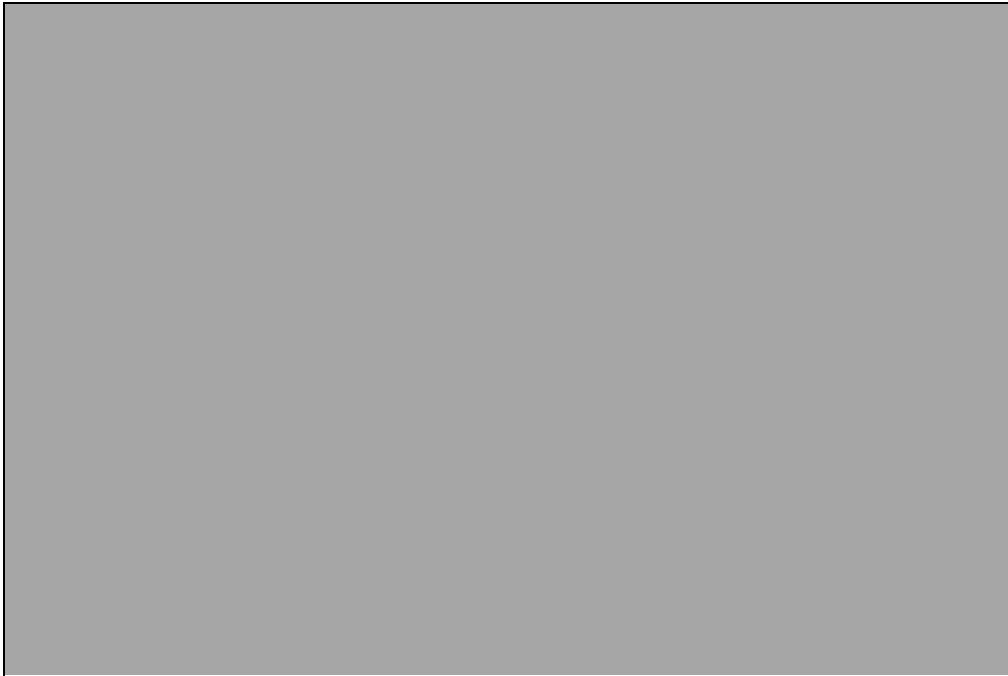
In **ViewController.m**, add this method right before `@end`:

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

Apple introduced this method in iOS 7. It allows you to show or hide the status bar in the `UIViewController` instance containing the code.

Quick and easy, right?

Build and run, and take a final look at the cleanest SpriteKit game ever.



And with that, you're done with the project setup—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 in the root folder as **PompaDroid – Starter Project.zip**—unzip it and open up the Xcode project inside.

Build and run, and you'll see it's simply an empty Sprite Kit template. Now you can get straight to filling it with code!

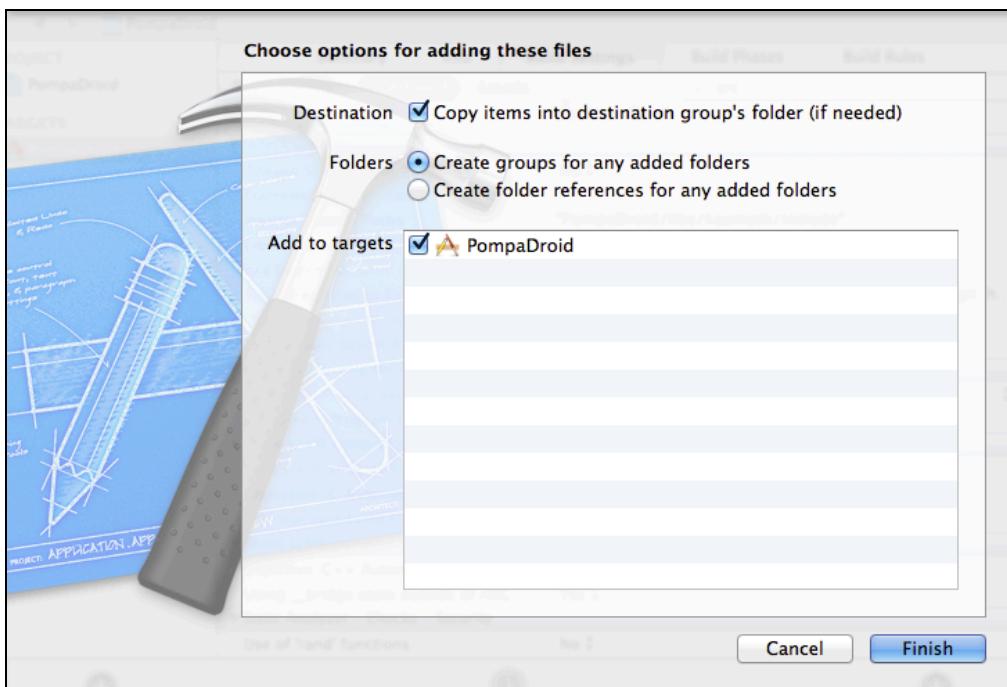
Your first challenge in building the game is to create a title screen. You don't want the game to drop the player into the middle of the action before introducing itself.

You don't need a fancy title screen, so you'll simply display a title and a start message.

Before you begin, you need the files containing the images you're going to use for the title scene. This starter kit includes a **Resources** directory that contains a subdirectory named **Images**.

Before you add the Images folder into your project, control-click (or right-click) on the **PompaDroid** group in Xcode, select **New Group** and give it the name **Resources**.

Now get 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**.



Going forward, make sure you apply the above options for all the files and folders you add to this project, except when otherwise specified.

Take a quick look through the files. These are the files you'll use for the title menu:

- **bg_title.png**: The background image for the title scene.
- **txt_title.png**: An image with the game's title, "PompaDroid".
- **txt_touchtostart.png**: An image with the text, "Touch to Start".

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

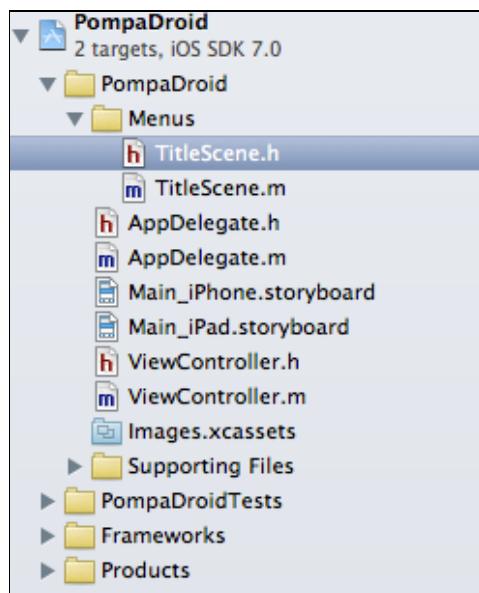
Note: I prepared these images in Adobe Photoshop, but I could very well have used a different image editor. There's nothing special about them—they are simply static images that you'll display as sprites. You can open **TitleScreen.psd** inside the **Raw** folder included in the starter kit to see how I created these images.

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

Now right-click (or control-click) on the newly-created **Menus** group, but instead of selecting New Group, this time select **New File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Name the new file **TitleScene**, make it a subclass of **SKScene** and click **Next** again.

Note: An **SKScene**, which we'll refer to simply as a scene (also known as a screen or a stage), is crucial to Sprite Kit'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*. The active **SKScene** controls game events and all the active nodes (**SKNodes**) contained within it.

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



Now that **TitleScene** exists, you can take steps to transition the game into this scene. By default, it is **ViewController**'s job to handle scene presentation.

Go to **ViewController.m** and add this to the top of the file:

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

This allows you to call instances of **TitleScene** from **ViewController.m**. You want **ViewController** to present **TitleScene** as the first scene.

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

```
//1
-(void)viewWillLayoutSubviews
{
    [super viewWillLayoutSubviews];

    SKView *skView = (SKView *)self.view;

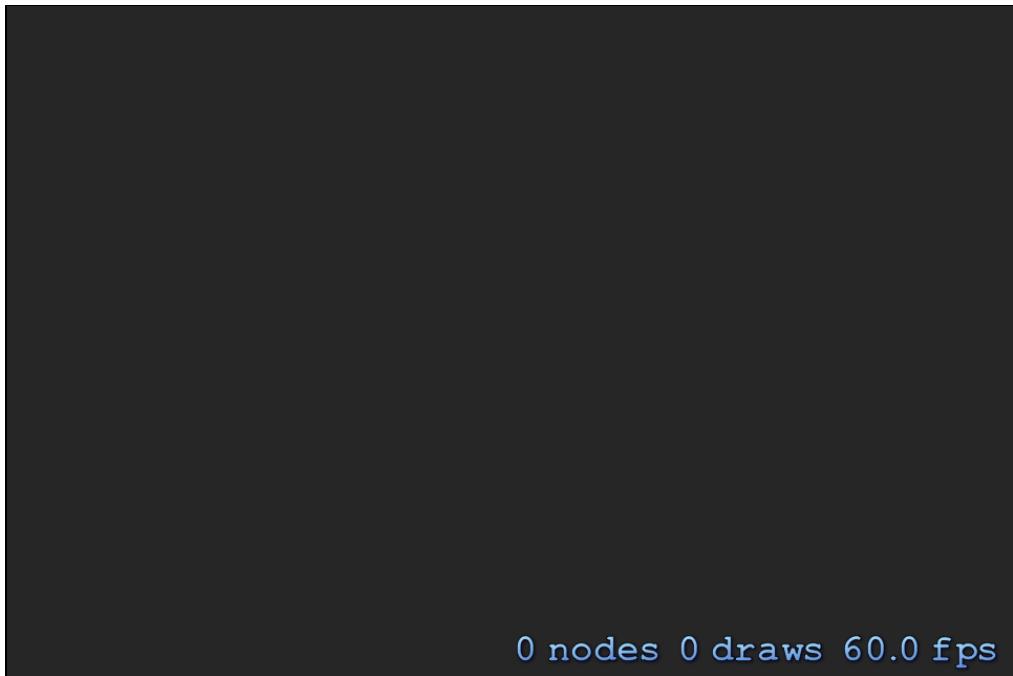
    //2
    if (!skView.scene)
    {
        skView.showsFPS = YES;
        skView.showsNodeCount = YES;
        //3
        skView.showsDrawCount = YES;

        //4
        SKScene *titleScene = [TitleScene
sceneWithSize:skView.bounds.size];
        [skView presentScene:titleScene];
    }
}
```

Remember the lines of code you removed from **ViewController** earlier? Well, you just rewrote them, but with a few key differences:

1. You wrote the code in **viewWillLayoutSubviews** instead of **viewDidLoad**. By default, **ViewController** loads in portrait orientation and only changes to landscape orientation after the view has already loaded. The dimensions (width and height) of the view depend on the view's orientation. Since **viewWillLayoutSubviews** executes after the orientation change, you can be certain that the dimensions of the view are correct at this stage.
2. Unlike **viewDidLoad**, **viewWillLayoutSubviews** can be executed more than once, so you add a condition to only execute the code if the view does not have a scene yet.
3. The first two properties of **SKView** show or hide the FPS and node count discussed earlier. You add a third statistic: the draw count, which is the number of OpenGL draws on the screen. There is usually one draw call per object displayed on the screen, but you can reduce this number by using Sprite Kit's batch rendering, which you'll learn about later on. Suffice it to say, the more draw calls, the greater the risk of poor performance due to a heavy processor load.
4. Instead of the now-forgotten **MyScene**, you create a **TitleScene** that is as big as the **ViewController**'s view. **presentScene** tells the view to set **TitleScene** as the active view and display its contents on the screen.

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



It's a whole lot of nothing, but that's what you want—a fresh, clean scene to work with. ViewController's responsibility ends with presenting the correct SKScene. It is the SKScene's job to provide the game with appearance and behavior, so it's time to work with TitleScene.

Go to **TitleScene.m** and add the following methods inside **@implementation**:

```
- (instancetype)initWithSize:(CGSize)size
{
    if (self = [super initWithSize:size])
    {
        [self setTitle];
    }

    return self;
}

- (void)setTitle
{
    SKSpriteNode *titleBG =
    [SKSpriteNode spriteNodeWithImageNamed:@"bg_title"];

    titleBG.position = CGPointMake(CGRectGetMidX(self.frame),
                                   CGRectGetMidY(self.frame));

    [self addChild:titleBG];
}
```

You call `setTitle` from `initWithSize`. In here, you create an `SKSpriteNode` using the background image you added previously, position it at the center of the scene and add it as a child of the layer.

SKSpriteNodes, 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: Sprite Kit 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 scene 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.



It's a good thing there's a way to define shortcuts! Let's make this 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 the checkbox **PompaDroid** in **Targets** is un-ticked. 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 BOUNDS [[UIScreen mainScreen] bounds].size
#define SCREEN (UIInterfaceOrientationIsPortrait([UIApplication
sharedApplication]. statusBarOrientation) ? CGSizeMake(BOUNDS.width,
BOUNDS.height) : CGSizeMake(BOUNDS.height, BOUNDS.width))
#define CENTER CGPointMake(SCREEN.width * 0.5, SCREEN.height * 0.5)
#define OFFSCREEN CGPointMake(-SCREEN.width, -SCREEN.height)
```

```
#define IS_RETINA() ([[UIScreen mainScreen]
    respondsToSelector:@selector(scale)] && [[UIScreen mainScreen] scale] ==
2)
#define IS_IPHONE5() ([UIScreen mainScreen].bounds.size.height == 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:

- **BOUNDS**: The screen's dimensions in points in the default portrait orientation.
- **SCREEN**: The screen's dimension in points based on the current device orientation.
- **CENTER**: The center of the screen in points based on the current device orientation.
- **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-inches 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 you work on from here on out, but it's a hassle 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 every file in your project automatically imports without you needing to do a thing. You can import your **Defines.h** in this file to make it available everywhere as well.

In the **Supporting Files** group, open **PompaDroid-Prefix.pch** and do the following:

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

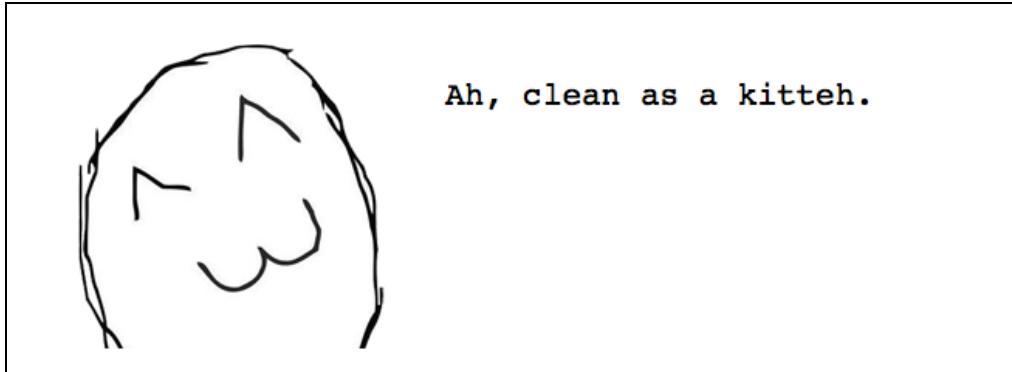
Build so that **Defines.h** gets populated throughout your project, but don't run just yet.

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

```
- (void)setupTitle
{
    SKSpriteNode *titleBG =
    [SKSpriteNode spriteNodeWithImageNamed:@"bg_title"];
    titleBG.position = CENTER;
```

```
[self addChild:titleBG];  
}
```

As you can see, you've just simplified the code by making use of the **CENTER** macro. Now doesn't that look pretty? ☺



In Xcode, select either the iPhone Retina (3.5-inch) or iPhone Retina (4-inch) simulator. I'll explain why you've chosen this iPhone simulator soon.

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



There's one little but important detail that remains: you need to 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]
```

```
SKSpriteNode *title = [SKSpriteNode  
spriteNodeWithImageNamed:@"txt_title"];  
title.position = CGPointMake(CENTER.x, CENTER.y + 66);  
[self addChild:title];  
  
SKSpriteNode *start = [SKSpriteNode  
spriteNodeWithImageNamed:@"txt_touchtostart"];  
start.position = CGPointMake(CENTER.x, CENTER.y - 37.5 );  
[self addChild:start];
```

That's simple enough: You add the remaining two images as sprites and position them from the center.

Build and run the game, and you will see:



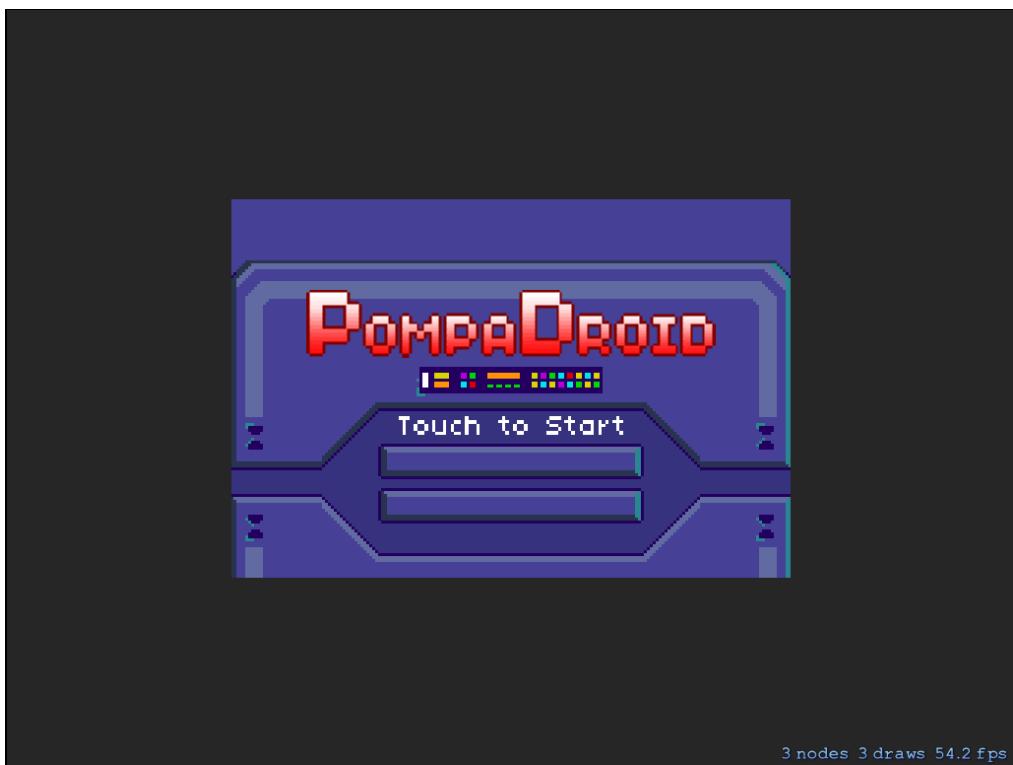
Note: The two sprites appear on top of the background image because in the code, you've added them after the background image. Optionally, you could add these sprites first and then manipulate the `zPosition` property to change the order in which the sprites are drawn. Keep this in mind—you may need this information for your own projects!

Supporting universal display

In the previous section, I specifically asked you to choose an iPhone simulator because, well, you hadn't done anything to support the iPad yet.

If you tried to run the game on an iPad, you would get very different results.

For example, if you ran it on an iPad simulator, you would see the following:



In Sprite Kit, by default each device resolution needs its own version of an image, with different suffixes to signify their purpose. Taking **bg_title.png** as an example, to support all display types, there should also be the following:

- **bg_title@2x.png** for iPhone retina.
- **bg_title~ipad.png** for iPad non-retina.
- **bg_title@2x~ipad.png** for iPad retina (2x the size of the iPad non-retina scale).

This would be true for all images supporting universal display.

If a retina version of an image is missing, Sprite Kit will double the scale of the non-retina version. In this case, Sprite Kit has automatically doubled the size of **bg_title.png** for you.

Note: You do not necessarily need to include all of these options if you can figure out a way to reuse images. For example, in our games at White Widget 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 4-inch iPhone we often use the same sprites as for the 3.5-inch iPhone, and just include different background images that are wider.

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—it also saves memory and can earn you style points. 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. Then you handle the scaling in the program itself. This will help you keep texture memory at a minimum.

The downside to this strategy, though, is that you can't take direct advantage of Sprite Kit automatically selecting art with the proper resolution for each device.

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

```
if (IS_IPAD())
{
    [titleBG setScale:2.0];
    [title setScale:2.0];
    [start setScale:2.0];
}
```

If the device is an iPad, you double the scale. However, it'd be a bit tedious to do this check every time you create a sprite. Fortunately, there is a cleaner way that involves defining some macros.



Switch to **Defines.h** and add this line right after the existing code, but above the final `#endif`:

```
#define kPointFactor (IS_IPAD() ? 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 this kind of syntax, but that's why you put it in **Defines.h**. ☺

You've defined a very useful macro in `kPointFactor`. Sprite Kit only does automatic point-to-pixel conversion when going from non-retina to retina, so you will need this to convert points from iPhone to iPad.

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

```
- (void)setupTitle
{
    SKSpriteNode *titleBG =
    [SKSpriteNode spriteNodeWithImageNamed:@"bg_title"];

    titleBG.position = CENTER;
    [self addChild:titleBG];

    SKSpriteNode *title =
    [SKSpriteNode spriteNodeWithImageNamed:@"txt_title"];

    title.position =
    CGPointMake(CENTER.x, CENTER.y + 66 * kPointFactor);
    [self addChild:title];

    SKSpriteNode *start =
    [SKSpriteNode spriteNodeWithImageNamed:@"txt_touchoftostart"];
```

```
start.position =  
CGPointMake(CENTER.x, CENTER.y - 37.5 * kPointFactor);  
[self addChild:start];  
  
[titleBG setScale:kPointFactor];  
[title setScale:kPointFactor];  
[start setScale:kPointFactor];  
}
```

The new code first multiplies the position values by **kPointFactor** to adjust them for iPad and then it sets the scale of all sprites to **kPointFactor** to size them accordingly.

In Xcode, select the iPad simulator and then build and run your project. It should now look like this:



Try it for the remaining devices: iPhone Retina (4-inch) and iPad Retina.

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

568 pixels is the longest width you need, because 568 pixels is the length of the widest iOS device to date (the 4-inch iPhone). Doubling this value results in 1136, which is the 4-inch iPhone's pixel width, even longer than the iPad's pixel width of the 1024.

384 pixels is the longest height you need, because doubling this results in 768, which is the pixel height of the tallest iOS device to date (the iPad).

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



This shows how the background image will look on each iOS device, and compares the screen size to the background image size. Note that the background image is sometimes slightly wider and/or taller than the screen size, but that is OK because our artist designed the areas along the side and top of the image to be non-critical. It's often more convenient to have some of the background image cut off and be able to reuse the same background image everywhere, instead of having to make a different background image for each device type.

From now on, whenever you build and run, you can test on any device(s) you like. For the remainder of this project, I will be using the iPhone Retina (4-inch) simulator for all examples.

You may have noticed one minor problem: the art seems blurred at higher resolutions. This is because by default, when Sprite Kit scales up an image, it applies anti-aliasing, which is a technique to diminish jagged pixels in an image.

However, this has the effect of making your pixel art blurry – which is completely the opposite of what you want in a pixel art game.

To disable anti-aliasing, add these lines at the end of `setupTitle`:

```
titleBG.texture.filteringMode = SKTextureFilteringNearest;
title.texture.filteringMode = SKTextureFilteringNearest;
start.texture.filteringMode = SKTextureFilteringNearest;
```

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



It may be hard to detect a change in this screenshot or on the Simulator, but if you have a retina device, I recommend running the game with and without the above lines of code so you can see the difference. You should witness a definite improvement.

Now you know how to implement pixel art for all devices.

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\Cocoa Touch\Objective-C class** template and click Next. Enter **SKScene** for Subclass of, name the new file **GameScene** and click Next.

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

Open **GameScene.m** and add this to the top of the file:

```
#import "GameLayer.h"  
#import "HudLayer.h"
```

Still in **GameScene.m**, add this inside the `@implementation`:

```
- (instancetype)initWithSize:(CGSize)size  
{  
    if (self = [super initWithSize:size])  
    {  
        GameLayer *gameLayer = [GameLayer node];  
        [self addChild:gameLayer];  
  
        HudLayer *hudLayer = [HudLayer node];  
        [self addChild:hudLayer];  
    }  
    return self;  
}
```

The code creates one instance of **GameLayer** and one instance of **HudLayer**, and adds them to the **GameScene**.

As you may have guessed from the names, **GameLayer** will contain the 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.

Next, you'll make the game switch to this new scene when the user taps the main menu. Go back to **TitleScene.m** and add this to the top of the file:

```
#import "GameScene.h"
```

Still in **TitleScene.m**, add this inside the `@implementation`:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event  
{  
    GameScene *gameScene = [GameScene sceneWithSize:self.size];  
    SKTransition *fade = [SKTransition fadeWithDuration:1.0];  
  
    [self.view presentScene:gameScene  
                  transition:fade];  
}
```

When the user touches the screen, the game fires the `touchesBegan` method. Here, you create a transition to **GameScene** by telling the presenting view of **TitleScene** to fade into **GameScene**.

Before you check your work, why not add a bit of life to **TitleScene** by making the "Touch to Start" sprite blink? That sounds great, but there's a slight snag. Unlike

Cocos2D, Sprite Kit doesn't come packaged with a simple way to make a sprite blink.

Not to worry. The raywenderlich.com team has you covered.

Introducing SKTUtils

Some functions, like the aforementioned blink, are commonly found in games. Rather than make you continuously rewrite these in each game you create, we've gathered them all together into a handy little library called **SKTUtils**.

If you've read *iOS Games by Tutorials*, then you're already familiar with **SKTUtils**. This starter kit contains an expanded version of the library with added features that you'll use throughout this book.

You can find **SKTUtils** in the root folder for this book. Drag the entire **SKTUtils** folder into the Project Navigator in Xcode. Make sure **Copy items into destination group's folder (if needed)** and the **PompaDroid target** are both checked. Click **Finish**.

Even if you've used **SKTUtils** before, peek around the contents of the library to get a feel for what's inside. I'll inform you each time you use something from this library at different points in the book.

To get your first taste of **SKTUtils**, open **TitleScene.m** and add the following:

```
//add to top of file
#import "SKAction+SKTEExtras.h"

//add these lines at the end of setupTitle
start.name = @"StartText";
SKAction *blinkAction = [SKAction blinkWithDuration:5.0 blinks:10];
SKAction *repeatAction = [SKAction repeatActionForever:blinkAction];
[start runAction:repeatAction];
```

First you give the start sprite a name. A name is just a string that you associate with a sprite so you can easily find it again. A name can be handy if you want a quick way to get a reference to a sprite that you added to the layer in another method. Alternatively, you can create an instance variable or property to keep track of the sprite, but sometimes naming it is easier.

Next, you create a few Sprite Kit actions called **SKActions**. 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 Sprite Kit node will perform the action.

You can create the above **blinkAction** thanks to **SKAction+SKTEExtras.h**, which contains additional action methods for **SKAction**. **blinkAction** tells the node performing the action to blink ten times in five seconds, while **repeatAction** tells the node to repeat the inner action until it is told to stop.

Now add these lines at the end of **touchesBegan**:

```
//add these lines at the beginning of touchesBegan
SKSpriteNode *start = (SKSpriteNode *)[self
childNodeWithName:@"StartText"];
[start removeAllActions];
start.hidden = YES;
```

When the user taps the screen, `start` should disappear. You assigned `start` a name of "StartText" in `setupTitle`, and retrieved `start` again in `touchesBegan:` using its name. As mentioned before, in your own games you may want to do this using a property or instance variable. You employ the `name` property here to illustrate how to use `childNodeWithName:`.

Once you have a reference to the start button node, you use `removeAllActions` to stop the blinking and also make the node invisible.

Note: The `SKSpriteNode *` inside the parenthesis tells the compiler that whatever the method returns is going to be an object of type `SKSpriteNode`. 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, `childNodeWithName` returns an object of type `SKNode`, and `SKSpriteNode` is a subclass of `SKNode`.

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



Onward... into nothingness!

Before moving on, make sure to include `SKTUtils` to your pre-compiled header file so you don't have to worry about importing it later on.

Open `PompaDroid-Prefix.pch` and add the following:

```
//add after #import "Defines.h"  
#import "SKTUtils.h"
```

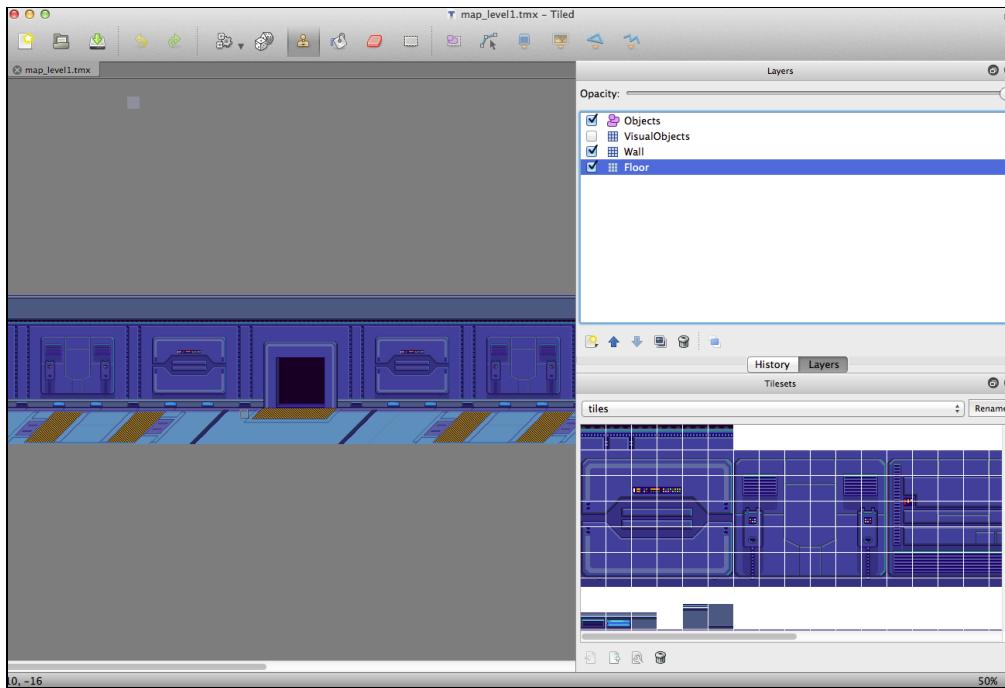
Loading the tile map

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

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

- **tiles.png**: The tiles that make up 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.

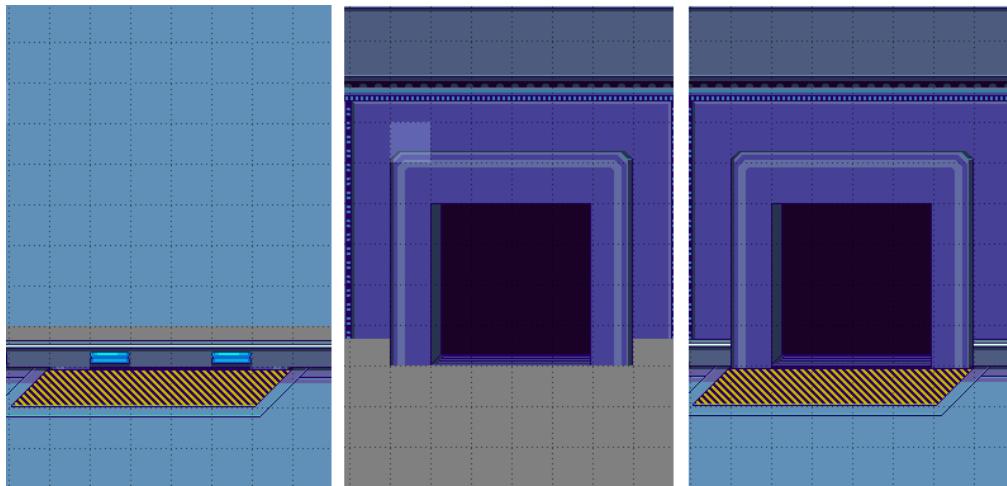
The tile map was made using the Tiled map editor, available at <http://www.mapeditor.org>. If you have this application installed, open **map_level1.tmx** to see what's inside:



Looking at this map, there are a few important things to keep in mind:

- 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. To see tile rows, enable the grid view by selecting **View\Show Grid** from the menu. You will see that the fourth row of tiles from the bottom 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 tile variations by combining two in one spot.



- The walkable floor tiles are in the first three rows from the bottom.
- The **Objects** layer is responsible for determining where to place game objects like punchable trash cans. You'll write some code that reads the location of these objects to place game objects there later.
- The **VisualObjects** layer is just meant to be a preview of what the game will look like after you write the code to place the game objects using the Objects layer. Since it's a preview only, that's why it's turned off.

Note: If you want to know more about creating tile maps, check out Chapters 14-16 of *iOS Games by Tutorials*, "Beginning Tile Maps", "More Tile Maps", and "Imported Tile Maps."

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

As of the time of writing, Sprite Kit doesn't have built-in support for TMX files. There are hints of private APIs in the code that make me believe the developers at Apple are working on it, but right now, you have no choice but to either code TMX tile map support yourself or use a third-party library for it.

The open source community has been working hard on various solutions of its own, and for PompaDroid, you will use the solution created by Jeremy Stone, JSTileMap.

JSTileMap is a custom subclass of SKNode that renders TMX files in Sprite Kit. It parses a TMX file and instantiates various objects to model the contents of the file, such as TMXLayers for each tile layer and SKSpriteNodes for each tile. It's a work in progress, so these things may change in the future.

Note: Because an open source project like **JSTileMap** can change quickly and often, you're not going to download the latest code to follow along with this chapter.

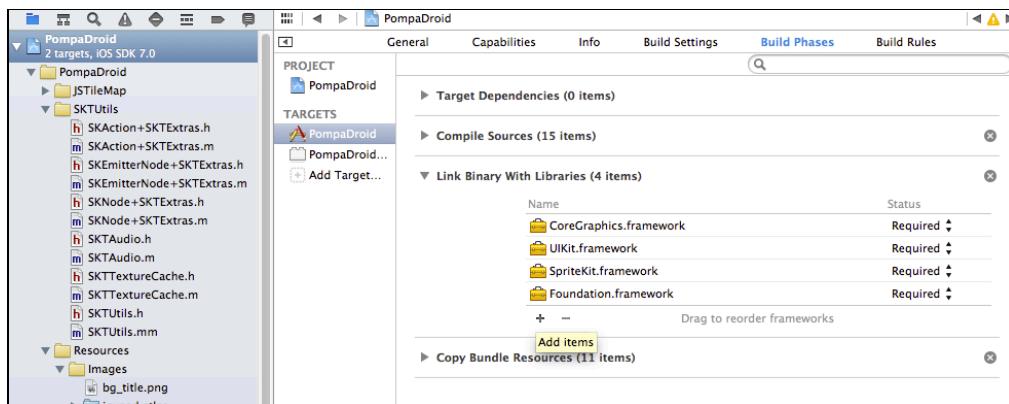
Instead, we've included a folder in this chapter's resources named **JSTileMap** that includes a version of the code that we know works at the time of this writing. Of course, after going through the chapter, you are encouraged to download the latest version of **JSTileMap** to use in your own apps.

This chapter won't be covering the inner workings of **JSTileMap** and its supporting classes, so you should take a look at the source code if you want a detailed understanding of how it works.

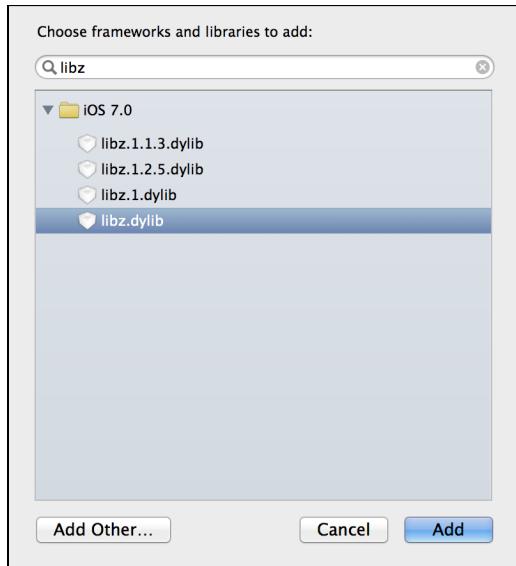
Find the **JSTileMap** folder in the root folder for this book. Inside Xcode, choose **File\Add Files to "PompaDroid"...** and select the **JSTileMap** folder. Be sure **Copy items into destination group's folder (if needed)**, **Create groups for any added folders** and the **PompaDroid** target are all checked, and then click **Add**.

Your app would fail to build if you tried it right now, because part of what you just imported was a compression utility named **LFCGzipUtility** that requires you to link your app with an additional library. You need this compression utility because Tiled compresses the map data to make TMX files smaller.

Choose the **PompaDroid** project in the Project Navigator and then choose the **PompaDroid** target. Inside the **General** tab, click the **+** button in the section labeled **Linked Frameworks and Libraries**, as shown below:



Select the library named **libz.dylib** and click **Add**.



Build your project to make sure everything compiles OK.

Now that `JSTileMap` is in place, you can get back to coding. Open `GameLayer.h` and add the following:

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

//add below the @interface declaration
@property (strong, nonatomic) JSTileMap *tileMap;
```

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

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

- (void)initTileMap:(NSString *)fileName
{
    self.tileMap = [JSTileMap mapNamed:fileName];
    [self.tileMap setScale:kPointFactor];
    [self addChild:self.tileMap];
}
```

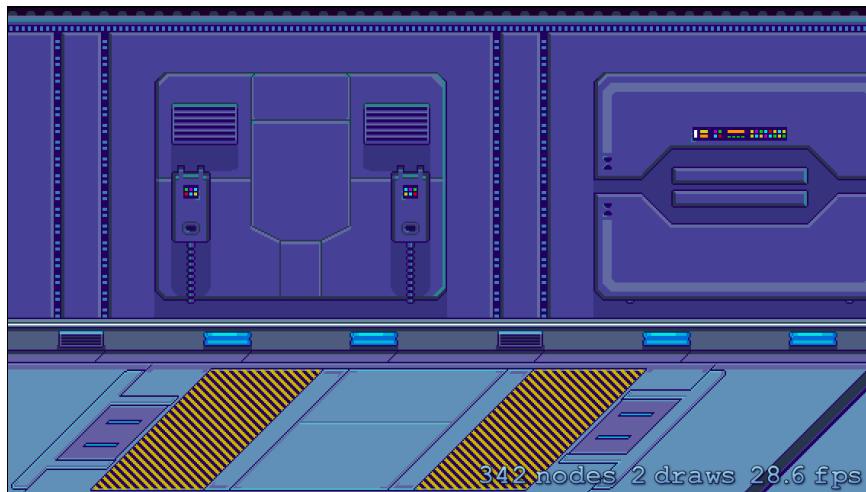
You create a `tileMap` property so you can access the tile map from anywhere in the class. You'll use this later to check the properties of the tile map. You could also access the tile map using the `name` attribute as you did before, but when you need

to reference an object frequently, it's always better to use a property because it's faster and more convenient.

initTileMap: creates the tile map based on the filename provided. The method makes sure the tile map scales according to the device by setting the map's scale property to **kPointFactor**.

Finally, you add the tile map to the `GameLayer` node.

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 beginning of your first level!

At this point, you've accomplished a lot. You've:

- Created a Sprite Kit project;
- Constructed a main menu with some simple Sprite Kit actions;
- Enabled a transition between two different scenes;
- And you've 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—funky hair-do and all.

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 use the Wall and Floor layers to modify the beginning of the level so that it has a slightly different look. Build and run, and make sure your changes appear as intended.

Chapter 2: Walk This Way

Now that your game has a map, how about adding a hero?

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

An iOS beat 'em up game typically requires you to do the following:

- Display a joypad control on the screen;
- Enable moving the hero around the screen with the joypad;
- Animate the hero as s/he moves;
- Scroll the layer as the hero moves.

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

Walking in style

Before you start adding the joypad to the game and allowing the hero to move, you're going to work on the animation system for the hero. That way, when you do add the joypad control, your hero can move in style.

After all, it's easy enough to add a hero, but hardly worthwhile unless your hero looks alive rather than wooden—and that calls for animations.

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

But knowing how to run an animation isn't enough—you also need to know the proper time to play it. 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 way to transition between a sprite's animations is to use a state machine.

A state machine is simply an algorithm that keeps track of an object's state and executes different code based on what state it is in.

A single state machine can have only one active state at any given 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. Let's say there are only three:

- Stay 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 s/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 a coherent 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: When is it appropriate to play a given animation? For the game's implementation, when you switch states, you will also switch the character's animation.

That's enough theory for now. It's time to get back to 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 chapter, since each action represents a state, I'll use the words **action** and **state** interchangeably.

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

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

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

Here you add an enumeration and a structure:

1. `ActionState` is an enumeration of the states that your sprite class will have. You haven't created the class yet, but you're about to. These states 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\Cocoa Touch\Objective-C class** template and click **Next**. Enter **SKSpriteNode** 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 (strong, nonatomic) SKSpriteNode *shadow;

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

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

//attributes
@property (assign, nonatomic) CGFloat walkSpeed;
@property (assign, nonatomic) CGFloat runSpeed;
@property (assign, nonatomic) CGFloat hitPoints;
@property (assign, nonatomic) CGFloat attackDamage;
@property (assign, nonatomic) CGFloat jumpAttackDamage;
@property (assign, nonatomic) CGFloat runAttackDamage;
@property (assign, nonatomic) CGFloat maxHitPoints;
@property (assign, nonatomic) CGFloat 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 `SKActions` to be executed for each state. The `SKActions` will be a combination of executing sprite animations and other events triggered when the character switches states.
3. **States:** `actionState` 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:** These properties define the character's statistics, such as speed, hit points and the damage and knockback force inflicted by the character's attacks.

But wait, there's more!



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

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

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

//collision
@property (strong, nonatomic) NSMutableArray *contactPoints;
@property (strong, nonatomic) NSMutableArray *attackPoints;
@property (assign, nonatomic) CGFloat detectionRadius;

- (CGRect)feetCollisionRect;
```

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

1. **Movement:** These are dynamic values you'll use 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 your sprites will have

a canvas size that is much larger than the image contained inside so that the sprite frames inside an animation line up with each other.

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**:

```
// updates
- (void)update:(NSTimeInterval)delta;

// action methods
- (void)idle;

- (void)attack;

- (void)hurtWithDamage:(CGFloat)damage
                 force:(CGFloat)force
                direction:(CGPoint)direction;

- (void)knockoutWithDamage:(CGFloat)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)exitFrom:(CGPoint)origin
               to:(CGPoint)destination;
```

```
- (void)jumpRiseWithDirection:(CGPoint)direction;
- (void)jumpCutoff;
- (void)jumpFall;
- (void)jumpLand;
- (void)jumpAttack;
- (void)runAttack;
- (void)reset;

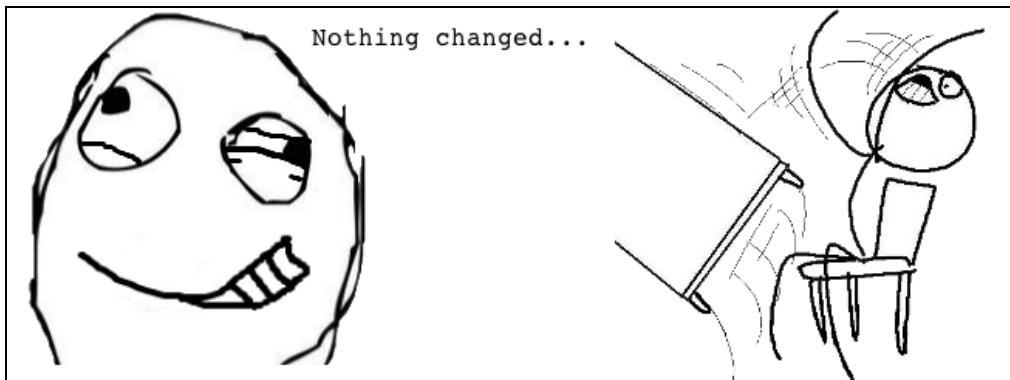
//contact point methods
- (NSMutableArray *)contactPointArray:(NSUInteger)size;
- (void)modifyContactPointAtIndex:(const NSUInteger)pointIndex
                           offset:(const CGPoint)offset
                           radius:(const CGFloat)radius;
- (void)modifyAttackPointAtIndex:(const NSUInteger)pointIndex
                           offset:(const CGPoint)offset
                           radius:(const CGFloat)radius;
- (void)modifyPoint:(ContactPoint *)point
                           offset:(const CGPoint)offset
                           radius:(const CGFloat)radius;
- (ContactPoint)contactPointWithOffset:(const CGPoint)offset
                           radius:(const CGFloat)radius;
- (void)setContactPointsForAction:(ActionState)actionState;

// factory methods
- (NSMutableArray *)texturesWithPrefix:(NSString *)prefix
                           startFrameIdx:(NSInteger)startFrameIdx
                           frameCount:(NSInteger)frameCount;
- (SKAction *)animateActionForGroup:(NSMutableArray *)group
                           timePerFrame:(NSTimeInterval)timeInterval
                           frameCount:(NSInteger)frameCount;
- (SKAction *)animateActionForTextures:(NSMutableArray *)textures
                           timePerFrame:(NSTimeInterval)timeInterval;
```

Don't worry. All of this advanced planning will smooth and speed your work later on! Treat it like an initial checklist of what you need to define.

1. **Update:** This method will execute at every interval, like a repeating timer. More on this later.

2. **Action Methods:** You won't use the SKActions (from the actions section) directly. Instead, you'll use these methods to trigger each state to make the code centralized and clean.
 3. **Contact Point Methods:** These are helper methods to create and adjust collision points.
 4. **Factory Methods:** These are methods that make it easier to create animations.
- That's it. Build and run your code to make sure everything still works as expected. The project builds and runs fine, but you should be seeing a lot of warnings in Xcode, since you declared several methods in **ActionSprite.h**, but haven't yet defined them in **ActionSprite.m**. You'll do that now.



Pompadour go!

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

Select the **Characters** group, go to **File\New\New File**, choose the **iOS\Cocoa Touch\Objective-C class** template and click **Next**. Enter **Hero** as the name of the new file and **ActionSprite** for Subclass of. Click **Next** again and then **Create**.

Open **Hero.m** and add the following:

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

//add this method
- (instancetype)init
{
    SKTextureCache *cache = [SKTextureCache sharedInstance];
    SKTexture *texture = [cache textureNamed:@"hero_idle_00"];

    self = [super initWithTexture:texture];

    if (self)
    {
        NSMutableArray *idleFrames =
            [NSMutableArray arrayWithCapacity:6];
```

```
for (NSInteger i = 0; i < 6; ++i) {
    NSString *name =
        [NSString stringWithFormat:@"hero_idle_%02ld", (long)i];
    SKTexture *texture = [cache textureNamed:name];
    [idleFrames addObject:texture];
}

SKAction *idleAnimation =
    [SKAction animateWithTextures:idleFrames
                           timePerFrame:1.0/12.0];

self.idleAction =
    [SKAction repeatActionForever:idleAnimation];
}

return self;
}
```

You create the hero character with an initial texture (or frame, in the context of an animation sequence), prepare an `NSMutableArray` containing all the other frames belonging to the idle animation and create the `SKAction` that plays this animation.

Some important points:

- You're initializing the hero sprite differently from the title scene sprites. Instead of using a filename directly, you use `SKTextureCache` and the name of the sprite's texture. You will learn more about this soon—for now, just keep it in mind.
- An `NSMutableArray` is an indexed collection of objects. In this case, you use one to keep track of the sprite frames in the idle animation.
- `animateWithTextures:` uses an array of textures to change a sprite's texture at intervals defined by the delay parameter. `1.0/12.0` means 12 frames per second.

To test out your hero, first go to `GameScene.m` and add the following:

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

//replace initWithFrame
- (instancetype)initWithFrame:(CGSize)size
{
    if (self = [super initWithFrame:size])
    {
        SKTextureAtlas *atlas =
            [SKTextureAtlas atlasNamed:@"sprites"];

        [[SKTextureCache sharedInstance]
            addTexturesFromAtlas:atlas
            filteringMode:SKTextureFilteringNearest];

        GameLayer *gameLayer = [GameLayer node];
        [self addChild:gameLayer];
    }
}
```

```

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

```

Then, switch to **GameLayer.h** and make the following changes:

```

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

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

```

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

```

//replace init

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

//add before @end
- (void)initHero
{
    self.hero = [Hero node];

    [self.hero setScale:kPointFactor];
    self.hero.position = CGPointMake(100 * kPointFactor,
                                    100 * kPointFactor);

    [self addChild:self.hero];
    [self.hero idle];
}

```

SKTTextureCache is another class bundled inside SKTUtils. Its job is to store persistent and reusable copies of textures loaded from a **texture atlas**—a list of textures bundled together in one image.

Note: We'll cover texture atlases and sprite sheets more completely in the next section, but in the meantime, get prepared by checking out this fun and educational video:

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

Once a texture is in memory, you can simply call on SKTTextureCache to quickly provide you with textures for sprites. This is quite useful for games that require a lot of animations and thus have continuously changing textures. Without SKTTextureCache, you might need to create a new texture each time you animate your SKSpriteNode and this would have a bad impact on performance.

You use SKTTextureCache to load the list of textures from **sprites.atlasc**, found in your **Resources\Images** group in Xcode.

You also tell SKTTextureCache to set the filtering mode of the textures to SKTextureFilteringNearest after storing them. Remember from Chapter 1, this is the filtering mode that resizable pixel art requires.

Any texture that is in SKTTextureCache can now be used by any sprite, which is why you can instantiate the hero this way, instead of with **initWithFile:**.

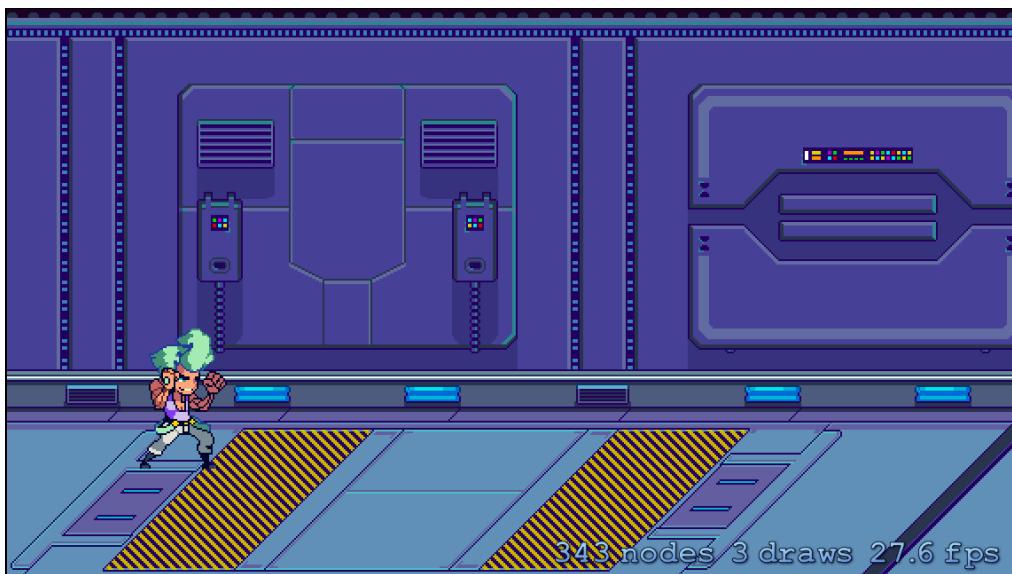
Next, you need to define the **idle** method. Go to **ActionSprite.m** and add the following inside **@implementation**:

```
- (void)idle
{
    if (self.actionState != kActionStateIdle) {

        [self removeAllActions];
        [self runAction:self.idleAction];
        self.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

The Pompadoured Hero is now in the house! Or at least, he's 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.png**, both contained in **sprites.atlasc**.

But where did these 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:

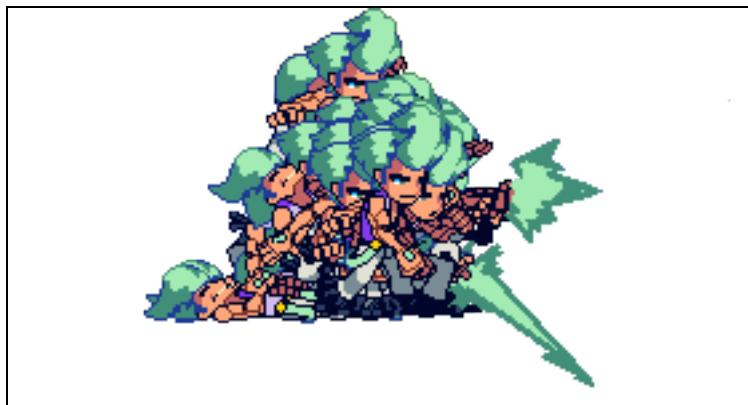
1. **Texture file:** The texture atlas that contains all the sprite frames crammed together. In this case, it is **sprites.png**.
2. **Definition file:** The property list (**sprites.plist**) file contains information about each sprite frame, such as its location in the texture and its dimensions.
3. **Atlas folder:** The folder (**sprites.atlasc**) should contain both the texture file and the definition file. Atlas folders should have the **.atlasc** file extension to indicate that these files it contains are part of a pre-generated texture atlas.

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

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 something about how the artist, Hunter Russell, drew each sprite frame.

Inside the root directory of the files that came with this book there is a folder named **Raw**. Inside this folder is a file named **HeroFrames.psd**. Take a look at the drawing and you'll see that the canvas size is much bigger than the hero. Hunter created each frame for the hero on a 280x150-pixel canvas, but his 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 this: 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.

In some animations, the character or something that is attached to the character, such as a weapon, needs to move some distance from the center of the canvas, like when he's jumping or falling down. That's why you need to make the canvas bigger—so you have room to put the sprite at the right spot in those animations that require more relative movement.

However, when the character moves around the screen (as when the user presses or taps the movement button), it is the entire character canvas that moves within the layer. Since there's a lot of transparent space that surrounds your sprite in order to accommodate the animations, you can't use the size of the sprite itself for collision detection – you'll need to make separate (smaller) shapes for that, as you'll learn about later.

Note that when you make animations for a sprite like you see here, you shouldn't move your sprite much outside the base position, except for moving things like the character's head and limbs. Your game engine will be responsible for moving the sprite itself in response to the user's actions.

Note: Texture atlases are most useful when you want to squeeze all the performance you can get from the device.

First, you reduce the number of draw calls, primarily because of Sprite Kit's batch drawing. Pay attention to the draw call counter positioned in between the node counter and FPS as you work through this starter kit. If you were to add five sprites to the scene without an atlas, the draw count would be five. With an atlas, the draw count would be just one.

Second, when any of your images have a canvas size that is a lot bigger than the actual drawing, you're able to reduce the texture memory used by trimming out the blank spaces from the texture atlas.

Creating a directional pad

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

The logical thing is to move the hero 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're going to make your own animated 8-directional D-pad.

Begin 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 **Controls** group you just created, go to **File\New\File...**, choose the **iOS\Cocoa Touch\Objective-C class** template and click **Next**. Enter **ActionDPad** as the name of the class and **SKSpriteNode** for Subclass of. Click **Next** once again and then click **Create**.

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

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

typedef NS_ENUM(NSInteger, ActionDPadDirection) {
    kActionDPadDirectionCenter = 0,
    kActionDPadDirectionUp,
    kActionDPadDirectionUpRight,
    kActionDPadDirectionRight,
    kActionDPadDirectionDownRight,
    kActionDPadDirectionDown,
    kActionDPadDirectionDownLeft,
    kActionDPadDirectionLeft,
    kActionDPadDirectionUpLeft
};

@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 : SKSpriteNode

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

+ (instancetype)dPadWithPrefix:(NSString *)filePrefix
                           radius:(CGFloat)radius;
- (instancetype)initWithPrefix:(NSString *)filePrefix
                           radius:(CGFloat)radius;

- (void)update:(NSTimeInterval)delta;

@end
```

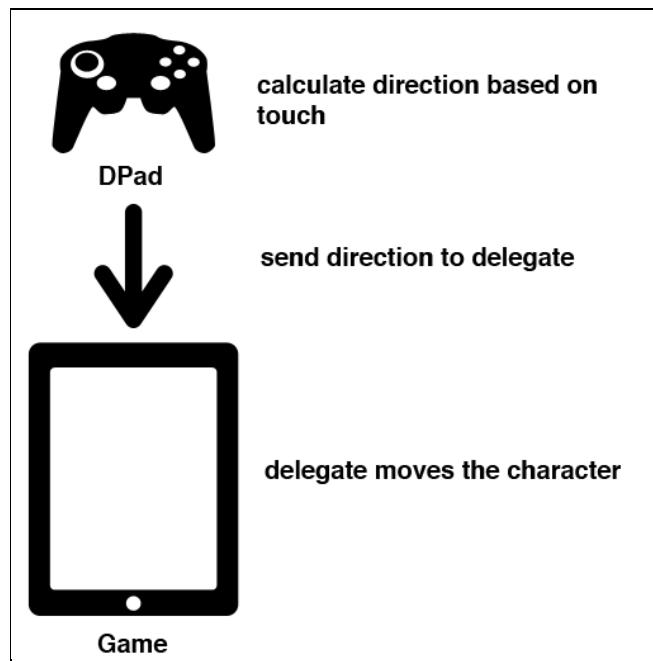
There are a lot of declarations above, but here are the important ones:

- **ActionDPadDirection:** This enumeration represents the D-pad's nine directions. Wait, didn't I say it was going to be an 8-directional D-pad? You are including the center "direction" here, because you also need to know when the joystick is centered so you don't move the hero at all.
- You store both the current and previously pressed directions of the D-pad. The previous direction will be a private property in the implementation file.
- **prefix:** You'll use the filename prefix for the D-pad image file to animate the D-pad.
- **delegate:** This is the delegate of the D-pad, explained in detail below.
- **isHeld:** A Boolean that returns YES as long as the player is touching the D-pad.
- **radius:** The radius of the circle formed by the D-pad.

For the ActionDPad, you're 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 specified by you, **ActionDPad** will pass on responsibility to the delegate class, such as when the user has moved the D-pad in a particular direction.

This keeps ActionDPad ignorant of the game's logic, thus allowing you to reuse it in any other game you may want to develop.

The diagram below illustrates what will happen when the user touches the D-pad:



When **ActionDPad** detects a touch that is within the radius of the D-pad, it will calculate the direction (**ActionDPadDirection**) of that touch and send a message to its delegate indicating the direction. Anything that happens after that is not **ActionDPad**'s concern.

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 it can call any of these methods whenever it wants to pass something onto the delegate.

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

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

//add inside @implementation
+ (instancetype)dPadWithPrefix:(NSString *)filePrefix
                           radius:(CGFloat)radius
{
    return [[self alloc] initWithPrefix:filePrefix
                               radius:radius];
}

- (instancetype)initWithPrefix:(NSString *)filePrefix
                           radius:(CGFloat)radius
{
    NSString *filename =
    [filePrefix stringByAppendingString:@"_center"];

    SKTTextureCache *cache = [SKTTextureCache sharedInstance];
    self = [super initWithTexture:[cache textureNamed:filename]];

    if (self) {
        _radius = radius;
        _direction = kActionDPadDirectionCenter;
        _isHeld = NO;
        _prefix = filePrefix;
        self.userInteractionEnabled = YES;
    }

    return self;
}
```

You've done most of these before. Just take note that you're using `initWithTexture:` and `SKTTextureCache`, so expect to have a sprite sheet for the D-pad images. You also enable user interaction for the node so that it can receive touch input later on.

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`, without the 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 the Xcode automatically synthesizes properties. Each property also gets its 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:

```
// insert after last #import and before @implementation

@interface ActionDPad()

@property (assign, nonatomic) CGFloat radius;
@property (strong, nonatomic) NSString *prefix;

@property (assign, nonatomic)
ActionDPadDirection previousDirection;
@property (assign, nonatomic) NSUInteger touchHash;

@end

// 1
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    if (self.touchHash == 0)
    {
        for (UITouch *touch in touches) {

            CGPoint location = [touch locationInNode:self.parent];
            CGFloat distanceSQ = CGPointDistanceSQ(location,
                                                    self.position);

            if (distanceSQ <= self.radius * self.radius) {
                //get angle 8 directions
                self.touchHash = touch.hash;
                [self updateDirectionForTouchLocation:location];
                self.isHeld = YES;
            }
        }
    }
}
```

```
        }

    }

// 2
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    for (UITouch *touch in touches) {
        if (touch.hash == self.touchHash)
        {
            CGPoint location = [touch locationInNode:self.parent];
            [self updateDirectionForTouchLocation:location];
        }
    }
}

// 3
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    self.direction = kActionDPadDirectionCenter;
    self.isHeld = NO;

    for (UITouch *touch in touches) {
        if (touch.hash == self.touchHash)
        {
            NSString *txtName =
                [NSString stringWithFormat:@"%@_center", self.prefix];

            SKTexture *texture =
                [[SKTTextureCache sharedInstance] textureNamed:txtName];

            [self setTexture:texture];
            [self.delegate actionDPadTouchEnded:self];

            self.touchHash = 0;
        }
    }
}

// 4
- (void)update:(NSTimeInterval)delta
{
    if (self.isHeld) {
        [self.delegate actionDPad:self
                           isHoldingDirection:self.direction];
    }
}

// 5 & 6
- (void)updateDirectionForTouchLocation:(CGPoint)location
{
    CGPoint point = CGPointMakeSubtract(location, self.position);
    CGFloat radians = CGPointMakeToAngle(point);
```

```
CGFloat degrees = -1 * RadiansToDegrees(radians);
NSString *suffix = @"_center";

self.previousDirection = self.direction;

if (degrees <= 22.5 && degrees >= -22.5) {

    self.direction = kActionDPadDirectionRight;
    suffix = @"_right";
} else if (degrees > 22.5 && degrees < 67.5) {

    self.direction = kActionDPadDirectionDownRight;
    suffix = @"_downright";
} else if (degrees >= 67.5 && degrees <= 112.5) {

    self.direction = kActionDPadDirectionDown;
    suffix = @"_down";
} else if (degrees > 112.5 && degrees < 157.5) {

    self.direction = kActionDPadDirectionDownLeft;
    suffix = @"_downleft";
} else if (degrees >= 157.5 || degrees <= -157.5) {

    self.direction = kActionDPadDirectionLeft;
    suffix = @"_left";
} else if (degrees < -22.5 && degrees > -67.5) {

    self.direction = kActionDPadDirectionUpRight;
    suffix = @"_upright";
} else if (degrees <= -67.5 && degrees >= -112.5) {

    self.direction = kActionDPadDirectionUp;
    suffix = @"_up";
} else if (degrees < -112.5 && degrees > -157.5) {

    self.direction = kActionDPadDirectionUpLeft;
    suffix = @"_upleft";
}

NSString *textureName =
    [NSString stringWithFormat:@"%@%@", self.prefix, suffix];

[self setTexture:[[SKTTextureCache sharedInstance]
    textureNamed:textureName]];

if (self.isHeld) {

    if (self.previousDirection != self.direction) {

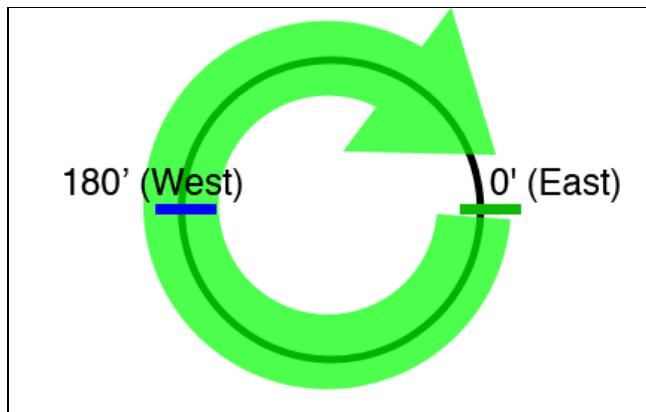
        [self.delegate actionDPad:self
            didChangeDirectionTo:self.direction];
    }
} else {
```

```
[self.delegate actionDPad:self  
    didChangeDirectionTo:self.direction];  
}  
}
```

Don't be intimidated by the long wall of code—it's very simple once you break it down:

1. **touchesBegan** checks if the location of the touch is inside the D-pad's circle. It does this by checking if the distance of the touch from the center of the D-pad is shorter than the radius of the D-pad. If yes, it switches on the **isHeld** Boolean property and triggers an update to the direction value. You also store the touch's hash value in the **touchHash** property. This hash value is a unique non-zero ID number for the touch, so you can differentiate it from other touches later on.
2. **touchesMoved** simply triggers an update of the direction value every time the player drags their finger to a new position. By comparing the **touchHash** value against all active touches, you make sure that you only update the direction if it is the finger that was used to touch the D-pad the first time.
3. **touchesEnded** switches off the **isHeld** flag, centers the direction, resets the displayed frame to the center frame and notifies the delegate that the touch has ended. You also reset the **touchHash** value to 0,
4. **update** constantly passes the direction value to the delegate as long as the user is touching the ActionDPad.
5. **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. Whenever the user selects a new direction, this method passes the direction value to the delegate.
6. **updateDirectionForTouchLocation** also sets a suffix that corresponds to the filename of each sprite frame. There is a different image for each touched direction, plus the neutral direction—nine images in all. **setTexture:** changes the displayed frame for the sprite by combining the prefix and suffix values taken from the direction.

The angle values, in degrees, may look weird to you—the opposite of what you learned in math class! Take a look at this circle:



In mathematics, 0 degrees is usually at the east-most point of the circle, becoming positive in the counterclockwise direction (opposite to the arrow direction in the diagram).

Since you multiply the angle by -1 (which reverses the direction), the angle in your code becomes positive in the clockwise direction (to match the diagram).

This means that if the touch happens to land in the area of the D-pad enclosed by the angles -22.5 and 22.5 degrees, the touch will correspond to selecting right on the D-pad, and so on.

Note: This way of thinking about angles is just a convention chosen for this book, just like how values grow larger in a CGRect by moving right and down on the screen of an iOS device.

You could just as well have chosen to increase the value of angles in a counterclockwise direction. There's no right or wrong here, as long as you're consistent.

OK, that's the D-pad class. Now you need to add it to your game and use it. You need to make it part of the heads-up display (HUD).

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

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

//add before the @end
@property (strong, nonatomic) ActionDPad *dPad;

//add this method declaration inside @interface
- (void)update:(NSTimeInterval)delta;
```

Switch to **HudLayer.m** and add the following methods:

```
- (instancetype)init
{
```

```

if (self = [super init])
{
    //directional pad
    CGFloat radius = 64.0 * kPointFactor;
    _dPad = [ActionDPad dPadWithPrefix:@"dpad" radius:radius];
    _dPad.position = CGPointMake(radius, radius);
    _dPad.alpha = 0.5;
    [self addChild:_dPad];
}

return self;
}

- (void)update:(NSTimeInterval)delta
{
    [self.dPad update:delta];
}

```

You instantiate an **ActionDPad** and set its radius. You multiply the radius by **kPointFactor** so that it automatically converts for iPad. You also call ActionDPad's update loop from HudLayer's update loop.

Before creating the ActionDPad, you need to load the **joypad.atlas** texture atlas, which you will do in a short while. This atlas contains the D-pad image files. Look inside and you'll see that each direction's filename has a "dpad" prefix.

One noticeable difference between the D-pad and hero sprites is that you didn't set the former's scale to **kPointFactor**. This is because your D-pad art isn't drawn in the pixel style, and so instead your texture atlas contains images for each device resolution, namely:

- **filename.png** for iPhone non-retina.
- **filename@2x.png** for iPhone retina and iPad non-retina.
- **filename@2x~ipad.png** for iPad retina.

By default, Sprite Kit will look for filename~ipad.png for the iPad version, but in this case you want it to use filename@2x.png for both the retina iPhone and non-retina iPad. It's a good thing that SKTextureCache 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.

This mechanism is turned off by default, so go to **GameScene.m** and add the following to `initWithSize:`:

```

//add these after the first addTexturesFromAtlas
[[SKTextureCache sharedInstance]
    setEnableFallbackSuffixes:YES];

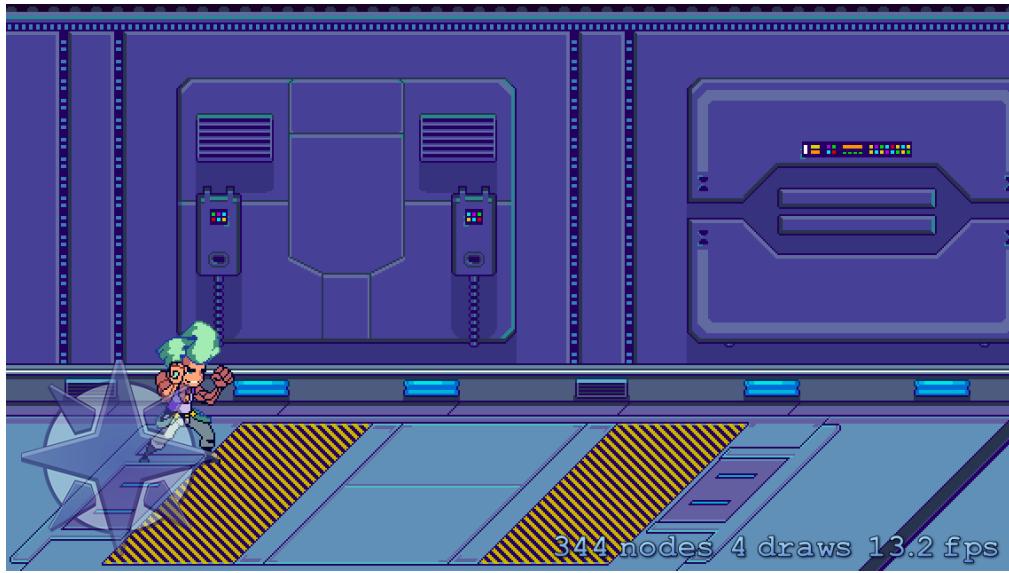
atlas = [SKTextureAtlas atlasNamed:@"joypad"];
[[SKTextureCache sharedInstance]

```

```
addTexturesFromAtlas:atlas  
filteringMode:SKTextureFilteringLinear];
```

After enabling fallback suffixes for the texture cache, you load all the textures from the **joypad.atlas** texture atlas found in your Resources\Images group.

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



I know I don't have to tell you to try out that D-pad. Although it doesn't make the hero move yet, you will see shading appear on each area you touch. Neat!

Note: You may have noticed that the folder **joypad.atlas** is packaged differently from **sprites.atlasc**, and it doesn't look like a texture atlas at all because the images aren't packed together into one sheet.

Xcode has the ability to create texture atlases automatically, as long as you group images together in a **.atlas** folder. When you build your project, Xcode will pack all the images found in **joypad.atlas** into a **joypad.atlasc**, in a similar manner as **sprites.atlasc**.

In sum, **joypad.atlas** is a texture atlas that Xcode will automatically create, while **sprites.atlasc** is a texture atlas that was manually created using TexturePacker.

Moving the hero

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: Your overarching 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 the 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 an array of animation frames. You already declared one earlier: `texturesWithPrefix:startFrameIdx:frameCount:`. Now it’s time to implement it.

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

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

//add this before @end
- (NSMutableArray *)texturesWithPrefix:(NSString *)prefix
    startFrameIdx:(NSInteger)startFrameIdx
    frameCount:(NSInteger)frameCount
{
    NSInteger idxCount = frameCount + startFrameIdx;
    NSMutableArray *textures =
        [NSMutableArray arrayWithCapacity:frameCount];

    SKTexture *texture;

    for (NSInteger i = startFrameIdx; i < idxCount; i++) {
        NSString *name =
            [NSString stringWithFormat:@"%@_%02ld", prefix, (long)i];

        texture = [[SKTTextureCache sharedInstance]
            textureNamed:name];
        [textures addObject:texture];
    }

    return textures;
}
```

The only difference between this method and the one you used earlier to create the array of idle textures is that because of its parameters, this one is reusable.

The method creates an array with textures 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 an `NSMutableArray` and a `for` loop every time you want a texture array.

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

```
- (instancetype)init
{
    SKTextureCache *cache = [SKTextureCache sharedInstance];
    SKTexture *texture = [cache textureNamed:@"hero_idle_00"];

    self = [super initWithTexture:texture];

    if (self)
    {
        NSMutableArray *textures =
            [self texturesWithPrefix:@"hero_idle"
                startFrameIdx:0 frameCount:6];

        SKAction *idleAnimation =
            [SKAction animateWithTextures:textures
                timePerFrame:1.0/12.0];

        self.idleAction =
            [SKAction repeatActionForever:idleAnimation];

        textures =
            [self texturesWithPrefix:@"hero_walk"
                startFrameIdx:0 frameCount:8];

        SKAction *walkAnimation =
            [SKAction animateWithTextures:textures
                timePerFrame:1.0/12.0];

        self.walkAction =
            [SKAction repeatActionForever:walkAnimation];

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

    return self;
}
```

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

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

```
- (void)walkWithDirection:(CGPoint)direction
{
    if (self.actionState == kActionStateIdle ||
        self.actionState == kActionStateRun) {
```

```

[self removeAllActions];
[self runAction:self.walkAction];
self.actionState = kActionStateWalk;
[self moveWithDirection:direction];

} else if (self.actionState == kActionStateWalk) {
    [self moveWithDirection:direction];
}
}

- (void)moveWithDirection:(CGPoint)direction
{
    if (self.actionState == kActionStateWalk) {

        self.velocity = CGPointMake(direction.x * self.walkSpeed,
                                     direction.y * self.walkSpeed);
        [self flipSpriteForVelocity:self.velocity];

    } else if (self.actionState == kActionStateRun) {

        self.velocity = CGPointMake(direction.x * self.runSpeed,
                                     direction.y * self.runSpeed);
        [self flipSpriteForVelocity:self.velocity];

    } else if (self.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.xScale = self.directionX * kPointFactor;
}

```

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

Note: In Sprite Kit, a negative **xScale** 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 user touches the D-pad. 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 immediately after @interface GameLayer : SKNode
<ActionDPadDelegate>

//add before @end
@property (weak, nonatomic) HudLayer *hud;
```

You add a weak reference to a **HudLayer** instance within **GameLayer**. You also make **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 initWithFrame:size]) right after
[self addChild:hudLayer]
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:

```
#pragma mark - ActionDPadDelegate methods

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

- (void)actionDPad:(ActionDPad *)actionDPad
isHoldingDirection:(ActionDPadDirection)direction
```

```
{  
    CGPoint directionVector = [self vectorForDirection:direction];  
    [self.hero moveWithDirection:directionVector];  
}  
  
- (void)actionDPadTouchUpInside:(ActionDPad *)actionDPad  
{  
    if (self.hero.actionState == kActionStateWalk ||  
        self.hero.actionState == kActionStateRun) {  
        [self.hero idle];  
    }  
}
```

You trigger the hero's `walkWithDirection:` and `moveWithDirection:` methods every time the user touches the ActionDPad, and trigger the hero's `idle` method every time the user stops touching the ActionDPad.

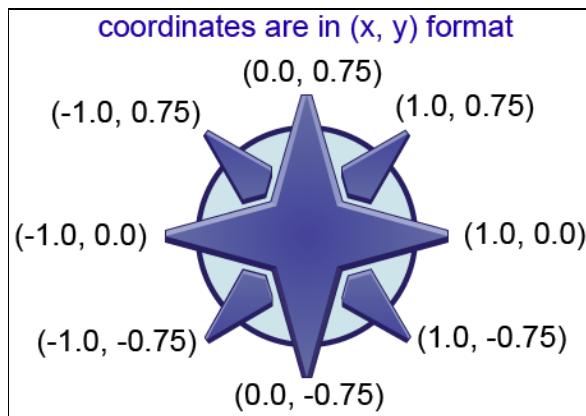
ActionDPad sends an enum value for direction to its delegate, but ActionSprite expects a normal vector value of -1.0 to 1.0 in (x, y) coordinates, so you use a conversion method, which you will write next.

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

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

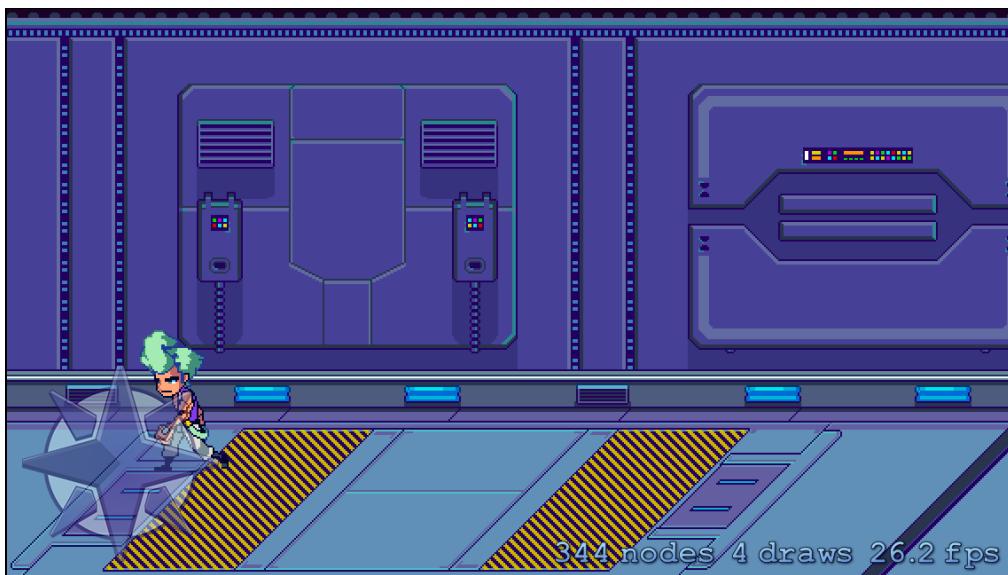
```
        case kActionDPadDirectionUpLeft:  
            return CGPointMake(-maxX, maxY);  
            break;  
        default:  
            return CGPointMakeZero;  
            break;  
    }  
}
```

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



For each direction, the method assigns the corresponding x- and y-values. You cap the maximum value for the y-axis at 0.75 to make the vertical movement slower than the horizontal movement. Feel free to change this and experiment with the effects. Higher values will make the hero move faster, while lower values 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... the animation's running, but he's not actually moving. What gives?

A little sprite navigation

Take a look at `moveWithDirection`: again in **Hero.m** and notice that it doesn't do anything except change the hero's velocity. Where is the code that changes 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 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:(NSTimeInterval)delta
{
    if (self.actionState == kActionStateWalk) {
        CGPoint point = CGPointMultiplyScalar(self.velocity, delta);
        self.desiredPosition = CGPointAdd(self.position, point);
    }
}
```

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 displacement of the sprite to its current position. This is why you multiply the velocity by a time delta passed into `update`: (the time interval) to arrive at the correct displacement you need to update the sprite's position.

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 as an approach that's easy to understand and implement, though not as one that is extremely accurate. But this isn't a physics simulation, so 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)
self.centerToBottom = 39.0 * kPointFactor;
self.centerToSides = 29.0 * kPointFactor;
```

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

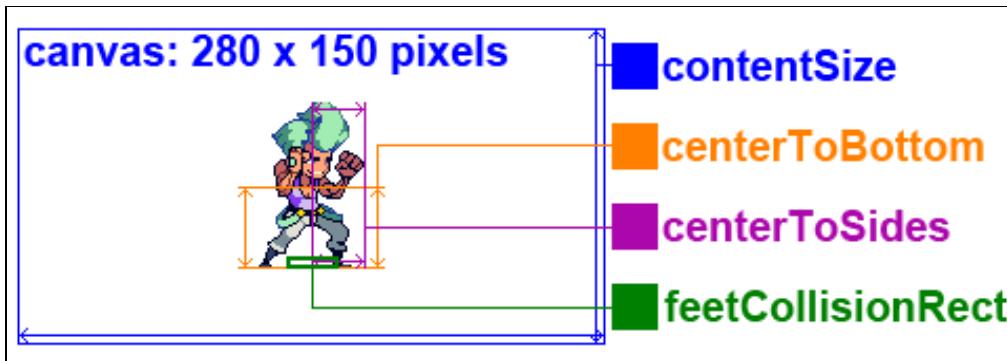
```
- (CGRect)feetCollisionRect
{
    CGRect feetRect =
        CGRectMake(self.desiredPosition.x - self.centerToSides,
                  self.desiredPosition.y - self.centerToBottom,
                  self.centerToSides * 2,
                  5.0 * kPointFactor);

    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 occupies a 280x150 pixel canvas, but the actual drawing only takes up a fraction of that space. Each SKSpriteNode has a frame property that gives you the position and size of the sprite. The frame property is useful in cases where the sprite takes up the whole frame, but it's 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're defining here, like the `feetCollisionRect` method, you already 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.

Go to `GameLayer.h` and add this method declaration:

```
- (void)update:(NSTimeInterval)delta;
```

Switch to `GameLayer.m` and add the following two methods at the bottom of the file:

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

- (void)updatePositions
{
    CGFloat mapWidth =
        self.tileMap.mapSize.width *
        self.tileMap.tileSize.width *
        kPointFactor;

    CGFloat floorHeight =
        3 * self.tileMap.tileSize.height * kPointFactor;

    CGFloat posX = MIN(mapWidth -
        self.hero.feetCollisionRect.size.width/2,
        MAX(self.hero.feetCollisionRect.size.width/2,
            self.hero.desiredPosition.x));

    CGFloat posY = MIN(floorHeight + (self.hero.centerToBottom -
        self.hero.feetCollisionRect.size.height), MAX(self.hero.centerToBottom,
            self.hero.desiredPosition.y));

    self.hero.position = CGPointMake(posX, posY);
}
```

First you write `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** to scale 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 bounds of the sprite. Remember that the canvas you're using for the sprites is much bigger than the actual drawings, 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 (x-coordinate 0.0) 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 (y-coordinate 0.0).

If the desired position of the ActionSprite is within the boundaries 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 a maximum number.

Lastly, you need to tell the game to repeatedly execute the update: method. This is called your game loop, or update loop. In Sprite Kit, each SKScene is already equipped with its own update loop, much like the update loop you added to GameLayer and to ActionSprite.

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

```
//add these before @implementation

@interface GameScene()

@property (strong, nonatomic) GameLayer *gameLayer;
@property (strong, nonatomic) HudLayer *hudLayer;
@property (assign, nonatomic) NSTimeInterval lastUpdateTime;

@end

//replace initWithSize
- (instancetype)initWithSize:(CGSize)size
{
    if (self = [super initWithSize:size])
    {
        SKTextureAtlas *atlas =
        [SKTextureAtlas atlasNamed:@"sprites"];

        [[SKTextureCache sharedInstance]
```

```
addTexturesFromAtlas:atlas
filteringMode:SKTextureFilteringNearest];

[[SKTextureCache sharedInstance]
setEnableFallbackSuffixes:YES];

atlas = [SKTextureAtlas atlasNamed:@"joypad"];

[[SKTextureCache sharedInstance]
addTexturesFromAtlas:atlas
filteringMode:SKTextureFilteringLinear];

_gameLayer = [GameLayer node];
[self addChild:_gameLayer];

_hudLayer = [HudLayer node];
[self addChild:_hudLayer];

_hudLayer.dPad.delegate = _gameLayer;
_gameLayer.hud = _hudLayer;
}

return self;
}

//add this method
- (void)update:(NSTimeInterval)currentTime
{
    if (self.lastUpdateTime <= 0) {
        self.lastUpdateTime = currentTime;
    }

    NSTimeInterval delta = currentTime - self.lastUpdateTime;
    self.lastUpdateTime = currentTime;

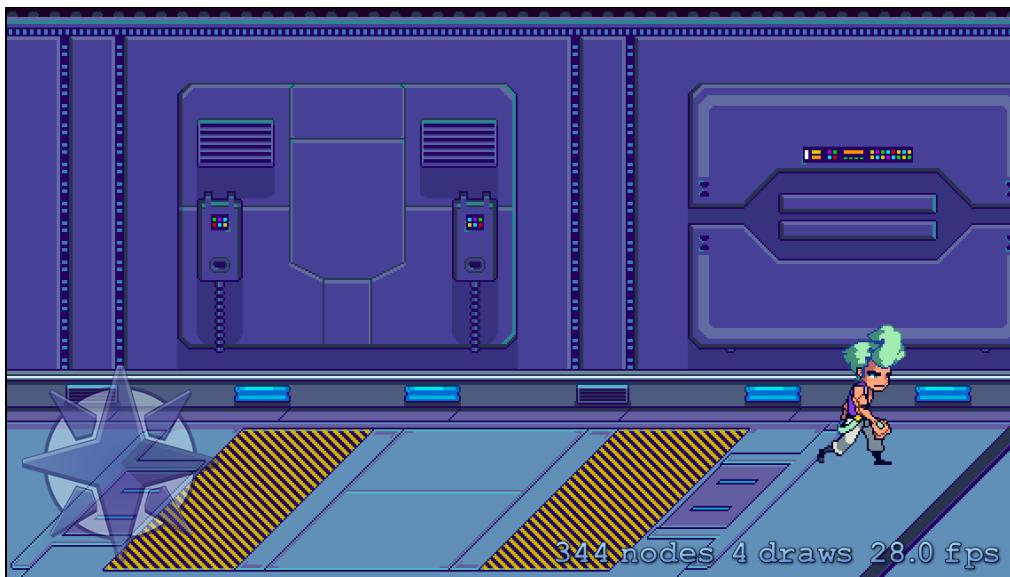
    [self.gameLayer update:delta];
    [self.hudLayer update:delta];
}
```

First, you create private properties for GameLayer and HudLayer so that you can access them in another method, and this change is reflected in `initWithSize:`.

Then you write GameScene's update loop. An SKScene's update method will be triggered at every frame update. Remember the FPS (frames per second) statistic that you have displayed on the lower-right of the device's screen? That's how many times update will run every second.

By default, an SKScene's update loop is able to get the current running time of the game. This means that `currentTime` counts from 0 to N, where N is how long the game has been running. For your position calculations in GameLayer to work properly, you need the **delta time**—the time interval between frames. To get this, you subtract the `currentTime` in the current update from the `currentTime` of the previous update, which you store in the `lastUpdateTime` property.

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 (you'll address that in another chapter) and focus on the second, 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.

Open `GameLayer.m` and add the following method:

```
//add this in update: right after [self updatePositions];
[self setViewpointCenter:self.hero.position];

- (void)setViewpointCenter:(CGPoint)position {
    NSInteger x = MAX(position.x, CENTER.x);
    NSInteger y = MAX(position.y, CENTER.y);

    x = MIN(x, (self.tileMap.mapSize.width *
    self.tileMap.tileSize.width * kPointFactor)
        - CENTER.x);

    y = MIN(y, (self.tileMap.mapSize.height * self.tileMap.tileSize.height
    * kPointFactor)
        - CENTER.y);

    CGPoint actualPosition = CGPointMake(x, y);
```

```
CGPoint viewPoint = CGPointMakeSubtract(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 moves, it takes all of its children along for the ride.

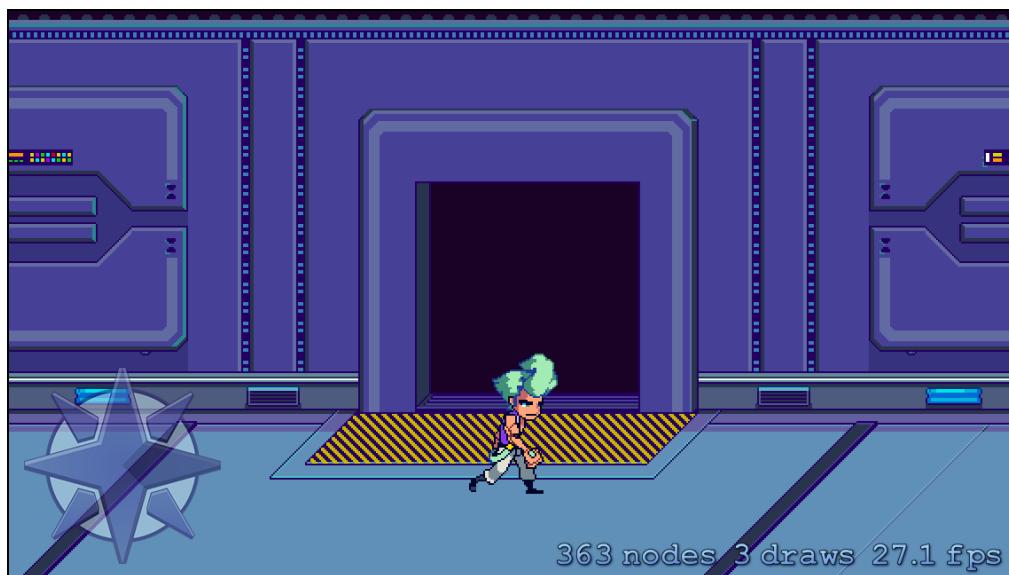
The **CENTER** macro, you might recall, simply returns half of the screen's dimensions and not the actual center of the entire map. Subtracting from **CENTER** makes sure that the "camera" doesn't move out of bounds.

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. You should now see the hero at all times and the background should 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).



You've reached that magic moment in the development process where your creation is just 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, and 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**. Simply replace those with your own versions. You can then build the sprite sheet with TexturePacker 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. It's time to add some pep to his step by giving him the ability to run.

Then, you'll add the standard controls for a beat 'em up game: jumping and punching.

And of course, your hero needs something to punch, so you'll add enemy robots to 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. That gets boring pretty fast.

You could increase the hero's `walkSpeed` attribute to make him quicker, but there's a better option—allow him to run. What else are you going to do with `kActionStateRun`?

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

```
- (void)moveWithDirection:(CGPoint)direction
{
    if (self.actionState == kActionStateWalk) {

        self.velocity = CGPointMake(direction.x * self.walkSpeed,
                                    direction.y * self.walkSpeed);
        [self flipSpriteForVelocity:self.velocity];

    } else if (self.actionState == kActionStateRun) {

        self.velocity = CGPointMake(direction.x * self.runSpeed,
                                    direction.y * self.runSpeed);
        [self flipSpriteForVelocity:self.velocity];
    }
}
```

```
    } else if (self.actionState == kActionStateIdle) {  
        [self walkWithDirection:direction];  
    }  
}
```

Maybe you noticed something peculiar about this method when you created it originally. In the first else-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, you expect it 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 no one 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 (self.actionState == kActionStateIdle ||  
        self.actionState == kActionStateWalk) {  
  
        [self removeAllActions];  
        [self runAction:self.runAction];  
        self.actionState = kActionStateRun;  
        [self moveWithDirection:direction];  
    }  
}  
  
//replace the if statement condition inside update: with this  
if (self.actionState == kActionStateWalk ||  
    self.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 the method to change `desiredPosition` when the current state is `kActionStateRun`.

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

```
//add inside the curly braces of if (self) after self.directionX
//run animation
textures = [self texturesWithPrefix:@"hero_run"
                           startFrameIdx:0 frameCount:8];

SKAction *runAnimation =
[SKAction animateWithTextures:textures timePerFrame:1.0/12.0];

self.runAction = [SKAction repeatActionForever: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 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 you'll count the previous and current taps as a double tap.
- **The previous direction pressed.** If the previously and currently pressed directions are the same, then the hero should run.

You can combine these two variables to determine if the player tapped the same direction arrow twice in rapid succession.

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

```
//insert after #import "GameLayer.h"
@interface GameLayer ()

@property (assign, nonatomic) CGFloat runDelay;
@property (assign, nonatomic) ActionDPadDirection previousDirection;
```

```
@end
```

`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` will simply store the last direction returned by the D-pad.

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

```
//add inside update: right after [self updatePositions]

if (self.runDelay > 0) {
    self.runDelay -= delta;
}

//replace this method with the following:

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

    // 1
    if (self.runDelay > 0 &&
        self.previousDirection == direction &&
        (direction == kActionDPadDirectionRight ||
         direction == kActionDPadDirectionLeft)) {

        [self.hero runWithDirection:directionVector];
    }
    // 2
    else if (self.hero.actionState == kActionStateRun &&
              abs(self.previousDirection - direction) <= 1) {

        [self.hero moveWithDirection:directionVector];
    }
    // 3
    else {
        [self.hero walkWithDirection:directionVector];
        self.previousDirection = direction;
        self.runDelay = 0.2;
    }
}
```

You make sure `runDelay` counts down to zero by decrementing the delta time at every game update. Then you make three significant changes to your handling of D-pad direction changes:

1. You check if the player selected the same direction in the allowed time interval and call `runWithDirection:` if so.
2. You let the hero run in his current direction only if the newly selected direction is the same. You'll revisit this in a bit.

3. Here you make the hero walk, either because it's the first time the hero has 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 first else-if statement looks weird in how it determines whether the direction has changed. 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.



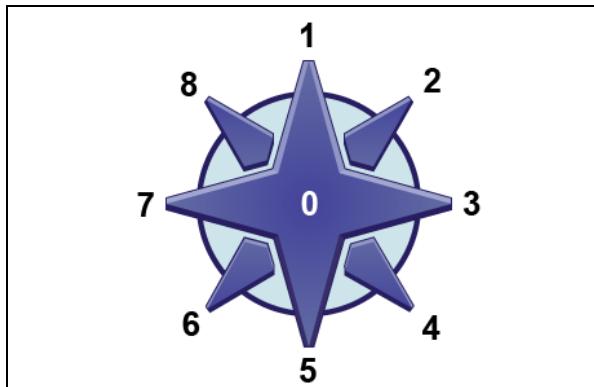
To make sense of this, open **ActionDPad.h** and revisit the definition of the **ActionDPadDirection** enumeration:

```
typedef NS_ENUM(NSInteger, ActionDPadDirection) {  
    kActionDPadDirectionCenter = 0,  
    kActionDPadDirectionUp,  
    kActionDPadDirectionUpRight,  
    kActionDPadDirectionRight,  
    kActionDPadDirectionDownRight,  
    kActionDPadDirectionDown,  
    kActionDPadDirectionDownLeft,  
    kActionDPadDirectionLeft,  
    kActionDPadDirectionUpLeft  
};
```

An enumeration is simply a bunch of integers grouped together using words. Enumerations help make code readable and limit the numbers you're allowed to use for a given situation, which helps avoid coding errors.

You have given the first number, 0, 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** has a value of 1, **kActionDPadDirectionUpRight** is 2, **kActionDPadDirectionRight** is 3 and so on.

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



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 value could be 6, 7 or 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 will stop running. If running right, **previousDirection** will be 3, and if **currentDirection** has a value other than 2, 3 or 4, then the hero also will stop 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\Cocoa Touch\Objective-C class** template and click **Next**. Enter **ActionButton** for Class and **SKSpriteNode** for Subclass of. Click **Next** and then **Create** to create the new files.

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

```
#import <SpriteKit/SpriteKit.h>
@class ActionButton;

@protocol ActionButtonDelegate <NSObject>

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

@end

@interface ActionButton : SKSpriteNode

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

+ (instancetype)buttonWithPrefix:(NSString *)filePrefix
                           radius:(CGFloat)radius;

- (instancetype)initWithPrefix:(NSString *)filePrefix
                           radius:(CGFloat)radius;

- (void)update:(NSTimeInterval)delta;

@end
```

This is very similar to the way you set up ActionDPad—much simpler, even! You declare protocol methods that ActionButton's delegate should implement. This way, you can reuse ActionButton 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 public properties:

- **delegate:** ActionButton will notify the delegate class when it is pressed, held or released.
- **isHeld:** A Boolean that returns true as long as the user is touching the action button.

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

Switch to **ActionButton.m** and add the following:

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

// add below #import "SKTTextureCache.h"
@interface ActionButton()

@property (assign, nonatomic) CGFloat radius;
@property (strong, nonatomic) NSString *prefix;
@property (assign, nonatomic) NSUInteger touchHash;

@end

//add below @implementation
+ (instancetype)buttonWithPrefix:(NSString *)filePrefix
                           radius:(CGFloat)radius
{
    return [[self alloc] initWithPrefix:filePrefix radius:radius];
}

- (instancetype)initWithPrefix:(NSString *)filePrefix
                           radius:(CGFloat)radius
{
    NSString *filename =
        [filePrefix stringByAppendingString:@"_normal"];

    SKTexture *texture =
        [[SKTTextureCache sharedInstance] textureNamed:filename];

    self = [super initWithTexture:texture];

    if (self) {
        _radius = radius;
        _prefix = filePrefix;
        _isHeld = NO;
        self.userInteractionEnabled = YES;
    }
    return self;
}

- (void)update:(NSTimeInterval)delta
{
    if (self.isHeld) {
        [self.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”. If the prefix string were “button”, then the sprite frame’s name would be “button_normal”.

After storing the important values to the instance variables, you enable touches for the node. Then you write the update: method, which calls the delegate’s **actionButtonIsHeld:** method as long as the user is touching the button.

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

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    if (self.touchHash == 0)
    {
        for (UITouch *touch in touches) {

            CGPoint location = [touch locationInNode:self.parent];
            CGFloat distanceSQ = CGPointDistanceSQ(location,
                                                    self.position);

            if (distanceSQ <= self.radius * self.radius) {
                self.touchHash = touch.hash;
                self.isHeld = YES;

                NSString *name =
                    [NSString stringWithFormat:@"%@_selected", self.prefix];

                SKTexture *texture =
                    [[SKTTextureCache sharedInstance] textureNamed:name];

                [self setTexture:texture];

                [self.delegate actionButtonWasPressed:self];
            }
        }
    }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{
    for (UITouch *touch in touches) {
        if (self.touchHash == touch.hash)
        {
            self.isHeld = NO;
            self.touchHash = 0;
            NSString *name =
                [NSString stringWithFormat:@"%@_normal", self.prefix];

            SKTexture *texture =
                [[SKTTextureCache sharedInstance] textureNamed:name];
        }
    }
}
```

```
[self setTexture:texture];  
[self.delegate actionButtonWasReleased:self];  
}  
}  
}
```

touchesBegan: checks if the touch location is within the bounds of the button. If it is, then the method changes the button's texture to its selected state, sets `self.isHeld` to YES and calls the delegate's `actionButtonWasPressed:`. You also store the hash value of the touch, like in ActionDPad, to identify it later in `touchesEnded:`.

Once the touch leaves the button, `touchesEnded:` turns off the `isHeld` Boolean, resets `touchHash`, switches the sprite's texture back to its unselected state and calls the delegate's `actionButtonWasReleased:`.

With this code, 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. You'll use one button to make the hero attack and the other 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 first @property  
@property (strong, nonatomic) ActionButton *buttonA;  
@property (strong, nonatomic) ActionButton *buttonB;
```

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

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

```
//add inside curly braces of if (self = [super init]) after [self  
addChild:_dPad];  
  
CGFloat buttonRadius = radius / 2.0;  
CGFloat padding = 8.0 * kPointFactor;  
  
_buttonB = [ActionButton buttonWithType:@"button_b"  
           radius:buttonRadius];  
  
_buttonB.position =  
    CGPointMake(SCREEN.width - buttonRadius - padding,  
               buttonRadius * 2 + padding);  
  
_buttonB.alpha = 0.5;  
_buttonB.name = @"ButtonB";  
[self addChild:_buttonB];  
  
_buttonA =  
    [ActionButton buttonWithType:@"button_a"]
```

```
radius:buttonRadius];  
  
_buttonA.position =  
    CGPointMake(_buttonB.position.x - radius - padding,  
               buttonRadius + padding);  
  
_buttonA.alpha = 0.5;  
_buttonA.name = @"ButtonA";  
[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 name property.

You are using the name property to hold a unique identifying string 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 name 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 actually trigger some action, which is what you'll take up next.

A simple jab attack

It's time to start putting the "beat 'em up" into your beat 'em up game. This is where the A and B action buttons come into play.

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, simply 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 `SKAction` in `Hero`.

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

```
//add inside if (self)
    textures = [self texturesWithPrefix:@"hero_attack_00"
                                startFrameIdx:0
                                frameCount:3];

    SKAction *attackAnimation =
        [SKAction animateWithTextures:textures
            timePerFrame:1.0/15.0];

    self.attackAction =
        [SKAction sequence:@[attackAnimation, [SKAction
    performSelector:@selector(idle) onTarget:self]]];
```

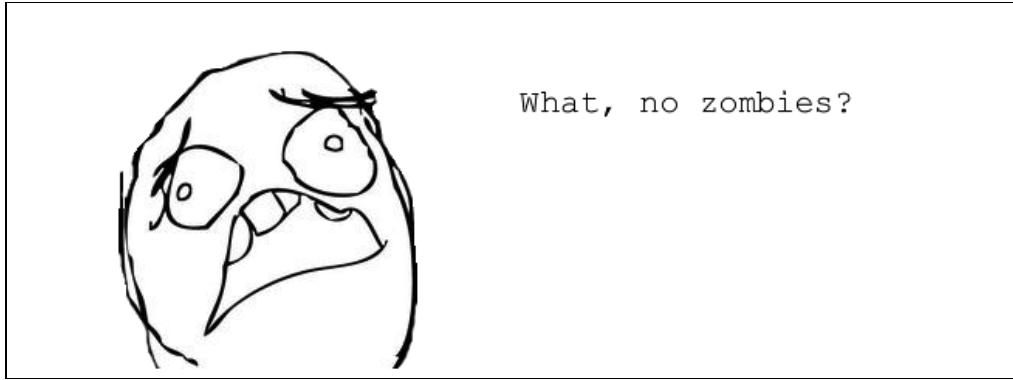
There are differences here compared with previous actions. The attack animation runs faster than the idle animation at 15 frames per second. Plus, the attack action only animates the attack animation once, and quickly switches back to the idle state by calling `idle`.

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

```
- (void)attack
{
    if (self.actionState == kActionStateIdle ||
        self.actionState == kActionStateWalk ||
        self.actionState == kActionStateAttack) {

        [self removeAllActions];
        [self runAction:self.attackAction];
        self.actionState = kActionStateAttack;
    }
}
```

Here you see a few more restrictions to the attack 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 in the process of 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 below the last `ActionDPadDelegate` method:

```
#pragma mark - ActionButtonDelegate methods  
  
- (void)actionButtonWasPressed:(ActionButton *)actionButton  
{  
    if ([actionButton.name isEqualToString:@"ButtonA"]) {  
        [self.hero attack];  
    }  
}  
  
- (void)actionButtonIsHeld:(ActionButton *)actionButton{  
}  
  
- (void)actionButtonWasReleased:(ActionButton *)actionButton{  
}
```

You make **GameLayer** implement the `ActionButtonDelegate` protocol so it can act as a delegate for the buttons. You then implement the protocol methods that **GameLayer**, as `ActionButton`'s delegate, needs to implement. You execute `actionButtonIsHeld:` and `actionButtonWasReleased:` whenever the player respectively holds or releases any button.

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



If the player presses either of the two buttons, then you execute `actionButtonWasPressed:.` You have to know which button the player is pressing, so you use the `name` property 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 right after _gameLayer.hud = _hudLayer  
_hudLayer.buttonA.delegate = _gameLayer;  
_hudLayer.buttonB.delegate = _gameLayer;
```

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

Build and run, and jab away!

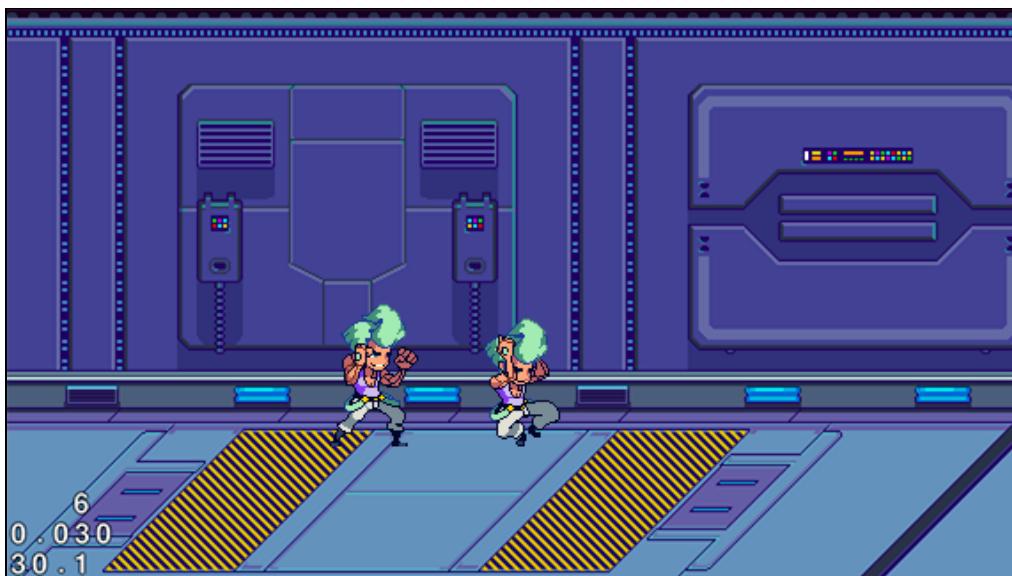


Jump up and get down!

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

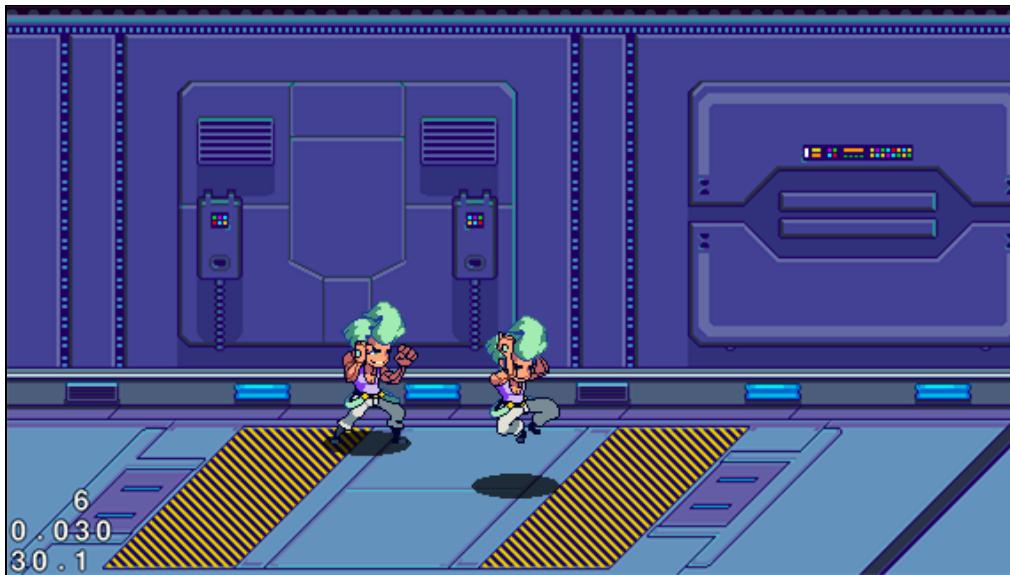


The character on the right is jumping, but he looks like he is just squatting on the same plane as the character on the left. It also looks like these two characters should collide when they bump into each other, but they should not. What's a sprite to do?

The shadow effect

Visually, this is easy to remedy: you can fake some three-dimensional depth using simple shadows. You'll always position the shadows on the ground position of the character—for the standing character, this is right below his feet; for the jumping character, 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, which 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—that is, how fast the character ascends 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 in how the sprite's position on the screen is decided.

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 will start with an initial jumping force, `kJumpForce`. There's also a gravity force, `kGravity`, that will constantly drag down the hero's velocity. This way, the

hero will leap up with a high velocity and then gradually slow down until he falls back to the ground.

The height of the hero's jump will depend on how long the player touches 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;
    self.shadow.position =
        CGPointMake(groundColor.x,
                    groundPosition.y - self.centerToBottom);
}

- (void)jumpRiseWithDirection:(CGPoint)direction
{
    if (self.actionState == kActionStateIdle) {
        [self jumpRise];
    }
    else if (self.actionState == kActionStateWalk ||
              self.actionState == kActionStateJumpLand) {

        self.velocity = CGPointMake(direction.x * self.walkSpeed,
                                    direction.y * self.walkSpeed);
        [self flipSpriteForVelocity:self.velocity];
        [self jumpRise];
    }
    else if (self.actionState == kActionStateRun) {

        self.velocity = CGPointMake(direction.x * self.runSpeed,
                                    direction.y * self.walkSpeed);
        [self flipSpriteForVelocity:self.velocity];
        [self jumpRise];
    }
}

- (void)jumpRise
{
    if (self.actionState == kActionStateIdle ||
        self.actionState == kActionStateWalk ||
        self.actionState == kActionStateRun ||
        self.actionState == kActionStateJumpLand) {

        [self removeAllActions];
        [self runAction:self.jumpRiseAction];
        self.jumpVelocity = kJumpForce;
        self.actionState = kActionStateJumpRise;
    }
}
```

You override the `setGroundPosition:` setter method and make sure that the shadow's position is always relative to the `groundPosition` of the `ActionSprite`. Here you have to use the `_groundPosition` instance variable directly instead of using the property to avoid getting into an infinite loop.

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. Recall that you haven't defined `jumpRiseAction` yet. This action will be defined in the individual subclass, not in `ActionSprite.m`.

Unlike other action states, jumping contains multiple sequential 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
- `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 (self.actionState == kActionStateJumpRise) {

        if (self.jumpVelocity > kJumpCutoff) {
            self.jumpVelocity = kJumpCutoff;
        }
    }
}

- (void)jumpFall
{
    if (self.actionState == kActionStateJumpRise ||
        self.actionState == kActionStateJumpAttack) {

        self.actionState = kActionStateJumpFall;
        [self runAction:self.jumpFallAction];
    }
}

- (void)jumpLand
{
```

```
if (self.actionState == kActionStateJumpFall ||  
    self.actionState == kActionStateRecover) {  
  
    self.jumpHeight = 0;  
    self.jumpVelocity = 0;  
  
    self.actionState = kActionStateJumpLand;  
    [self runAction:self.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. You'll define `jumpFallAction` later.

`jumpLand` zeroes out both `jumpHeight` and `jumpVelocity`, changes the state to `kActionStateJumpLand` and executes `jumpLandAction`, which you'll define for each individual `ActionSprite` subclass.

Both `groundPosition` 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:(NSTimeInterval)delta  
{  
  
    if (self.actionState == kActionStateWalk ||  
        self.actionState == kActionStateRun) {  
  
        CGPoint point = CGPointMultiplyScalar(self.velocity, delta);  
        self.desiredPosition =  
            CGPointAdd(self.groundPosition, point);  
    } else if (self.actionState == kActionStateJumpRise) {  
  
        CGPoint point = CGPointMultiplyScalar(self.velocity, delta);  
        self.desiredPosition =  
            CGPointAdd(self.groundPosition, point);  
  
        self.jumpVelocity -= kGravity * delta;  
        self.jumpHeight += self.jumpVelocity * delta;  
  
        if (self.jumpVelocity <= kJumpForce/2) {  
            [self jumpFall];  
        }  
    }  
}
```

```
    } else if (self.actionState == kActionStateJumpFall) {  
  
        CGPoint point = CGPointMakeMultiplyScalar(self.velocity, delta);  
        self.desiredPosition =  
            CGPointMakeAdd(self.groundPosition, point);  
  
        self.jumpVelocity -= kGravity * delta;  
        self.jumpHeight += self.jumpVelocity * delta;  
  
        if (self.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.

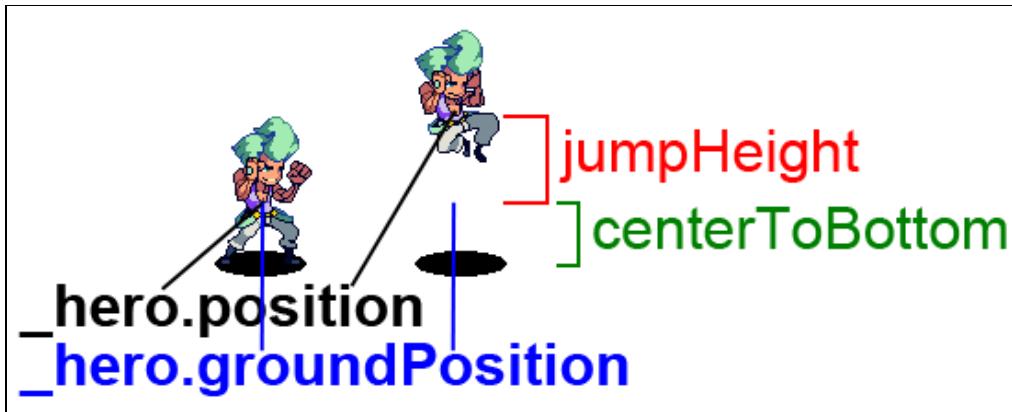
Once **jumpVelocity** falls below half of its original value, then the **jumpFall** method is executed.

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 self.hero.position = CGPointMake(posX, posY) with the  
following:  
self.hero.groundPosition = CGPointMake(posX, posY);  
self.hero.position = CGPointMake(self.hero.groundPosition.x,  
                                self.hero.groundPosition.y +  
                                self.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 **groundColor**. The onscreen position is then a combination of both the **groundColor** 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 **groundColor** 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.name isEqualToString:@"ButtonA"]) {
        [self.hero attack];
    } else if ([actionButton.name isEqualToString:@"ButtonB"]) {
        CGPoint directionVector =
        [self vectorForDirection:self.hud.dPad.direction];
        [self.hero jumpRiseWithDirection:directionVector];
    }
}

- (void)actionButtonWasReleased:(ActionButton *)actionButton
{
    if ([actionButton.name isEqualToString:@"ButtonB"]) {
        [self.hero jumpCutoff];
    }
}
```

Here you add code to handle the case where the user taps the B button. Remember that a delegate object can act as delegate to multiple objects, which is why you added an else-if statement to detect the B button in the method above.

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 player releases the B button, the hero cuts off his jump.

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

Go to `initHero` and add this line:

```
//add before [self addChild:self.hero]
[self.hero.shadow setScale:kPointFactor];
[self addChild:self.hero.shadow];
```

You add the shadow to the scene before the hero himself. This is to make sure that the game draws the shadow first so it appears behind the hero. You set the scale to `kPointFactor` so that the image scales according to the device.

You're almost there! All that's left 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 = [SKSpriteNode spriteNodeWithTexture:[[SKTextureCache
sharedInstance] textureNamed:@"shadow_character"]];
self.shadow.alpha = 0.75;

//jump animation
NSArray *jumpRiseFrames = @[[[SKTextureCache sharedInstance]
textureNamed:@"hero_jump_05"], [[SKTextureCache sharedInstance]
textureNamed:@"hero_jump_00"]];
self.jumpRiseAction = [SKAction animateWithTextures:jumpRiseFrames
timePerFrame:1.0/12.0];

//fall animation
self.jumpFallAction = [SKAction animateWithTextures:[self
texturesWithPrefix:@"hero_jump" startFrameIdx:1 frameCount:4]
timePerFrame:1.0/12.0];

//land animation
self.jumpLandAction = [SKAction sequence:@[[SKAction
setTexture:[[SKTextureCache sharedInstance]
textureNamed:@"hero_jump_05"]], [SKAction waitForDuration:1.0/12.0],
[SKAction performSelector:@selector(idle) onTarget:self]]];
```

You just did the following:

1. You create the **shadow** sprite using the **shadow_character.png** sprite frame included in the sprite sheet.
2. You create the **jumpRise** animation and action using two specific frames from the sprite sheet. You can't use the convenience function for creating animations here because the two sprite frames aren't in sequence.
3. You create the **jumpFall** animation and action using the now-familiar method.
4. You create the **jumpLand** animation and action. This action doesn't use the conventional animation action because you only have a single frame for the landing action. The `setTexture` SKAction changes the texture of the sprite to what you want before the sprite transitions to the idle action after 1/12th of a second.

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 put together the **Hero** class, it should be easy to create a new character using different images. For the next character, though, you'll spice things up by assembling it piece-by-piece.

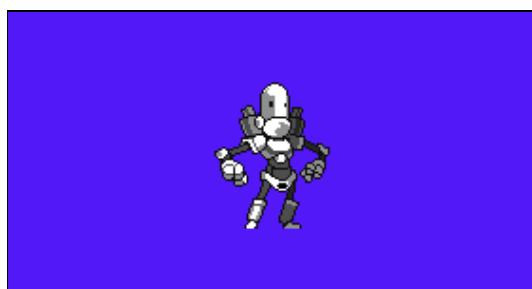
Assembling the robot

Introducing the **Robot**:

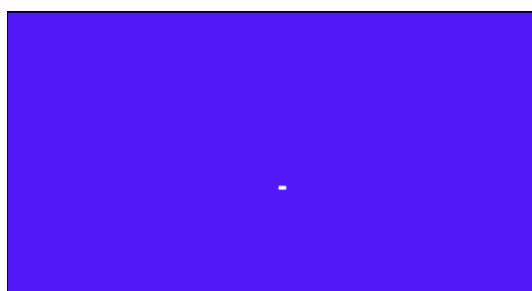


What you see here is the robot's complete form. In reality, it is drawn using three separate sprite components.

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\Cocoa Touch\Objective-C class** template and click **Next**. Enter **Robot** for Class and **ActionSprite** for Subclass of. Click **Next** and then **Create**.

Open **Robot.h** and add the following properties:

```
//add before @end
@property (strong, nonatomic) SKSpriteNode *belt;
@property (strong, nonatomic) SKSpriteNode *smoke;
```

Switch to **Robot.m** and add the following:

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

//add this method
- (instancetype)init
{
    SKTTextureCache *cache = [SKTTextureCache sharedInstance];
    NSString *textureName = @"robot_base_idle_00";
    SKTexture *texture = [cache textureNamed:textureName];

    self = [super initWithTexture:texture];
    if (self) {

        SKTexture *beltTeture =
            [cache textureNamed:@"robot_belt_idle_00"];
        self.belt =
            [SKSpriteNode spriteNodeWithTexture:beltTeture];

        SKTexture *smokeTexture =
            [cache textureNamed:@"robot_smoke_idle_00"];
        self.smoke =
            [SKSpriteNode spriteNodeWithTexture:smokeTexture];

        SKTexture *shadowTexture =
```

```
[cache textureNamed:@"shadow_character"];
self.shadow =
[SKSpriteNode spriteNodeWithTexture:shadowTexture];

self.shadow.alpha = 0.75;

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 you 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 by overriding certain methods in the SKSpriteNode superclass. That's just a fancy way of saying that you will replace, or add to, the implementation of a method that already exists in the superclass.

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

```
- (void)setPosition:(CGPoint)position
{
    [super setPosition:position];
    self.belt.position = position;
    self.smoke.position = position;
}

- (void)setXScale:(CGFloat)xScale
{
    [super setXScale:xScale];
    self.belt.xScale = xScale;
    self.smoke.xScale = xScale;
}

- (void)setYScale:(CGFloat)yScale
{
    [super setYScale:yScale];
    self.belt.yScale = yScale;
    self.smoke.yScale = yScale;
}

- (void)setScale:(CGFloat)scale
{
    [super setScale:scale];
    self.belt.scale = scale;
    self.smoke.scale = scale;
}
```

```

}

- (void)setHidden:(BOOL)hidden
{
    [super setHidden:hidden];
    self.belt.hidden = hidden;
    self.smoke.hidden = hidden;
}

- (void)setZPosition:(CGFloat)zPosition
{
    [super setZPosition:zPosition];
    self.smoke.zPosition = zPosition;
    self.belt.zPosition = zPosition;
}

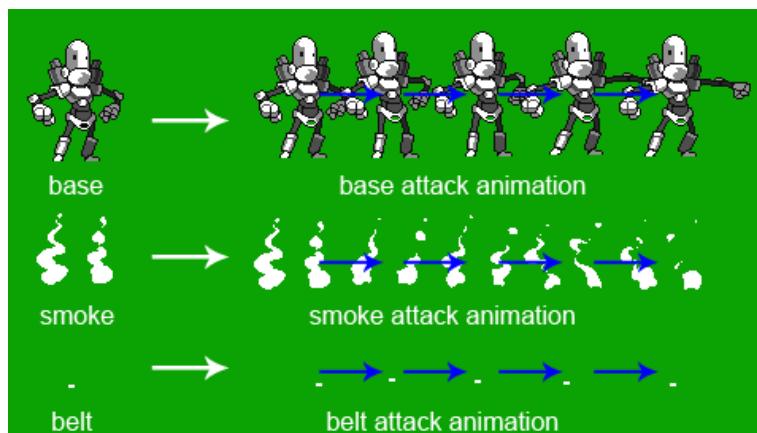
```

All of these methods are originally implemented in `SKSpriteNode`. When you set the position of a sprite through the `position` property, internally the `setPosition:` setter method is executed. The same goes for the `scale`, `hidden` and `zPosition` properties, and for every other property for that matter!

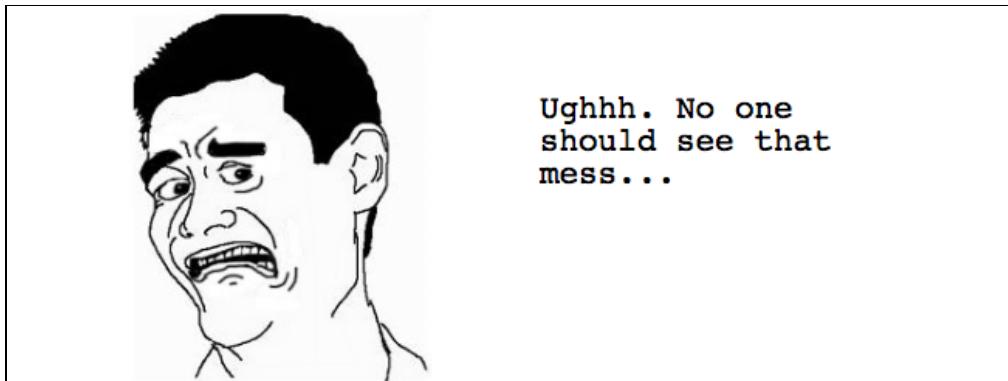
Now, whenever the base sprite's properties change, the belt and smoke sprite's properties change along with it.

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:



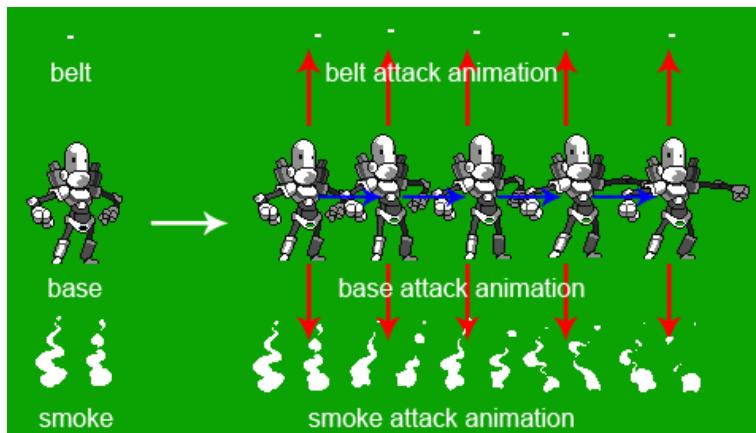
But there's a problem: if you create three separate animation actions and have each of these sprites executing its own animations, there is a big possibility that they will go out of sync with one another, especially since you are dealing with very short intervals per frame. Not to mention that it will be three times the work to start and stop each animation.



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 does.

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.

The running actions should correspond like this:



The base sprite should only need to run the base animation action, and then every step taken by that animation action should trigger the same step on the other two animation actions. This will ensure that all three actions are synchronized at all times.

The first step in implementing the above is to make actions that you can group.

Select the **Controls** group in Xcode, go to **File\New\New File**, choose the **iOS\Cocoa Touch\Objective-C class** template and click **Next**. Enter **AnimationMember** for Class and **NSObject** for Subclass of. Click **Next** and then **Create**.

AnimationMember will pair a group of animation frames to a target sprite. The goal is for each target sprite to animate through its animation frames when a grouped animation action starts.

Open **AnimationMember.h** and add the following before @end:

```
@property (strong, nonatomic) NSMutableArray *textures;
@property (weak, nonatomic) SKSpriteNode *target;
@property (assign, nonatomic) NSInteger currentIndex;

+ (instancetype)animationWithTextures:(NSMutableArray *)textures
                                target:(SKSpriteNode *)target;

- (instancetype)initWithTextures:(NSMutableArray *)textures
                                target:(SKSpriteNode *)target;

- (void)animateToIndex:(NSInteger)index;
```

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

```
+ (instancetype)animationWithTextures:(NSMutableArray *)textures
                                target:(SKSpriteNode *)target
{
    return [[self alloc] initWithTextures:textures target:target];
}

- (instancetype)initWithTextures:(NSMutableArray *)textures
                                target:(SKSpriteNode *)target
{
    if (self = [super init]) {

        _textures = textures;
        _target = target;
        _currentIndex = 0;
    }
    return self;
}

- (void)animateToIndex:(NSInteger)index
{
    if (index < self.textures.count) {

        SKTexture *texture = self.textures[index];

        if (texture != self.target.texture) {
            [self.target setTexture:texture];
            self.currentIndex = index;
        }
    }
}
```

AnimationMember just acts as the mediator between a sprite and its animation. You initialize it with a target sprite and an array of textures that comprise that sprite's animation.

Note: AnimationMember's target property is a weak reference so as to avoid retain cycles. Remember that if two objects reference each other, SKSpriteNode and AnimationMember in this case, at least one of those references has to be weak.

animateToIndex: is able to pick out a specific frame from the texture array and change the SKSpriteNode's image to the retrieved frame.

The next step is to group these animation members together. To do this, you have to create your own custom SKAction that can animate a group of AnimationMember objects.

Switch to **ActionSprite.m** and add the following:

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

//add this method

- (SKAction *)animateActionForGroup:(NSMutableArray *)group
    timePerFrame:(NSTimeInterval)timeInterval
    frameCount:(NSInteger)frameCount
{
    NSTimeInterval duration = timeInterval * frameCount;

    SKAction *action = [SKAction customActionWithDuration:duration
actionBlock:^(SKNode *node, CGFloat elapsedTime){

        int index = elapsedTime / timeInterval;
        for (int i = 0; i < group.count; ++i)
        {
            AnimationMember *animationMember = group[i];
            if (animationMember)
                [animationMember animateToIndex:index];
        }
    }];
    return action;
}
```

animateActionForGroup takes in a group of AnimationMembers and animates them using the timeInterval and frameCount.

The code inside actionBlock: will run in a loop, much like the update: loop. At every interval, this action computes the index for the frame that the animation members need to display.

The animation has to switch frames at every timeInterval, so you divide the current elapsedTime by timeInterval to get the index of the next frame. Then, you tell each AnimationMember in the group to animate to that index.

You need one `AnimationMember` for each part of the robot (body, belt and smoke) for each animation. This will be quite a lot of work, so it's a good idea to make a helper method that produces grouped animations for the `Robot` class.

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

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

//add this method
- (SKAction *)animateActionForActionWord:(NSString *)actionKeyWord
timePerFrame:(NSTimeInterval)timeInterval
frameCount:(NSUInteger)frameCount
{
    AnimationMember *baseAnimation = [AnimationMember
animationWithTextures:[self texturesWithPrefix:[NSString
stringWithFormat:@"robot_base_%@"], actionKeyWord] startFrameIdx:0
frameCount:frameCount] target:self];

    AnimationMember *beltAnimation = [AnimationMember
animationWithTextures:[self texturesWithPrefix:[NSString
stringWithFormat:@"robot_belt_%@"], actionKeyWord] startFrameIdx:0
frameCount:frameCount] target:_belt];

    AnimationMember *smokeAnimation = [AnimationMember
animationWithTextures:[self texturesWithPrefix:[NSString
stringWithFormat:@"robot_smoke_%@"], actionKeyWord] startFrameIdx:0
frameCount:frameCount] target:_smoke];

    NSMutableArray *actionGroup = [@[baseAnimation, beltAnimation,
smokeAnimation] mutableCopy];

    return [self animateActionForGroup:actionGroup
                           timePerFrame:timeInterval
                           frameCount:frameCount];
}
```

This method makes it easy to create a grouped animation action for the robot in one shot. 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:

- An `AnimationMember` with sprite frame animations targeting the base sprite.
- An `AnimationMember` with sprite frame animations targeting the `belt` sprite.
- An `AnimationMember` with sprite frame animations targeting the `smoke` sprite.
- A custom animation `SKAction` using an array containing the three `AnimationMembers` above as the member animations.

Each animation contains a sprite frame name that starts with a descriptive prefix (`robot_base_`, `robot_smoke_` or `robot_belt_`) and combines this with the action suffix. The string formed should reflect the texture name of each texture.

You are expecting all of these animations to have the same number of frames and the same interval 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.alpha = 0.75

//idle animation
SKAction *idleAnimationGroup =
    [self animateActionForActionWord:@"idle"
        timePerFrame:1.0/12.0
        frameCount:5];
self.idleAction =
    [SKAction repeatActionForever:idleAnimationGroup];

//attack animation
SKAction *attackAnimationGroup =
    [self animateActionForActionWord:@"attack"
        timePerFrame:1.0/15.0
        frameCount:5];

self.attackAction =
    [SKAction sequence:@[attackAnimationGroup, [SKAction
performSelector:@selector(idle) onTarget:self]]];

//walk animation
SKAction *walkAnimationGroup =
    [self animateActionForActionWord:@"walk"
        timePerFrame:1.0/12.0
        frameCount:6];

self.walkAction =
    [SKAction repeatActionForever:walkAnimationGroup];
```

You create three actions—idle, attack and walk—using the helper method you just wrote for the robot. These work much the same way as the hero's animation actions, except that triggering these actions animates three sprites at once.

Your robot army

Now jump straight to filling the game level with a horde of these robots. Switch to **GameLayer.h** and add the following property:

```
@property (strong, nonatomic) NSMutableArray *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 below – (void) initHero
– (void) initRobots {

    NSInteger robotCount = 50;
    self.robots = [NSMutableArray arrayWithCapacity:robotCount];

    for (NSInteger i = 0; i < robotCount; i++) {

        Robot *robot = [Robot node];
        [self addChild:robot.shadow];
        [self addChild:robot.smoke];
        [self addChild:robot];
        [self addChild:robot.belt];
        [self.robots addObject:robot];

        int minX = SCREEN.width + robot.centerToSides;
        int maxX = self.tileMap.mapSize.width *
                    self.tileMap.tileSize.width *
                    kPointFactor - robot.centerToSides;
        int minY = robot.centerToBottom;
        int maxY = 3 * self.tileMap.tileSize.height *
                    kPointFactor + robot.centerToBottom;

        robot.xScale = -kPointFactor;
        robot.yScale = kPointFactor;
        [robot.shadow setScale:kPointFactor];
        robot.position = CGPointMake(RandomIntRange(minX, maxX),
                                     RandomIntRange(minY, maxY));
        robot.groundPosition = robot.position;
        robot.desiredPosition = robot.position;
        [robot idle];
    }
}
```

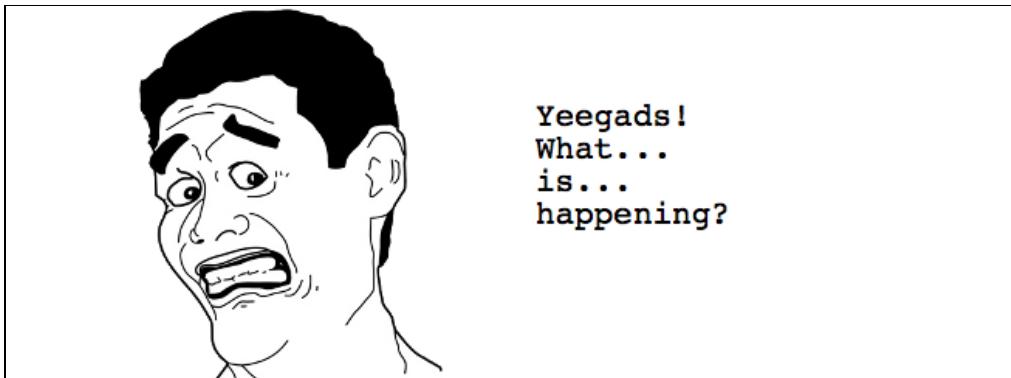
The above does the following:

1. It creates an array of 50 robots and adds all of their components to the batch node.
2. It uses the random functions in SKTUtils to place the 50 robots across the tile map's floor. It 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. The code scales the components to support universal display. The robots' xScale values are negative so that they initially face left.
4. Finally, the code 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.



Sprite Kit can handle drawing order through a node's `zPosition` property. Nodes with higher `zPositions` are drawn in front of objects with lower `zPositions`, and these `zPositions` are added from parent node to child node.

To fix the broken drawing sequence, you need to handle the `zPosition` dynamically. Every time a sprite moves across the screen vertically, its `zPosition` should change. The higher a sprite is on the screen, the lower its `zPosition` should be.

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

```
//add inside update:, after [self updatePositions];
[self reorderActors];

//add these methods
- (CGFloat)getZFromYPosition:(CGFloat)yPosition
{
```

```
    return SCREEN.height - yPosition;
}

- (void)reorderActors
{
    CGFloat zPosition =
        [self getZFromYPosition:self.hero.shadow.position.y];

    self.hero.zPosition = zPosition;

    for (Robot *robot in self.robots) {
        zPosition =
            [self getZFromYPosition:robot.shadow.position.y];
        robot.zPosition = zPosition;
    }
}
```

`getZFromYPosition:` returns an integer ranging from 0 to the total screen height. Every time the game updates the sprite positions, `reorderActors` changes the `zPosition` of each character based on how far it is from the bottom of the map. As the character moves higher along the y-axis, the resulting `zPosition` goes down.

You use `groundPosition` instead of `position` so that jumping does not affect the `zPosition` of the characters.

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

```
- (void)setZPosition:(CGFloat)zPosition
{
    [super setZPosition:zPosition];

    if (self.shadow) {
        self.shadow.zPosition = zPosition;
    }
}
```

You override `setZPosition:` in `ActionSprite` so that the sprite's shadow follows its `zPosition` all the time. Remember that `setZPosition:` is the setter method for `SKNode`'s `zPosition` property. This change in how `zPosition` is handled will affect `ActionSprite` subclasses, such as `Hero` and `Robot`.

Build and run, and the drawing sequence should now be fixed. Well, sort of...



As soon as you fix one thing, something else breaks! Now that the characters are all drawn in the right order, they appear in front of the D-pad and buttons!

This is because you made the `zPosition` of the sprites higher than the `zPosition` of the D-pad and buttons. Now why'd you do a thing like that? ;]

To fix this, go to **GameScene.m** and add this line inside `initWithSize:`:

```
//add after _hudLayer = [HudLayer node];
_hudLayer.zPosition = _gameLayer.zPosition + SCREEN.height;
```

In `getZFromYPosition:` earlier, a character's max `zPosition` is equal to the height of the screen. For the whole HudLayer to be on top of everything in GameLayer, its `zPosition` has to be higher than GameLayer's `zPosition` plus the highest `zPosition` of any one of GameLayer's child nodes.

In effect, all the children of HudLayer will have a higher `zPosition` than any children of GameLayer.

Build and run to see the issue resolved once and for all.



No fighting yet, boys. No one's blown the whistle.

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 is a special reason why the robots are white and comprised of three different parts.

If you've ever seen games like *Tiny Tower* or *Pocket Planes*, you may have wondered how they are 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, which involves drawing the same sprite using different color sets or palettes. In Sprite Kit, changing the color of a sprite is much easier thanks to `SKSpriteNode`'s `color` property.

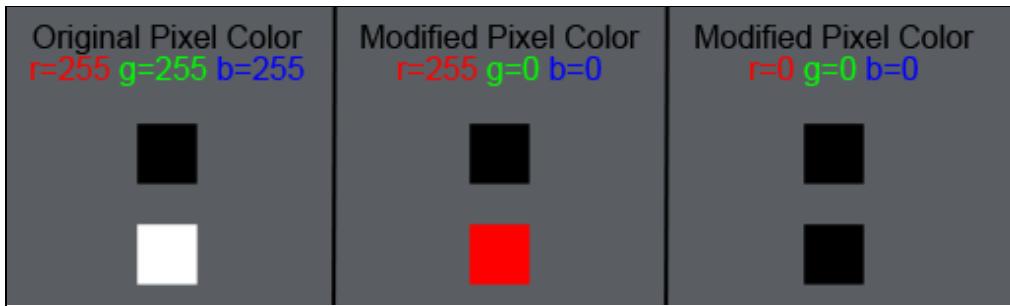
Note: You'll notice that `SKSpriteNode`'s `color` property isn't of data type `UIColor`. Rather, it is an `SKColor`. A what, you ask? Not to worry—`SKColor` resolves to `UIColor` in iOS and `NSColor` in OS X.

Changing the `color` property of an `SKSpriteNode` 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 1, where 0 is no color and 1 is full color for that particular color component (red, green or blue).

By default, an `SKSpriteNode`'s color value is 1 red, 1 green and 1 blue, meaning that all three represent the `SKSpriteNode` in its original full color. When it is 0 red, 0 green and 0 blue, the `SKSpriteNode` 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 (1, 1, 1) for the original pixel color, you cannot change the color of an `SKSpriteNode` 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's affected by color tinting.

This is why the robot is colored the way it is in the game:



Based on the `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 that keeps the shading consistent.

As you might suspect, if you change the `color` property of an `SKSpriteNode`, it will affect the whole image. This is why your robots have three different components: so that the belt, smoke and base colors can all be different. If the robot's body is green, for example, the smoke doesn't necessarily need to be green as well.

Stay tuned—there will be green smoke! Right now it's time to put all that theory to the test.

Go to **Defines.h** and add this definition:

```
typedef NS_ENUM(NSInteger, ColorSet) {  
    kColorLess = 0,  
    kColorCopper,  
    kColorSilver,  
    kColorGold,  
    kColorRandom,  
};
```

You will use these definitions as presets to determine how to color each robot.

Switch to **Robot.h** and add this property:

```
@property (assign, nonatomic) ColorSet colorSet;
```

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

```
- (void)setColorSet:(ColorSet)colorSet  
{  
    _colorSet = colorSet;  
  
    if (colorSet == kColorLess) {  
  
        self.color = [UIColor whiteColor];  
        self.belt.color = [UIColor whiteColor];  
        self.smoke.color = [UIColor whiteColor];  
  
    } else if (colorSet == kColorCopper) {  
  
        self.color = [UIColor colorWithRed:255/255.0  
                                     green:193/255.0  
                                      blue:158/255.0  
                                         alpha:1.0];  
  
        self.belt.color = [UIColor colorWithRed:99/255.0  
                                     green:162/255.0  
                                      blue:1.0  
                                         alpha:1.0];  
  
        self.smoke.color = [UIColor colorWithRed:220/255.0  
                                     green:219/255.0  
                                      blue:182/255.0  
                                         alpha:1.0];  
  
    } else if (colorSet == kColorSilver) {  
  
        self.color = [UIColor whiteColor];
```

```
self.belt.color = [UIColor colorWithRed:99/255.0
                                green:1.0
                                 blue:128/255.0
                                alpha:1.0];

self.smoke.color = [UIColor colorWithRed:128/255.0
                                green:128/255.0
                                 blue:128/255.0
                                alpha:1.0];

} else if (colorSet == kColorGold) {

    self.color = [UIColor colorWithRed:233/255.0
                                green:177/255.0
                                 blue:0
                                alpha:1.0];

    self.belt.color = [UIColor colorWithRed:109/255.0
                                green:40/255.0
                                 blue:25/255.0
                                alpha:1.0];

    self.smoke.color = [UIColor colorWithRed:222/255.0
                                green:129/255.0
                                 blue:82/255.0
                                alpha:1.0];

} else if (colorSet == kColorRandom) {

    self.color =
        [UIColor colorWithRed:RandomFloatRange(0, 1)
                     green:RandomFloatRange(0, 1)
                      blue:RandomFloatRange(0, 1)
                     alpha:1.0];

    self.belt.color =
        [UIColor colorWithRed:RandomFloatRange(0, 1)
                     green:RandomFloatRange(0, 1)
                      blue:RandomFloatRange(0, 1)
                     alpha:1.0];

    self.smoke.color =
        [UIColor colorWithRed:RandomFloatRange(0, 1)
                     green:RandomFloatRange(0, 1)
                      blue:RandomFloatRange(0, 1)
                     alpha:1.0];
}

}
```

Whenever you change the `colorSet` property of a robot, you're also changing the base, belt and smoke colors according to the preset.

kColorLess changes all colors to [UIColor whiteColor], which is just a shortcut definition for an all 1.0 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.

Still in **Robot.m**, add the following lines to **init**:

```
//add inside if (self)
self.colorBlendFactor = 1.0;
self.belt.colorBlendFactor = 1.0;
self.smoke.colorBlendFactor = 1.0;
```

An SKSpriteNode's **colorBlendFactor** controls how much a sprite's texture color is affected by the **color** property. You set **colorBlendFactor** for the three sprites to 1, which is the maximum value for the property, so the sprites get fully tinted the way I described before.

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 random colors for their base, belt and smoke sprites.

Build and run. The robots have gone psychedelic!



Building on the first two chapters, you've added a run action, buttons that trigger punching and jumping, and colorful enemies. Not bad for one chapter!

Stay tuned for the next chapter, where the Pompadoured Hero will get his first chance to beat these robots into scrap metal!

Challenge: If you did the challenge from Chapter 2, you have replaced the hero's idle and walk animations with your own drawings. For this chapter, replace the rest of the animations as well to fully animate your personalized hero.

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 the Pompadoured Hero run, jump and punch, and added an army of colorful robots to the scene.

However, there's a big problem with the current implementation. Your hero's punches don't connect with anything! If this is a beat 'em up game, he should be able to crack some heads, right?

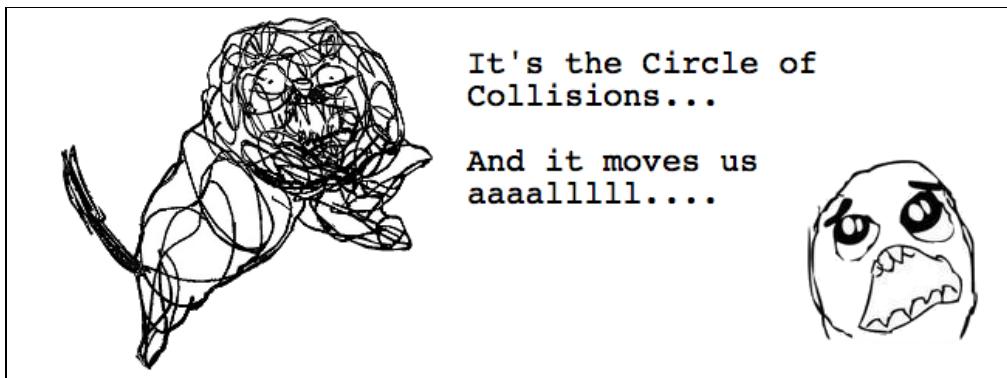
In this chapter, you will implement collision detection so punching will actually damage the robots. Along the way, you'll learn how to handle some tricky challenges such as how to make the collision detection vary based on the animation frame and how to add debug drawing.

Get ready to lay some smack down!

Your collision strategy

To get the hero's punches to connect with the robots, you need to implement a strategy for detecting collisions. One way to do this is to create collision boxes or rectangles to represent each character and check if these rectangles intersect.

For this game, however, you'll employ a more versatile and precise system that uses 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, such as a hand when it punches or a short energy burst.
3. **Detection circles:** 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, therefore, will 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 you'll use it 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 2, you declared properties for the aforementioned circles of the ActionSprite. For reference, here they are again:

```
//collision
@property (strong, nonatomic) NSMutableArray *contactPoints;
@property (strong, nonatomic) NSMutableArray *attackPoints;
@property (assign, nonatomic) CGFloat detectionRadius;
```

These are as follows:

- **contactPoints:** An array of contact circles.
- **attackPoints:** An array 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.

Now open **ActionSprite.m** and add the following helper method:

```
- (NSMutableArray *)contactPointArray:(NSUInteger)size
{
    NSMutableArray *array =
        [[NSMutableArray alloc] initWithCapacity:size];

    for (NSInteger i = 0; i < size; i++) {

        ContactPoint contactPoint;
        contactPoint.offset = CGPointMakeZero;
        contactPoint.position = CGPointMakeZero;
        contactPoint.radius = 0;

        NSValue *value =
        [NSValue valueWithBytes:&contactPoint
            objCType:@encode(ContactPoint)];

        [array addObject:value];
    }

    return array;
}
```

You'll use `contactPointArray:` later on to populate the attack point and contact point arrays. Notice that since `ContactPoint` is technically a C struct and not an Objective-C object, you can't add it directly to the `NSMutableArray`. This is why you first have to store it inside an `NSValue` before adding it to the array.

With that helper method in place, switch to **Hero.m** and add the following:

```
//add to the end of init inside the curly braces of if (self)

self.detectionRadius = 100.0 * kPointFactor;

self.attackPoints = [self contactPointArray:3];
self.contactPoints = [self contactPointArray:4];

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 calculated this number in the "Optional Exercise" section that is coming up.

You also create two mutable arrays, one to hold attack points and the other to hold contact points. `contactPointArray`: initializes these arrays with placeholder `ContactPoints` that you'll populate later.

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.contactPoints = [self contactPointArray:4];
self.attackPoints = [self contactPointArray:1];

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 many hit points or as much attack damage or attack force as the hero. This is logical, because otherwise your hero would never be able to defeat a horde of robots. ☺

At some point, you'll 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 this easier.

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

```
// 1
- (ContactPoint)contactPointWithOffset:(const CGPoint)offset
                                    radius:(const CGFloat)radius
{
    ContactPoint contactPoint;

    contactPoint.offset =
        CGPointMakeMultiplyScalar(offset, kPointFactor);

    contactPoint.radius = radius * kPointFactor;

    contactPoint.position =
        CGPointMakeAdd(self.position, contactPoint.offset);

    return contactPoint;
```

```

}

// 2
- (void)modifyPoint:(ContactPoint *)point
    offset:(const CGPoint)offset
    radius:(const CGFloat)radius
{
    point->offset = CGPointMakeMultiplyScalar(offset, kPointFactor);
    point->radius = radius * kPointFactor;
    point->position = CGPointMakeAdd(self.position, point->offset);
}

//3
- (void)modifyAttackPointAtIndex:(const NSUInteger)pointIndex
    offset:(const CGPoint)offset
    radius:(const CGFloat)radius
{
    NSValue *value = self.attackPoints[pointIndex];
    ContactPoint contactPoint;
    [value getValue:&contactPoint];

    [self modifyPoint:&contactPoint offset:offset radius:radius];

    self.attackPoints[pointIndex] =
        [NSValue valueWithBytes:&contactPoint
            objCType:@encode(ContactPoint)];
}

//3
- (void)modifyContactPointAtIndex:(const NSUInteger)pointIndex
    offset:(const CGPoint)offset
    radius:(const CGFloat)radius
{
    NSValue *value = self.contactPoints[pointIndex];
    ContactPoint contactPoint;
    [value getValue:&contactPoint];

    [self modifyPoint:&contactPoint offset:offset radius:radius];

    self.contactPoints[pointIndex] =
        [NSValue valueWithBytes:&contactPoint
            objCType:@encode(ContactPoint)];
}

```

Let's go over this section by section:

1. **contactPointWithOffset**: creates and returns 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:offset:radius** 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.

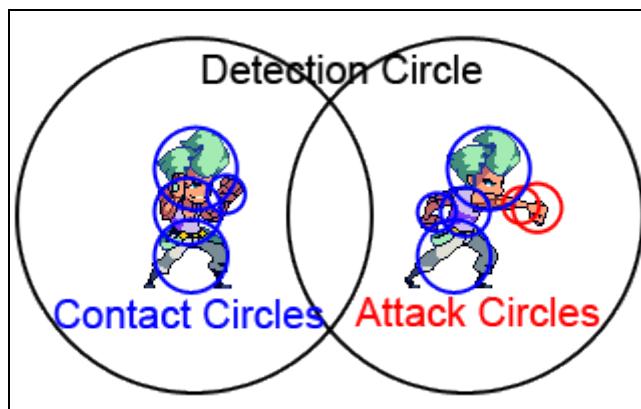
You also multiply `kPointFactor` for iPad conversion so you don't have to do the calculations individually later on.

Optional exercise: finding contact points

In this starter kit, I've figured out the contact points for each animation frame for you. You may be curious to learn how I came up with them so you can do the same with your own artwork.

If so, keep reading! But if you'd rather 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:



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?

In fact, there is such a program! Though it wasn't intended for this purpose, [PhysicsEditor](#) fits the bill nicely.

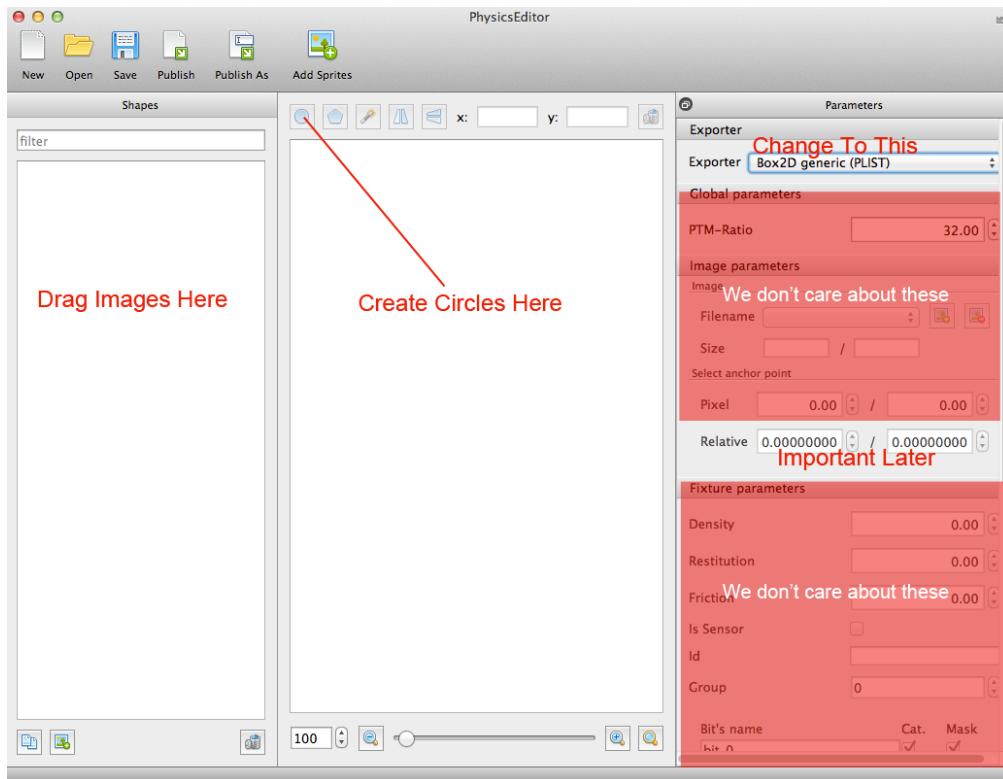
Note: PhysicsEditor is a tool for creating collision shapes for different physics engines such as Box2D and Chipmunk, two commonly-used 2D game physics engines. You can get PhysicsEditor [here](#):

<http://www.codeandweb.com/physicseditor/download>

If you haven't already done so, download PhysicsEditor and install it, and then fire it up. You will get a blank project with three panels/columns.

PhysicsEditor is pretty straightforward. On the left, you put all the images you want to work with. 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, which is simply figure out the offset and radius for each shape.

First, select **Box2D generic (PLIST)** as the Exporter from the right panel:



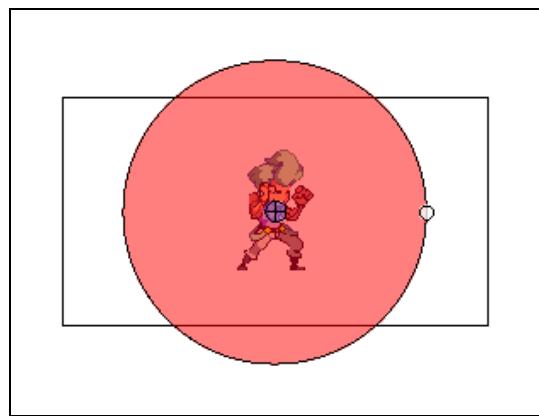
This will open the **Relative** options—the anchor point of the sprite, or in other words, the x- and y-origin of the sprite. In this starter kit, you've been using the center of the sprite, coordinates (0.5, 0.5), as the origin, so for PhysicsEditor to give out correct offset values, it also needs to 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**.

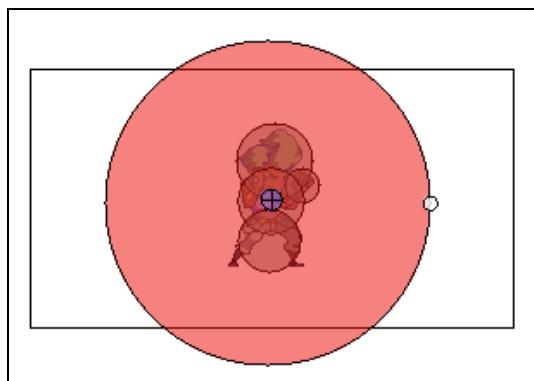
Next, in the center panel, change the magnification, configurable via a slider at the bottom of the panel, to a comfortable level, and then tap on the circle button on the upper part of this panel to create a circle shape. A new circle will appear at the lower-left portion of the frame.

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 for the hero and arrange and resize them like this:



For the hero's attack circle, drag **hero_attack_00_01.png** to the left panel, change Relative to 0.5 / 0.5 and create a circle near the hero's fist.



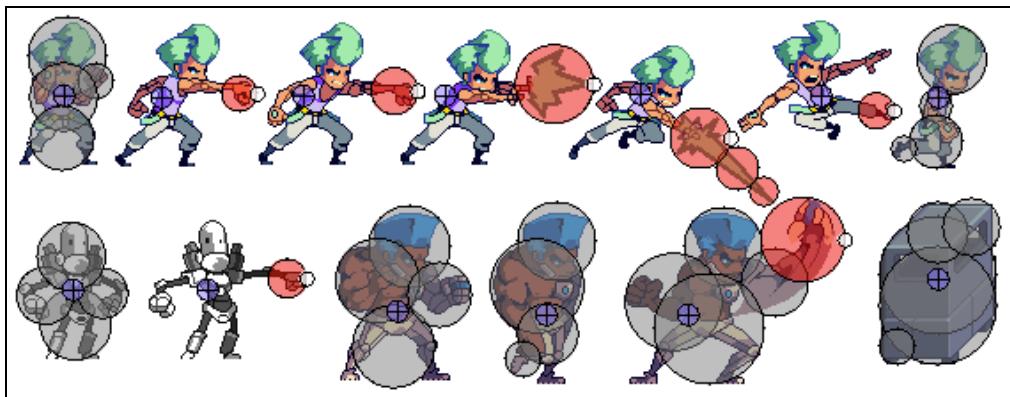
Add all the other relevant images from the folder named **Raw** in the starter kit to the left panel and define their circles one by one.

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 have to use your best judgment. For reference, here are some of my own mappings (excluding detection circles):



There are two frames present above that aren't included in the previous list: the walk frame of the hero and the walk frame of the boss (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 think you can keep using the same positions for some actions, there's no need to define circles for them. In this case, if you were to not define circles for the walk frame, it would simply use the same positions as the action that comes before it, which would most likely be the idle action.

When you're done, click **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, go to the **Raw** folder of your starter kit to find the **Circles.plist** file I created for you, which you can use to continue from this point.

You only want to look at the information contained in **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	Dictionary (2 items)
anchorpoint	String	{ 0.5000,0.5000 }
fixtures	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	Dictionary (2 items)
radius	Number	20 ← radius
position	String	{ 1.728,-19.498 } ← offset
Item 1	Dictionary	(10 items)
Item 2	circles	Dictionary (10 items)
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	Dictionary (2 items)

Inside one of the **images**, 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'll see the **radius** and the **position** (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 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.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
{

    for (NSInteger i = 0; i < self.contactPoints.count; i++) {

        NSValue *value = self.contactPoints[i];
        ContactPoint contactPoint;
        [value getValue:&contactPoint];

        CGPoint offset = CGPointMake(contactPoint.offset.x *
                                      self.directionX,
                                      contactPoint.offset.y);

        contactPoint.position = CGPointMakeAdd(self.position, offset);

        self.contactPoints[i] =
            [NSValue valueWithBytes:&contactPoint
                           objCType:@encode(ContactPoint)];
    }

    for (NSInteger i = 0; i < self.attackPoints.count; i++) {

        NSValue *value = self.attackPoints[i];
        ContactPoint contactPoint;
        [value getValue:&contactPoint];

        CGPoint offset = CGPointMake(contactPoint.offset.x *
                                      self.directionX,
                                      contactPoint.offset.y);

        contactPoint.position = CGPointMakeAdd(self.position, offset);

        self.attackPoints[i] =
    }
}
```

```

        [NSValue valueWithBytes:&contactPoint
                      objCType:@encode(ContactPoint)];
    }
}
// 4
- (void)setPosition:(CGPoint)position
{
    [super setPosition:position];
    [self transformPoints];
}

```

Let's go over this one method at a time:

1. **setActionState**: sets the `actionState` instance variable and 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. You execute this method every time the position of `ActionSprite` changes. Doing this ensures that all the contact and attack circles follow `ActionSprite` wherever it goes.
4. You override `setPosition` to call `transformPoints` each time the position of the sprite changes.

Now go to `Hero.m` and add the `setContactPointsForAction`: method:

```

- (void)setContactPointsForAction:(ActionState)actionState
{
    if (actionState == kActionStateIdle) {

        [self modifyContactPointAtIndex:0
                                  offset:CGPointMake(3.0, 23.0)
                                  radius:19.0];

        [self modifyContactPointAtIndex:1
                                  offset:CGPointMake(17.0, 10.0)
                                  radius:10.0];

        [self modifyContactPointAtIndex:2
                                  offset:CGPointZero
                                  radius:19.0];

        [self modifyContactPointAtIndex:3
                                  offset:CGPointMake(0.0, -21.0)
                                  radius:20.0];

    } else if (actionState == kActionStateWalk) {

```

```
[self modifyContactAtIndex:0
    offset:CGPointMake(8.0, 23.0)
    radius:19.0];

[self modifyContactAtIndex:1
    offset:CGPointMake(12.0, 4.0)
    radius:4.0];

[self modifyContactAtIndex:2
    offset:CGPointZero
    radius:10.0];

[self modifyContactAtIndex:3
    offset:CGPointMake(0.0, -21.0)
    radius:20.0];

} else if (actionState == kActionStateAttack) {

    [self modifyContactAtIndex:0
        offset:CGPointMake(15.0, 23.0)
        radius:19.0];

    [self modifyContactAtIndex:1
        offset:CGPointMake(24.5, 4.0)
        radius:6.0];

    [self modifyContactAtIndex:2
        offset:CGPointZero
        radius:16.0];

    [self modifyContactAtIndex:3
        offset:CGPointMake(0.0, -21.0)
        radius:20.0];

    [self modifyAttackAtIndex:0
        offset:CGPointMake(41.0, 3.0)
        radius:10.0];

    [self modifyAttackAtIndex:1
        offset:CGPointMake(41.0, 3.0)
        radius:10.0];

    [self modifyAttackAtIndex:2
        offset:CGPointMake(41.0, 3.0)
        radius:10.0];

} else if (actionState == kActionStateAttackTwo) {

    [self modifyAttackAtIndex:0
        offset:CGPointMake(51.6, 2.4)
        radius:13.0];

    [self modifyAttackAtIndex:1
```

```
        offset:CGPointMake(51.6, 2.4)
        radius:13.0];

    [self modifyAttackPointAtIndex:2
        offset:CGPointMake(51.6, 2.4)
        radius:13.0];

} else if (actionState == kActionStateAttackThree) {

    [self modifyAttackPointAtIndex:0
        offset:CGPointMake(61.8, 6.2)
        radius:22.0];

    [self modifyAttackPointAtIndex:1
        offset:CGPointMake(61.8, 6.2)
        radius:22.0];

    [self modifyAttackPointAtIndex:2
        offset:CGPointMake(61.8, 6.2)
        radius:22.0];

} else if (actionState == kActionStateRunAttack) {

    [self modifyAttackPointAtIndex:0
        offset:CGPointMake(31.2, -8.8)
        radius:10.0];

    [self modifyAttackPointAtIndex:1
        offset:CGPointMake(31.2, -8.8)
        radius:10.0];

    [self modifyAttackPointAtIndex:2
        offset:CGPointMake(31.2, -8.8)
        radius:10.0];

} else if (actionState == kActionStateJumpAttack) {

    [self modifyAttackPointAtIndex:2
        offset:CGPointMake(70.0, -55.0)
        radius:8.0];

    [self modifyAttackPointAtIndex:1
        offset:CGPointMake(55.0, -42.0)
        radius:12.0];

    [self modifyAttackPointAtIndex:0
        offset:CGPointMake(34.0, -25.0)
        radius:17.0];
}

}
```

That's quite a bit of code! Fortunately, it's all pretty simple to understand. This method moves all of the hero's contact and attack points based on the offsets and

radii taken from **Circles.plist** for each **actionState**. You can either use the values here for convenience, or substitute the values from your own **Circles.plist** (which may vary slightly).

Note: Some of the 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 would 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:CGPointMake(1.7, 19.5)
            radius:20.0];

        [self modifyContactPointAtIndex:1
            offset:CGPointMake(-15.5, 3.5)
            radius:16.0];

        [self modifyContactPointAtIndex:2
            offset:CGPointMake(17.0, 2.1)
            radius:14.0];

        [self modifyContactPointAtIndex:3
            offset:CGPointMake(-0.8, -18.5)
            radius:19.0];
    } else if (actionState == kActionStateWalk) {

        [self modifyContactPointAtIndex:0
            offset:CGPointMake(8.0, 23.0)
            radius:19.0];

        [self modifyContactPointAtIndex:1
            offset:CGPointMake(12.0, 4.0)
            radius:4.0];

        [self modifyContactPointAtIndex:2
            offset:CGPointZero
            radius:10.0];

        [self modifyContactPointAtIndex:3
            offset:CGPointMake(0.0, -21.0)]
    }
}
```

```
        radius:20.0];  
  
    } else if (actionState == kActionStateAttack) {  
  
        [self modifyContactPointAtIndex:0  
            offset:CGPointMake(15.0, 23.0)  
            radius:19.0];  
  
        [self modifyContactPointAtIndex:1  
            offset:CGPointMake(24.5, 4.0)  
            radius:6.0];  
  
        [self modifyContactPointAtIndex:2  
            offset:CGPointZero  
            radius:16.0];  
  
        [self modifyContactPointAtIndex:3  
            offset:CGPointMake(0.0, -21.0)  
            radius:20.0];  
  
        [self modifyAttackPointAtIndex:0  
            offset:CGPointMake(45.0, 6.5)  
            radius:10.0];  
    }  
}
```

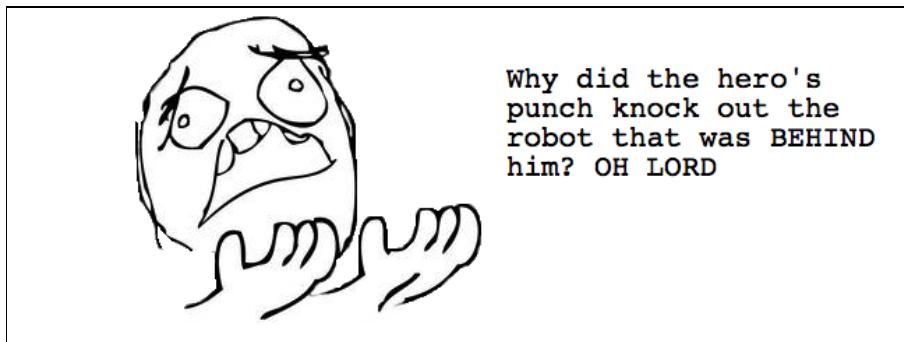
This is more of the same. The robot contains fewer arrangements because it has fewer actions than the hero.

Build and run. For now, just make sure the code compiles correctly and that everything still works on all devices. You have yet to implement actual collision detection, so you won't see much of a functional difference in the game.



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 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 use debug drawing for all the circles and rectangles you've created.

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.

Next, you'll make a new standalone node layer that handles debug drawing.

Select the **Game** group in Xcode, go to **File\New\New File**, choose the **iOS\Cocoa Touch\Objective-C class** template and click **Next**. Enter **DebugLayer** for Class and **SKNode** for Subclass of. Click **Next** and then **Create**.

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

```
#import <SpriteKit/SpriteKit.h>
#import "GameLayer.h"

@interface DebugLayer : SKNode

@property (strong, nonatomic) NSMutableArray *shapeNodes;

@property (weak, nonatomic) GameLayer *gameLayer;

+ (instancetype)nodeWithGameLayer:(GameLayer *)gameLayer;
- (instancetype)initWithGameLayer:(GameLayer *)gameLayer;
- (void)update:(NSTimeInterval)delta;

@end
```

Switch to **DebugLayer.m** and replace its contents with the following:

```
#import "DebugLayer.h"

@interface DebugLayer()

@property (assign, nonatomic) NSInteger shapeNodeIndex;

@end

@implementation DebugLayer

+ (instancetype)nodeWithGameLayer:(GameLayer *)gameLayer
{
    return [[self alloc] initWithGameLayer:gameLayer];
}

- (instancetype)initWithGameLayer:(GameLayer *)gameLayer
{
    if (self = [super init]) {

        _shapeNodes = [NSMutableArray arrayWithCapacity:30];
        _gameLayer = gameLayer;

        for (int i = 0; i < 30; ++i) {
            SKShapeNode *shapeNode = [SKShapeNode node];
            shapeNode.hidden = YES;
            shapeNode.antialiased = NO;
            [self addChild:shapeNode];
            [_shapeNodes addObject:shapeNode];
        }
    }

    return self;
}

@end
```

When you create an instance of DebugLayer, you need to link it to an instance of GameLayer that contains the objects whose collision shapes need to be drawn. You also create and cache 30 SKShapeNodes that DebugLayer can use.

SKShapeNodes are like SKSpriteNodes, but they contain geometric shapes rather than textures or images.

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

```
// 1
- (SKShapeNode *)getNextShapeNode
{
    SKShapeNode *shapeNode;

    if (self.shapeNodeIndex < self.shapeNodes.count) {
```

```
shapeNode = self.shapeNodes[self.shapeNodeIndex];
shapeNode.hidden = NO;

} else {

    shapeNode = [SKShapeNode node];
    shapeNode.antialiased = NO;
    [self addChild:shapeNode];
    [self.shapeNodes addObject:shapeNode];
}

self.shapeNodeIndex++;

return shapeNode;
}

// 2
- (void)drawCircleAtPosition:(CGPoint)position
    radius:(CGFloat)radius
    color:(UIColor *)color
    zPosition:(CGFloat)zPosition
{
    SKShapeNode *shapeNode = [self getNextShapeNode];
    shapeNode.position = position;
    shapeNode.path = CGPathCreateWithEllipseInRect((CGRect){ { -radius, -radius }, { radius * 2, radius * 2 } }, NULL);
    shapeNode.strokeColor = color;
    shapeNode.zPosition = zPosition;
}

// 3
- (void)drawRect:(CGRect)rect
    color:(UIColor *)color
    zPosition:(CGFloat)zPosition
{
    SKShapeNode *shapeNode = [self getNextShapeNode];
    shapeNode.position = CGPointMakeZero;
    shapeNode.path = CGPathCreateWithRect(rect, NULL);
    shapeNode.strokeColor = color;
    shapeNode.zPosition = zPosition;
}

// 4
- (void)drawShapesForActionSprite:(ActionSprite *)sprite
{
    if (!sprite.hidden) {

        [self drawCircleAtPosition:sprite.position
            radius:sprite.detectionRadius
            color:[UIColor blueColor]
            zPosition:sprite.zPosition];
    }
}
```

```

for (NSInteger i = 0; i < sprite.contactPoints.count; i++){

    NSValue *value = sprite.contactPoints[i];
    ContactPoint contactPoint;
    [value getValue:&contactPoint];

    [self drawCircleAtPosition:contactPoint.position
                      radius:contactPoint.radius
                      color:[UIColor greenColor]
                     zPosition:sprite.zPosition];
}

for (NSInteger i = 0; i < sprite.attackPoints.count; i++) {

    NSValue *value = sprite.attackPoints[i];
    ContactPoint attackPoint;
    [value getValue:&attackPoint];

    [self drawCircleAtPosition:attackPoint.position
                      radius:attackPoint.radius
                      color:[UIColor redColor]
                     zPosition:sprite.zPosition];
}

[self drawRect:sprite.feetCollisionRect
          color:[UIColor yellowColor]
         zPosition:sprite.zPosition];
}
}

```

You're adding the following:

1. `getNextShapeNode` gets a shape node using the `shapeNodeIndex` property. This integer represents the index of the next available unused `SKShapeNode` from the collection of `SKShapeNodes` created during initialization. When it has reached the end of the array, it creates a new `SKShapeNode`. The method also increments the `shapeNodeIndex` variable.
2. `drawCircleAtPosition:radius:color:zPosition:` retrieves the next available `SKShapeNode` using `getNextShapeNode` and gives it a circular shape using the node's `path` property, as well as a position, radius, stroke color and `zPosition`.
3. `drawRect:color:zPosition:` retrieves the next available `SKShapeNode` and gives it a rectangular shape, stroke color and `zPosition`. It sets the `position` property to `CGPointZero` because the `CGRect` will have the actual position value.
4. `drawShapesForActionSprite:` draws the debug shapes for a given `ActionSprite` class. Each `SKShapeNode` corresponds to a shape measurement from `ActionSprite`. Detection circles will be blue, contact circles green, attack circles red and feet rectangles yellow.

Add one more method to **DebugLayer.m**:

```

- (void)update:(NSTimeInterval)delta
{
    self.position = self.gameLayer.position;
    self.shapeNodeIndex = 0;

    [self drawShapesForActionSprite:self.gameLayer.hero];

    for (ActionSprite *robot in self.gameLayer.robots) {
        [self drawShapesForActionSprite:robot];
    }
}

```

At every time step in the game loop, DebugLayer will follow the position of GameLayer because GameLayer scrolls as the hero moves across the map. You reset the value of shapeNodeIndex to 0 so that calls to getNextShape node start from the beginning of the shapeNodes array. Then, you use drawShapesForActionSprite to draw the debug shapes for the hero and all the robots in GameLayer.

As you should know by now, DebugLayer's update method, like GameLayer's and HudLayer's, needs to run at every update step of GameScene.

Open **GameScene.m** and add the following:

```

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

//add this property inside @interface GameScene()
#if DRAW_DEBUG_SHAPES
@property (strong, nonatomic) DebugLayer *debugLayer;
#endif

//add to the end of if (self = [super initWithFrame:size])
#if DRAW_DEBUG_SHAPES
_debugLayer = [DebugLayer nodeWithGameLayer:_gameLayer];
_debugLayer.zPosition = _gameLayer.zPosition + 1;
[self addChild:_debugLayer];
#endif

//add to the end of update:
#if DRAW_DEBUG_SHAPES
[self.debugLayer update:delta];
#endif

```

Note the `#if DRAW_DEBUG_SHAPES` and `#endif` preprocessor directives before and after the code—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.

You create an instance of DebugLayer in GameScene and make sure it updates the debug shapes in the game loop.

Build and run, and you should see all the shapes on the screen!



How's that for visual confirmation of all your hard work?

Try out all of the different states (walking, running, jumping and punching) and make sure the shapes are OK for each.

Note: The way debug drawing works at this point isn't very efficient. Just take one look at the center number in the lower-right corner—the number of OpenGL draw calls the game is executing—and you will notice that it has shot up to a very high value.

Unlike `SKSpriteNodes`, `SKShapeNodes` aren't drawn in a batch, so every `SKShapeNode` added to the scene is one more draw call.

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 you've positioned the shapes correctly, so there's no need to keep debug drawing enabled.

Now that your 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:

- A **collision trigger**, or when to check for collisions.
- **Collision detection**, or a means to check for intersecting circles.

- **Collision resolution**, or the appropriate response for each type of collision.

When you implemented movement with the D-pad, you actually went through these same three steps.

With the D-pad, 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, when you implement collision detection you will already be sure that a given **ActionSprite** will react properly, and when you implement your collision trigger you will already know how to detect collisions.

Resolving collisions

It's time to bring on the pain! Go to **ActionSprite.m** and add the following method:

```
- (void)hurtWithDamage:(CGFloat)damage
                  force:(CGFloat)force
                 direction:(CGPoint)direction
{
    if (self.actionState > kActionStateNone &&
        self.actionState < kActionStateKnockedOut) {

        if (self.jumpHeight > 0) {

            [self knockoutWithDamage:damage direction:direction];
        } else {

            [self removeAllActions];
            [self runAction:self.hurtAction];
            self.actionState = kActionStateHurt;
            self.hitPoints -= damage;
            self.desiredPosition =
                CGPointMake(self.position.x + direction.x * force,
                           self.position.y);

            if (self.hitPoints <= 0) {
                [self knockoutWithDamage:0 direction:direction];
            }
        }
    }
}
```

If the sprite is in the middle of a jump during the collision, then you knock out the character via **knockoutWithDamage:**. If not, then **hurtWithDamage:** subtracts the damage amount from the current **hitPoints**, moves the **desiredPosition** using the force value multiplied by the x- and y-directions, and executes the hurt action.

If **hitPoints** reaches or falls below zero, then you knock out the sprite, but this time with zero damage because the damage has already been applied.

Still in **ActionSprite.m**, add **knockoutWithDamage**:

```
- (void)knockoutWithDamage:(CGFloat)damage
                      direction:(CGPoint)direction
{
    if (self.actionState != kActionStateKnockedOut &&
        self.actionState != kActionStateDead &&
        self.actionState != kActionStateRecover &&
        self.actionState != kActionStateNone) {

        self.hitPoints -= damage;
        [self removeAllActions];
        [self runAction:_knockedOutAction];
        self.jumpVelocity = kJumpForce / 2.0;
        self.velocity = CGPointMake(direction.x * self.runSpeed,
                                    direction.y * self.runSpeed);
        [self flipSpriteForVelocity:CGPointMake(-self.velocity.x,
                                                self.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 is not only forced back, it also rises 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 a knockout. A sprite can either die or shake it off and recover.



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 **GameLayer** that he has died so that **GameLayer** can execute the necessary game logic.

As with ActionDPad andActionButton, you will use the delegation pattern.

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 (weak, nonatomic) id <ActionSpriteDelegate> delegate;
```

This should be familiar by now. When the **ActionSprite** attacks, the **delegate** should execute **actionSpriteDidAttack:**, passing in a reference to the **ActionSprite** that performed the action. Similarly, when an **ActionSprite** dies, then the **delegate** should execute **actionSpriteDidDie:**.

Now all ActionSprites can die peacefully, knowing that the game will attend to their death. ☺

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

```
- (void)die
{
    self.actionState = kActionStateDead;
    self.velocity = CGPointMakeZero;
    self.jumpHeight = 0;
    self.jumpVelocity = 0;
    self.hitPoints = 0.0;
    [self.delegate actionSpriteDidDie:self];
    [self runAction:self.dieAction];
}

- (void)recover
{
    if (self.actionState == kActionStateKnockedOut) {

        self.actionState = kActionStateNone;
        self.velocity = CGPointMakeZero;
        self.jumpVelocity = 0;
        self.jumpHeight = 0;

        [self performSelector:@selector(getUp)
                      withObject:nil
                      afterDelay:0.5];
    }
}
```

```
- (void)setUp
{
    self.actionState = kActionStateRecover;
    [self runAction:self.recoverAction];
}
```

When the **ActionSprite** dies, it simply changes its 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, while in **recover**, it has absolutely no action state and zero movement values. After half a second, **recover** executes **setUp**, which changes the **ActionSprite**'s state to **kActionStateRecover** and executes the recover animation action.

Still in **ActionSprite.m**, add this clause to the **if-else** block in **update**:

```
else if (_actionState == kActionStateKnockedOut) {

    self.desiredPosition =
        CGPointMakeAdd(self.groundPosition,
                      CGPointMakeMultiplyScalar(self.velocity, delta));

    self.jumpVelocity -= kGravity * delta;
    self.jumpHeight += self.jumpVelocity * delta;

    if (self.jumpHeight <= 0) {
        if (self.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 in mid-air.

In the above code, when an **ActionSprite** is knocked out, the **update:** method updates the **desiredPosition** to move the sprite across the screen and waits 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:

```
//add to top of file
```

```
#import "SKAction+SKTExtras.h"

//inside init (below the existing animation blocks)
//hurt animation
self.hurtAction = [SKAction sequence:@[[SKAction
animateWithTextures:[self texturesWithPrefix:@"hero_hurt"
startFrameIdx:0 frameCount:3] timePerFrame:1.0/12.0], [SKAction
performSelector:@selector(idle) onTarget:self]]];

//knocked out animation
self.knockedOutAction = [SKAction animateWithTextures:[self
texturesWithPrefix:@"hero_knockout" startFrameIdx:0 frameCount:5]
timePerFrame:1.0/12.0];

//die action
self.dieAction = [SKAction blinkWithDuration:2.0 blinks:10.0];

//recover animation
self.recoverAction = [SKAction sequence:@[[[SKAction
animateWithTextures:[self texturesWithPrefix:@"hero_getup"
startFrameIdx:0 frameCount:6] timePerFrame:1.0/12.0], [SKAction
performSelector:@selector(jumpLand) onTarget:self]]];
```

Switch to **Robot.m** and add the following:

```
//add to top of file
#import "SKAction+SKTExtras.h"

//add to init (again, below the existing animation blocks)
//hurt animation
SKAction *hurtAnimationGroup = [self animateActionForActionWord:@"hurt"
timePerFrame:1.0/12.0 frameCount:3];
self.hurtAction = [SKAction sequence:@[hurtAnimationGroup, [SKAction
performSelector:@selector(idle) onTarget:self]]];

//knocked out animation
self.knockedOutAction = [self animateActionForActionWord:@"knockout"
timePerFrame:1.0/12.0 frameCount:5];

//die action
self.dieAction = [SKAction blinkWithDuration:2.0 blinks:10];

//recover animation
SKAction *recoverAnimationGroup = [self
animateActionForActionWord:@"getup" timePerFrame:1.0/12.0 frameCount:6];
self.recoverAction = [SKAction sequence:@[recoverAnimationGroup,
[SKAction performSelector:@selector(idle) onTarget:self]]];
```

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 simply makes the sprite blink ten times in two seconds.

Detecting Collisions

Are you ready for phase two? Open **GameLayer.h** and make the following changes:

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

Here you simply mark the layer as implementing ActionSpriteDelegate so it is notified when an action sprite attacks another sprite or dies.

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

```
//add this in initHero, after self.hero = [Hero node];  
self.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  
    CGFloat planeDist =  
        attacker.shadow.position.y - target.shadow.position.y;  
  
    if (fabsf(planeDist) <= kPlaneHeight) {  
  
        NSInteger i, j;  
        CGFloat combinedRadius =  
            attacker.detectionRadius + target.detectionRadius;  
  
        //initial detection  
        if (CGPointDistanceSQ(attacker.position, target.position)  
            <= combinedRadius * combinedRadius) {  
  
            NSInteger attackPointCount = attacker.attackPoints.count;  
            NSInteger contactPointCount = target.contactPoints.count;  
  
            ContactPoint attackPoint, contactPoint;
```

```
//secondary detection
for (i = 0; i < attackPointCount; i++) {

    NSValue *value = attacker.attackPoints[i];
    [value getValue:&attackPoint];

    for (j = 0; j < contactPointCount; j++) {

        NSValue *value = target.contactPoints[j];
        [value getValue:&contactPoint];

        combinedRadius =
            attackPoint.radius + contactPoint.radius;

        if (CGPointDistanceSQ(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:

```
CGFloat planeDist = attacker.shadow.position.y -
    target.shadow.position.y;

if (fabsf(planeDist) <= kPlaneHeight)
```

If they are too far apart along the y-axis, then there is no need to check if their collision shapes hit one another. Otherwise, you would have to consider the first two events to be collisions, which would make for some weird gameplay.

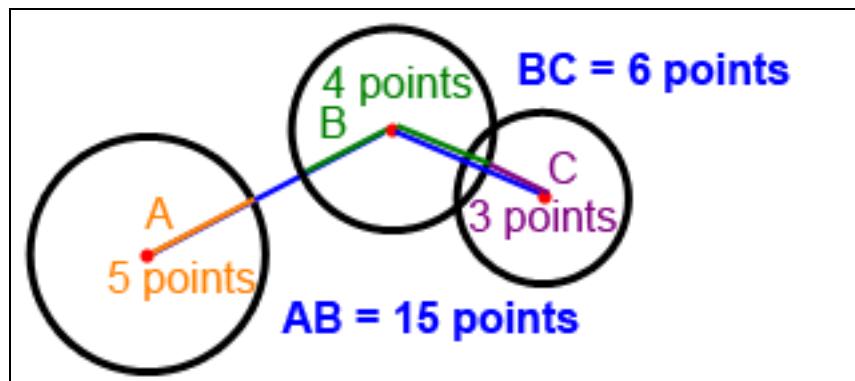
If the method passes the y-coordinate check, then it continues to the next phase: the initial detection, which consists of this section:

```
CGFloat combinedRadius =
    attacker.detectionRadius + target.detectionRadius;

//initial detection
if (CGPointDistanceSQ(attacker.position, target.position)
    <= combinedRadius * combinedRadius) {
```

The above code determines if the detection circles of the two sprites intersect by checking the distance between their center positions.

If the 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 `CGPointDistanceSQ`:

```
if (CGPointDistance(attacker.position, target.position) <=
    combinedRadius)
```

This is good, and would also work. However, this requires a square root operation, which is expensive in terms of computing power. Doing it too often can result in a

performance hit. Instead, you simply square both sides and compare the results, which is more efficient.

`CGPointDistanceSQ` gets the distance between two points right before performing the square root operation, 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:

```
NSInteger attackPointCount = attacker.attackPoints.count;
NSInteger contactPointCount = target.contactPoints.count;

ContactPoint attackPoint, contactPoint;

//secondary detection
for (i = 0; i < attackPointCount; i++) {

    NSValue *value = attacker.attackPoints[i];
    [value getValue:&attackPoint];

    for (j = 0; j < contactPointCount; j++) {

        NSValue *value = target.contactPoints[j];
        [value getValue:&contactPoint];

        combinedRadius =
            attackPoint.radius + contactPoint.radius;

        if (CGPointDistanceSQ(attackPoint.position,
        contactPoint.position) <= combinedRadius * combinedRadius) {

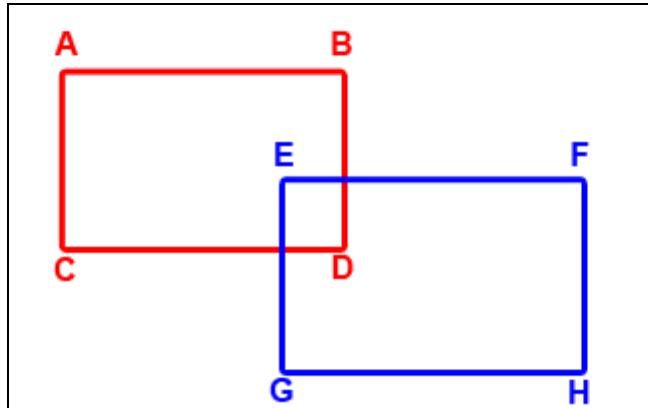
            //attack point collided with contact point
            position->x = attackPoint.position.x;
            position->y = attackPoint.position.y;
            return YES;
        }
    }
}
```

In this phase, you loop through all of the attacker's attack circles and check if they intersect with any of the target's contact circles using the same distance method as before. An intersection between any attack and contact circle triggers this piece of code:

```
//attack point collided with contact point
position->x = attackPoint.position.x;
position->y = attackPoint.position.y;
return YES;
```

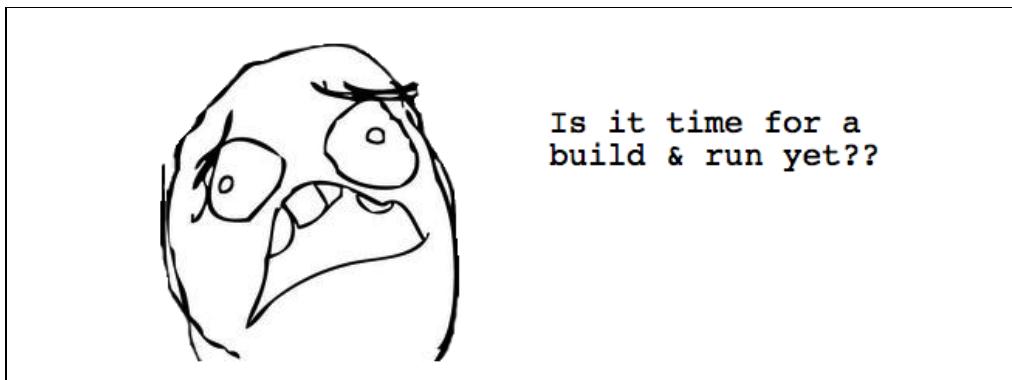
You store the position of the attack in the `ContactPoint` that was passed by reference into `collisionBetweenAttacker:` and return `YES` to signify that a collision happened.

Note: If you've been wondering why you're using 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 intersect, you would have to check a lot of points. You would check if points 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 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 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 any of the other four frames that precede and follow the punch.

You need to detect the exact moment when the sprite's displayed frame changes to the one showing the attack. A good place to do this is when the sprite's texture changes in `setTexture:`.

The general flow of triggering collisions is:

1. ActionSprite animates using an `SKAction`.
2. `SKAction` triggers `setTexture:` during animation.
3. `setTexture:` informs `GameLayer` to check collisions for specific frames/textures.
4. `GameLayer` resolves the collision.

Because the default `SKAction` `animateWithTextures:` method doesn't use `setTexture:` to change an `SKSpriteNode`'s texture, it is quite difficult to follow this plan. For steps 1 and 2 to work, you need to use the custom action you created to animate the robot in the previous chapter—which you get by calling the method `animateActionForGroup:timePerFrame:frameCount:`.

If you remember, this method creates a custom `SKAction` that uses `AnimationMembers` to animate sprites, and these `AnimationMembers` trigger `SKSpriteNode`'s `setTexture:` method when the action is run.

Since you already have a helper method that creates such custom actions, it's the perfect place to start.

Open **ActionSprite.m** and add the following method:

```
- (SKAction *)animateActionForTextures:(NSMutableArray *)textures
timePerFrame:(NSTimeInterval)timeInterval
{
    AnimationMember *animationMember =
        [AnimationMember animationWithTextures:textures
                           target:self];
    NSMutableArray *actionGroup =
        [NSMutableArray arrayWithObject:animationMember];
    SKAction *action =
        [self animateActionForGroup:actionGroup
                           timePerFrame:timeInterval
                           frameCount:textures.count];
    return action;
}
```

`animateActionForTextures:` works the same way as the `SKAction` `animateWithTextures:` method. It creates an action that animates the `ActionSprite` that created it by using the `animateActionForGroup:timePerFrame:frameCount` method you wrote earlier.

Next, go to **Hero.m** and do the following:

```
//replace the following line in init

SKAction *attackAnimation =
    [SKAction animateWithTextures:textures
        timePerFrame:1.0/15.0];

//with this line

SKAction *attackAnimation =
    [self animateActionForTextures:textures
        timePerFrame:1.0/15.0];

//add this method

- (void)setTexture:(SKTexture *)texture
{
    [super setTexture:texture];

    SKTexture *attackTexture =
        [[SKTextureCache sharedInstance]
            textureNamed:@"hero_attack_00_01"];

    if (texture == attackTexture) {
        [self.delegate actionSpriteDidAttack:self];
    }
}
```

Similarly, switch to **Robot.m** and add the following method:

```
- (void)setTexture:(SKTexture *)texture
{
    [super setTexture:texture];

    SKTexture *attackTexture =
        [[SKTextureCache sharedInstance]
            textureNamed:@"robot_base_attack_03"];

    if (texture == attackTexture) {
        [self.delegate actionSpriteDidAttack:self];
    }
}
```

You're changing the hero's attackAnimation to use the custom SKAction that employs `setTexture:` to animate sprites. The Robot class already has this custom action so there's no need to replace it.

When `setTexture:` is triggered, you first call `setTexture:` on the super class so that the sprite still changes the displayed texture. You then check to see if the new texture is equal to the exact texture 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 == self.hero) {

        CGPoint attackPosition;
        Robot *robot;
        for (robot in self.robots) {

            // 2
            if (robot.actionState < kActionStateKnockedOut &&
                robot.actionState != kActionStateNone) {

                if ([self collisionBetweenAttacker:self.hero
                                              andTarget:robot
                                         atPosition:&attackPosition]) {

                    [robot hurtWithDamage:self.hero.attackDamage
                               force:self.hero.attackForce
                               direction:CGPointMake(self.hero.directionX, 0.0)];

                    didHit = YES;
                }
            }
        }

        return didHit;
    } else {

        // 3
        if (self.hero.actionState < kActionStateKnockedOut &&
            self.hero.actionState != kActionStateNone) {

            CGPoint attackPosition;
            if ([self collisionBetweenAttacker:actionSprite
                                          andTarget:self.hero
                                         atPosition:&attackPosition]) {

                [self.hero hurtWithDamage:actionSprite.attackDamage
                               force:actionSprite.attackForce
                               direction:CGPointMake(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 (one 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, your collision handling is now complete!

Build and run, and put some hurt on those robots!



Knockback

Wait-a-minute... when the hero hurts or knocks out a robot, shouldn't his attack force affect the robot's position? Yes, yes it 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]

for (Robot *robot in self.robots) {
    [robot update:delta];
}
```

```
//add inside updatePositions, after setting _hero.position

for (Robot *robot in self.robots) {

    if (robot.actionState > kActionStateNone) {

        posY = MIN(floorHeight + (robot.centerToBottom -
robot.feetCollisionRect.size.height), MAX(robot.centerToBottom,
robot.desiredPosition.y));

        robot.groundPosition =
            CGPointMake(robot.desiredPosition.x, posY);

        robot.position =
            CGPointMake(robot.groundPosition.x,
                        robot.groundPosition.y + robot.jumpHeight);

        if (robot.actionState == kActionStateDead &&
            self.hero.groundPosition.x - robot.groundPosition.x
            >= SCREEN.width + robot.size.width/2) {

            robot.hidden = YES;
            [robot reset];
        }
    }
}
```

You update all the robots in the same way you updated the hero earlier, but with slight modifications.

First, you don't restrict the x-position of the robots. You allow the robots 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 you make the robot invisible and reset it.

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 = self.maxHitPoints;
}
```

This sends the sprite to the off-screen coordinates 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!



After all that coding, this may be the most rewarding fight of your life, especially considering that no one's fighting back!

Stay tuned for the next chapter, where it will be the poor robots' turn to deal the damage.

Challenge: If you've completed 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 a two-fisted strike, 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 to take a pummeling.

In this chapter, you're going to change that. You'll add some brains to your bots, converting them into seek-and-destroy death machines that have it 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 by giving the Pompadoured Hero the dramatic entrance he deserves.

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 to help them decide what to do in every situation. The AI 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 will be only four things a robot can opt to do once it sees the hero:

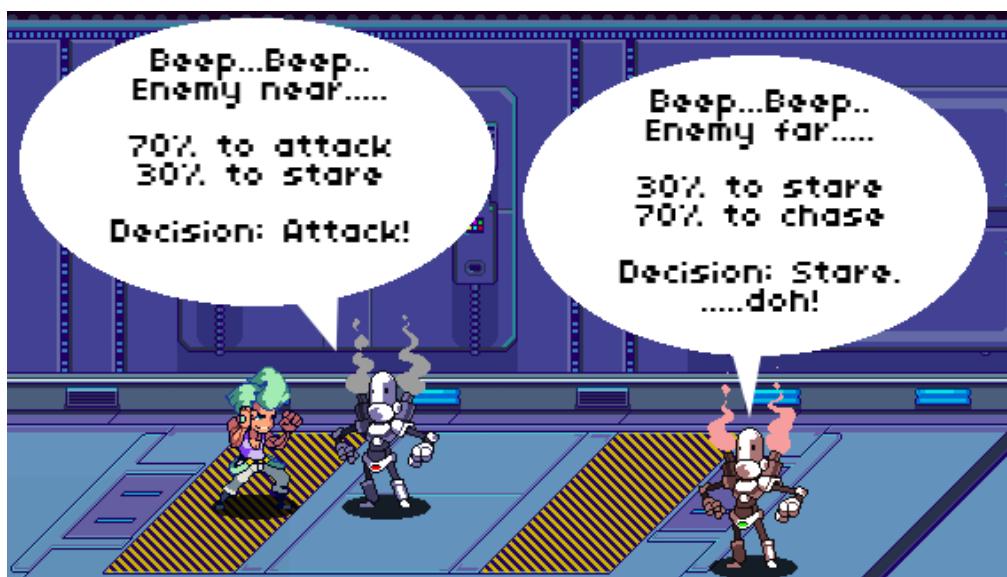
1. Stand idle.
2. Move closer to the hero to surround him.

3. Chase or move towards the hero.
4. Attack the hero head-on.

For every situation, the decision will depend 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 options should have a different selection probability. Further, these probabilities should 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 chooses to remain idle.

In every situation, the robot should weigh its options and make a choice based on the weight of each option. The higher the probability percentage for an option, the more likely the robot should be to choose it above the alternative.

It's time to give your robots some brains!

Go to **Defines.h** and add these definitions:

```
typedef NS_ENUM(NSInteger, AIDecision)
{
    kDecisionAttack = 0,
    kDecisionStayPut,
    kDecisionChase,
    kDecisionMove
};
```

Next, select the **PompaDroid** group in Xcode, go to **File\New\Group** and name the new group **AI**.

Then select the **AI** group, go to **File\New\File**, choose the **iOS\Cocoa Touch\Objective-C 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

@property (assign, nonatomic) AIDecision decision;
@property (assign, nonatomic) NSInteger weight;

+ (instancetype)decisionWithDecision:(AIDecision)decision
                                andWeight:(CGFloat)weight;

- (instancetype)initWithDecision:(AIDecision)decision
                                andWeight:(CGFloat)weight;

@end
```

I'll explain this in a moment, but first switch to **WeightedDecision.m** and add the implementation:

```
+ (instancetype)decisionWithDecision:(AIDecision)decision
                                andWeight:(CGFloat)weight
{
    return [[self alloc] initWithDecision:decision
                                andWeight:weight];
}

- (instancetype)initWithDecision:(AIDecision)decision
                                andWeight:(CGFloat)weight
{
    if (self = [super init]) {
        _decision = decision;
        _weight = weight;
    }

    return self;
}
```

AIDecision is an enumeration that 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) that the robot will make that decision. There's nothing in this class except initializer methods. It merely stores the decision type and the weight.

Select the **AI** group, go to **File\New\File**, choose the **iOS\Cocoa Touch\Objective-C 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 "ActionSprite.h"
#import "WeightedDecision.h"

@interface ArtificialIntelligence : NSObject

@property (assign, nonatomic) CGFloat decisionDuration;
@property (weak, nonatomic) ActionSprite *controlledSprite;
@property (weak, nonatomic) ActionSprite *targetSprite;
@property (assign, nonatomic) AIDecision decision;
@property (strong, nonatomic) NSMutableArray *availableDecisions;

+ (instancetype)aiWithControlledSprite:(ActionSprite *)controlledSprite
targetSprite:(ActionSprite *)targetSprite;

- (instancetype)initWithControlledSprite:(ActionSprite *)
controlledSprite targetSprite:(ActionSprite *)targetSprite;

- (void)update:(NSTimeInterval)delta;

@end
```

First, you import the necessary classes that the AI will use. In addition, you declare the following properties:

- **decisionDuration**: How long a decision lasts before a new decision is made.
- **controlledSprite**: The ActionSprite that the AI is controlling.
- **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 replace its contents with the following:

```
#import "ArtificialIntelligence.h"

@interface ArtificialIntelligence()

@property (strong, nonatomic) WeightedDecision *attackDecision;
@property (strong, nonatomic) WeightedDecision *idleDecision;
@property (strong, nonatomic) WeightedDecision *chaseDecision;
@property (strong, nonatomic) WeightedDecision *moveDecision;

@end
```

```
@implementation ArtificialIntelligence

+ (instancetype)aiWithControlledSprite:(ActionSprite *)controlledSprite
targetSprite:(ActionSprite *)targetSprite
{
    return [[self alloc] initWithControlledSprite:controlledSprite
targetSprite:targetSprite];
}

- (instancetype)initWithControlledSprite:(ActionSprite
*)controlledSprite targetSprite:(ActionSprite *)targetSprite
{
    if (self = [super init]) {

        _controlledSprite = controlledSprite;
        _targetSprite = targetSprite;
        _availableDecisions = [NSMutableArray 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;
}

@end
```

An AI object 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;

    self.attackDecision.weight = attackWeight;
    self.idleDecision.weight = idleWeight;
    self.chaseDecision.weight = chaseWeight;
    self.moveDecision.weight = moveWeight;

    int choice = RandomIntRange(1, totalWeight);
    NSInteger minInclusive = 1;
    NSInteger maxExclusive = minInclusive;
    NSInteger decisionWeight;

    WeightedDecision *weightedDecision;

    for (weightedDecision in self.availableDecisions) {
        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. The method assigns each decision 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) {
        [self.controlledSprite attack];
        self.decisionDuration = RandomFloatRange(0.25, 1.0);
    }
}
```

```
    } else if (_decision == kDecisionStayPut) {  
  
        [self.controlledSprite idle];  
        self.decisionDuration = RandomFloatRange(0.25, 1.5);  
    }  
    else if (_decision == kDecisionChase) {  
  
        NSValue *value = self.controlledSprite.attackPoints[0];  
  
        ContactPoint contactPoint;  
        [value getValue:&contactPoint];  
  
        CGFloat reachDistance = self.targetSprite.centerToSides +  
        contactPoint.offset.x + contactPoint.radius;  
        CGPoint reachPosition =  
        CGPointMake(self.targetSprite.groundPosition.x + (RandomSign() *  
        reachDistance), self.targetSprite.groundPosition.y);  
        CGPoint moveDirection =  
        CGPointNormalize(CGPointSubtract(reachPosition,  
        self.controlledSprite.groundPosition));  
        [self.controlledSprite walkWithDirection:moveDirection];  
        self.decisionDuration = RandomFloatRange(0.5, 1.0);  
  
    } else if (_decision == kDecisionMove) {  
  
        CGFloat randomX = RandomSign() * RandomFloatRange(20.0 *  
        kPointFactor, 100.0 * kPointFactor);  
        CGFloat randomY = RandomSign() * RandomFloatRange(10.0 *  
        kPointFactor, 40.0 * kPointFactor);  
        CGPoint randomPoint = CGPointMake(self.targetSprite.groundPosition.x  
        + randomX, self.targetSprite.groundPosition.y + randomY);  
        CGPoint moveDirection =  
        CGPointNormalize(CGPointSubtract(randomPoint,  
        self.controlledSprite.groundPosition));  
        [self.controlledSprite walkWithDirection:moveDirection];  
        self.decisionDuration = RandomFloatRange(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 AI then sets **decisionDuration** so that the sprite will make the next decision at a random interval between 0.25 and 1 second.

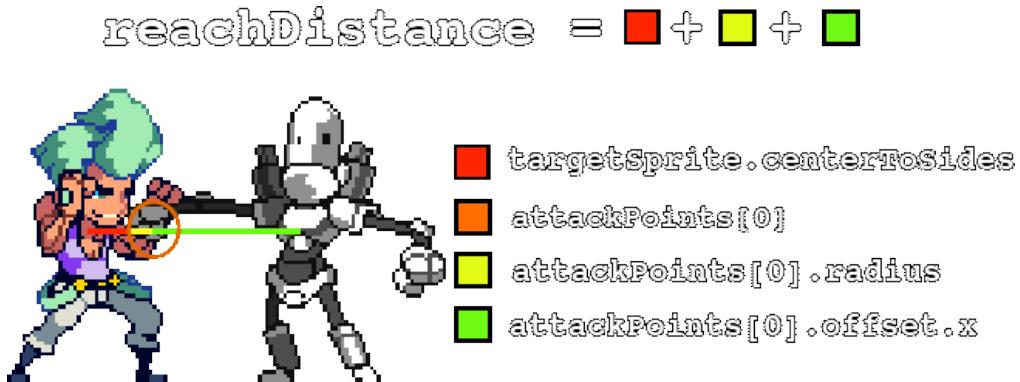
- **If the decision is to stay put:**

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 computes 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 this, 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 (you'll choose randomly) of the `targetSprite` where the `controlledSprite` can reach it.

Then, the AI 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 you use in moving `ActionSprite`.

The AI then tells `ActionSprite` to walk toward 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 picks a random position near the `targetSprite`, gets the direction from `controlledSprite` to this position and asks 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:(NSTimeInterval)delta
{
    if (self.targetSprite && self.controlledSprite &&
        self.controlledSprite.actionState > kActionStateNone) {
```

```
//1
CGFloat distanceSQ =
CGPointDistanceSQ(self.controlledSprite.groundPosition,
self.targetSprite.groundPosition);

CGFloat planeDist = fabsf(self.controlledSprite.shadow.position.y -
self.targetSprite.shadow.position.y);

CGFloat combinedRadius = self.controlledSprite.detectionRadius +
self.targetSprite.detectionRadius;

BOOL samePlane = NO;
BOOL canReach = NO;
BOOL tooFar = YES;
BOOL canMove = NO;

//2
if (self.controlledSprite.actionState == kActionStateWalk ||
    self.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

            NSInteger attackPointCount =
                self.controlledSprite.attackPoints.count;

            NSInteger contactPointCount =
                self.targetSprite.contactPoints.count;

            NSInteger i, j;
            ContactPoint attackPoint, contactPoint;
            for (i = 0; i < attackPointCount; i++) {

                NSValue *value =
                    self.controlledSprite.attackPoints[i];

                [value getValue:&attackPoint];

                for (j = 0; j < contactPointCount; j++) {
```

```
        NSValue *value =
            self.targetSprite.contactPoints[j];

        [value getValue:&contactPoint];

        combinedRadius =
            attackPoint.radius + contactPoint.radius;

        if (CGPointDistanceSQ(attackPoint.position,
contactPoint.position) <= combinedRadius * combinedRadius) {
            canReach = YES;
            break;
        }
    }

    if (canReach) {
        break;
    }
}
}

//4
if (canReach && _decision == kDecisionChase) {
    self.decision = kDecisionStayPut;
}

//5
if (self.decisionDuration > 0) {
    self.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 it's doing section by section:

1. **update:** will run constantly as long as the **controlledSprite** is alive and 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 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, 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 sprite can reach the target.
5. The AI counts **decisionDuration** down to zero. Once finished, it makes 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.

Your AI is complete. Now to inject some brains into the robots!

Go to **GameLayer.h** and add this property:

```
@property (strong, nonatomic) NSMutableArray *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 =
        [NSMutableArray arrayWithCapacity:self.robots.count];

    ArtificialIntelligence *brain;

    for (Robot *robot in self.robots) {

        brain =
            [ArtificialIntelligence aiWithControlledSprite:robot
                                                     targetSprite:self.hero];

        [self.brains addObject:brain];
    }
}

//add inside update:, before Robot *robot
for (ArtificialIntelligence *brain in self.brains) {
    [brain update:delta];
}
```

You create a brain for each 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 game doesn't set the robot's attack circle 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 = [self contactPointArray:1];

[self modifyAttackPointAtIndex:0
                         offset:CGPointMake(45.0, 6.5)
                         radius:10.0];
```

The above code positions the attack circle of the robot during initialization.

Build and run, and 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:

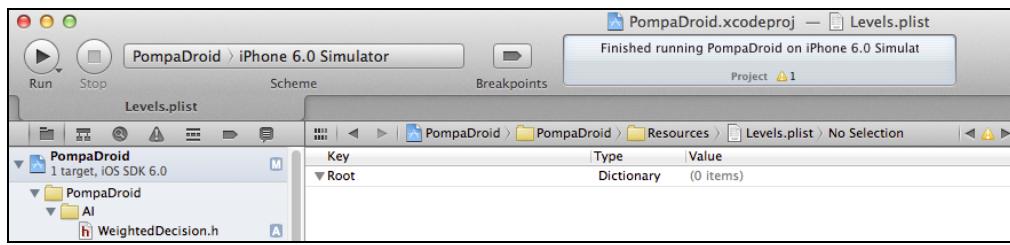
1. **Free Roam Events:** These allow the player to move around the map freely.
2. **Battle Events:** During these, for a fixed area on the map, a predetermined number of robots will come and fight the player.
3. **Scripted Events:** These are events where the player cannot control the hero.

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 is just an XML file 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.

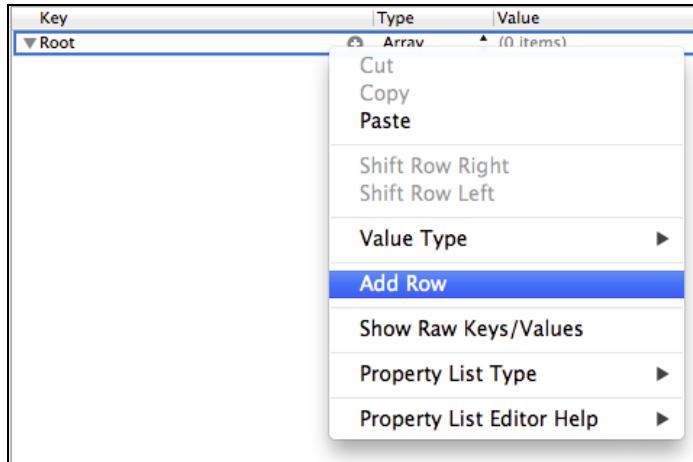
In Xcode, select the **Resources** folder, go to **File\New\File**, choose **iOS\Resource\Property List** and click **Next**. Name the file **Levels** and click **Create**.

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:** You'll use the key to retrieve this row's value. Note that string-based keys are only used for items in dictionaries. Since your item is in an array, it's 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.

In **Levels.plist**, the rows under Root will be the levels in the game, where Item 0 is the first level. Since each 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 as **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

The **TileMap** entry will hold the filename of the **TMXTiledMap** you will load for this level (**map_level1.tmx** in this case).

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 that 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 and 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 can't assign it a new Key. The Key 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 will be 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 by the hero stepping onto the 15th column of tiles from the left side of the map.

Next, create another new row under Item 0 of BattleEvents, 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 the attributes of each enemy. 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 corresponds 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 characters 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 it will spawn at the left edge of the screen, whereas a value of 1 means it will spawn at the right edge.

Both the Row and Offset values make it easy to position each enemy for a battle event 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 in the resources under **Versions\Chapter5**, 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 NS_ENUM(NSInteger, EventState)
{
    kEventStateScripted = 0,
    kEventStateFreeWalk,
    kEventStateBattle,
    kEventStateEnd
};

typedef NS_ENUM(NSInteger, EnemyType)
{
    kEnemyRobot = 0,
    kEnemyBoss
};
```

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 properties
@property (strong, nonatomic) NSMutableArray *battleEvents;
@property (strong, nonatomic) NSDictionary *currentEvent;
@property (assign, nonatomic) EventState eventState;
```

Then head over to **GameLayer.m** and add the following properties to the private interface on top:

```
@property (assign, nonatomic) NSInteger activeEnemies;
@property (assign, nonatomic) CGFloat viewPointOffset;
@property (assign, nonatomic) CGFloat eventCenter;
```

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.

Still in **GameLayer.m**, add 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:(NSInteger)level
{
    NSString *levelsPlist = [[NSBundle mainBundle]
pathForResource:@"Levels" ofType:@"plist"];

    NSMutableArray *levelArray = [[NSMutableArray alloc]
initWithContentsOfFile:levelsPlist];

    NSDictionary *levelData = [[NSDictionary alloc]
initWithDictionary:[levelArray objectAtIndex:level]];

    NSString *tileMap = [levelData objectForKey:@"TileMap"];

    [self initTileMap:tileMap];

    //store the events
    self.battleEvents = [NSMutableArray arrayWithArray:[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 in `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 up in the property list. If, for example, you loaded the first level (level 0), it would look something like this:

Key	Type	Value
Root	Array	(1 item)
levelArray	Dictionary	(2 items)
Item 0	String	map_level1.tmx
levelData	Array	(1 item)
TileMap	Dictionary	(2 items)
_battleEvents	String	map_level1.tmx
battleEvents	Array	(1 item)

Build and run! Check that the tile map shows up correctly.



If the tile map is still there, it means the game loaded **Levels.plist** successfully, which means **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 {
    NSInteger robotCount = 50;
    self.robots = [NSMutableArray arrayWithCapacity:robotCount];

    for (NSInteger i = 0; i < robotCount; i++) {

        Robot *robot = [Robot node];
        robot.delegate = self;
        [self addChild:robot.shadow];
        [self addChild:robot.smoke];
        [self addChild:robot];
        [self addChild:robot.belt];
        [self.robots addObject:robot];

        [robot setScale:kPointFactor]; //scaling simplified
        [robot.shadow setScale:kPointFactor];
        robot.position = OFFSCREEN; //this changed
        robot.groundPosition = robot.position;
        robot.desiredPosition = robot.position;
        robot.hidden = YES;
        //this line was removed: [robot idle];
        robot.colorSet = kColorRandom;
    }
}
```

The new method has a few changes:

- **Scaling simplified:** It doesn't really matter if `xScale` is positive or negative now since it will be set when the robots move, so you just have to make sure the robot is scaled properly as per the device.
- **OFFSCREEN position:** Now when you create a robot, you send it 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 Sprite Kit 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 at all times. Instead, you want to spawn robots as you need them, based on the current battle event.

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

```
- (void)spawnEnemies:(NSMutableArray *)enemies
                 fromOrigin:(CGFloat)origin
{
    for (NSDictionary *enemyData in enemies) {
        NSInteger row = [enemyData[@"Row"] integerValue];
        NSInteger type = [enemyData[@"Type"] integerValue];
        CGFloat offset = [enemyData[@"Offset"] floatValue];

        if (type == kEnemyRobot) {
            NSInteger color = [enemyData[@"Color"] integerValue];
            //get an unused robot
            for (Robot *robot in self.robots) {

                if (robot.actionState == kActionStateNone) {
                    [robot removeAllActions];
                    robot.hidden = YES;
                    robot.groundPosition = CGPointMake(origin + (offset *
(CENTER.x + robot.centerToSides)), robot.centerToBottom +
self.tileMap.tileSize.height * row * kPointFactor);
                    robot.position = robot.groundPosition;
                    robot.desiredPosition = robot.groundPosition;
                    [robot setColorSet:color];
                    [robot idle];
                    robot.hidden = NO;
                    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 in the tile map.

`spawnEnemies:fromOrigin:` cycles through all the enemy data contained in `enemies` and retrieves the Type, Color, Row and Offset for each enemy. Next, if it determines that the Type of the enemy is `kEnemyRobot` (or 0 in numerical form), then it traverses through the robots array and retrieves an unused robot.

Once it gets an inactive robot, it changes the properties of that robot based on the attributes taken from the enemy data.

The only special calculation you're doing here is for the position. You're calculating it as shown in this diagram:



Let's discuss it per coordinate:

- **The x-coordinate:**

You always measure the x-position from the origin, which is the center of the current screen. You then decide the robot's x-position 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 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 to the left.

- **The y-coordinate:**

The y-position is simpler. You measure it using the height of each tile and the robot's `centerToBottom` attribute. Since the robot's position is always based on the

center, to put 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 (self.eventState == kEventStateBattle &&  
        self.activeEnemies <= 0) {  
  
        CGFloat maxCenterX = self.tileMap.mapSize.width *  
            self.tileMap.tileSize.width * kPointFactor - CENTER.x;  
  
        CGFloat cameraX = MAX(MIN(self.hero.position.x, maxCenterX),  
            CENTER.x);  
  
        self.viewPointOffset = cameraX - self.eventCenter;  
        self.eventState = kEventStateFreeWalk;  
    } else if (self.eventState == kEventStateFreeWalk) {  
  
        [self cycleEvents];  
    }  
}  
  
- (void)cycleEvents  
{  
  
    NSDictionary *event;  
    NSInteger column;  
  
    CGFloat tileSizeWidth =  
        self.tileMap.tileSize.width * kPointFactor;  
  
    for (event in self.battleEvents) {  
  
        column = [event[@"Column"] integerValue];  
  
        CGFloat maxCenterX = self.tileMap.mapSize.width *  
            self.tileMap.tileSize.width * kPointFactor - CENTER.x;
```

```

CGFloat columnPosition = column * tileSize - tileSize/2;

self.eventCenter =
    MAX(MIN(columnPosition, maxCenterX), CENTER.x);

//1
if (self.hero.position.x >= self.eventCenter) {

    self.currentEvent = event;
    self.eventState = kEventStateBattle;
    NSMutableArray *enemyData =
        [NSMutableArray arrayWithArray:event[@"Enemies"]];
    self.activeEnemies = enemyData.count;
    [self spawnEnemies:enemyData fromOrigin:self.eventCenter];
    [self setViewpointCenter:CGPointMake(self.eventCenter,
    self.hero.position.y)];
    break;
}
}

if (self.eventState == kEventStateBattle) {
    [self.battleEvents removeObject:self.currentEvent];
}
}
}

```

You call `updateEvent` from `update:` so 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.

Before you test the game again, add this method header to `GameLayer.h`:

```
- (void)startGame;
```

Switch to `GameLayer.m` and add the method implementation:

```

- (void)startGame
{
    self.eventState = kEventStateFreeWalk;
}

```

```
}
```

Lastly, switch to **GameScene.m** and add this method:

```
- (void)didMoveToView:(SKView *)view
{
    [self.gameLayer startGame];
}
```

`didMoveToView` is called when `ViewController` presents `GameScene`, so this simply changes the `eventState` to `kEventStateFreeWalk` once that happens.

Build and run the game, and walk towards the 15th column (if you are using the scene configuration mentioned earlier in this chapter) to experience your first battle event!



Fighting is mandatory!

Of course, if you play for a bit, you'll realize that you can run the hero toward 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 (_self.eventState == kEventStateFreeWalk ||
    _self.eventState == kEventStateScripted) {

    [self setViewpointCenter:_self.hero.position];
```

```
}

//replace updatePositions with the following

- (void)updatePositions
{
    CGFloat mapWidth = self.tileMap.mapSize.width *
self.tileMap.tileSize.width * kPointFactor;

    CGFloat floorHeight = 3 * self.tileMap.tileSize.height * kPointFactor;

    CGFloat posX, posY;

    if (self.hero.actionState > kActionStateNone)
    {
        // 1
        if (self.eventState == kEventStateFreeWalk)
        {
            posX = MIN(mapWidth - self.hero.feetCollisionRect.size.width/2,
MAX(self.hero.feetCollisionRect.size.width/2,
self.hero.desiredPosition.x));

            posY = MIN(floorHeight + (self.hero.centerToBottom -
self.hero.feetCollisionRect.size.height), MAX(self.hero.centerToBottom,
self.hero.desiredPosition.y));

            self.hero.groundPosition = CGPointMake(posX, posY);

            self.hero.position = CGPointMake(self.hero.groundPosition.x,
self.hero.groundPosition.y + self.hero.jumpHeight);
        }
        // 2
        else if (self.eventState == kEventStateBattle)
        {
            posX = MIN(MIN(mapWidth -
self.hero.feetCollisionRect.size.width/2, self.eventCenter + CENTER.x -
self.hero.feetCollisionRect.size.width/2), MAX(self.eventCenter -
CENTER.x + self.hero.feetCollisionRect.size.width/2,
self.hero.desiredPosition.x));
            posY = MIN(floorHeight + (self.hero.centerToBottom -
self.hero.feetCollisionRect.size.height), MAX(self.hero.centerToBottom,
self.hero.desiredPosition.y));

            self.hero.groundPosition = CGPointMake(posX, posY);

            self.hero.position = CGPointMake(self.hero.groundPosition.x,
self.hero.groundPosition.y + self.hero.jumpHeight);
        }
    }

    Robot *robot;
    for (robot in self.robots)
```

```
{  
    if (robot.actionState > kActionStateNone)  
    {  
        posY = MIN(floorHeight + (robot.centerToBottom -  
        robot.feetCollisionRect.size.height), MAX(robot.centerToBottom,  
        robot.desiredPosition.y));  
  
        robot.groundPosition = CGPointMake(robot.desiredPosition.x, posY);  
  
        robot.position = CGPointMake(robot.groundPosition.x,  
        robot.groundPosition.y + robot.jumpHeight);  
  
        if (robot.actionState == kActionStateDead &&  
        self.hero.groundPosition.x - robot.groundPosition.x >= SCREEN.width +  
        robot.size.width/2)  
        {  
            robot.hidden = YES;  
            [robot reset];  
        }  
    }  
}
```

In `update:`, you restrict the call to `setViewpointCenter:` so that it happens only during 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 game updates the position of the hero.

1. When the `eventState` is `kEventStateFreeWalk`, the game only clamps the hero's position from the edges of the map, like before.
2. When the `eventState` is `kEventStateBattle`, then the game does an additional clamping 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 all 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 afterward. Take a look back at `updateEvent` and you'll see that the condition for leaving an event is when `activeEnemies` becomes zero.

```
- (void)updateEvent
{
    if (self.eventState == kEventStateBattle &&
        self.activeEnemies <= 0) {
```

You set `activeEnemies` at the start of the battle event in `cycleEvents`, but you don't update it after that. Every time the hero defeats an enemy, this number should decrease.

Thanks to your delegating `ActionSprite` 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 == self.hero) {
        //future code here
    } else {
        self.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 any 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.

Simply put, this looks ugly. Instead of snapping back, the camera should pan smoothly 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 = CGPointMake(viewPoint.x + self.viewPointOffset,  
viewPoint.y);  
  
//add this to the end of update:  
  
if (self.viewPointOffset < 0) {  
  
    self.viewPointOffset += SCREEN.width * delta;  
  
    if (self.viewPointOffset >= 0) {  
        self.viewPointOffset = 0;  
    }  
}
```

```

} else if (self.viewPointOffset > 0) {

    self.viewPointOffset -= SCREEN.width * delta;

    if (self.viewPointOffset <= 0) {
        self.viewPointOffset = 0;
    }
}

```

setViewPointCenter: temporarily adjusts its focus using **viewPointOffset**. This means that when a battle ends, **setViewPointCenter:** will still be focused on the **eventCenter** position. In **update:**, you gradually reduce the value of **viewPointOffset** back to zero. The rate of decrement for **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 you're reusing dead robots to spawn the new robots, and if you position a dead robot at the edge of the screen, its body extends toward the visible part of the screen.

The fix for this is easy: reset the robot sprites before reusing them.

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

```

- (void)reset
{

```

```
[self setTexture:[[SKTTextureCache sharedInstance]
textureNamed:@"robot_base_idle_00"]];

[self.belt setTexture:[[SKTTextureCache sharedInstance]
textureNamed:@"robot_belt_idle_00"]];

[self.smoke setTexture:[[SKTTextureCache sharedInstance]
textureNamed:@"robot_smoke_idle_00"]];

[super reset];
}
```

The method simply changes all the robot sprites to their idle frames during a reset. This method overrides the parent class's reset method, which you're already calling in updatePositions when a robot is dead and off screen.

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 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 onscreen. 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.h** and add this protocol method:

```
//add inside @protocol ActionSpriteDelegate <NSObject> before the first
@end
- (void)actionSpriteDidFinishAutomatedWalking:(ActionSprite
*)actionSprite;
```

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

```
- (void)enterFrom:(CGPoint)origin to:(CGPoint)destination
{
    CGFloat diffX = fabsf(destination.x - origin.x);
    CGFloat diffY = fabsf(destination.y - origin.y);

    if (diffX > 0) {
        self.directionX = 1.0;
    } else {
        self.directionX = -1.0;
    }

    self.xScale = self.directionX * kPointFactor;

    NSTimeInterval duration = MAX(diffX, diffY) / self.walkSpeed;

    self.actionState = kActionStateAutomated;
    [self removeAllActions];
    [self runAction:self.walkAction];

    SKAction *moveAction =
        [SKAction moveTo:destination duration:duration];

    SKAction *blockAction = [SKAction runBlock:^{
        [self.delegate actionSpriteDidFinishAutomatedWalking:self];
        [self idle];
    }];

    [self runAction:[SKAction sequence:@[moveAction,
                                         blockAction]]];
}
```

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, and 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. After moving the sprite, it informs the delegate that the sprite has finished walking and 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 (self.actionState == kActionStateAutomated) {

    self.groundPosition = self.position;
    self.desiredPosition = self.groundPosition;
}
```

SKAction can only modify the **position** attribute of an SKSpriteNode. Since **ActionSprite** is moved using **groundPosition** and **desiredPosition**, you just plug the value of position into them whenever **actionState** is **kActionStateAutomated**.

Switch to **GameLayer.m** and do the following:

```
//replace startGame

- (void)startGame
{
    self.eventState = kEventStateScripted;

    CGPoint destination = CGPointMake(64.0 * kPointFactor,
                                      self.hero.position.y);

    [self.hero enterFrom:self.hero.position to:destination];
}

//add this method below actionSpriteDidAttack:

- (void)actionSpriteDidFinishAutomatedWalking:(ActionSprite
*)actionSprite
{
    self.eventState = kEventStateFreeWalk;
}
```

Instead of starting out with **kEventStateFreeWalk**, you now start with **kEventStateScripted**. Then you make the hero walk to a certain position. Once the hero finishes walking, you trigger the delegate method and the event changes to a free roam.

Still in **GameLayer.m**, make the following changes to **initHero** (or replace the method entirely):

```
- (void)initHero
{
    self.hero = [Hero node];
    self.hero.delegate = self;

    [self.hero setScale:kPointFactor];
    [self.hero.shadow setScale:kPointFactor];

    [self addChild:self.hero.shadow];
    [self addChild:self.hero];
```

```
//change self.hero.position = CGPointMake(100 * kPointFactor, 100  
*kPointFactor); to  
  
    self.hero.position = CGPointMake(-self.hero.centerToSides,  
                                     80 * kPointFactor);  
  
    //add the following two lines  
    self.hero.desiredPosition = self.hero.position;  
    self.hero.groundPosition = self.hero.position;  
  
    //remove [self.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 remove the player's control over the hero during scripted events.

Still in **GameLayer.m**, make the following changes to the indicated methods:

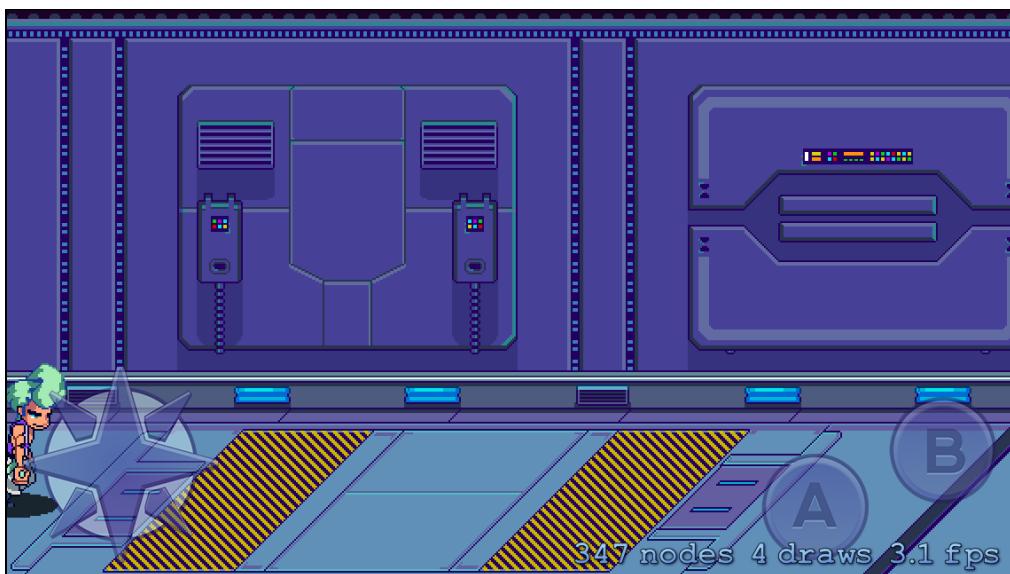
```
- (void)actionDPad:(ActionDPad *)actionDPad  
didChangeDirectionTo:(ActionDPadDirection)direction  
{  
    //add this if-statement at the top  
    if (self.eventState == kEventStateScripted) return;  
  
    //do not modify the rest of the code in this method  
}  
  
- (void)actionDPadTouchUpInside:(ActionDPad *)actionDPad  
{  
    //modify the if condition as shown  
    if (self.eventState != kEventStateScripted &&  
        (self.hero.actionState == kActionStateWalk ||  
         self.hero.actionState == kActionStateRun)) {  
  
        [self.hero idle];  
    }  
}  
  
- (void)actionButtonWasPressed:(ActionButton *)actionButton  
{  
    //add this if-statement to the top of the method  
    if (self.eventState == kEventStateScripted) return;  
  
    //do not modify the rest of the code in this method  
}  
  
- (void)actionButtonWasReleased:(ActionButton *)actionButton  
{
```

```
//add this if-statement to the top of the method
if (self.eventState == kEventStateScripted) return;

//do not modify the rest of the code in this method
}
```

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!



This is a good spot to take a break. You have a pretty cool game on your hands: brainy robots, scripted events and a flashy entrance! What more could you want? How about combo attacks for your hero and support for multiple levels? That's what you're going to add in the next chapter.

Challenge: Modify the AI to have a new “running” state where the robot tries to move away from the hero if its health is low. This will give you more practice setting up 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 additional attacks. First you’ll add two combination actions to his arsenal: a jump punch and a running kick. Then you’ll give him the 1-2-3 punch, a chain of punches guaranteed to lay a robot flat.

But with great power comes great responsibility—to take your share of punches from the robots, that is. You’ll modify the game so that if the robots punch the hero enough consecutive times, the hero himself will fall to the ground.

After that, you’ll enable the game to support multiple levels and finally, you’ll investigate some memory issues.

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 a solid variety of actions, but the hero has only one option for attacking: the 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 (self.actionState == kActionStateJumpRise ||
        self.actionState == kActionStateJumpFall) {

        self.velocity = CGPointMakeZero;
        [self removeAllActions];
        self.actionState = kActionStateJumpAttack;
        [self runAction:self.jumpAttackAction];
    }
}

- (void)runAttack
{
    if (self.actionState == kActionStateRun) {

        [self removeAllActions];
        self.actionState = kActionStateRunAttack;
        [self runAction:self.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 (self.actionState == kActionStateJumpRise ||
          self.actionState == kActionStateJumpFall) {

    [self jumpAttack];
}

else if (self.actionState == kActionStateRun) {
    [self runAttack];
}

//change the first if-statement in update: to this

if (self.actionState == kActionStateWalk ||
    self.actionState == kActionStateRun ||
    self.actionState == kActionStateRunAttack) {

    CGPoint point = CGPointMultiplyScalar(self.velocity, delta);
    self.desiredPosition = CGPointAdd(self.groundPosition, point);
}

```

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

//jump attack action
SKAction *jumpAttackAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"hero_jumpattack" startFrameIdx:0 frameCount:5]
timePerFrame:1.0/10.0];
self.jumpAttackAction = [SKAction sequence:@[jumpAttackAnimation,
[SKAction performSelector:@selector(jumpFall) onTarget:self]]];

//run attack action
SKAction *runAttackAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"hero_runattack" startFrameIdx:0 frameCount:6]
timePerFrame:1.0/10.0];
self.runAttackAction = [SKAction sequence:@[runAttackAnimation,
[SKAction performSelector:@selector(idle) onTarget:self]]];

//add these attributes to init along with the other attributes
self.jumpAttackDamage = 15.0;
self.runAttackDamage = 15.0;
```

Build, run, and try out these new actions, 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 the player can only perform the jump attack once per jump, do the following in **ActionSprite.m**:

```
//between the last #import but before @implementation, add the
//following:

@interface ActionSprite()

@property (assign, nonatomic) BOOL didJumpAttack;

@end
```

Still in **ActionSprite.m**, modify the following methods as per the comments:

```
- (void)jumpLand
{
    if (self.actionState == kActionStateJumpFall ||
        self.actionState == kActionStateRecover) {

        self.jumpHeight = 0;
        self.jumpVelocity = 0;
        //add this line
        self.didJumpAttack = NO;

        self.actionState = kActionStateJumpLand;
        [self runAction:self.jumpLandAction];
    }
}

- (void)jumpAttack
{
    //add this condition
    if (!self.didJumpAttack &&
        (self.actionState == kActionStateJumpRise ||
         self.actionState == kActionStateJumpFall)) {

        self.velocity = CGPointMakeZero;
        [self removeAllActions];
        self.actionState = kActionStateJumpAttack;

        //add this line
        self.didJumpAttack = YES;
        [self runAction:self.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 there can only be one jump attack per jump.

Build and run again, and this time try out your newfound abilities on the robots.



Unfortunately, as you've discovered, the new attacks have no effect 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 `setTexture:` with the following:

```
- (void)setTexture:(SKTexture *)texture
{
    [super setTexture:texture];

    SKTexture *attackTexture =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_attack_00_01"];

    //add these new textures
    SKTexture *runAttackTexture =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_runattack_02"];

    SKTexture *runAttackTexture2 =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_runattack_03"];

    SKTexture *jumpAttackTexture =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_jumpattack_02"];

    //include them in the condition
```

```
if (texture == attackTexture ||
    texture == runAttackTexture ||
    texture == runAttackTexture2 ||
    texture == jumpAttackTexture) {

    [self.delegate actionSpriteDidAttack:self];
}
```

You reference textures that show the hero in his attack pose while running and jumping, and tell the delegate that the hero is attacking when these textures 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:self.hero andTarget:robot
atPosition:&attackPosition])
if (self.hero.actionState == kActionStateJumpAttack) {

[robot knockoutWithDamage:self.hero.jumpAttackDamage
direction:CGPointMake(self.hero.directionX,
0)];

} else if (self.hero.actionState == kActionStateRunAttack) {

[robot knockoutWithDamage:self.hero.runAttackDamage
direction:CGPointMake(self.hero.directionX,
0)];

} else {

[robot hurtWithDamage:self.hero.attackDamage
force:self.hero.attackForce
direction:CGPointMake(self.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 get knocked down!

The game is pretty exciting at this point, but the player might sometimes encounter a scenario where they have to fight off an overwhelming crowd of robots and get pummeled to death before they're able to respond. It's the stuff of which nightmares are made!



One of the reasons the hero is so helpless in this situation is that when he's being hit continuously, there's no chance of escape. While the hero is showing the hurt animation, he isn't able to move or attack, but he can still 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 number 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.m** and add the following private class extension with a few properties:

```
//add after the last #import but before @interface
@interface Hero()

@property (assign, nonatomic) CGFloat hurtTolerance;
@property (assign, nonatomic) CGFloat recoveryRate;
@property (assign, nonatomic) CGFloat hurtLimit;

@end
```

These are:

- **hurtTolerance**: This is, in effect, the hero's secondary hit points variable. Whenever the hero gets 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.

Still in **Hero.m**, 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:(CGFloat)damage
    force:(CGFloat)force
    direction:(CGPoint)direction
{
    [super hurtWithDamage:damage force:force direction:direction];
    if (self.actionState == kActionStateHurt) {
```

```
    self.hurtTolerance -= damage;

    if (self.hurtTolerance <= 0) {
        [self knockoutWithDamage:0 direction:direction];
    }
}

- (void)knockoutWithDamage:(CGFloat)damage
                      direction:(CGPoint)direction
{
    [super knockoutWithDamage:damage direction:direction];

    if (self.actionState == kActionStateKnockedOut) {
        self.hurtTolerance = self.hurtLimit;
    }
}

- (void)update:(NSTimeInterval)delta
{
    [super update:delta];

    if (self.hurtTolerance < self.hurtLimit) {

        self.hurtTolerance +=
            self.hurtLimit * delta / self.recoveryRate;

        if (self.hurtTolerance >= self.hurtLimit) {
            self.hurtTolerance = self.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 want to add logic on top of 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.



The 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 you manage the attack action. Whenever an attack connects with an enemy, you'll give the hero 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 the following properties:

```
//add under attributes
@property (assign, nonatomic) CGFloat actionDelay;
@property (assign, nonatomic) CGFloat attackDelayTime;
```

Now open **ActionSprite.m** and make the following changes to the existing code:

```
//add inside the if statement of idle
self.actionDelay = 0.0;

//add to the end of update:, before the last curly brace
if (self.actionDelay > 0) {
    self.actionDelay -= delta;
```

```
if (self.actionDelay <= 0) {
    self.actionDelay = 0;
}

//modify attack to look like this
- (void)attack
{
    //add actionDelay as condition
    if (self.actionState == kActionStateIdle ||
        self.actionState == kActionStateWalk ||
        (self.actionState == kActionStateAttack &&
         self.actionDelay <= 0)) {

        [self removeAllActions];
        [self runAction:self.attackAction];
        self.actionState = kActionStateAttack;
        //set actionDelay to the value of attackDelayTime
        self.actionDelay = self.attackDelayTime;

    } else if (self.actionState == kActionStateJumpRise ||
               self.actionState == kActionStateJumpFall) {

        [self jumpAttack];

    } else if (self.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**:

```
//set this property init along with the other attributes
self.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.m** and add the following:

```
//add below @property (assign, nonatomic) CGFloat hurtLimit;
@property (assign, nonatomic) CGFloat attackTwoDelayTime;
@property (assign, nonatomic) CGFloat attackThreeDelayTime;
@property (assign, nonatomic) CGFloat chainTimer;
```

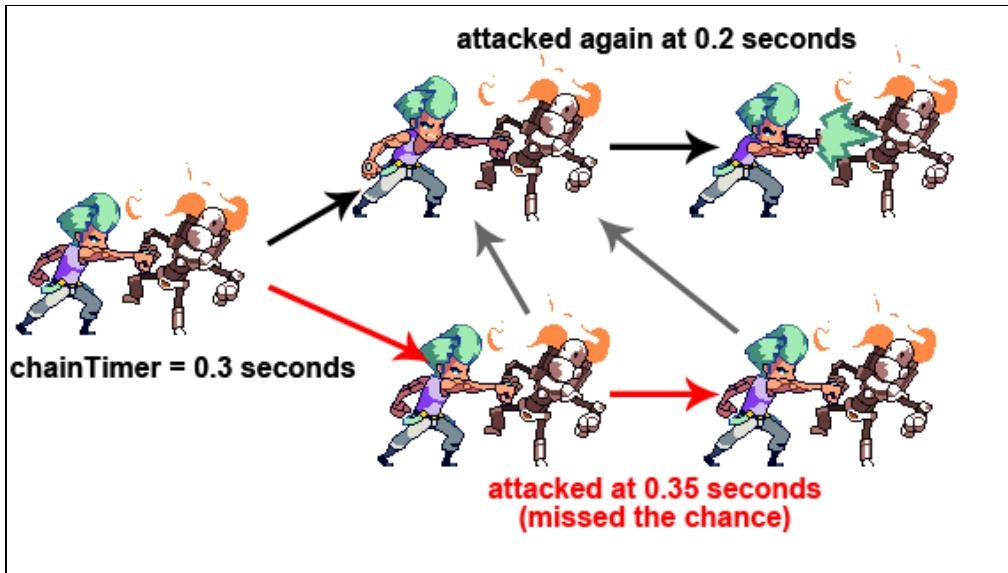
Now turn your attention to **Hero.h** and add the following public properties:

```
@property (strong, nonatomic) SKAction *attackTwoAction;
@property (strong, nonatomic) SKAction *attackThreeAction;
@property (assign, nonatomic) CGFloat attackTwoDamage;
@property (assign, nonatomic) CGFloat 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 player triggers the second attack 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 the player triggers it within the 0.3-second limit, the hero will execute the final, more powerful punch.

You need to make a couple of changes to **Hero.m**. First, do the following:

```
//add these actions in init along with the other actions
//attack two action
SKAction *attackTwoAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"hero_attack_01" startFrameIdx:0 frameCount:3]
timePerFrame:1.0/12.0];
self.attackTwoAction = [SKAction sequence:@[attackTwoAnimation,
[SKAction performSelector:@selector(idle) onTarget:self]]];

//attack three action
SKAction *attackThreeAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"hero_attack_02" startFrameIdx:0 frameCount:5]
timePerFrame:1.0/10.0];
self.attackThreeAction = [SKAction sequence:@[attackThreeAnimation,
[SKAction performSelector:@selector(idle) onTarget:self]]];

//add these attributes in init along with the other attributes
_attackTwoDelayTime = 0.14;
_attackThreeDelayTime = 0.45;
_chainTimer = 0;
_attackTwoDamage = 10.0;
```

```
_attackThreeDamage = 20.0;

//add to the beginning of update:, after [super update:delta]

if (self.chainTimer > 0) {

    self.chainTimer -= delta;

    if (self.chainTimer <= 0) {
        self.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 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 &&
        self.actionDelay <= 0 &&
        self.chainTimer > 0) {

        self.chainTimer = 0;
        [self removeAllActions];
        [self runAction:self.attackTwoAction];
        self.actionState = kActionStateAttackTwo;
        self.actionDelay = self.attackTwoDelayTime;

    } else if (self.actionState == kActionStateAttackTwo &&
               self.actionDelay <= 0 && self.chainTimer > 0) {

        self.chainTimer = 0;
        [self removeAllActions];
        [self runAction:self.attackThreeAction];
        self.actionState = kActionStateAttackThree;
        self.actionDelay = self.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 `setTexture:` with the following (or modify it according to the comments):

```
- (void)setTexture:(SKTexture *)texture
{
    [super setTexture:texture];

    SKTexture *attackTexture =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_attack_00_01"];

    SKTexture *runAttackTexture =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_runattack_02"];

    SKTexture *runAttackTexture2 =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_runattack_03"];

    SKTexture *jumpAttackTexture =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_jumpattack_02"];

    //add these new textures
    SKTexture *attackTexture2 =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_attack_01_01"];

    SKTexture *attackTexture3 =
        [[SKTextureCache sharedInstance]
         textureNamed:@"hero_attack_02_02"];

    //change the conditions
    if (texture == attackTexture || texture == attackTexture2) {

        if ([self.delegate actionSpriteDidAttack:self]) {
            self.chainTimer = 0.3;
        }
    } else if (texture == attackTexture3) {

        [self.delegate actionSpriteDidAttack:self];
    } else if (texture == runAttackTexture ||
               texture == runAttackTexture2 ||
               texture == jumpAttackTexture) {

        [self.delegate actionSpriteDidAttack:self];
    }
}
```

```
}
```

You add the textures for the second and third attacks to the list of textures 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 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. The third attack is in a separate `else-if` block because you will put something different there later.

Build, run, and try out your two new actions.



What's left to do? If you tried to attack a robot with the 1-2-3 punch, then you certainly noticed that the collision resolution code behaves the same as 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 (self.hero.actionState == kActionStateJumpAttack) {
```

```
[robot knockoutWithDamage:self.hero.jumpAttackDamage
    direction:CGPointMake(self.hero.directionX,
    0)];
} else if (self.hero.actionState == kActionStateRunAttack) {

[robot knockoutWithDamage:self.hero.runAttackDamage
    direction:CGPointMake(self.hero.directionX,
    0)];
//add this else-if statement
} else if (self.hero.actionState == kActionStateAttackThree) {

[robot knockoutWithDamage:self.hero.attackThreeDamage
    direction:CGPointMake(self.hero.directionX,
    0)];
//add this else-if statement
} else if (self.hero.actionState == kActionStateAttackTwo) {

[robot hurtWithDamage:self.hero.attackTwoDamage
    force:self.hero.attackForce
    direction:CGPointMake(self.hero.directionX, 0.0)];
} else {

[robot hurtWithDamage:self.hero.attackDamage
    force:self.hero.attackForce
    direction:CGPointMake(self.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**. If the hero then 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

You now have a fairly complex game with several different attack modes, but you're still playing the same level from Chapter 1!

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 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 SKScenes in Sprite Kit 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 logic to support multiple levels. When the player completes a level, simply 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 (assign, nonatomic) NSInteger totalLevels;
@property (assign, nonatomic) NSInteger currentLevel;

//add these methods
+ (instancetype)nodeWithLevel:(NSInteger)level;
- (instancetype)initWithLevel:(NSInteger)level;

```

Now switch to **GameLayer.m** and do the following:

```

//add these methods
+ (instancetype)nodeWithLevel:(NSInteger)level
{
    return [[self alloc] initWithLevel:level];
}
- (instancetype)initWithLevel:(NSInteger)level
{
    if (self = [super init]) {
        [self loadLevel:level];
        [self initHero];
        [self initRobots];
        [self initBrains];
    }
    return self;
}

//add these to the end of loadLevel:
self.totalLevels = levelArray.count;
self.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
+ (instancetype)sceneWithSize:(CGSize)size
                           level:(NSUInteger)level;

- (instancetype)initWithSize:(CGSize)size
                           level:(NSUInteger)level;

```

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

```

+ (instancetype)sceneWithSize:(CGSize)size
                           level:(NSUInteger)level
{

```

```
    return [[self alloc] initWithSize:size level:level];
}

//added level to the initializer
- (instancetype)initWithSize:(CGSize)size
                      level:(NSUInteger)level
{
    if (self = [super initWithSize:size])
    {
        SKTextureAtlas *atlas =
        [SKTextureAtlas atlasNamed:@"sprites"];

        [[SKTextureCache sharedInstance]
         addTexturesFromAtlas:atlas
         filteringMode:SKTextureFilteringNearest];

        [[SKTextureCache sharedInstance]
         setEnableFallbackSuffixes:YES];

        atlas = [SKTextureAtlas atlasNamed:@"joypad"];

        [[SKTextureCache sharedInstance]
         addTexturesFromAtlas:atlas
         filteringMode:SKTextureFilteringLinear];

        //added level number
        _gameLayer = [GameLayer nodeWithLevel:level];
        [self addChild:_gameLayer];

        _hudLayer = [HudLayer node];
        [self addChild:_hudLayer];

        _hudLayer.zPosition = _gameLayer.zPosition + SCREEN.height;
        _hudLayer.dPad.delegate = _gameLayer;
        _gameLayer.hud = _hudLayer;

        _hudLayer.buttonA.delegate = _gameLayer;
        _hudLayer.buttonB.delegate = _gameLayer;

#if DRAW_DEBUG_SHAPES
        _debugLayer = [DebugLayer nodeWithGameLayer:_gameLayer];
        _debugLayer.zPosition = _gameLayer.zPosition + 1;
        [self addChild:_debugLayer];
#endif
    }

    return self;
}
```

Similar to `GameLayer`, you now add the level number when creating a new `GameScene`. `GameScene` will use the level number and pass it on to `GameLayer`.

To test if your new initializers work, go to **TitleScene.m** and change **touchesBegan:withEvent:** as follows:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
    SKSpriteNode *start =
        (SKSpriteNode *)[self childNodeWithName:@"StartText"];

    [start removeAllActions];
    start.hidden = YES;

    //add level number 0 to GameScene creation

    SKScene *gameScene =
        [GameScene sceneWithSize:self.size level:0];

    [self.view presentScene:gameScene
                      transition:[SKTransition fadeWithDuration:1.0]];
}
```

This makes **GameScene** load the first level when the player touches the screen in the title scene.

Build and run to confirm that the first level still loads properly.



All right, it still works!

You have to increment the level number and reload GameScene when you want to go to the next level. Here's the dilemma: GameLayer can determine when a level has ended, but it is ViewController's view (SKView) that has to load a new GameScene, so there has to be a way for these two to communicate.

Notifications

One way of getting GameLayer to communicate with ViewController is through the scene property of SKNode and the view property of SKScene, but for this starter kit, you will be using the Notification Center.

The Notification Center is like a global message board where objects can send and receive messages between one other. An object can subscribe to a specific type of message and it will get notified as soon as another object sends that message type.

Open **ViewController.m** and do the following:

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

//add in viewDidLoad, after [super viewDidLoad]
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(didReceivePresentGameSceneNotification:)
name:@"PresentGame" object:nil];

//add this method
- (void)didReceivePresentGameSceneNotification:(NSNotification *)notification
{
    NSDictionary *attributes = notification.userInfo;
    NSInteger level = [attributes[@"Level"] integerValue];

    SKView *skView = (SKView *)self.view;

    GameScene *gameScene =
        [GameScene sceneWithSize:skView.bounds.size level:level];

    [[SKTextureCache sharedInstance]
        loadTexturesFromAtlas:[SKTextureAtlas atlasNamed:@"sprites"]
        filteringMode:SKTextureFilteringNearest];

    [skView presentScene:gameScene
                transition:[SKTransition fadeWithDuration:1.0]];
}
```

In viewDidLoad, you register ViewController as an observer of the “PresentGame” notification. If another object posts a notification with the same name, that will trigger didReceivePresentGameSceneNotification::.

When an object posts a notification, it does so with the NSNotification object. This object can have its own dictionary of attributes in case the object needs to send custom parameters.

When ViewController receives the “PresentGame” notification, you get the level number through the “Level” attribute contained in the notification. Then you use this to load a new instance of GameScene with the appropriate level number.

Note: SKTextureCache's `loadTexturesFromAtlas:filteringMode:` method refreshes the textures loaded from a specific atlas and re-implements the filtering mode indicated.

You only need this because of a **bug in Sprite Kit**. Sprite Kit sometimes forgets the filtering mode you set for some of the previously loaded textures. Because of this, you might occasionally lose the crispness of the pixel art sprites whenever a scene transition occurs.

Don't worry. Knowing Apple, this will most likely get fixed in the next version of Sprite Kit. For now, you have this workaround.

You can test this method by replacing `TitleScene`'s transition to `GameScene`.

Open `TitleScene.m` and do the following in `touchesBegan:withEvent:`:

```
//replace this line
//[[self.view presentScene:[GameScene sceneWithSize:self.size level:0]
transition:[SKTransition fadeWithDuration:1.0]]];

//with this
[[NSNotificationCenter defaultCenter]
postNotificationName:@"PresentGame" object:self userInfo:@{@"Level" :
@0}];
```

Build and run, and check if the first level still loads properly.



Of course it still works. ☺

The next level

Now that ViewController can receive notifications from anywhere, it will be easy to ask it to load a new GameScene from GameLayer.

Go to **GameLayer.m** and do the following:

```
//replace updateEvent

- (void)updateEvent
{
    if (self.eventState == kEventStateBattle &&
        self.activeEnemies <= 0) {

        CGFloat maxCenterX =
            self.tileMap.mapSize.width *
            self.tileMap.tileSize.width *
            kPointFactor - CENTER.x;

        CGFloat cameraX =
            MAX(MIN(self.hero.position.x, maxCenterX), CENTER.x);

        self.viewPointOffset = cameraX - self.eventCenter;

        //add this
        if (self.battleEvents.count == 0) {
            [self exitLevel];
        } else {
            self.eventState = kEventStateFreeWalk;
        }

    } else if (self.eventState == kEventStateFreeWalk) {
        //modified this part
        [self cycleEvents];

        //add this
    } else if (self.eventState == kEventStateScripted) {

        CGFloat exitX =
            self.tileMap.tileSize.width *
            self.tileMap.mapSize.width *
            kPointFactor + self.hero.centerToSides;

        if (self.hero.position.x >= exitX) {

            self.eventState = kEventStateEnd;
            if (self.currentLevel < self.totalLevels - 1) {

                //next level
                [[NSNotificationCenter defaultCenter]
                    postNotificationName:@"PresentGame"
                    object:nil
                    userInfo:@{@"Level" : @(self.currentLevel + 1)}];
            }
        }
    }
}
```

```

        } else {
            //end game
        }
    }

//add this new method
- (void)exitLevel
{
    self.eventState = kEventStateScripted;

    CGFloat exitX =
        self.tileMap.tileSize.width *
        self.tileMap.mapSize.width *
        kPointFactor + self.hero.centerToSides;

    [self.hero exitFrom:self.hero.position
                  to:CGPointMake(exitX, self.hero.position.y)];
}

```

You change `updateEvent` to check for a winning condition. If `eventState` is `kEventStateBattle` and there are no more battle events left, then you trigger `exitLevel`. Otherwise, you change the state back to `kEventStateFreeWalk`.

`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 is out of sight, `updateEvent` changes the event to `kEventStateEnd`, signaling the end of the level. If the current level is not the last, 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, go to `ActionSprite.m` and add this method:

```

- (void)exitFrom:(CGPoint)origin to:(CGPoint)destination
{
    CGFloat diffX = fabsf(destination.x - origin.x);
    CGFloat diffY = fabsf(destination.y - origin.y);

    if (diffX > 0) {
        self.directionX = 1.0;
    } else {
        self.directionX = -1.0;
    }

    self.xScale = self.directionX * kPointFactor;

    NSTimeInterval duration = MAX(diffX, diffY) / self.walkSpeed;
}

```

```
self.actionState = kActionStateAutomated;
[self removeAllActions];
[self runAction:self.walkAction];
[self runAction:[SKAction moveTo:destination
                           duration:duration]];
}
```

This is almost the same as `enterFrom:to:`, with one minor difference—the sprite won't go back to the idle state after reaching its destination.

Build and run, and see if you can make it through all the levels defined in **Levels.plist**. Exit stage right.



Optional: a simple loading scene

Sprite Kit doesn't transition to new scenes very smoothly and takes awhile to create, load and present scenes. Whenever a scene is presented, you'll experience a noticeable delay before you see the new scene onscreen.

If you're loading a resource-heavy scene such as `GameScene`, it would be better to show a loading scene during this delay so the player won't think the game is stuck.

Select the **Menus** group, go to **File\New\File**, choose the **iOS\Cocoa Touch\Objective-C class** template and click **Next**. Enter **SKScene** for Subclass of, click **Next** and name the new file **LoadingScene**.

Open **LoadingScene.m** and add this method:

```
- (instancetype)initWithSize:(CGSize)size
{
    if (self = [super initWithSize:size]) {
        self.scaleMode = SKSceneScaleModeAspectFit;
```

```
SKLabelNode *loadingLabel =
    [SKLabelNode labelNodeWithFontNamed:@"04b03"];

    loadingLabel.fontSize = 40 * kPointFactor;

    loadingLabel.horizontalAlignmentMode =
        SKLabelHorizontalAlignmentModeLeft;

    loadingLabel.text = @"Loading...";

    loadingLabel.position =
        CGPointMake(SCREEN.width - 185 * kPointFactor,
                    20 * kPointFactor);

    [self addChild:loadingLabel];
}

return self;
}
```

SKLabelNode is a subclass of SKNode that can display text. You create a left-aligned SKLabelNode with a 40-point Arial font and give it the appropriate text to display, and then you place it in the lower-right corner of the screen.

This must be one of the simplest scenes you've ever created!

Now switch to **ViewController.m** and do the following:

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

//replace didReceivePresentGameSceneNotification
-(void)didReceivePresentGameSceneNotification:(NSNotification *)
notification
{
    NSDictionary *attributes = notification.userInfo;
    NSInteger level = [attributes[@"Level"] integerValue];

    SKView *skView = (SKView *)self.view;
    //presented loading scene first

    SKScene *loadingScene =
        [LoadingScene sceneWithSize:skView.bounds.size];

    [skView presentScene:loadingScene
                  transition:[SKTransition fadeWithDuration:1.0]];

    //added this

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT
        , 0), ^{

```

```
GameScene *gameScene =
    [GameScene sceneWithSize:skView.bounds.size level:level];

dispatch_async(dispatch_get_main_queue(), ^{
    [[SKTextureCache sharedInstance]
        loadTexturesFromAtlas:
        [SKTextureAtlas atlasNamed:@"sprites"]
        filteringMode:SKTextureFilteringNearest];

    [skView presentScene:gameScene
        transition:[SKTransition fadeWithDuration:1.0]];
});
});
```

You rewrite `didReceivePresentGameSceneNotification:` so that it initially loads `LoadingScene` instead of `GameScene`. Then, you create a background block through this line:

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT
, 0), ^{
    //background code here});
```

This command instructs the game to execute the code inside the curly braces in a background thread/queue, while your whole game is running on what is called the main thread/queue.

Each queue is serial in nature, such that only one process can happen at any given time in that queue. This means that when the main queue is loading a new scene, the game has to wait for it to finish before it can do something else.

When you run something on a different thread/queue, the game's main queue doesn't have to wait for it and can happily continue whatever it is it wants to do.

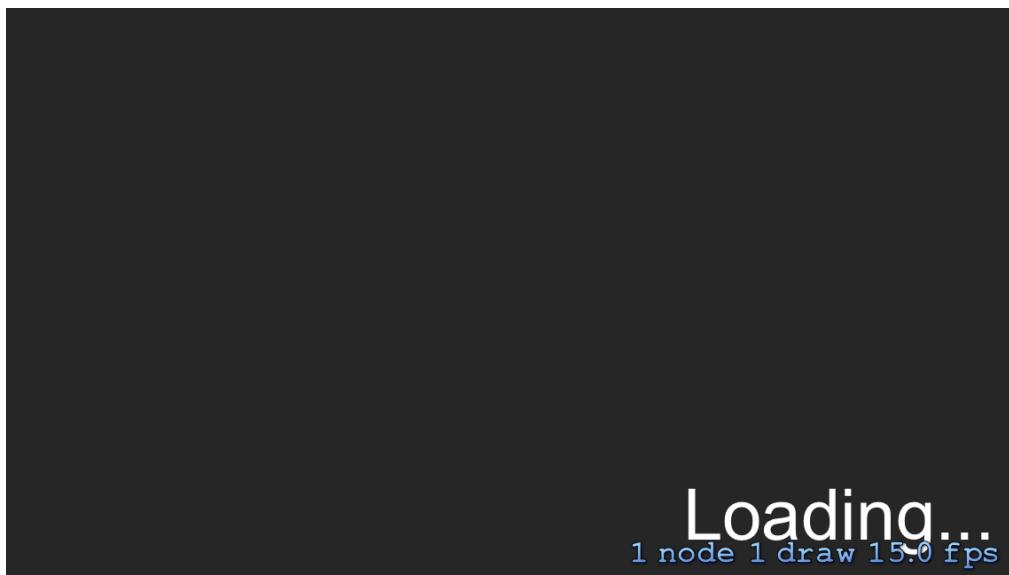
Hence, you create the `GameScene` inside this background instead. Then, right after that, you have this line of code:

```
dispatch_async(dispatch_get_main_queue(), ^{
    [[SKTextureCache sharedInstance]
        loadTexturesFromAtlas:
        [SKTextureAtlas atlasNamed:@"sprites"]
        filteringMode:SKTextureFilteringNearest];

    [skView presentScene:gameScene
        transition:[SKTransition fadeWithDuration:1.0]];
});
```

This tells the game to execute the code inside the curly braces in the main queue. In iOS, any code that updates the screen has to be done on the main thread, so you moved the code that presents `GameScene` inside this block.

Build and run, and check out your new loading scene.



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 memory allocation.

Since everything is "automatic," you wouldn't think there could be any memory issues such as leaks, right?

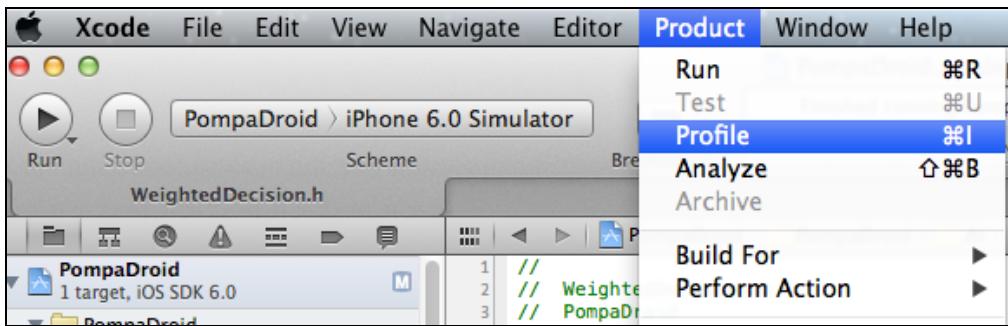
To definitively answer that question, you're going to profile your project for memory leaks.

To complete this section, you're going to have to play through an entire level several times. To expedite things, 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.

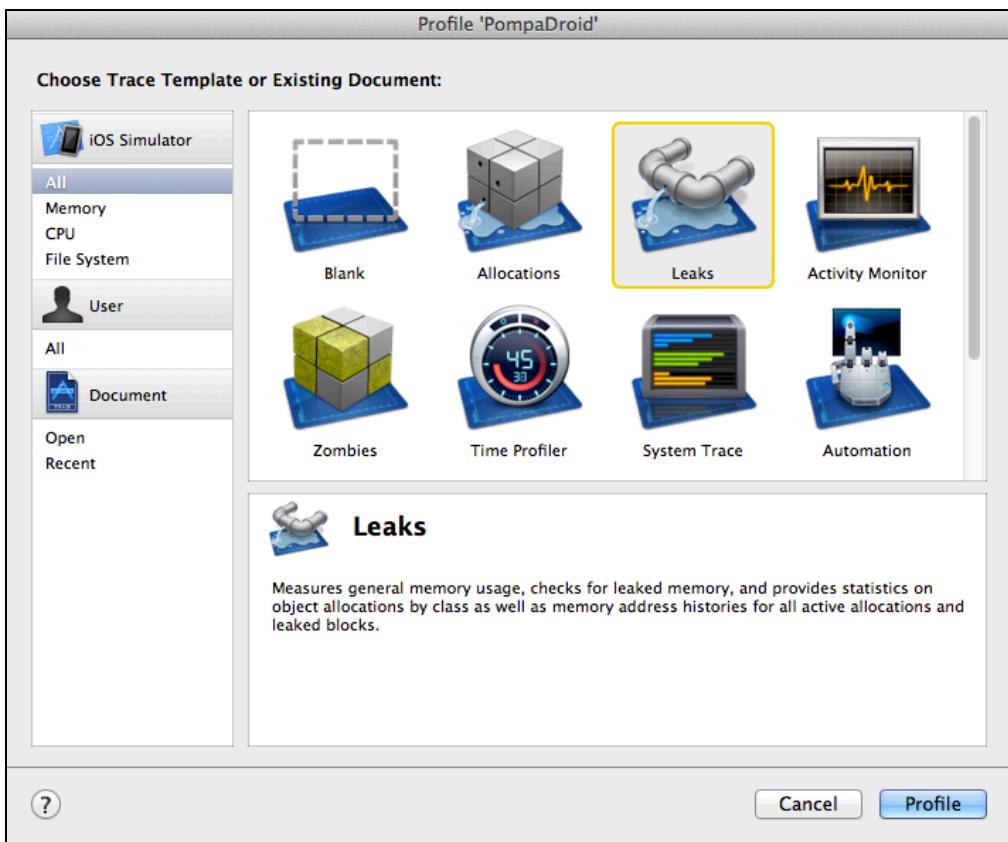
Note: Profiling your application simply means putting its specific aspects to the test. This could be anything from running CPU and GPU diagnostics to checking memory allocation and leaks.

Do you have a leaky project?

Select **Product\Profile** (or use the Command+I keyboard shortcut) to run Instruments, the profiling tool.

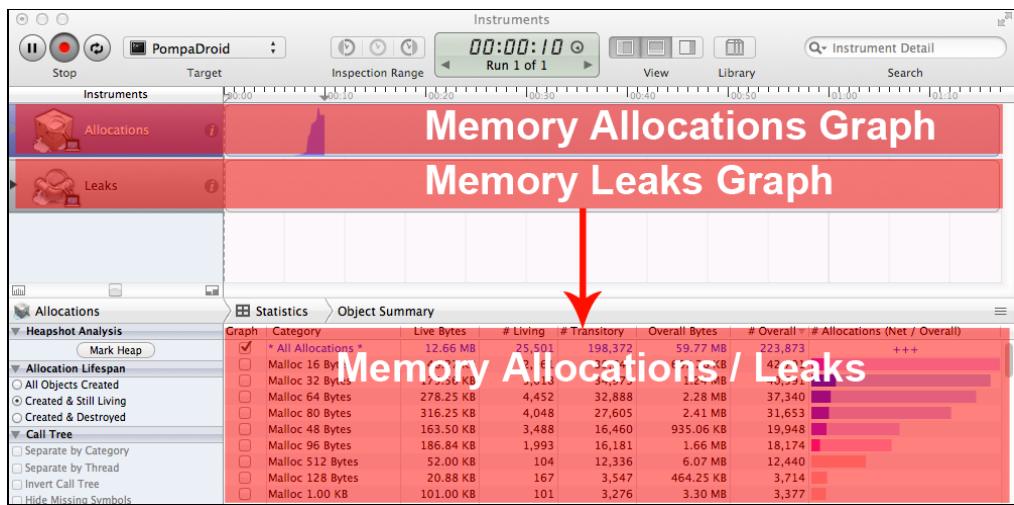


Wait a few seconds for your game to compile for the tool. When it's done, Instruments should ask you what kind of test you want to perform:



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:

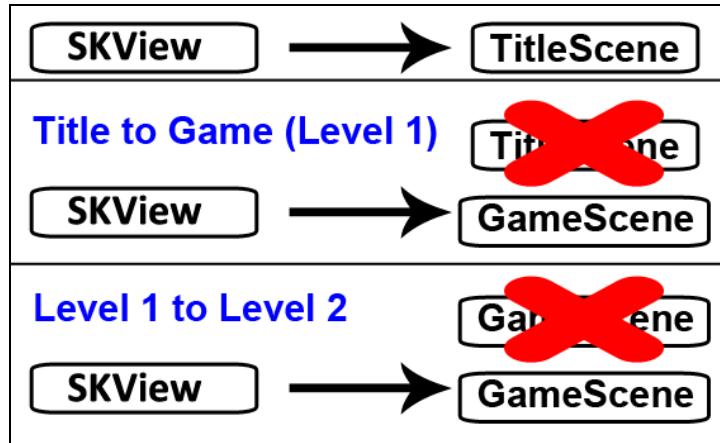


By default, the memory allocations graph is selected, with the bottom panel showing all the allocations for the objects your code created. Click on the memory leaks graph and the bottom panel will show you all your leaking objects.

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 begin to run the game, showing you the title scene.

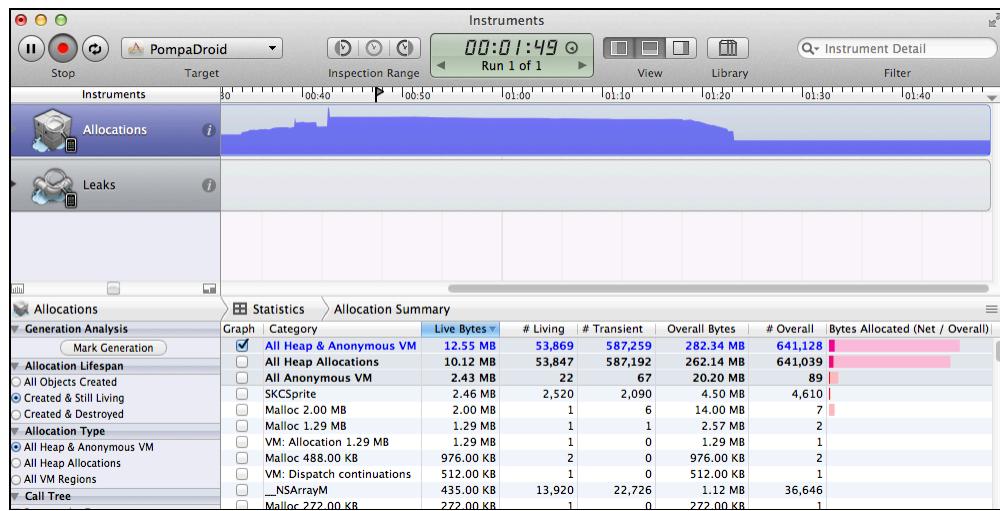
Memory leaks occur when a program doesn't properly discard/deallocate its objects. Sprite Kit discards objects in groups, depending on their SKScene membership. When the game transitions from one SKScene to another, it deallocates the old SKScene from memory along with all objects inside it. Or at least, 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.

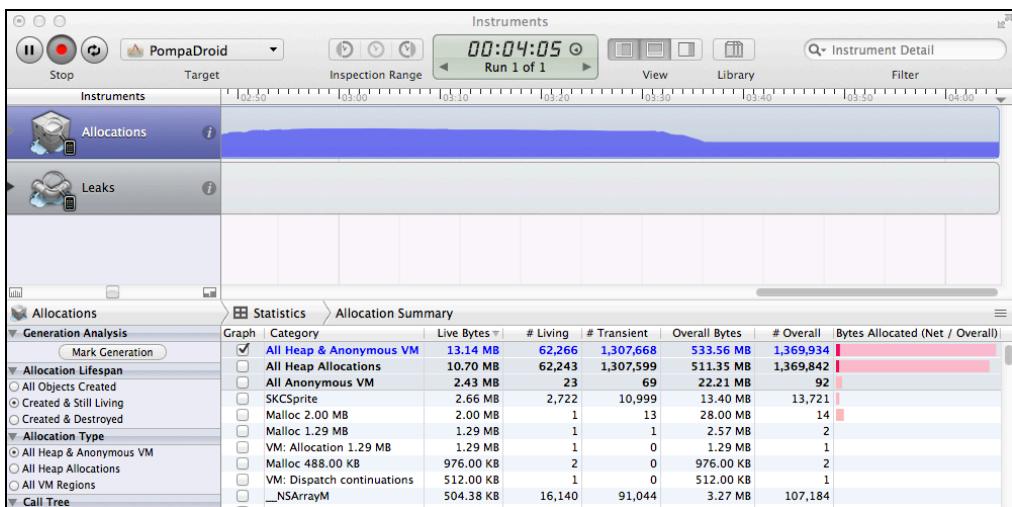
Now make the game transition from **TitleScene** to **GameScene** by tapping on the screen. Wait for **GameScene** to load completely and then look at the memory leaks graph.



The memory leaks graph is empty, which means there are no leaking objects in **TitleScene**.

Note: When profiling your project, it's best to use an actual test device instead of the Simulator, which can be inaccurate and inconsistent in reflecting the correct memory diagnosis.

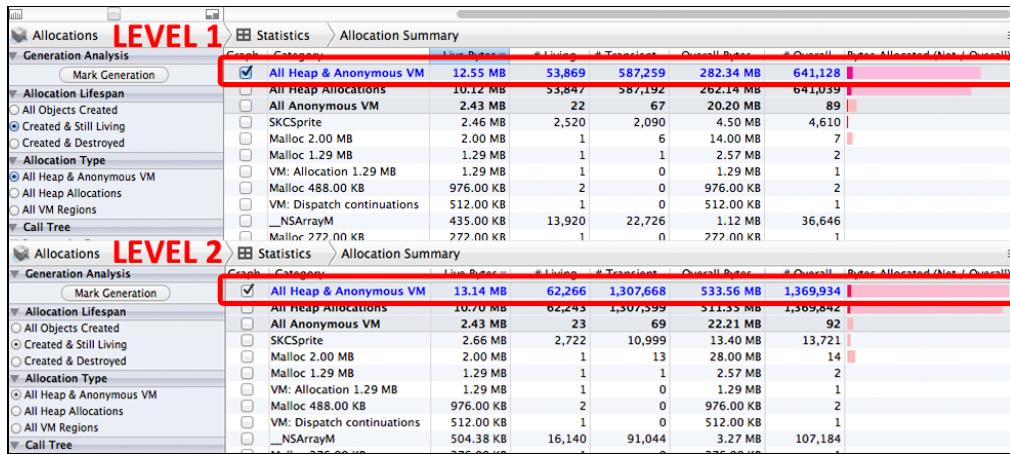
So far, so good! Now play until the end of the first level and let the game transition to the second level. After the second level loads, observe the memory leaks graph again.



The leaks section still comes up empty. That's a relief! It seems too good to be true, doesn't it? Well, I'm sorry to say that it is.

Look back at both Level 1 and Level 2's memory diagnostics again and observe the memory allocations graph. Specifically, look at the **All Heap & Anonymous VM** category to observe how much memory your app uses.

Here they are side by side:



From Level 1 to Level 2, you have an increase of approximately 0.5 MB of memory usage. Live memory can gradually grow as you transition from scene to scene depending on what Sprite Kit caches behind the scenes, but a growth of 0.5 MB from loading the same scene again raises a red flag.

This means that some objects got left behind when the game deallocated the first GameScene. They still exist somewhere, but the active part of your game is no longer using them. What's worse is that they don't appear as leaks because they are still somehow linked to a part of your game.

When you have this type of memory issue, it most probably means that some objects don't get properly deallocated. You can use breakpoints in your code to determine which objects those are.

For this exercise, you will be looking at three possible culprits: GameScene, GameLayer and ActionSprite.

Add this method to **GameScene.m**, **GameLayer.m** and **ActionSprite.m**:

```
- (void)dealloc
{
}
```

Every class has a dealloc method that gets called automatically when an object is removed from memory. You wrote it down so you can intercept calls to dealloc using breakpoints. In Xcode, click on the line number inside dealloc to insert a breakpoint. You should see a blue marker appear on the left side of the code.

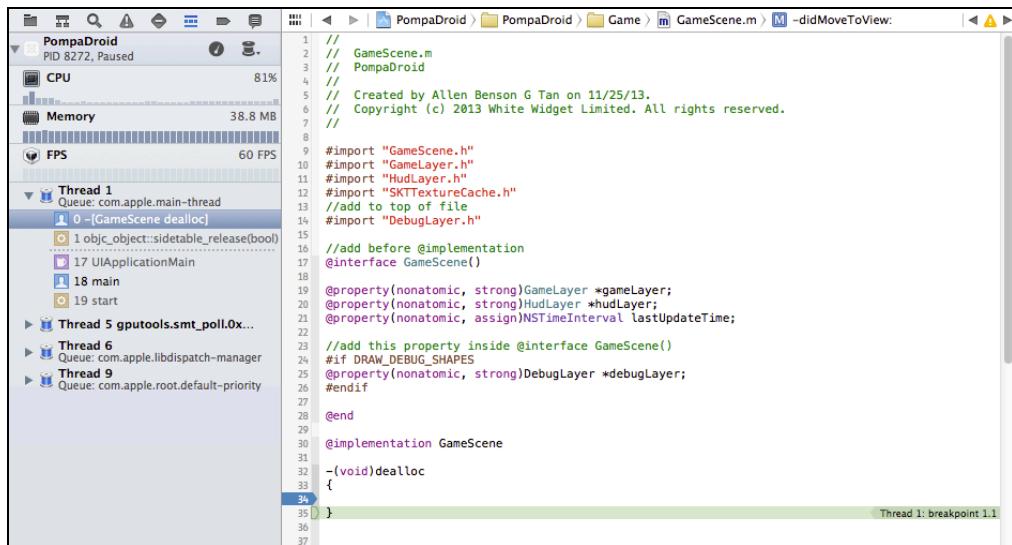
```

32 -(void)dealloc
33 {
34 }
35 }

```

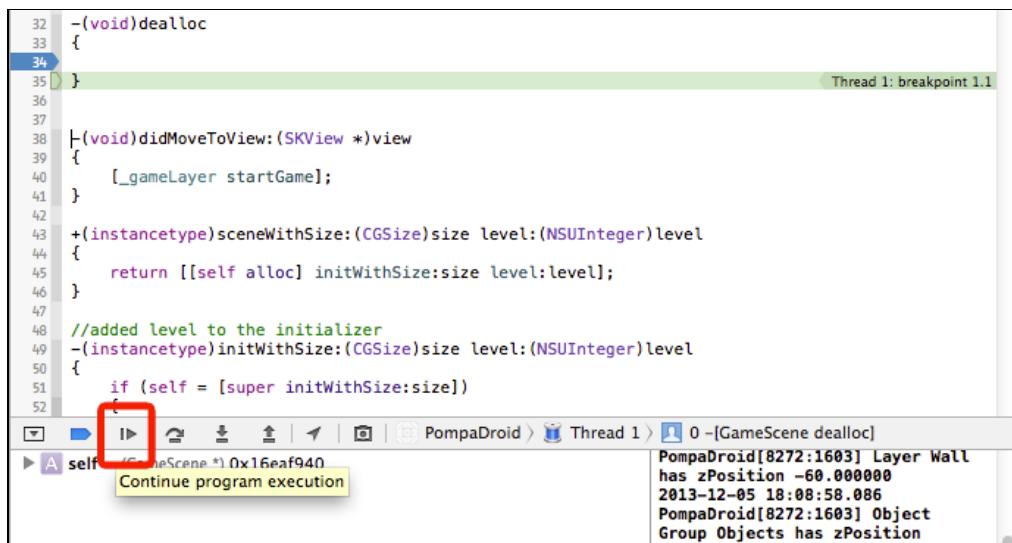
Do this for all three classes: GameScene, GameLayer and ActionSprite.

Build and run, and finish the first level.



After you finish the first level, Xcode should show you something similar to the above. The code has stopped on the breakpoint you created before. This means that the program passed through this line of code and that GameScene gets deallocated. As expected, it's the first of the three, so no problems here.

To continue, press the play button located at the left side of your Debug Area (this area is below the code) as shown in this screenshot:



As soon as you press play, the game will stop again when it crosses the next breakpoint.

```

1 // GameLayer.m
2 // PompaDroid
3 /**
4 * Created by Allen Benson G Tan on 11/25/13.
5 * Copyright (c) 2013 White Widget Limited. All rights reserved.
6 */
7
8 #import "GameLayer.h"
9 #import "Robot.h"
10 #import "ArtificialIntelligence.h"
11
12 @implementation GameLayer
13
14 /*
15 -(void)dealloc
16 {
17     NSLog(@"DEALLOCED LAYER");
18     [_hero cleanup];
19     Robot *robot;
20     for (robot in _robots)
21     {
22         [robot cleanup];
23     }
24 }*/
25
26 -(void)dealloc
27 {
28
29 }
30
31 //replace_startGame
32 -(void)startGame
33 {
34     _eventState = kEventStateScripted;
35 }
```

This time it stopped on GameLayer's dealloc, so it means that GameLayer is discarded properly.

Press the play button again and see what happens.

```

1 // GameLayer.m
2 // PompaDroid
3 /**
4 * Created by Allen Benson G Tan on 11/25/13.
5 * Copyright (c) 2013 White Widget Limited. All rights reserved.
6 */
7
8 #import "GameLayer.h"
9 #import "Robot.h"
10 #import "ArtificialIntelligence.h"
11
12 @implementation GameLayer
13
14 /*
15 -(void)dealloc
16 {
17     NSLog(@"DEALLOCED LAYER");
18     [_hero cleanup];
19     Robot *robot;
20     for (robot in _robots)
21     {
22         [robot cleanup];
23     }
24 }*/
25
26 -(void)dealloc
27 {
28
29 }
30
31 //replace_startGame
32 -(void)startGame
33 {
34     _eventState = kEventStateScripted;
35 }
```

The game now shows the first level, and no other breakpoint was triggered! This means that all the ActionSprites inside weren't discarded after GameLayer was deallocated.

This is very bad indeed. In every GameScene, there are around 51 ActionSprites (50 Robots and a Hero) that take up permanent space in memory. That's a lot of wasted memory.

Now the question that must be on your mind right now is probably: Shouldn't ARC have prevented this?



The problem and the solution

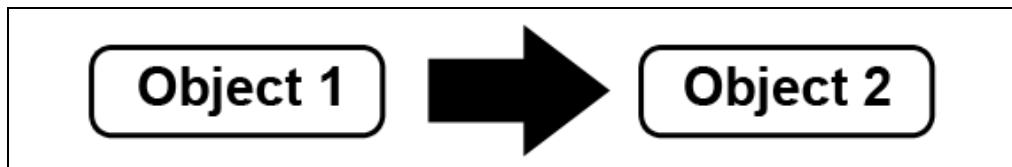
In Objective-C, an object is only cleaned up when no other object references it. This is why ARC includes the term “reference counting.” For each object, the compiler counts how many other objects still reference it. When there are no references left for an object, it gets removed from memory.

A memory leak occurs when an object isn’t deleted because it is still referenced somewhere, but the game doesn’t use it anymore and all access to it from the current state of the program has been lost.

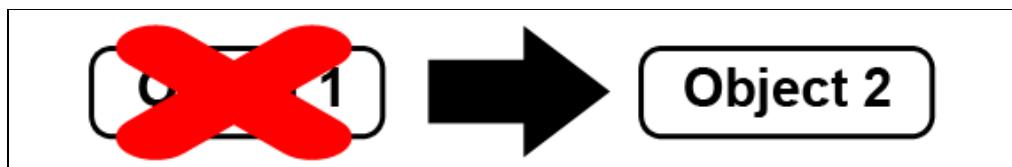
ARC’s job is to properly remove references to objects when they are not needed. However, there is one thing that confuses ARC: **retain cycles**.

A retain cycle occurs when two objects reference each other and it causes confusion about how to delete these objects.

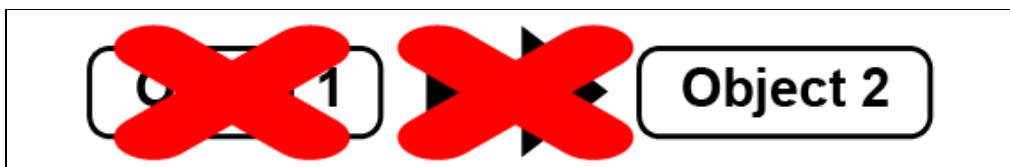
Imagine a normal situation where only one object references another:



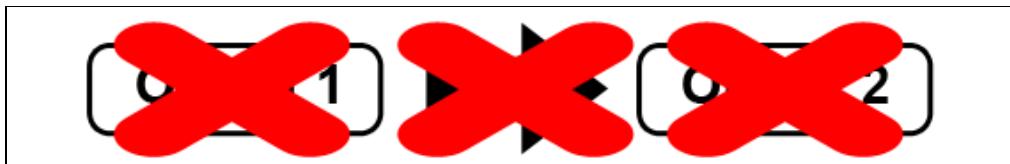
Object 1 is claiming ownership of Object 2. When nothing is claiming Object 1 anymore, then Object 1 will be removed from memory:



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 SKScene 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.

In Sprite Kit, an `SKAction` that you create using `performSelector:target:` keeps a strong reference to its target.

You created some actions that did the following:

```
[SKAction performSelector:@selector(idle) onTarget:self]
```

The target in this example is the `ActionSprite` that created the action. When you created the hero's attack action, for example, you used this line in `Hero.m`:

```
self.attackAction = [SKAction sequence:@[attackAnimation, [SKAction performSelector:@selector(idle) onTarget:self]]];
```

Here, the `SKAction` strongly references `Hero`. As long as `attackAction` exists, `Hero` cannot be removed from memory.

However, `attackAction` itself is also strongly referenced by `Hero` via this property:

```
@property (nonatomic, strong) SKAction *attackAction;
```

`attackAction` cannot be deleted because it is strongly referenced by `Hero`, and `Hero` cannot be deleted because it is strongly referenced by `attackAction` (because of the inner `SKAction`). 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 you're no longer using the `ActionSprite`. 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 to set up a pre-step to `dealloc` for these `ActionSprites`.

Open **ActionSprite.h** and add this method prototype:

```
- (void)cleanup;
```

Switch 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;
}
```

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 `SKActions` they contained before. Now all that's left is to trigger the `cleanup` function when you are sure that you don't need `ActionSprite` anymore. In the

earlier exercise, you determined that GameLayer's dealloc does get triggered, so that is probably the best place to activate the clean up process.

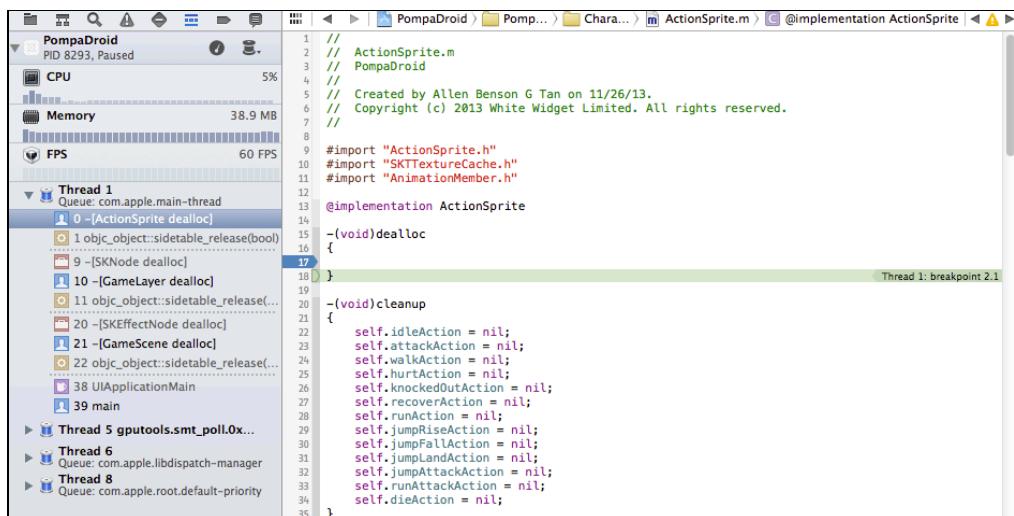
Open **GameLayer.m** and replace dealloc:

```
//replace dealloc
- (void)dealloc
{
    [self.hero cleanup];

    Robot *robot;
    for (robot in self.robots) {
        [robot cleanup];
    }
}
```

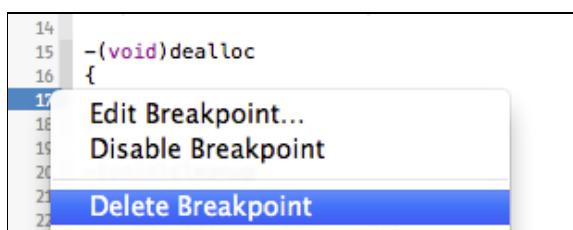
When GameLayer gets deallocated, you tell all ActionSprites to clean up after themselves.

Build and run, and perform the same breakpoint exercise you did earlier.



The game should now stop at the breakpoint inside ActionSprite's dealloc. Success!

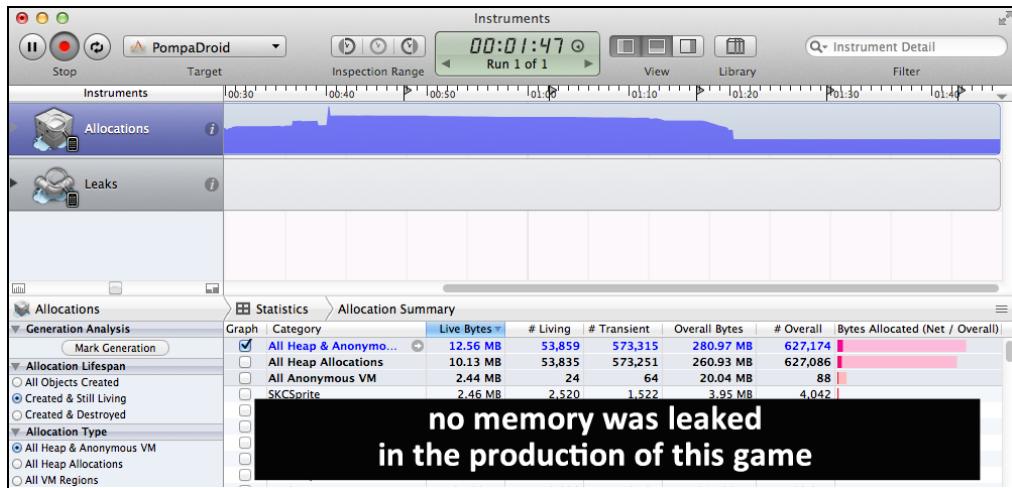
Remove the breakpoints by right-clicking on the blue markers and selecting **Delete Breakpoint**.



Now you can continue to check for any remaining memory issues.

Select **Product\Profile** (or use the Command+I keyboard shortcut) to run Instruments again and select the **Leaks** tool.

Profile your game using the Leaks instrument again and play until you get to the second level.



You'll notice that there are no more leaks and the memory allocation growth isn't as drastic as before. That's what you want to see! 😊

There is one more memory-related issue you have to fix. When an object subscribes to a notification using `NSNotificationCenter`, that same object must unsubscribe from it.

This is because `NSNotificationCenter` keeps an “unsafe unretained” pointer to its registered subscribers. If it ever tries to deliver a notification to a deallocated subscriber, your app will crash.

To avoid this problem, go to **ViewController.m** and add this method:

```
- (void)dealloc
{
    [[NSNotificationCenter defaultCenter]
        removeObserver:self name:@"PresentGame" object:nil];
}
```

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 to your liking. Here are some ideas:

Make it possible for the player to trigger the 1-2-3 punch even if they'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 make the hero run, you have to double-tap quite rapidly and less experienced players might miss this. Increase the time between taps to make running easier.

It currently takes a long time to beat up a robot. Make them easier to kill!

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, you gave your hero 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.

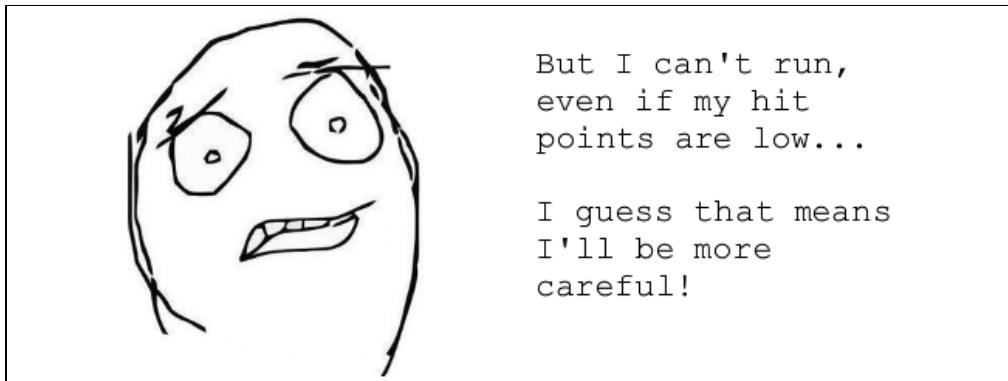
Then, in the second half of the chapter, you'll enhance the robot army by making some robots more powerful than others—and more powerful than any robot presently is. Finally, you'll create a Big Bad Boss enemy that will await the hero in the final battle.

Are you 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.

For one thing, during the game it's important for the player to know how many hit points the hero has left. A player needs to know the odds against the robot hordes!



But I can't run,
even if my hit
points are low...

I guess that means
I'll be more
careful!

You will display the hero's hit points as text, but you don't want to use an ordinary system font. To keep in line with the game's style, you'll use a custom font.

Note: By default, the iPhone can use only system fonts such as Arial and Helvetica. To use other fonts, you have to add the font files to your project, which will allow your game to use those fonts as-is.

First add the font file 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, and 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 **04b03.ttf** in the Fonts folder. This is a custom true-type font that you will use for Pompadroid.

Go to your **Project Navigator** and open **Supporting Files\PompaDroid-Info.plist**. This is the property list that contains various configurations for your app. Since it's a property list, you can use it as you did **Levels.plist** before.

Note: To learn more about Info.plist and what it can do for you, check out chapter 23 in *iOS 6 by Tutorials*, "Secrets of Info.plist."

Add a new row at the bottom of this list and choose **Fonts provided by application**.

▼ Information Property List		Dictionary
Localization native development r...		String
Bundle display name	✚ ✚	String
Executable file		String
Bundle identifier		String
InfoDictionary version		String
Bundle name		String
Bundle OS Type code		String
Bundle versions string, short		String
Bundle creator OS Type code		String
Bundle version		String
Application requires iPhone envir...		Boolean
Main storyboard file base name		String
Main storyboard file base name (i...		String
► Required device capabilities		Array
Status bar is initially hidden		Boolean
► Supported interface orientations		Array
► Supported interface orientations (...)		Array
► Fonts provided by application	✚ ✚	Array

Expand this new entry to see **Item 0**. Enter **04b03.ttf** as the value of Item 0, like so:

▼ Fonts provided by application	Array	(1 item)
Item 0	✚ ✚	String

This tells the application to load 04b03.ttf as a custom font.

Now you can use this font to represent the hero's hit points, which you want to display as part of the HUD. Naturally, then, you will add them to **HudLayer**. But first you need some definitions.

Select the **Controls** group in Xcode, go to **File\New\File...**, choose the **iOS\Cocoa Touch\Objective-C category** template and click **Next**. Enter **UIColor** for Category on, and **BGSK** for Category. Click **Next** and then click **Create**.

Open **UIColor+BGSK.h** and replace its contents with the following:

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

@interface UIColor (BGSK)

+ (UIColor *)fullHPCOLOR;
+ (UIColor *)midHPCOLOR;
+ (UIColor *)lowHPCOLOR;

@end
```

Switch to **UIColor+BGSK.m** and replace its contents with the following:

```
#import "UIColor+BGSK.h"

@implementation UIColor (BGSK)
```

```

+ (UIColor *)fullHPColor
{
    return [UIColor colorWithRed:95/255.0
                           green:1.0
                            blue:106/255.0
                           alpha:1.0];
}

+ (UIColor *)midHPColor
{
    return [UIColor colorWithRed:1.0
                           green:165/255.0
                            blue:0.0
                           alpha:1.0];
}

+(UIColor *)lowHPColor
{
    return [UIColor colorWithRed:1.0
                           green:50/255.0
                            blue:23/255.0
                           alpha:1.0];
}

@end

```

You add a category to the UIColor class. Categories let you add functionality to an already existing class without having to modify the class or make a subclass.

To make a category, you add a parenthesis after the class name in both the header and implementation files. The name in the parenthesis is the name of the category. In this case, you name your category BGSK.

You want to have additional color definitions in UIColor, so you add three methods in the BGSK category: the green fullHPColor, the orange midHPColor and the red lowHPColor. You'll use these to color the labels later.

Open **HudLayer.h** and do the following:

```
//add this method prototype
- (void)setHitPoints:(CGFloat)newHP fromMaxHP:(CGFloat)maxHP;
```

You'll use **setHitPoints:fromMaxHP:** to change the text displayed by the hit points label.

Switch to **HudLayer.m** and do the following:

```
//add to top of file
#import "UIColor+BGSK.h"

//add after last #import and before @implementation
```

```
@interface HudLayer()  
  
@property (strong, nonatomic) SKLabelNode *hitPointsLabel;  
  
@end  
  
//add inside init, right after [self addChild:_buttonA]  
CGFloat xPadding = 10.0 * kPointFactor;  
CGFloat yPadding = 40.0 * kPointFactor;  
_hitPointsLabel = [SKLabelNode labelNodeWithFontNamed:@"04b03"];  
_hitPointsLabel.text = @"0";  
_hitPointsLabel.fontSize = 32 * kPointFactor;  
_hitPointsLabel.horizontalAlignmentMode =  
    SKLabelHorizontalAlignmentModeLeft;  
_hitPointsLabel.position =  
    CGPointMake(xPadding, SCREEN.height - yPadding);  
_hitPointsLabel.colorBlendFactor = 1.0;  
[self addChild:_hitPointsLabel];  
  
//add this method  
- (void)setHitPoints:(CGFloat)newHP fromMaxHP:(CGFloat)maxHP{  
  
    int wholeHP = newHP;  
  
    self.hitPointsLabel.text =  
        [NSString stringWithFormat:@"%d", wholeHP];  
  
    CGFloat ratio = newHP / maxHP;  
  
    if (ratio > 0.6) {  
        self.hitPointsLabel.color = [UIColor fullHPColor];  
    } else if (ratio > 0.2) {  
        self.hitPointsLabel.color = [UIColor midHPColor];  
    } else {  
        self.hitPointsLabel.color = [UIColor lowHPColor];  
    }  
}
```

SKLabelNode displays text using a system or custom font. The property hitPointsLabel will be the label containing the hero's hit points.

You initialize a new SKLabelNode using the font you added to the project, with an initial text of "0" and a font size of 32 points. Then you position it in the top-left corner of the screen.

You left-align the text of the label by changing the value of horizontalAlignment. This also changes the origin coordinates of the label, or which point of the label is anchored at your specified position. No matter how long or short the text gets, its position from the left side of the screen will remain the same.

You also set colorBlendFactor to 1.0 so the label can change colors.

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 color the label red.

This coloring system helps alert the player to the hero's health situation at a glance, without requiring them 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 and then use `stringWithFormat:` to convert the integer into text.

Now go to `GameLayer.m` and do the following:

```
//add inside actionSpriteDidAttack:, right after
[self.hero hurtWithDamage:actionSprite.attackDamage
force:actionSprite.attackForce
direction:CGPointMake(actionSprite.directionX, 0.0)];

[self.hud setHitPoints:self.hero.hitPoints
fromMaxHP:self.hero.maxHitPoints];

//add this inside actionSpriteDidDie:, inside the curly braces of if
(actionSprite == self.hero)

[self.hud setHitPoints:0 fromMaxHP:self.hero.maxHitPoints];

//add this setter 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 scripted or battle event ends, you can tell the player to move right by flashing a simple message.

Open **HudLayer.h** and do the following:

```
//add this method prototype
- (void)showGoMessage;
```

Now switch to **HudLayer.m** and do the following:

```
//add to top of file
#import "SKAction+SKTExtras.h"

//add inside @interface block, below hitPointsLabel
@property (strong, nonatomic) SKLabelNode *goLabel;

//add inside init, after [self addChild:_hitPointsLabel]
_goLabel = [SKLabelNode labelNodeWithFontNamed:@"04b03"];
_goLabel.text = @"[GO]";
_goLabel.fontSize = 32 * kPointFactor;
_goLabel.horizontalAlignmentMode =
    SKLabelHorizontalAlignmentModeRight;
_goLabel.position = CGPointMake(SCREEN.width - xPadding,
                                SCREEN.height - yPadding);
_goLabel.colorBlendFactor = 1.0;
_goLabel.color = [UIColor fullHPColor];
_goLabel.hidden = YES;
[self addChild:_goLabel];

//add this method
- (void)showGoMessage
{
    [self.goLabel removeAllActions];
```

```
[self.goLabel runAction:[SKAction sequence:@[[SKAction
showNode:self.goLabel], [SKAction blinkWithDuration:3.0 blinks:6],
[SKAction hideNode:self.goLabel]]]];
}
```

This creates `goLabel`, which is similar to `hitPointsLabel`. This time, the `horizontalAlignmentMode` is to the right of the label, which you place in the upper-right corner of the screen. You set the label's `hidden` property to YES so 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.

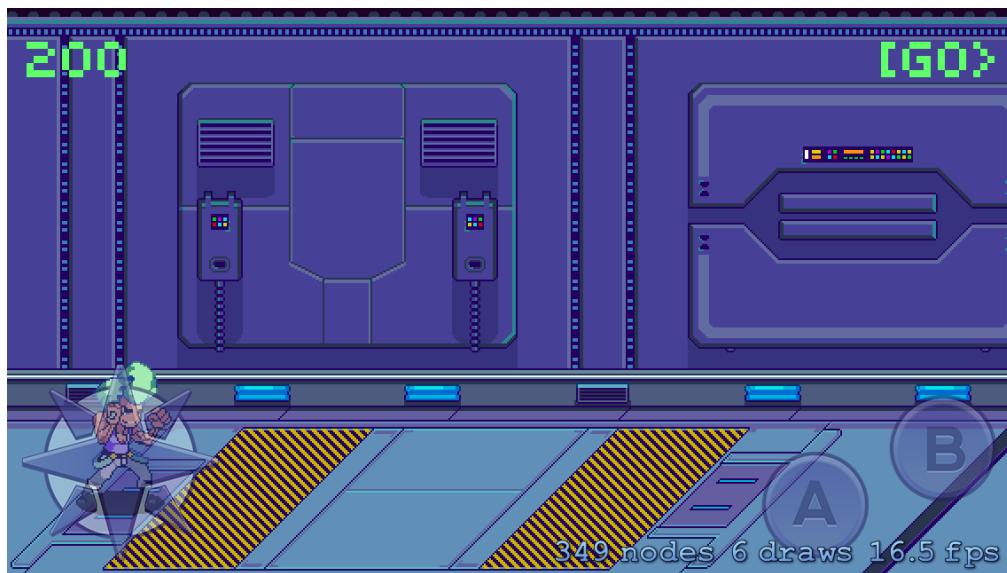
Switch to `GameLayer.m` and make the following changes:

```
//add this setter method
- (void)setEventState:(EventState)eventState
{
    _eventState = eventState;

    if (_eventState == kEventStateFreeWalk) {
        [_hud showGoMessage];
    }
}
```

You change the setter method for `eventState` to ensure the game shows the GO message when the `eventState` becomes `kEventStateFreeWalk`.

Build and run, and your new message should flash in the top-right corner.



To spoil the player with even more information, you can also add a level indicator. In case the player has lost track of their progress, this indicator will show the current level name at the start of every level.



Once again, open **HudLayer.h** and do the following:

```
//add this method prototype
- (void)displayLevel:(NSInteger)level;
```

Switch to **HudLayer.m** and do the following:

```
//add inside @interface block, below goLabel
@property (strong, nonatomic) SKLabelNode *centerLabel;

//add inside init, after [self addChild:_goLabel]
_centerLabel = [SKLabelNode labelNodeWithFontNamed:@"04b03"];
_centerLabel.text = @"LEVEL 1";
_centerLabel.fontSize = 32 * kPointFactor;
_centerLabel.colorBlendFactor = 1.0;
_centerLabel.color = [UIColor midHPColor];
_centerLabel.hidden = YES;
[self addChild:_centerLabel];

//add this method
- (void)displayLevel:(NSInteger)level
{
    self.centerLabel.text =
    [NSString stringWithFormat:@"LEVEL %ld", (long)level];

    [self.centerLabel runAction:[SKAction sequence:@[[SKAction
moveTo:CGPointMake(SCREEN.width + 112 * kPointFactor, CENTER.y)
duration:0], [SKAction showNode:self.centerLabel], [SKAction
moveTo:CENTER duration:0.2], [SKAction waitForDuration:1.0], [SKAction
moveTo:CGPointMake(-112 * kPointFactor, CENTER.y) duration:0.2],
[SKAction hideNode:self.centerLabel]]]];
}
```

The above code creates a new label named **centerLabel**, the text of which **displayLevel** can change to show the name of 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 add the following:

```
//add this line at the beginning of - (void)startGame {...}  
[self.hud displayLevel:self.currentLevel + 1];
```

This makes the game show the name of 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.

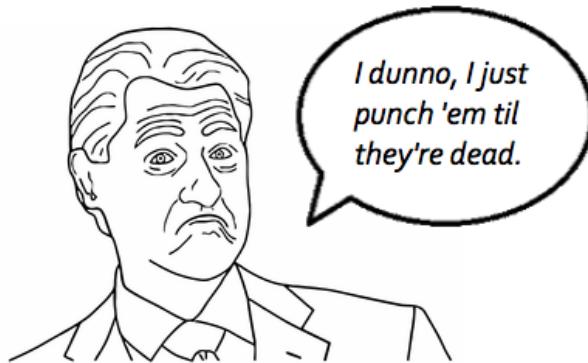


Now your players should feel very well informed. ☺

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 inflicts. 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, as 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\Cocoa Touch\Objective-C class** template and click **Next**. Enter **SKLabelNode** for Subclass of, click **Next** and name the new file **DamageNumber**.

Open **DamageNumber.h** and add the following:

```
//add this property
@property (strong, nonatomic) SKAction *damageAction;

//add these method prototypes
- (void)showWithValue:(NSInteger)value
    fromOrigin:(CGPoint)origin;

- (void)cleanup;
```

This defines a property you'll use to store the action that moves the number a bit after it appears, as well as a helper method to present a damage number when the hero hits a robot.

Switch to **DamageNumber.m** and add the following:

```
//add to top of file
#import "SKAction+SKTEExtras.h"

//add the following methods
- (void)cleanup
{
    self.damageAction = nil;
}

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

```

    self.fontName = @"04b03";
    self.fontSize = 20 * kPointFactor;
    self.colorBlendFactor = 1.0;
    self.color = [UIColor redColor];
    self.damageAction = [SKAction sequence:@[[SKAction showNode:self],
    [SKAction moveBy:CGVectorMake(0.0, 40.0 * kPointFactor) duration:0.6],
    [SKAction hideNode:self]]];

}
return self;
}

- (void)showWithValue:(NSInteger)value
              fromOrigin:(CGPoint)origin
{
    self.text = [NSString stringWithFormat:@"%ld", (long)value];
    self.position = origin;
    [self removeAllActions];
    [self runAction:self.damageAction];
}

```

This creates a subclass of SKLabelNode that does the following:

- It uses the 04b03 font.
- It stores an action named **damageAction**. This action makes the label appear and move up 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**.
- **cleanup** sets **damageAction** to nil. Since you keep a strong reference to the action, it's safer to let this go in case the action also strongly references **DamageNumber**.

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 (strong, nonatomic) NSMutableArray *damageNumbers;
```

Switch to **GameLayer.m** and do the following:

```

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

//add inside the dealloc method
DamageNumber *damageNumber;
for (damageNumber in self.damageNumbers) {
    [damageNumber cleanup];
}

//add to initWithLevel: right after [self initBrains]

```

```
[self initEffects];

//add these methods
- (void)initEffects
{
    self.damageNumbers = [NSMutableArray arrayWithCapacity:20];

    DamageNumber *number;

    for (NSInteger i = 0; i < 20; i++) {
        number = [DamageNumber node];
        number.hidden = YES;
        number.position = OFFSCREEN;
        [self addChild:number];
        [self.damageNumbers addObject:number];
    }
}

- (DamageNumber *)getDamageNumber
{
    DamageNumber *number;
    for (number in self.damageNumbers) {
        if (![number hasActions]) {
            return number;
        }
    }
    return number;
}
```

In `initWithLevel:`, you create 20 `DamageNumbers`, make them invisible, set them somewhere offscreen and store them in an array. `getDamageNumber` retrieves any available `DamageNumber`.

This is similar to how you fetch robots. To speed things up and 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:` (new code is highlighted):

```
- (BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    BOOL didHit = NO;
    if (actionSprite == self.hero) {

        CGPoint attackPosition;
        Robot *robot;
        for (robot in self.robots) {
```

```
if (robot.actionState < kActionStateKnockedOut &&
    robot.actionState != kActionStateNone) {

    if ([self collisionBetweenAttacker:self.hero
        andTarget:robot
        atPosition:&attackPosition]) {

        //add these
        DamageNumber *damageNumber = [self getDamageNumber];

        damageNumber.zPosition =
            MAX(robot.zPosition, self.hero.zPosition) + 1;

        if (self.hero.actionState == kActionStateJumpAttack) {

            [robot knockoutWithDamage:self.hero.jumpAttackDamage
                direction:CGPointMake(self.hero.directionX,
            0)];

            //add this
            [damageNumber showWithValue:self.hero.jumpAttackDamage
            fromOrigin:robot.position];

        } else if (self.hero.actionState ==
            kActionStateRunAttack) {

            [robot knockoutWithDamage:self.hero.runAttackDamage
                direction:CGPointMake(self.hero.directionX,
            0)];

            //add this
            [damageNumber showWithValue:self.hero.runAttackDamage
            fromOrigin:robot.position];

        } else if (self.hero.actionState ==
            kActionStateAttackThree) {

            [robot knockoutWithDamage:self.hero.attackThreeDamage
                direction:CGPointMake(self.hero.directionX,
            0)];

            //add this
            [damageNumber showWithValue:self.hero.attackThreeDamage
            fromOrigin:robot.position];

        } else if (self.hero.actionState ==
            kActionStateAttackTwo) {

            [robot hurtWithDamage:self.hero.attackTwoDamage
                force:self.hero.attackForce
                direction:CGPointMake(self.hero.directionX,
            0.0)];
        }
    }
}
```

```
//add this
[damageNumber showWithValue:self.hero.attackTwoDamage
fromOrigin:robot.position];

} else {

    [robot hurtWithDamage:self.hero.attackDamage
        force:self.hero.attackForce
        direction:CGPointMake(self.hero.directionX,
0.0)];

    //add this
    [damageNumber showWithValue:self.hero.attackDamage
        fromOrigin:robot.position];
}

didHit = YES;
}
}

return didHit;

} else {

if (self.hero.actionState < kActionStateKnockedOut &&
    self.hero.actionState != kActionStateNone) {

    CGPoint attackPosition;
    if ([self collisionBetweenAttacker:actionSprite
        andTarget:self.hero
        atPosition:&attackPosition]) {

        [self.hero hurtWithDamage:actionSprite.attackDamage
            force:actionSprite.attackForce
            direction:CGPointMake(actionSprite.directionX,
0.0)];

        [self.hud setHitPoints:self.hero.hitPoints
            fromMaxHP:self.hero.maxHitPoints];

        didHit = YES;

        //add these
        DamageNumber *damageNumber = [self getDamageNumber];

        damageNumber.zPosition =
            MAX(actionSprite.zPosition, self.hero.zPosition) + 1;

        [damageNumber showWithValue:actionSprite.attackDamage
            fromOrigin:self.hero.position];
    }
}
```

```
        }  
    }  
  
    return didHit;  
}
```

Take note of the comments and highlights in the code above. The highlighted lines after the comments are the ones you need to add.

Whenever an attack connects from either a robot or the hero, the code fetches an unused **DamageNumber**, sets its zPosition to be one point above the character, 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 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, battles can look somewhat choreographed. You want to convey life-or-death scenarios, not dance routines.

Select the **Effects** group, go to **File\New\File**, choose the **iOS\Cocoa Touch\Objective-C class** template and click **Next**. Enter **SKSpriteNode** for Subclass of, click **Next** and name the new file **HitEffect**.

Open **HitEffect.h** and do the following:

```
//add this property
@property (strong, nonatomic) SKAction *effectAction;

//add these method prototypes
- (void)showEffectAtPosition:(CGPoint)position;
- (void)cleanup;
```

Similar to the way you implemented DamageNumber, here you create a property to store the action to display the effect, a helper method to show the effect at a given position and a cleanup method to avoid retain cycles.

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

```
//add to top of file
#import "SKTTextureCache.h"
#import "SKAction+SKTEExtras.h"

//add these methods
- (void)cleanup
{
    self.effectAction = nil;
}

- (instancetype)init
{

SKTexture *texture =
[[SKTTextureCache sharedInstance]
 textureNamed:@"hiteffect_00"];

if (self = [super initWithTexture:texture]) {

    NSMutableArray *textures =
    [NSMutableArray arrayWithCapacity:6];

    for (int i = 0; i < 6; i++) {

        [textures addObject:[[SKTTextureCache sharedInstance]
textureNamed:[NSString stringWithFormat:@"hiteffect_%02d", i]]];
    }

    self.effectAction = [SKAction sequence:@[[SKAction showNode:self],
[SKAction animateWithTextures:textures timePerFrame:1.0/12.0], [SKAction
hideNode:self]]];
}

    return self;
}

- (void)showEffectAtPosition:(CGPoint)position
{
    [self removeAllActions];
```

```
    self.position = position;
    [self runAction:self.effectAction];
}
```

To create a **HitEffect**, you mix some concepts from both **DamageNumber** and **ActionSprite**. Like **ActionSprite**, HitEffect is a subclass of **SKSpriteNode** and is created with an image from the texture atlas you added in Chapter 1.

Then, like **DamageNumber**, it has a stored action, **effectAction**, to make it appear and disappear. But unlike **DamageNumber**, in between appearing and disappearing, **HitEffect** will animate using sprite frames.

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 (strong, nonatomic) NSMutableArray *hitEffects;
```

Switch to **GameLayer.m** and do the following:

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

//add inside dealloc, before the last }
HitEffect *hitEffect;
for (hitEffect in self.hitEffects) {
    [hitEffect cleanup];
}

//add inside initEffects, after the first for loop
self.hitEffects = [NSMutableArray arrayWithCapacity:20];
HitEffect *effect;

for (NSInteger i = 0; i < 20; i++) {
    effect = [HitEffect node];
    effect.hidden = YES;
    [effect setScale:kPointFactor];
    effect.position = OFFSCREEN;
    [self addChild:effect];
    [self.hitEffects addObject:effect];
}

//add this method
- (HitEffect *)getHitEffect
{
    HitEffect *effect;
    for (effect in self.hitEffects) {
        if (![effect hasActions]) {
            return effect;
        }
    }
}
```

```
    }
    return effect;
}
```

This initializes 20 **HitEffects**, scales them to the device, makes them disappear and stores them in an array. You also add them as children of GameLayer.

getHitEffect works the same way as **getDamageNumber**. It retrieves an unused **hitEffect**.

Still in **GameLayer.m**, modify **actionSpriteDidAttack:** as follows (new code is highlighted):

```
- (BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    BOOL didHit = NO;
    if (actionSprite == self.hero) {

        CGPoint attackPosition;
        Robot *robot;
        for (robot in self.robots) {

            if (robot.actionState < kActionStateKnockedOut &&
                robot.actionState != kActionStateNone) {

                if ([self collisionBetweenAttacker:self.hero
                                              andTarget:robot
                                             atPosition:&attackPosition]) {

                    //add this
                    BOOL showEffect = YES;

                    DamageNumber *damageNumber = [self getDamageNumber];

                    damageNumber.zPosition =
                        MAX(robot.zPosition, self.hero.zPosition) + 1;

                    if (self.hero.actionState == kActionStateJumpAttack) {

                        [robot knockoutWithDamage:self.hero.jumpAttackDamage
                                         direction:CGPointMake(self.hero.directionX,
                                         0)];

                        [damageNumber showWithValue:self.hero.jumpAttackDamage
                                         fromOrigin:robot.position];

                    //add this
                    showEffect = NO;

                } else if (self.hero.actionState ==
                           kActionStateRunAttack) {


```

```
[robot knockoutWithDamage:self.hero.runAttackDamage
    direction:CGPointMake(self.hero.directionX,
0)];  
  
[damageNumber showWithValue:self.hero.runAttackDamage
    fromOrigin:robot.position];  
  
} else if (self.hero.actionState ==
    kActionStateAttackThree) {  
  
[robot knockoutWithDamage:self.hero.attackThreeDamage
    direction:CGPointMake(self.hero.directionX,
0)];  
  
[damageNumber showWithValue:self.hero.attackThreeDamage
    fromOrigin:robot.position];  
  
//add this
showEffect = NO;  
  
} else if (self.hero.actionState ==
    kActionStateAttackTwo) {  
  
[robot hurtWithDamage:self.hero.attackTwoDamage
    force:self.hero.attackForce
    direction:CGPointMake(self.hero.directionX,
0.0)];  
  
[damageNumber showWithValue:self.hero.attackTwoDamage
    fromOrigin:robot.position];  
  
} else {  
  
[robot hurtWithDamage:self.hero.attackDamage
    force:self.hero.attackForce
    direction:CGPointMake(self.hero.directionX,
0.0)];  
  
[damageNumber showWithValue:self.hero.attackDamage
    fromOrigin:robot.position];
}  
  
didHit = YES;  
  
//add this if statement and its contents
if (showEffect) {
    HitEffect *hitEffect = [self getHitEffect];
    hitEffect.zPosition = damageNumber.zPosition + 1;
    [hitEffect showEffectAtPosition:attackPosition];
}
}
```

```

    }

    return didHit;
} else {

    if (self.hero.actionState < kActionStateKnockedOut &&
        self.hero.actionState != kActionStateNone) {

        CGPoint attackPosition;
        if ([self collisionBetweenAttacker:actionSprite
                                         andTarget:self.hero
                                         atPosition:&attackPosition]) {

            [self.hero hurtWithDamage:actionSprite.attackDamage
                               force:actionSprite.attackForce
                               direction:CGPointMake(actionSprite.directionX,
                               0.0)];

            [self.hud setHitPoints:self.hero.hitPoints
                           fromMaxHP:self.hero.maxHitPoints];

            didHit = YES;

            DamageNumber *damageNumber = [self getDamageNumber];

            damageNumber.zPosition =
                MAX(actionSprite.zPosition, self.hero.zPosition) + 1;

            [damageNumber showWithValue:actionSprite.attackDamage
                           fromOrigin:self.hero.position];

            //add these
            HitEffect *hitEffect = [self getHitEffect];
            hitEffect.zPosition = damageNumber.zPosition + 1;
            [hitEffect showEffectAtPosition:attackPosition];
        }
    }
}

return didHit;
}

```

Once again, while the whole rewritten method is provided for clarity, only add the highlighted lines after the comments.

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, the silver bot, the 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 = RandomIntRange(100, 250);
self.attackDamage = RandomIntRange(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 fight against the big boss of all those robots. The boss will be a worthy adversary to the Pompadoured Hero—even matching the hero's fancy hairdo with his own hi-top fade!



A class of his own

First, create the **Boss** class. This should be a piece of cake if you remember how you created the **Hero** and **Robot** classes.

Select the **Characters** group, go to **File\New\File**, choose the **iOS\Cocoa Touch\Objective-C class** template and click **Next**. Enter **ActionSprite** for Subclass of, click **Next** and name the new file **Boss**.

Open **Boss.m** and add the following:

```
//add to top of file
#import "SKAction+SKTExtras.h"
#import "SKTTextureCache.h"

//add these methods
- (instancetype)init
{
    if (self = [super initWithTexture:[[SKTTextureCache sharedInstance]
textureNamed:@"boss_idle_00"]])
    {
        self.shadow = [SKSpriteNode spriteNodeWithTexture:[[SKTTextureCache
sharedInstance] textureNamed:@"shadow_character"]];
        self.shadow.alpha = 0.75;

    //idle animation
```

```
SKAction *idleAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"boss_idle" startFrameIdx:0 frameCount:5]
timePerFrame:1.0/10.0];
self.idleAction = [SKAction repeatActionForever:idleAnimation];

//attack animation
SKAction *attackAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"boss_attack" startFrameIdx:0 frameCount:5]
timePerFrame:1.0/8.0];
self.attackAction = [SKAction sequence:@[attackAnimation, [SKAction
performSelector:@selector(idle) onTarget:self]]];

//walk animation
SKAction *walkAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"boss_walk" startFrameIdx:0 frameCount:6]
timePerFrame:1.0/8.0];
self.walkAction = [SKAction repeatActionForever:walkAnimation];

//hurt animation
SKAction *hurtAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"boss_hurt" startFrameIdx:0 frameCount:3]
timePerFrame:1.0/12.0];
self.hurtAction = [SKAction sequence:@[hurtAnimation, [SKAction
performSelector:@selector(idle) onTarget:self]]];

//knocked out animation
self.knockedOutAction = [self animateActionForTextures:[self
texturesWithPrefix:@"boss_knockout" startFrameIdx:0 frameCount:4]
timePerFrame:1.0/12.0];

//die action
self.dieAction = [SKAction blinkWithDuration:2.0 blinks:10.0];

//recover animation
SKAction *recoverAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"boss_setup" startFrameIdx:0 frameCount:6]
timePerFrame:1.0/12.0];

self.recoverAction = [SKAction sequence:@[recoverAnimation,
[SKAction performSelector:@selector(idle) onTarget:self]]];

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.contactPoints = [self contactPointArray:4];
self.attackPoints = [self contactPointArray:1];

[self modifyAttackPointAtIndex:0]
```

```
        offset:CGPointMake(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:CGPointMake(7.0, 36.0)
            radius:23.0];

        [self modifyContactPointAtIndex:1
            offset:CGPointMake(-11.0, 17.0)
            radius:23.5];

        [self modifyContactPointAtIndex:2
            offset:CGPointMake(-2.0, -20.0)
            radius:23.0];

        [self modifyContactPointAtIndex:3
            offset:CGPointMake(24.0, 9.0)
            radius:18.0];
    } else if (actionState == kActionStateWalk) {

        [self modifyContactPointAtIndex:0
            offset:CGPointMake(6.0, 41.0)
            radius:22.0];

        [self modifyContactPointAtIndex:1
            offset:CGPointMake(-5.0, 16.0)
            radius:26.0];

        [self modifyContactPointAtIndex:2
            offset:CGPointMake(1.0, -11.0)
            radius:17.0];

        [self modifyContactPointAtIndex:3
            offset:CGPointMake(-13.0, -25.0)
            radius:10.0];
    } else if (actionState == kActionStateAttack) {

        [self modifyContactPointAtIndex:0
            offset:CGPointMake(20.0, 38.0)
```

```
        radius:22.0];

    [self modifyContactPointAtIndex:1
        offset:CGPointMake(-8.0, 7.0)
        radius:27.3];

    [self modifyContactPointAtIndex:2
        offset:CGPointMake(49.0, 18.0)
        radius:19.0];

    [self modifyContactPointAtIndex:3
        offset:CGPointMake(12.0, -8.0)
        radius:31.0];

    [self modifyAttackPointAtIndex:0
        offset:CGPointMake(65.0, 42.0)
        radius:23.7];
}

- (void)setTexture:(SKTexture *)texture
{
    [super setTexture:texture];

    SKTexture *attackTexture =
    [[SKTextureCache sharedInstance]
     textureNamed:@"boss_attack_01"];

    SKTexture *attackTexture2 =
    [[SKTextureCache sharedInstance]
     textureNamed:@"boss_attack_02"];

    if (texture == attackTexture || texture == attackTexture2) {
        [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 texture atlas 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.

One thing that will make the boss more difficult to defeat 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:(CGFloat)damage
    force:(CGFloat)force
    direction:(CGPoint)direction
{
    if (self.actionState > kActionStateNone &&
        self.actionState < kActionStateKnockedOut) {

        CGFloat ratio = self.hitPoints / self.maxHitPoints;

        if (ratio <= 0.1) {
            [self removeAllActions];
            [self runAction:self.hurtAction];
            self.actionState = kActionStateHurt;
        }

        self.hitPoints -= damage;

        self.desiredPosition =
            CGPointMake(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 NS_ENUM(NSInteger, BossType)
{
    kBossNone = 0,
    kBossMohawk
};

```

Then open `Levels.plist`. 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, which corresponds to **kBossNone**, then it means that level doesn't have a boss. If the value is other than 0, then the 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 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 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 to 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 (strong, nonatomic) Boss *boss;
```

Switch to **GameLayer.m** and do the following:

```
//add these lines at the end of dealloc
if (self.boss) {
    [self.boss cleanup];
}
//add this to loadLevel:, after self.currentLevel = level
BossType boss =
    [[levelData objectForKey:@"BossType"] integerValue];

[self initBossWithType:boss];

//add this method
- (void)initBossWithType:(BossType)type
{
    if (type == kBossMohawk) {

        self.boss = [Boss node];
        self.boss.delegate = self;
        [self addChild:_boss.shadow];
        [_boss.shadow setScale:kPointFactor];
        [self addChild:_boss];
        [_boss setScale:kPointFactor];
        self.boss.hidden = YES;
        self.boss.position = OFFSCREEN;
        self.boss.groundPosition = OFFSCREEN;
        self.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, simply 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 you've positioned the attack and contact circles of the boss enemy, 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 add later, included in debug drawing.

Bringing the boss to life

To finish adding support for the boss 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 `zPosition` of the boss in `reorderActors::`;
5. Implement collision handling in `actionSpriteDidAttack::`; and
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) {

    self.boss.groundPosition = CGPointMake(origin + (offset * (CENTER.x +
    self.boss.centerToSides)), self.boss.centerToBottom +
    self.tileMap.tileSize.height * row * kPointFactor);

    self.boss.position = self.boss.groundPosition;
    self.boss.desiredPosition = self.boss.groundPosition;
    [self.boss idle];
    self.boss.hidden = NO;

}
```

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 [self.hero update:delta]
if (self.boss) {
    [self.boss update:delta];
}

//add this to updatePositions, before the last curly brace
if (self.boss && self.boss.actionState > kActionStateNone) {
```

```

    posY = MIN(floorHeight + (self.boss.centerToBottom -
self.boss.feetCollisionRect.size.height), MAX(self.boss.centerToBottom,
self.boss.desiredPosition.y));

    self.boss.groundPosition = CGPointMake(self.boss.desiredPosition.x,
posY);

    self.boss.position = CGPointMake(self.boss.groundPosition.x,
self.boss.groundPosition.y + self.boss.jumpHeight);

}

//add this to reorderActors, before the last curly brace
if (self.boss) {
    zPosition =
        [self getZFromYPosition:self.boss.shadow.position.y];

    self.boss.zPosition = zPosition;
}

```

These are steps 2, 3, and 4—a triple smash! Everything you did to update the hero’s position, you do here for the boss’s position, but only if the boss exists. There’s nothing else in the above code that you haven’t encountered before.

Next, to implement collision handling, modify `actionSpriteDidAttack:` as follows (new code is highlighted):

```

- (BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    BOOL didHit = NO;
    if (actionSprite == self.hero) {

        CGPoint attackPosition;
        Robot *robot;
        for (robot in self.robots) {

            if (robot.actionState < kActionStateKnockedOut &&
                robot.actionState != kActionStateNone) {

                if ([self collisionBetweenAttacker:self.hero
                    andTarget:robot
                    atPosition:&attackPosition]) {

                    BOOL showEffect = YES;

                    DamageNumber *damageNumber = [self getDamageNumber];

                    damageNumber.zPosition =
                        MAX(robot.zPosition, self.hero.zPosition) + 1;

                    if (self.hero.actionState == kActionStateJumpAttack) {

```

```
[robot knockoutWithDamage:self.hero.jumpAttackDamage
    direction:CGPointMake(self.hero.directionX,
0)];

[damageNumber showWithValue:self.hero.jumpAttackDamage
    fromOrigin:robot.position];

showEffect = NO;

} else if (self.hero.actionState ==
    kActionStateRunAttack) {

[robot knockoutWithDamage:self.hero.runAttackDamage
    direction:CGPointMake(self.hero.directionX,
0)];

[damageNumber showWithValue:self.hero.runAttackDamage
    fromOrigin:robot.position];

} else if (self.hero.actionState ==
    kActionStateAttackThree) {

[robot knockoutWithDamage:self.hero.attackThreeDamage
    direction:CGPointMake(self.hero.directionX,
0)];

[damageNumber showWithValue:self.hero.attackThreeDamage
    fromOrigin:robot.position];

showEffect = NO;

} else if (self.hero.actionState ==
    kActionStateAttackTwo) {

[robot hurtWithDamage:self.hero.attackTwoDamage
    force:self.hero.attackForce
    direction:CGPointMake(self.hero.directionX,
0.0)];

[damageNumber showWithValue:self.hero.attackTwoDamage
    fromOrigin:robot.position];

} else {

[robot hurtWithDamage:self.hero.attackDamage
    force:self.hero.attackForce
    direction:CGPointMake(self.hero.directionX,
0.0)];

[damageNumber showWithValue:self.hero.attackDamage
    fromOrigin:robot.position];
}
```

```
didHit = YES;

if (showEffect) {
    HitEffect *hitEffect = [self getHitEffect];
    hitEffect.zPosition = damageNumber.zPosition + 1;
    [hitEffect showEffectAtPosition:attackPosition];
}
}

//add this new if block before return didHit;
if (self.boss &&
    self.boss.actionState < kActionStateKnockedOut &&
    self.boss.actionState != kActionStateNone) {

    if ([self collisionBetweenAttacker:self.hero
        andTarget:self.boss
        atPosition:&attackPosition]) {

        BOOL showEffect = YES;
        DamageNumber *damageNumber = [self getDamageNumber];
        damageNumber.zPosition = MAX(self.boss.zPosition,
                                      self.hero.zPosition) + 1;

        if (self.hero.actionState == kActionStateJumpAttack) {

            [self.boss hurtWithDamage:self.hero.jumpAttackDamage
            force:self.hero.attackForce direction:CGPointMake(self.hero.directionX, 0)];

            [damageNumber showWithValue:self.hero.jumpAttackDamage
            fromOrigin:self.boss.position];

            showEffect = NO;
        } else if (self.hero.actionState ==
                    kActionStateRunAttack) {

            [self.boss hurtWithDamage:self.hero.runAttackDamage
            force:self.hero.attackForce direction:CGPointMake(self.hero.directionX, 0)];

            [damageNumber showWithValue:self.hero.runAttackDamage
            fromOrigin:self.boss.position];

        } else if (self.hero.actionState ==
                    kActionStateAttackThree) {

            [self.boss hurtWithDamage:self.hero.attackThreeDamage
            force:self.hero.attackForce direction:CGPointMake(self.hero.directionX, 0)];

            [damageNumber showWithValue:self.hero.attackThreeDamage
            fromOrigin:self.boss.position];
        }
    }
}
```

```
        showEffect = NO;

    } else if (self.hero.actionState ==
        kActionStateAttackTwo) {

        [self.boss hurtWithDamage:self.hero.attackTwoDamage
force:self.hero.attackForce/2 direction:CGPointMake(self.hero.directionX, 0.0)];
        [damageNumber showWithValue:self.hero.attackTwoDamage
fromOrigin:self.boss.position];

    } else {

        [self.boss hurtWithDamage:self.hero.attackDamage force:0
direction:CGPointMake(self.hero.directionX, 0.0)];
        [damageNumber showWithValue:self.hero.attackDamage
fromOrigin:self.boss.position];
    }

    didHit = YES;

    if (showEffect) {
        HitEffect *hitEffect = [self getHitEffect];
        hitEffect.zPosition = damageNumber.zPosition + 1;
        [hitEffect showEffectAtPosition:attackPosition];
    }
}

return didHit;
}

//add this else if statement
else if (actionSprite == self.boss) {
    if (self.hero.actionState < kActionStateKnockedOut &&
        self.hero.actionState != kActionStateNone) {

        CGPoint attackPosition;

        if ([self collisionBetweenAttacker:self.boss
            andTarget:self.hero
            atPosition:&attackPosition]) {

            [self.hero knockoutWithDamage:self.boss.attackDamage
                direction:CGPointMake(actionSprite.directionX, 0.0)];

            [self.hud setHitPoints:self.hero.hitPoints
                fromMaxHP:self.hero.maxHitPoints];

            didHit = YES;
        }

        DamageNumber *damageNumber = [self getDamageNumber];
    }
}
```

```
[damageNumber showWithValue:self.boss.attackDamage
    fromOrigin:self.hero.position];

damageNumber.zPosition = MAX(self.boss.zPosition,
    self.hero.zPosition) + 1;

HitEffect *hitEffect = [self getHitEffect];
hitEffect.zPosition = damageNumber.zPosition + 1;
[hitEffect showEffectAtPosition:attackPosition];
}

}

else {

if (self.hero.actionState < kActionStateKnockedOut &&
    self.hero.actionState != kActionStateNone) {

CGPoint attackPosition;
if ([self collisionBetweenAttacker:actionSprite
    andTarget:self.hero
    atPosition:&attackPosition]) {

[self.hero hurtWithDamage:actionSprite.attackDamage
    force:actionSprite.attackForce
    direction:CGPointMake(actionSprite.directionX,
    0.0)];

[self.hud setHitPoints:self.hero.hitPoints
    fromMaxHP:self.hero.maxHitPoints];

didHit = YES;

DamageNumber *damageNumber = [self getDamageNumber];

damageNumber.zPosition =
    MAX(actionSprite.zPosition, self.hero.zPosition) + 1;

[damageNumber showWithValue:actionSprite.attackDamage
    fromOrigin:self.hero.position];

//add these
HitEffect *hitEffect = [self getHitEffect];
hitEffect.zPosition = damageNumber.zPosition + 1;
[hitEffect showEffectAtPosition:attackPosition];
}

}

return didHit;
}
```

The code above includes the whole method, but only the highlighted lines after the 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 =
        [NSMutableArray arrayWithCapacity:self.robots.count + 1];

    ArtificialIntelligence *brain =
        [ArtificialIntelligence aiWithControlledSprite:self.boss
                                                targetSprite:self.hero];

    [self.brains addObject:brain];

    for (Robot *robot in self.robots) {

        brain = [ArtificialIntelligence
                    aiWithControlledSprite:robot
                    targetSprite:self.hero];

        [self.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 an `SKLabelNode` to display the "Go" text that appears to encourage the player 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.

Chapter 8: Final Encounter

Welcome to the last chapter of the Beat 'Em Up Starter Kit! Everything you've done in the previous chapters has led up to this point—the final encounter!

In this chapter, you will add the final pieces of content to the game: the hero's gauntlet weapon and a destructible trashcan. Then, you'll polish to a shine with end game scenarios, music and sound effects, all important contributions to a production-ready game.

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 leader with an impressive haircut. 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 will be similar to an **ActionSprite**, but it will require some special handling to equip the character.

Let's give it a shot—pun intended. First, go to **Defines.h** and add this definition:

```
typedef NS_ENUM(NSInteger, WeaponState)
{
    kWepoanStateNone = 0,
    kWepoanStateEquipped,
    kWepoanStateUnequipped,
    kWepoanStateDestroyed
};
```

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, go to **File\New\File**, choose the **iOS\Cocoa Touch\Objective-C class** template and click **Next**. Enter **SKSpriteNode** for Subclass of, click **Next** and name the new file **Weapon**.

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

```
#import <SpriteKit/SpriteKit.h>
#import "AnimationMember.h"

@class Weapon;

@protocol WeaponDelegate <NSObject>

- (void)weaponDidReachLimit:(Weapon *)weapon;

@end

@interface Weapon : SKSpriteNode

//delegate
@property(nonatomic, weak) id <WeaponDelegate> delegate;

@property (strong, nonatomic) SKSpriteNode *shadow;
@property (strong, nonatomic) AnimationMember *attack;
@property (strong, nonatomic) AnimationMember *attackTwo;
@property (strong, nonatomic) AnimationMember *attackThree;
@property (strong, nonatomic) AnimationMember *idle;
@property (strong, nonatomic) AnimationMember *walk;
@property (strong, nonatomic) SKAction *droppedAction;
@property (strong, nonatomic) SKAction *destroyedAction;
@property (assign, nonatomic) CGFloat damageBonus;
@property (assign, nonatomic) NSInteger limit;
```

```
@property (assign, nonatomic) CGFloat jumpVelocity;
@property (assign, nonatomic) CGPoint velocity;
@property (assign, nonatomic) CGFloat jumpHeight;
@property (assign, nonatomic) CGPoint groundPosition;
@property (assign, nonatomic) WeaponState weaponState;
@property (assign, nonatomic) CGFloat centerToBottom;
@property (assign, nonatomic) CGFloat detectionRadius;

- (AnimationMember *)animationMemberWithPrefix:(NSString *)prefix
startFrameIdx:(NSUInteger)startFrameIdx
frameCount:(NSUInteger)frameCount
timePerFrame:(NSTimeInterval)timePerFrame target:(id)target;

- (void)used;

- (void)pickedUp;

- (void)droppedFrom:(CGFloat)height to:(CGPoint)destination;

- (void)reset;

- (void)update:(NSTimeInterval)delta;

- (void)cleanup;

@end
```

This is a weapon's base structure. Consider the similarities to **ActionSprite**:

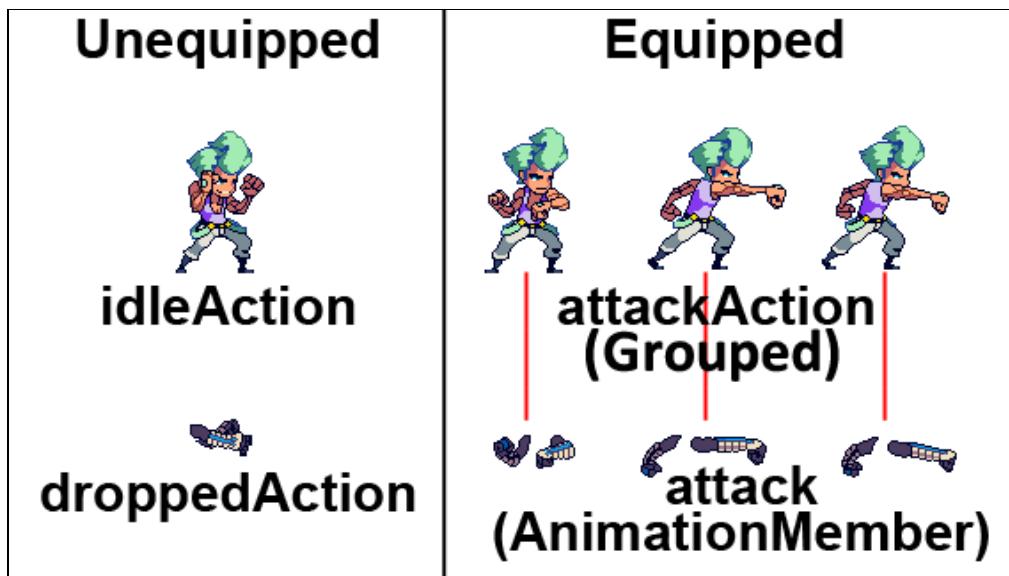
- Weapon is also a subclass of SKSpriteNode because it must display an image.
- It has a delegate: the **GameLayer**.
- Both ActionSprite and Weapon have shadow sprites.
- Both have actions and animations.
- Both have movement values: **jumpVelocity**, **velocity**, **jumpHeight** and **groundPosition**.
- Both have measurements: **centerToBottom** and **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 the hero gets when he's equipped with the weapon.
- Weapon has a **limit**. A weapon can only be used to hit the enemy a given number of times before it breaks.
- Weapon has only two actions it controls:
 - **droppedAction**: The animation for when the hero drops an equipped weapon.
 - **destroyedAction**: What happens when the weapon has reached its limit.

- Weapon has **AnimationMembers**, or animations that it doesn't control. You already encountered **AnimationMember** 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 the following:

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

- (NSMutableArray *)texturesWithPrefix:(NSString *)prefix
    startFrameIdx:(NSUInteger)startFrameIdx
    frameCount:(NSUInteger)frameCount
{
    NSUInteger idxCount = frameCount + startFrameIdx;

    NSMutableArray *textures =
    [NSMutableArray arrayWithCapacity:frameCount];

    SKTexture *texture;

    for (NSUInteger i = startFrameIdx; i < idxCount; i++) {

        texture = [[SKTextureCache sharedInstance] textureNamed:[NSString
stringWithFormat:@"%@_%02lu", prefix, (unsigned long)i]];

        [textures addObject:texture];
    }

    return textures;
}
```

```
- (AnimationMember *)animationMemberWithPrefix:(NSString *)prefix
startFrameIdx:(NSUInteger)startFrameIdx
frameCount:(NSUInteger)frameCount
timePerFrame:(NSTimeInterval)timePerFrame target:(id)target
{
    NSMutableArray *textures =
        [self texturesWithPrefix:prefix
                           startFrameIdx:startFrameIdx
                           frameCount:frameCount];

    AnimationMember *animationMember =
        [AnimationMember animationWithTextures:textures
                           target:target];

    return animationMember;
}
```

These are helper methods for creating AnimationMembers. They are very similar to the helper methods you wrote for ActionSprite.

Still in **Weapon.m**, add these methods:

```
- (void)cleanup
{
    self.droppedAction = nil;
    self.destroyedAction = nil;
    self.attack = nil;
    self.attackTwo = nil;
    self.attackThree = nil;
    self.idle = nil;
    self.walk = nil;
}

- (void)reset
{
    self.hidden = YES;
    self.shadow.hidden = YES;
    self.weaponState = kWeaponStateNone;
    self.velocity = CGPointMakeZero;
    self.jumpVelocity = 0;
}

- (void)setHidden:(BOOL)hidden
{
    [super setHidden:hidden];
    self.shadow.hidden = hidden;
}

- (void)setGroundPosition:(CGPoint)groundPosition
{
    _groundPosition = groundPosition;
```

```
    self.shadow.position = CGPointMake(groundPosition.x, groundPosition.y  
- self.centerToBottom);  
  
}
```

You'll use `cleanup` later to make sure you don't run into memory issues like before. `reset` makes the weapon invisible and changes all relevant values back to their default values. `setHidden:` and `setGroundPosition:` are overridden property methods so that the `shadow` always follows the weapon.

Add this last batch of methods to **Weapon.m**:

```
- (void)used  
{  
    self.limit--;  
  
    if (self.limit <= 0) {  
        [self.delegate weaponDidReachLimit:self];  
        self.weaponState = kWeaponStateDestroyed;  
        [self runAction:self.destroyedAction];  
    }  
}  
  
- (void)pickedUp  
{  
    self.weaponState = kWeaponStateEquipped;  
    self.shadow.hidden = YES;  
}  
  
- (void)droppedFrom:(CGFloat)height to:(CGPoint)destination  
{  
    self.jumpVelocity = kJumpCutoff;  
    self.jumpHeight = height;  
    self.groundPosition = destination;  
    self.weaponState = kWeaponStateUnequipped;  
    self.shadow.hidden = NO;  
    [self runAction:self.droppedAction];  
}  
  
- (void)update:(NSTimeInterval)delta  
{  
  
    if (self.weaponState > kWeaponStateEquipped) {  
  
        self.groundPosition =  
            CGPointMakeAdd(self.groundPosition,  
                           CGPointMakeMultiplyScalar(self.velocity, delta));  
  
        self.jumpVelocity -= kGravity * delta;  
        self.jumpHeight += self.jumpVelocity * delta;  
  
        if (self.jumpHeight < 0) {  
            self.velocity = CGPointMakeZero;  
        }  
    }  
}
```

```
        self.jumpVelocity = 0;
        self.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 the spot it should land. You execute **droppedAction** so the weapon changes its image back to the default image and the **shadow** becomes visible again.
- **update**: When the weapon is unequipped, it falls back to the ground. The velocity, jump and position calculations are the same as in **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\Cocoa Touch\Objective-C class** template and click **Next**. Enter **Weapon** for Subclass of, click **Next** and name the new file **Gauntlets**.

Open **Gauntlets.m** and add the following:

```
//add to top of file
#import "SKTextureCache.h"
#import "SKAction+SKTExtras.h"

//add these methods
- (instancetype)init
{
    if (self = [super initWithTexture:[[SKTextureCache sharedInstance]
textureNamed:@"weapon_unequipped"]])
    {
        self.attack = [self animationMemberWithPrefix:@"weapon_attack_00"
startFrameIdx:0 frameCount:3 timePerFrame:1.0/15.0 target:self];

        self.attackTwo = [self animationMemberWithPrefix:@"weapon_attack_01"
startFrameIdx:0 frameCount:3 timePerFrame:1.0/12.0 target:self];
    }
}
```

```
    self.attackThree = [self
animationMemberWithPrefix:@"weapon_attack_02" startFrameIdx:0
frameCount:5 timePerFrame:1.0/10.0 target:self];

    self.idle = [self animationMemberWithPrefix:@"weapon_idle"
startFrameIdx:0 frameCount:6 timePerFrame:1.0/12.0 target:self];

    self.walk = [self animationMemberWithPrefix:@"weapon_walk"
startFrameIdx:0 frameCount:8 timePerFrame:1.0/12.0 target:self];

    SKTexture *dropTexture = [[SKTextureCache sharedInstance]
textureNamed:@"weapon_unequipped"];
    self.droppedAction = [SKAction setTexture:dropTexture];

    self.destroyedAction = [SKAction sequence:@[[SKAction
blinkWithDuration:2.0 blinks:5], [SKAction
performSelector:@selector(reset) onTarget:self]]];

    self.damageBonus = 20.0;
    self.centerToBottom = 5.0 * kPointFactor;

    self.shadow = [SKSpriteNode spriteNodeWithTexture:[[SKTextureCache
sharedInstance] textureNamed:@"shadow_weapon"]];

    self.shadow.alpha = 0.75;
    self.detectionRadius = 10.0 * kPointFactor;

    [self reset];
}

return self;
}

-(void)reset
{
    [super reset];
    self.limit = 20;
}
```

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. If the hero gets hit by an enemy, he'll automatically drop the gauntlets.

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 a value for its detection radius. You'll use this radius 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.

Before going any further, give the gauntlets a test in their unequipped state.

Go to **GameLayer.h** and add this property:

```
@property (strong, nonatomic) NSMutableArray *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
{
    self.weapons = [NSMutableArray arrayWithCapacity:3];

    Weapon *weapon;

    for (NSInteger i = 0; i < 3; i++) {

        weapon = [Gauntlets node];
        [weapon.shadow setScale:kPointFactor];
        [weapon setScale:kPointFactor];
        weapon.hidden = NO;
        weapon.weaponState = kWeaponStateUnequipped;

        CGFloat maxX = self.tileMap.mapSize.width *
            self.tileMap.tileSize.width * kPointFactor;

        CGFloat minY = weapon.centerToBottom;

        CGFloat maxY =
            3 * self.tileMap.tileSize.height *
            kPointFactor + weapon.centerToBottom;

        weapon.groundPosition =
            CGPointMake(RandomFloatRange(0, maxX),
                       RandomFloatRange(minY, maxY));

        [self addChild:weapon.shadow];
        [self addChild:weapon];
        [self.weapons addObject:weapon];
    }
}

//add to the end of dealloc method
Weapon *weapon;
for (weapon in self.weapons) {
```

```
[weapon cleanup];
}

//add this to update:, after [_hero update:delta]
Weapon *weapon;
for (weapon in self.weapons) {
    [weapon update:delta];
}

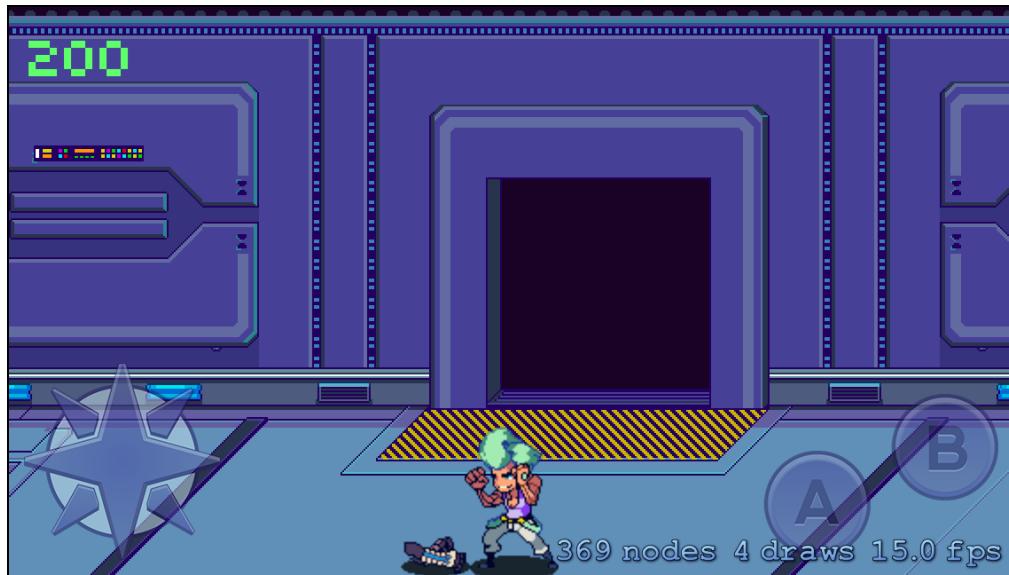
//add this to updatePositions, before the last curly brace
Weapon *weapon;
for (weapon in self.weapons) {
    if (weapon.weaponState > kWeaponStateEquipped) {
        weapon.position =
            CGPointMake(weapon.groundPosition.x,
                        weapon.groundPosition.y + weapon.jumpHeight);
    }
}

//add this to reorderActors, before the last curly brace

Weapon *weapon;
for (weapon in self.weapons) {
    if (weapon.weaponState != kWeaponStateEquipped) {
        zPosition =
            [self getZFromYPosition:weapon.shadow.position.y];
        weapon.zPosition = zPosition;
    }
}
```

The above creates three gauntlets and positions them randomly across the map. When a character drops a weapon, you update its position and `zPosition` 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, you need to fit the hero with weapon-wielding capabilities.

Equipping the hero

I bet you're eager to put those gauntlets in the hero's hands so 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"

//add the protocol to the @interface statement
@interface Hero : ActionSprite <WeaponDelegate>

//add this property
@property (weak, nonatomic) Weapon *weapon;

//add these methods
- (void)dropWeapon;
- (BOOL)pickUpWeapon:(Weapon *)weapon;
```

This makes the hero a delegate of the `Weapon` class, meaning the hero can now receive messages from weapon objects. The code also adds a weak reference to a weapon and some methods to drop and pick up weapons.

Now switch to **Hero.m** and add the following:

```
//add these properties inside @interface Hero()
@property (strong, nonatomic) NSMutableArray *attackGroup;
@property (strong, nonatomic) NSMutableArray *attackTwoGroup;
@property (strong, nonatomic) NSMutableArray *attackThreeGroup;
@property (strong, nonatomic) NSMutableArray *idleGroup;
```

```
@property (strong, nonatomic) NSMutableArray *walkGroup;
```

This creates five NSMutableArrays for the hero. These will act as groups of AnimationMembers for the attack, idle and walk actions.

Still in **Hero.m**, do the following inside **init**:

```
//replace these lines:  
NSMutableArray *textures =  
    [self texturesWithPrefix:@"hero_idle"  
        startFrameIdx:0 frameCount:6];  
  
SKAction *idleAnimation =  
    [SKAction animateWithTextures:textures  
        timePerFrame:1.0/12.0];  
  
//with these lines:  
_idleGroup = [NSMutableArray arrayWithCapacity:2];  
AnimationMember *idleMember = [AnimationMember  
    animationWithTextures:[self texturesWithPrefix:@"hero_idle"  
        startFrameIdx:0 frameCount:6] target:self];  
  
[_idleGroup addObject:idleMember];  
  
SKAction *idleAnimation = [self animateActionForGroup:_idleGroup  
    timePerFrame:1.0/12.0 frameCount:6];  
  
//replace these lines:  
textures =  
    [self texturesWithPrefix:@"hero_walk"  
        startFrameIdx:0 frameCount:8];  
  
SKAction *walkAnimation =  
    [SKAction animateWithTextures:textures  
        timePerFrame:1.0/12.0];  
  
//with these lines:  
_walkGroup = [NSMutableArray arrayWithCapacity:2];  
AnimationMember *walkMember = [AnimationMember  
    animationWithTextures:[self texturesWithPrefix:@"hero_walk"  
        startFrameIdx:0 frameCount:8] target:self];  
  
[_walkGroup addObject:walkMember];  
  
SKAction *walkAnimation = [self animateActionForGroup:_walkGroup  
    timePerFrame:1.0/12.0 frameCount:8];  
  
//replace this line:  
textures = [self texturesWithPrefix:@"hero_run"  
        startFrameIdx:0 frameCount:8];  
  
//with this line
```

```
NSArray * textures = [self texturesWithPrefix:@"hero_run"
                                         startFrameIdx:0 frameCount:8];

//replace these lines:
textures = [self texturesWithPrefix:@"hero_attack_00"
                                         startFrameIdx:0
                                         frameCount:3];

SKAction *attackAnimation =
    [self animateActionForTextures:textures
        timePerFrame:1.0/15.0];

//with these lines:
_attackGroup = [NSMutableArray arrayWithCapacity:2];
AnimationMember *attackMember = [AnimationMember
animationWithTextures:[self texturesWithPrefix:@"hero_attack_00"
startFrameIdx:0 frameCount:3] target:self];

[_attackGroup addObject:attackMember];

SKAction *attackAnimation = [self animateActionForGroup:_attackGroup
timePerFrame:1.0/15.0 frameCount:3];

//replace this line:
SKAction *attackTwoAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"hero_attack_01" startFrameIdx:0 frameCount:3]
timePerFrame:1.0/12.0];

//with these lines:
_attackTwoGroup = [NSMutableArray arrayWithCapacity:2];
AnimationMember *attackTwoMember = [AnimationMember
animationWithTextures:[self texturesWithPrefix:@"hero_attack_01"
startFrameIdx:0 frameCount:3] target:self];

[_attackTwoGroup addObject:attackTwoMember];

SKAction *attackTwoAnimation = [self
animateActionForGroup:_attackTwoGroup timePerFrame:1.0/12.0
frameCount:3];

//replace this line:
SKAction *attackThreeAnimation = [self animateActionForTextures:[self
texturesWithPrefix:@"hero_attack_02" startFrameIdx:0 frameCount:5]
timePerFrame:1.0/10.0];

//with these lines:
_attackThreeGroup = [NSMutableArray arrayWithCapacity:2];
AnimationMember *attackThreeMember = [AnimationMember
animationWithTextures:[self texturesWithPrefix:@"hero_attack_02"
startFrameIdx:0 frameCount:5] target:self];

[_attackThreeGroup addObject:attackThreeMember];
```

```
SKAction *attackThreeAnimation = [self  
animateActionForGroup:_attackThreeGroup timePerFrame:1.0/10.0  
frameCount:5];
```

You change the way you create the animations for the idle, walk and attack actions. Now, you first create a group for each action and then add the animation actions as members of these groups. This way, you can add and remove AnimationMembers to the groups as you please. The member animation you will add is the weapon's **AnimationMember** when a weapon is equipped.

Next, implement the two new methods in **Hero.m**:

```
- (BOOL)pickUpWeapon:(Weapon *)weapon  
{  
    if (self.actionState == kActionStateIdle) {  
  
        [self removeAllActions];  
        [weapon pickedUp];  
        [self setTexture:[[SKTextureCache sharedInstance]  
                           textureNamed:@"hero_jump_05"]];  
  
        [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 hero_jump_05 texture (because this image makes him crouch). After 1/12th of a second, the method calls **setWeapon**. 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.

You calculate the position for **droppedFrom:** 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 group actions.

To do that, add these methods to **Hero.m**:

```
- (void)removeWeaponAnimationMembers
{
    [_attackGroup removeObject:_weapon.attack];
    [_attackTwoGroup removeObject:_weapon.attackTwo];
    [_attackThreeGroup removeObject:_weapon.attackThree];
    [_idleGroup removeObject:_weapon.idle];
    [_walkGroup removeObject:_weapon.walk];
}

- (void)setWeapon:(Weapon *)weapon
{
    // 1
    if (_weapon) {
        [self removeWeaponAnimationMembers];
    }

    _weapon = weapon;

    // 2
    if (_weapon) {
        _weapon.delegate = self;
        _weapon.xScale = self.xScale;
        [self.attackGroup addObject:_weapon.attack];
        [self.attackTwoGroup addObject:_weapon.attackTwo];
        [self.attackThreeGroup addObject:_weapon.attackThree];
        [self.idleGroup addObject:_weapon.idle];
        [self.walkGroup addObject:_weapon.walk];
    }

    // 3
    self.velocity = CGPointMakeZero;
}
```

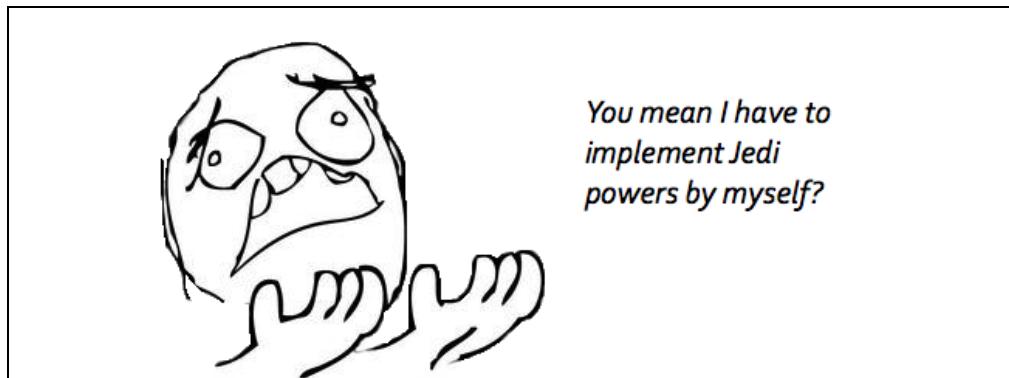
```
self.actionDelay = 0.0;

// 4
if (self.actionState == kActionStateIdle) {
    [self runAction:self.idleAction];
}
```

When you assign the hero 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 group 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 stops moving and resets his action delay.
4. If the hero was idle prior to setting a weapon, he returns to the idle state.

You're not finished with the hero yet! For now, though, it would be good to test if the hero can pick up a weapon. To test, you need to write the collision code between the hero and the weapon, such that the hero only picks up a weapon if it's 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
    CGFloat planeDist =
        player.shadow.position.y - weapon.shadow.position.y;

    if (fabsf(planeDist) <= kPlaneHeight) {
```

```

CGFloat combinedRadius =
    player.detectionRadius + weapon.detectionRadius;

NSInteger i;

// 2: initial detection
if (CGPointDistanceSQ(player.position, weapon.position) <=
    combinedRadius * combinedRadius) {

    NSInteger contactPointCount = player.contactPoints.count;
    ContactPoint contactPoint;

    // 3: secondary detection
    for (i = 0; i < contactPointCount; i++) {

        NSValue *value = player.contactPoints[i];
        [value getValue:&contactPoint];

        combinedRadius =
            contactPoint.radius + weapon.detectionRadius;

        if (CGPointDistanceSQ(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. You check if the two objects are on the same plane using `kPlaneHeight`.
2. You 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 simply check if any of the hero's contact circles intersect with the weapon's detection circle.

If the hero and the weapon pass all three tests, it means the hero is standing directly above the weapon and will be able to pick it up.

Still in `GameLayer.m`, modify `actionButtonWasPressed:` as follows (new code is highlighted):

```

- (void)actionButtonWasPressed:(ActionButton *)actionButton
{
    if (self.eventState == kEventStateScripted) return;
}

```

```
if ([actionButton.name isEqualToString:@"ButtonA"]) {  
    //replace the contents of this if statement  
    BOOL pickedUpWeapon = NO;  
  
    if (!self.hero.weapon) {  
  
        //check collision for all weapons  
        for (Weapon *weapon in self.weapons) {  
  
            if (weapon.weaponState == kWeaponStateUnequipped) {  
  
                if ([self collisionBetweenPlayer:self.hero  
                    andWeapon:weapon]) {  
  
                    pickedUpWeapon = [self.hero pickUpWeapon:weapon];  
                    weapon.zPosition = self.hero.zPosition + 1;  
                    break;  
                }  
            }  
        }  
  
        if (!pickedUpWeapon) {  
            [self.hero attack];  
        }  
    }  
}  
  
} else if ([actionButton.name isEqualToString:@"ButtonB"]) {  
  
    //replace the contents of this else if statement  
    if (self.hero.weapon) {  
        [self.hero dropWeapon];  
    } else {  
        CGPoint directionVector =  
            [self vectorForDirection:self.hud.dPad.direction];  
        [self.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 that 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.

Note: You'll notice something very strange happening with the scale of the gauntlet's texture whenever it animates. This is due to a bug in Sprite Kit, which I will discuss later. Just keep this in mind for now as it will be the last issue you tackle.

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 (self.weapon) {
        self.weapon.position = position;
    }
}

- (void)setXScale:(CGFloat)xScale
{
    [super setXScale:xScale];

    if (self.weapon) {
```

```
        self.weapon.xScale = xScale;
    }

- (void)setYScale:(CGFloat)yScale
{
    [super setYScale:yScale];

    if (self.weapon) {
        self.weapon.yScale = yScale;
    }
}

- (void)setScale:(CGFloat)scale
{
    [super setScale:scale];

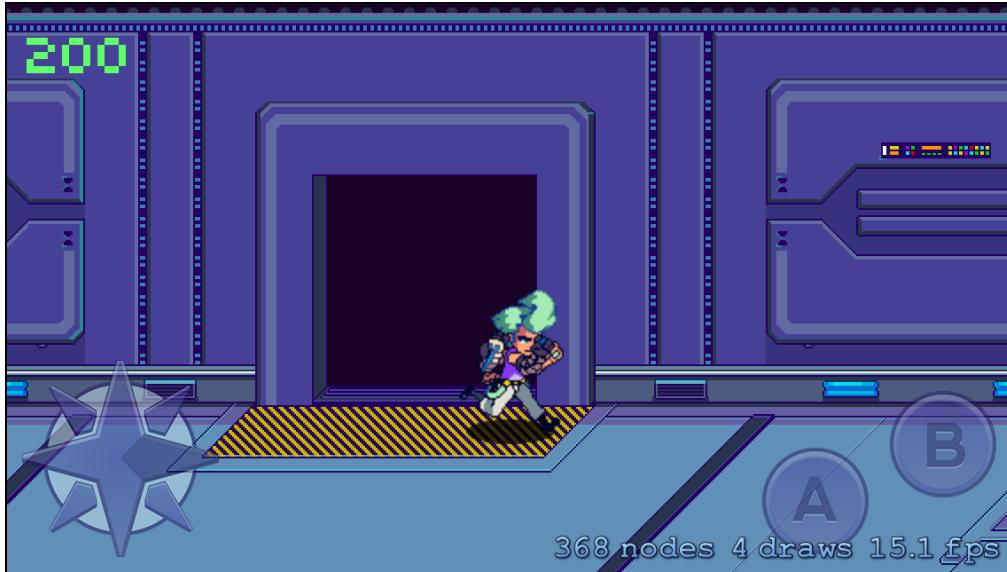
    if (self.weapon) {
        self.weapon.scale = scale;
    }
}

- (void)setZPosition:(CGFloat)zPosition
{
    [super setZPosition:zPosition];

    if (self.weapon) {
        self.weapon.zPosition = zPosition;
    }
}
```

Now when the hero has a weapon equipped, the weapon follows the hero's position values. 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:

- 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.
- When the hero gets hurt, the gauntlets don't follow the animation.
- When the hero is idle, the gauntlets don't follow his hands exactly.

It's tinkering time! Do the following in **GameLayer.m** (new code is highlighted):

```
//add “!self.hero.weapon &&” in the inner if statement of
actionDPad:didChangeDirectionTo:

- (void)actionDPad:(ActionDPad *)actionDPad
didChangeDirectionTo:(ActionDPadDirection)direction
{
    if (self.eventState == kEventStateScripted) return;

    CGPoint directionVector =
    [self vectorForDirection:direction];

    if (!self.hero.weapon && self.runDelay > 0 &&
        self.previousDirection == direction &&
        (direction == kActionDPadDirectionRight ||
        direction == kActionDPadDirectionLeft)) {

        [self.hero runWithDirection:directionVector];
    } else if (self.hero.actionState == kActionStateRun &&
               abs(self.previousDirection - direction) <= 1) {

        [self.hero moveWithDirection:directionVector];
    }
}
```

```

} else {

    [self.hero walkWithDirection:directionVector];
    self.previousDirection = direction;
    self.runDelay = 0.2;
}
}

```

Then in **Hero.m**, do the following:

```

//add this at the beginning (after the call to super) of
hurtWithDamage:force:direction:
if (self.weapon) {
    [self dropWeapon];
}

//add this at the beginning (after the call to super) of
knockoutWithDamage:direction:
if (self.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.

Build and run to check your work. Try to punch the air and observe what happens to the gauntlet sprite.



Right after the punch animation, the gauntlet's idle animation gets stretched for a split second. What's more, the gauntlet's idle animation still doesn't seem to be in sync with the hero.

As of writing, a bug in Sprite Kit is responsible for both of these issues. When an **SKSpriteNode** animates through a bunch of differently-sized images, it can make

mistakes in calculating the correct size of the texture to which it's set. This issue is more apparent when you use Xcode's automatic texture atlas generation feature (using `.atlas` instead of `.atlasc`).

I'm sure Apple will resolve this issue soon, so for now you'll implement a workaround.

Add this method to both **ActionSprite.m** and **Weapon.m**:

```
- (void)setTexture:(SKTexture *)texture
{
    // 1
    CGFloat xScale = self.xScale;
    CGFloat yScale = self.yScale;

    // 2
    self.xScale = 1.0;
    self.yScale = 1.0;

    [super setTexture:texture];

    // 3
    self.size = texture.size;

    //restore the previous xScale
    self.xScale = xScale;
    self.yScale = yScale;
}
```

As you know, when the sprites execute the custom animation action that you created a few chapters back, they repeatedly call `setTexture` to change their displayed texture.

In the code above, you replace the default `setTexture` method and add pre-steps and post-steps to it:

1. Before replacing the texture, you store the current scale values of the sprite.
2. You reset the scale values to their default values.
3. After replacing the texture, you make sure that the sprite's size is indeed the texture's size. Whenever you change the size of a sprite manually, it takes the current scale into account. This is why you had to reset the scale first before messing with the sprite's size.
4. You restore the previous scale values of the sprite.

That should fix it. 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:

```
- (CGFloat)attackDamage
{
    if (self.weapon) {
        return [super attackDamage] + self.weapon.damageBonus;
    }

    return [super attackDamage];
}

- (CGFloat)attackTwoDamage
{
    if (self.weapon) {
        return _attackTwoDamage + self.weapon.damageBonus;
    }
}
```

```
    return _attackTwoDamage;
}

- (CGFloat)attackThreeDamage
{
    if (self.weapon) {
        return _attackThreeDamage + self.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 (new code is highlighted):

```
//modify setTexture: (new code is highlighted)
- (void)setTexture:(SKTexture *)texture
{
    [super setTexture:texture];

    SKTexture *attackTexture =
    [[SKTextureCache sharedInstance]
     textureNamed:@"hero_attack_00_01"];

    SKTexture *runAttackTexture =
    [[SKTextureCache sharedInstance]
     textureNamed:@"hero_runattack_02"];

    SKTexture *runAttackTexture2 =
    [[SKTextureCache sharedInstance]
     textureNamed:@"hero_runattack_03"];

    SKTexture *jumpAttackTexture =
    [[SKTextureCache sharedInstance]
     textureNamed:@"hero_jumpattack_02"];

    //add these new textures
    SKTexture *attackTexture2 =
    [[SKTextureCache sharedInstance]
     textureNamed:@"hero_attack_01_01"];

    SKTexture *attackTexture3 =
    [[SKTextureCache sharedInstance]
     textureNamed:@"hero_attack_02_02"];

    if (texture == attackTexture || texture == attackTexture2) {
```

```
if ([self.delegate actionSpriteDidAttack:self]) {
    self.chainTimer = 0.3;

    //add this if statement
    if (self.weapon) {
        [self.weapon used];
    }
}

} else if (texture == attackTexture3) {

    //replace the contents of this else if statement
    if ([self.delegate actionSpriteDidAttack:self]) {
        if (self.weapon) {
            [self.weapon used];
        }
    }
}

} else if (texture == runAttackTexture ||
    texture == runAttackTexture2 ||
    texture == jumpAttackTexture) {

    [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 drops the weapon.

That’s it. Build, run, and finally have fun smashing 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. When designing a level, there may be times you want to have more control over where your weapons appear.

Here's an idea: why not add some new objects to the map that will 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.

First, go to **Defines.h** and add this:

```
typedef NS_ENUM(NSInteger, ObjectState)
{
    kObjectStateActive,
    kObjectStateDestroyed
};
```

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\Cocoa Touch\Objective-C class** template and click **Next**. Enter **SKSpriteNode** for Subclass of, click **Next** and name the new file **MapObject**.

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

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

@interface MapObject : SKSpriteNode

@property (assign, nonatomic) CGFloat detectionRadius;
@property (assign, nonatomic) ContactPoint *contactPoints;
@property (assign, nonatomic) NSInteger contactPointCount;
@property (assign, nonatomic) ObjectState objectState;

- (CGRect)collisionRect;

- (void)modifyContactPointAtIndex:(const NSUInteger)pointIndex
                           offset:(const CGPoint)offset
                           radius:(const CGFloat)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. Notice that instead of using an array of NSValues that wrap around ContactPoint structs, you're going to use the structs directly.
- **modifyContactPointAtIndex**: A method that helps adjust these points.
- **objectState**: The state of the object.
- **collisionRect**: You'll use this to detect collisions.
- **destroyed**: This will control what happens when the object is destroyed.

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

```
- (void)dealloc
{
    free(self.contactPoints);
}

- (void)destroyed
{
    self.objectState = kObjectStateDestroyed;
}

- (CGRect)collisionRect
{
    return CGRectZero;
```

```

}

- (void)setPosition:(CGPoint)position
{
    [super setPosition:position];
    [self transformPoints];
}

- (void)transformPoints
{
    for (NSInteger i = 0; i < _contactPointCount; i++) {
        CGPoint point = CGPointMake(self.contactPoints[i].offset.x,
                                    self.contactPoints[i].offset.y);

        self.contactPoints[i].position =
            CGPointMakeAdd(self.position, point);
    }
}

- (void)modifyContactPointAtIndex:(const NSUInteger)pointIndex
                           offset:(const CGPoint)offset
                           radius:(const CGFloat)radius
{
    ContactPoint *contactPoint = &self.contactPoints[pointIndex];
    [self modifyPoint:contactPoint offset:offset radius:radius];
}

- (void)modifyPoint:(ContactPoint *)point
               offset:(const CGPoint)offset
               radius:(const CGFloat)radius
{
    point->offset = CGPointMakeMultiplyScalar(offset, kPointFactor);
    point->radius = radius * kPointFactor;
    point->position = CGPointMakeAdd(self.position, point->offset);
}

```

Most of these methods have counterparts in **ActionSprite**, except for:

- **destroyed**: This simply changes the **objectState** to **kObjectStateDestroyed**.
- **collisionRect**: This returns an empty rectangle. The **collisionRect** should be specific per object, so you'll have to implement this in every **MapObject** subclass.

Your 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\Cocoa Touch \Objective-C class** template and click **Next**. Enter **MapObject** for Subclass of, click **Next** and name the new file **TrashCan**.

Go to **TrashCan.m** and do the following:

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

//add these methods
- (instancetype)init
{
    SKTexture *texture =
        [[SKTTextureCache sharedInstance] textureNamed:@"trashcan"];

    if (self = [super initWithTexture:texture]) {

        self.objectState = kObjectStateActive;
        self.detectionRadius = 57.0 * kPointFactor;
        self.contactPointCount = 5;

        self.contactPoints =
            malloc(sizeof(ContactPoint) * self.contactPointCount);

        [self modifyContactPointAtIndex:0
                               offset:CGPointMake(0.0, 2.0)
                               radius:33.0];

        [self modifyContactPointAtIndex:1
                               offset:CGPointMake(0.0, -15.0)
                               radius:33.0];

        [self modifyContactPointAtIndex:2
                               offset:CGPointMake(0.0, 26.0)
                               radius:17.0];

        [self modifyContactPointAtIndex:3
                               offset:CGPointMake(19.0, 29.0)
                               radius:16.0];

        [self modifyContactPointAtIndex:4
                               offset:CGPointMake(-23.0, -38.0)
                               radius:10.0];
    }
    return self;
}

- (void)destroyed
{
    [self setTexture:[[SKTTextureCache sharedInstance]
                     textureNamed:@"trashcan_hit"]];

    [super destroyed];
}

- (CGRect)collisionRect
{
    return CGRectMake(self.position.x - self.size.width/2,
                      self.position.y - self.size.height/2,
```

```
    64 * kPointFactor, 32 * kPointFactor);  
}
```

In `init`, you create an `SKSpriteNode` using the `trashcan.png` image found in the texture atlas. 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 `size` 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`.

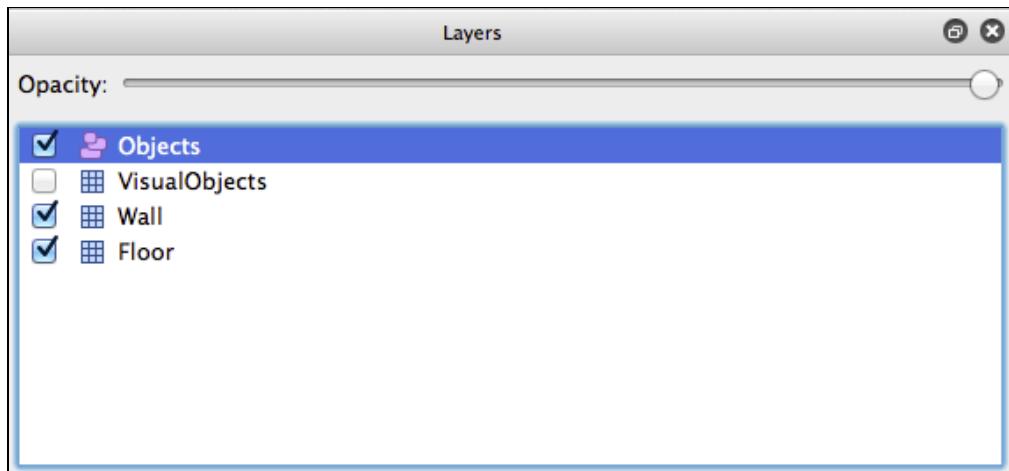
Now that you've finished the `TrashCan` class, you can put some trashcans on the map. You could define the position of each map object in `Levels.plist`, just as you placed the enemies and battle events on the map.

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!

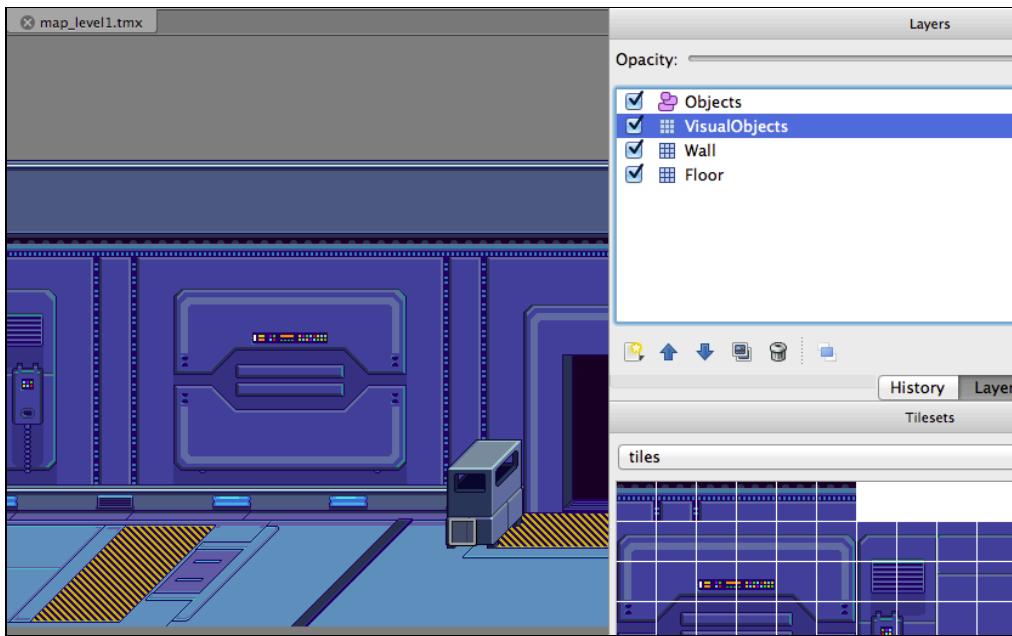
Lucky for you, the tile map you've been using, `map_level1.tmx`, already has the trashcans positioned properly. To see exactly how it was done, take a look at the file.

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** that has an un-ticked checkbox.



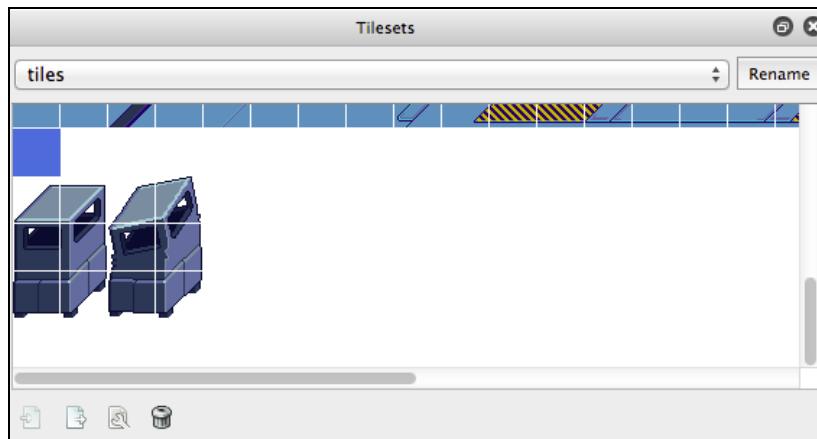
Check the **VisualObjects** checkbox. Once you've done this, take a look at the tile map to the left and you will see some trashcans appear.



If you leave the **VisualObjects** layer visible and save the file, your game will have trashcans in it. However, there's a reason I didn't set the **VisualObjects** layer 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.

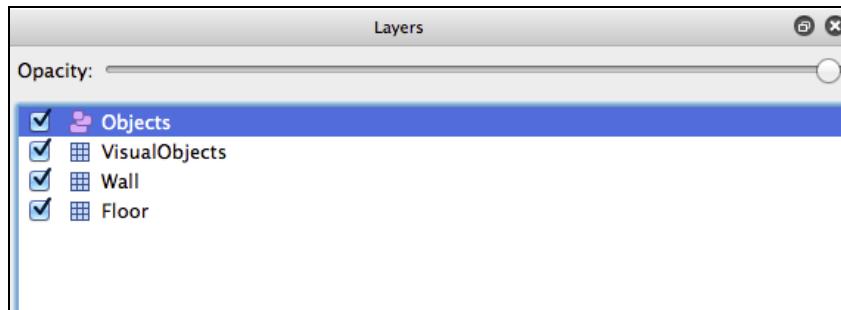


You need a trashcan to be treated as single object, a **TrashCan** object, and it needs to animate as one object as well. If your trashcan were made up of six tiles, you'd need to replace all six tiles just to animate it by one frame.

A better way is to 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.

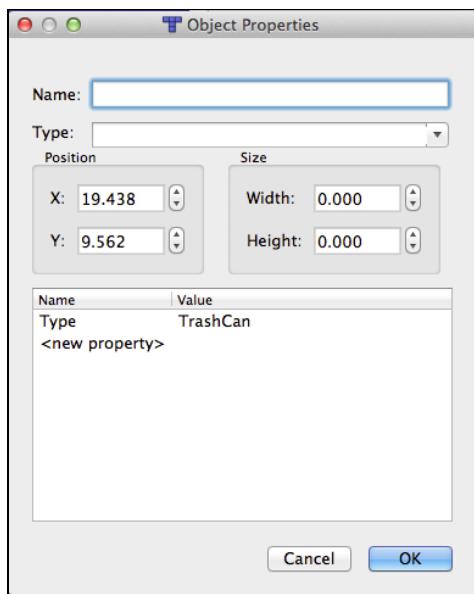


The icon beside the Objects layer is different from the other layers' icons because it's not a tile layer, or a layer used to place visual tiles—rather, it's an object layer, or a layer used to place markers with properties.

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 Sprite Kit, 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, I've pre-defined for you a property named **Type** 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 one way you can 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 it doesn't show up in the game.

Back in Xcode, open **GameLayer.h** and add this property:

```
@property (strong, nonatomic) NSMutableArray *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
{
    TMXObjectGroup *objectGroup = [self.tileMap groupNamed:@"Objects"];

    self.mapObjects = [NSMutableArray
arrayWithCapacity:objectGroup.objects.count];

    NSMutableDictionary *object;
    NSString *type;
    CGPoint position, coord, origin;

    for (object in [objectGroup objects]) {
        type = object[@"Type"];

        if (type && [type compare:@"TrashCan"] == NSOrderedSame) {

            position = CGPointMake([object[@"x"] floatValue], [object[@"y"]
floatValue]);

            coord = [self tileCoordForPosition:position];
            origin = [self tilePositionForCoord:coord
                                         anchorPoint:CGPointMake(0, 0)];

            TrashCan *trashCan = [TrashCan node];
            [trashCan setScale:kPointFactor];

            CGPoint actualOrigin = CGPointMultiplyScalar(origin,
kPointFactor);
            trashCan.position = CGPointMake(actualOrigin.x +
trashCan.size.width * trashCan.anchorPoint.x, actualOrigin.y +
trashCan.size.height * trashCan.anchorPoint.y);

            [self addChild:trashCan];
            [self.mapObjects addObject:trashCan];
        }
    }
}
- (CGPoint)tileCoordForPosition:(CGPoint)position
{
    CGFloat tileSizeWidth = self.tileMap.tileSize.width;
    CGFloat tileSizeHeight = self.tileMap.tileSize.height;
```

```

CGFloat levelHeight =
    self.tileMap.mapSize.height * tileHeight;

CGFloat x = floor(position.x / tileSize);
CGFloat y = floor((levelHeight - position.y) / tileSize);
return CGPointMake(x, y);

}

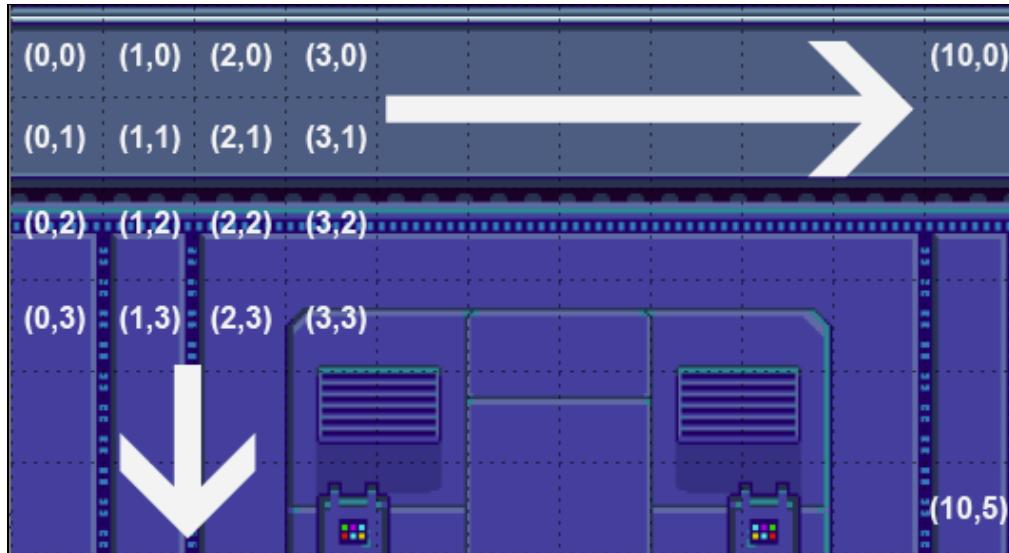
- (CGPoint)tilePositionForCoord:(CGPoint)coord
    anchorPoint:(CGPoint)anchorPoint
{
    CGFloat w = self.tileMap.tileSize.width;
    CGFloat h = self.tileMap.tileSize.height;
    return CGPointMake((coord.x * w) + (w * anchorPoint.x),
        ((self.tileMap.mapSize.height - coord.y - 1) * h) + (h *
    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 the helper method **tileCoordForPosition:** to get the tile coordinates of the object's position.

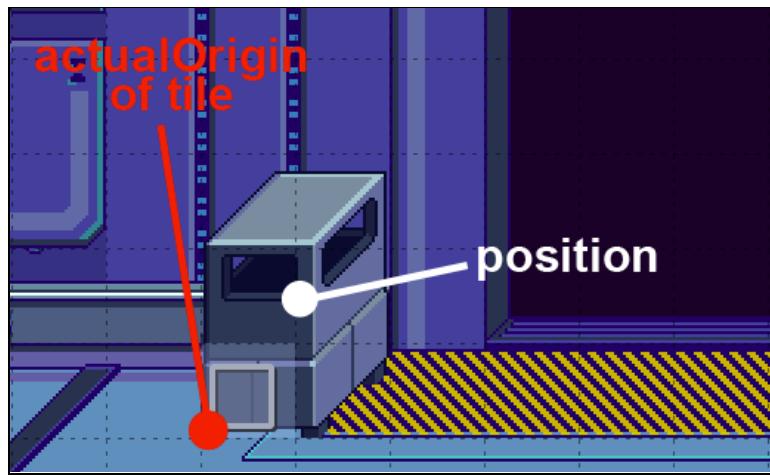
Each tile has a pair of coordinates, starting from (0, 0) for the upper-leftmost tile on the map. As you traverse right, the x-coordinate increases, and as you traverse downward, the y-coordinate increases.



The direction of the y-axis is opposite to what you're used to in Sprite Kit, where the (0, 0) coordinate is on the lower-left and y increases going upward.

Next, given the tile coordinates, you use another helper method, **tilePositionForCoord:**, to retrieve the position of the origin of the tile in Sprite Kit'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 an `SKSpriteNode` 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 trashcans scattered throughout 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, 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**:

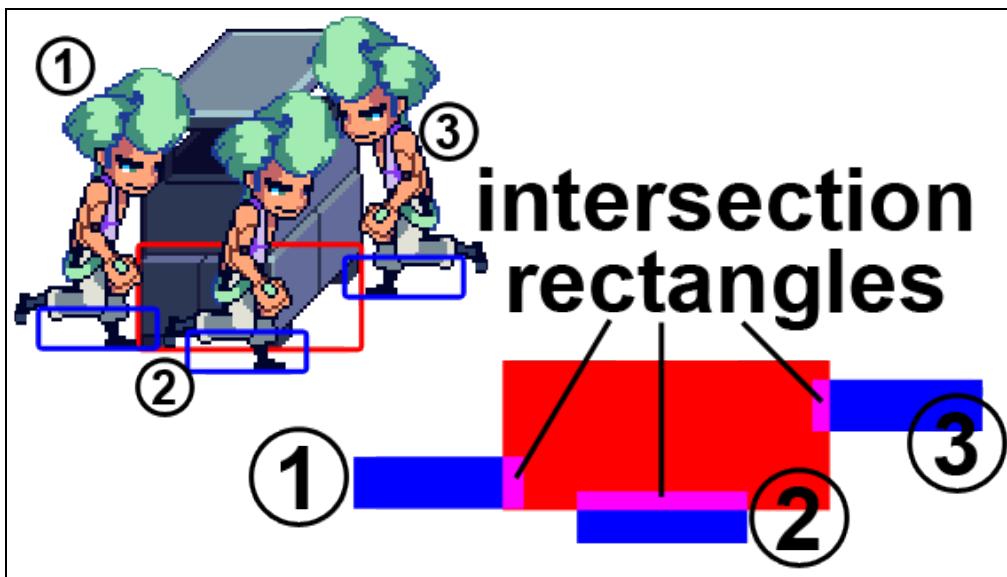
```
- (void)objectCollisionsForSprite:(ActionSprite *)sprite
{
    MapObject *mapObject;
    for (mapObject in self.mapObjects) {
        if (CGRectIntersectsRect(sprite.feetCollisionRect,
                               mapObject.collisionRect)) {
            CGFloat x = sprite.desiredPosition.x;
            CGFloat y = sprite.desiredPosition.y;

            CGRect intersect =
            CGRectIntersection(sprite.feetCollisionRect,
                               mapObject.collisionRect);

            if (intersect.size.width > intersect.size.height) {
                if (sprite.groundPosition.y < mapObject.position.y) {
                    y = sprite.desiredPosition.y - intersect.size.height;
                } else {
                    y = sprite.desiredPosition.y + intersect.size.height;
                }
            } else {
                if (sprite.groundPosition.x < mapObject.position.x) {
                    x = sprite.desiredPosition.x - intersect.size.width;
                } else {
                    x = sprite.desiredPosition.x + intersect.size.width;
                }
            }
            sprite.desiredPosition = CGPointMake(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 backward 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 make use of this collision method.

In `GameLayer.m`, modify `updatePositions` as follows:

```
//add inside if (self.hero.actionState > kActionStateNone), right after
the {
[self objectCollisionsForSprite:self.hero];

//add inside if (robot.actionState > kActionStateNone), right after the
{
[self objectCollisionsForSprite:robot];
```

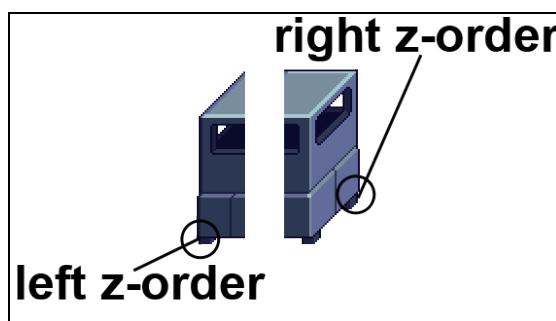
```
//add inside if (self.boss && self.boss.actionState > kActionStateNone),  
right after the {  
[self objectCollisionsForSprite:self.boss];
```

Build, run, and try colliding with the trashcan.



Because of the perspective of the trashcan, drawing it with the proper `zPosition` 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 `zPositions`.

In fact, splitting the image in half is a good way of handling `zPosition` for the trashcan.



The lower-left point of the trashcan's `collisionRect` will have a different `zPosition` than the upper-right point of the rectangle. Let's call these the left and right `zPositions`.

When a sprite bumps into the left side of the trashcan and moves above the left `zPosition`, it will appear behind the trashcan. But when the sprite bumps into the right side of the trashcan and is below the right `zPosition`, it will appear in front of the trashcan.

That's the ideal behavior. However, implementing this would require 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 `zPositions`.

Still in `GameLayer.m`, modify `reorderActors`:

```
//add at the end of the method, before the last }
MapObject *object;
for (object in self.mapObjects) {

    zPosition = [self getZFromYPosition:object.collisionRect.origin.y +
object.collisionRect.size.height/2];

    object.zPosition = zPosition;
}
```

You use the center of the `collisionRect` as the basis of the object's `zPosition`.

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 a 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
    CGFloat objectBottom = object.collisionRect.origin.y;

    CGFloat objectTop =
        objectBottom + object.collisionRect.size.height;

    CGFloat attackerBottom = attacker.feetCollisionRect.origin.y;

    CGFloat attackerTop =
        attackerBottom + attacker.feetCollisionRect.size.height;

    if ((attackerBottom > objectBottom &&
        attackerBottom < objectTop) ||
        (attackerTop > objectBottom &&
        attackerTop < objectTop)) {

        NSInteger i, j;
        CGFloat combinedRadius =
            attacker.detectionRadius + object.detectionRadius;

        //initial detection
        if (CGPointDistanceSQ(attacker.position, object.position)
            <= combinedRadius * combinedRadius) {

            NSInteger attackPointCount = attacker.attackPoints.count;
            NSInteger contactPointCount = object.contactPointCount;

            ContactPoint attackPoint, contactPoint;

            //secondary detection
            for (i = 0; i < attackPointCount; i++) {

                NSValue *value = attacker.attackPoints[i];
                [value getValue:&attackPoint];

                for (j = 0; j < contactPointCount; j++) {

                    contactPoint = object.contactPoints[j];

                    combinedRadius =
                        attackPoint.radius + contactPoint.radius;

                    if (CGPointDistanceSQ(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 in the first phase, where you check 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 simply 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;
    for (weapon in self.weapons) {
        if (weapon.weaponState == kWeaponStateNone){
            return weapon;
        }
    }
    return weapon;
}

//replace the contents of initWeapons
- (void)initWeapons
{
    self.weapons = [NSMutableArray arrayWithCapacity:3];
    Weapon *weapon;

    for (NSInteger i = 0; i < 3; i++) {
        weapon = [Gauntlets node];
        weapon.hidden = YES;
        [weapon.shadow setScale:kPointFactor];
        [weapon setScale:kPointFactor];
        weapon.groundPosition = OFFSCREEN;
        [self addChild:weapon.shadow];
        [self addChild:weapon];
        [self.weapons addObject:weapon];
    }
}

//add this to actionSpriteDidAttack:, inside if (actionSprite ==
//self.hero), right before returning didHit;
MapObject *mapObject;

```

```
for (mapObject in self.mapObjects) {  
  
    if ([self collisionBetweenAttacker:self.hero  
        andObject:mapObject  
        atPosition:&attackPosition]) {  
  
        HitEffect *hitEffect = [self getHitEffect];  
        hitEffect.zPosition = MAX(mapObject.zPosition,  
            self.hero.zPosition) + 1;  
  
        [hitEffect showEffectAtPosition:attackPosition];  
  
        if (mapObject.objectState != kObjectStateDestroyed) {  
  
            [mapObject destroyed];  
            Weapon *weapon = [self getWeapon];  
            [weapon droppedFrom:mapObject.size.height/2  
to:CGPointMake(mapObject.position.x, mapObject.position.y -  
mapObject.size.height/2)];  
            weapon.hidden = NO;  
        }  
    }  
}
```

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 already destroyed, then you destroy it and drop a weapon from its center.

Build, run, and scavenge for a pair of gauntlets!



Ending scenarios

If you've managed to get through the entire game and beat the final boss, you may 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 demonstrably when either all the enemies are wiped out or the hero dies.

To keep things simple, when the game ends, you'll simply 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:(SKColor *)color;
```

Switch to **HudLayer.m** and add the method implementation:

```
- (void)showMessage:(NSString *)message color:(UIColor *)color
{
    self.centerLabel.color = color;

    [self.centerLabel setText:message];

    [self.centerLabel runAction:[SKAction sequence:@[[SKAction
moveTo:CGPointMake(SCREEN.width + 50 * kPointFactor, CENTER.y)
duration:0], [SKAction showNode:self.centerLabel], [SKAction
moveTo:CENTER duration:0.2], [SKAction waitForDuration:1.0], [SKAction
moveTo:CGPointMake(-50 * kPointFactor, CENTER.y) duration:0.2],
[SKAction hideNode:self.centerLabel]]]];
}
```

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 "UIColor+BGSK.h"

//add this to actionSpriteDidDie:, inside if (actionSprite ==
self.hero), right after [self.hud setHitPoints:0
fromMaxHP:self.hero.maxHitPoints]

[self.hud showMessage:@"GAME OVER"
            color:[UIColor lowHPColor]];

[self runAction:[SKAction sequence:@[[SKAction waitForDuration:2.0],
[SKAction runBlock:^{[[NSNotificationCenter defaultCenter
```

```

postNotificationName:@"PresentTitle" object:nil]];]]];

//add this to the end of updateEvent, right after the //end game comment

[self hud showMessage:@"YOU WIN" color:[UIColor fullHPColor]];

[self runAction:[SKAction sequence:@[[SKAction waitForDuration:2.0],
[SKAction runBlock:^{
    [[NSNotificationCenter defaultCenter]
postNotificationName:@"PresentTitle" object:nil];
}]]];
```

When the hero dies, the HUD will show a message saying “GAME OVER” and send a notification named “PresentTitle” after two seconds. Likewise, when the hero wins on the last level, the HUD will show a “YOU WIN” message and send the same notification.

Remember that by design, you initiate scene transitions through the Notification Center, with ViewController as the recipient of such types of notifications. ViewController is currently subscribed to the “PresentGame” notification, so now you need to make it also listen for the “PresentTitle” notification.

Go to **ViewController.m** and do the following:

```

//add in viewDidLoad, right after [super viewDidLoad]
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(didReceivePresentTitleSceneNotification:)
name:@"PresentTitle" object:nil];

//add at the beginning of dealloc
[[NSNotificationCenter defaultCenter] removeObserver:self
name:@"PresentTitle" object:nil];

//add this method
- (void)didReceivePresentTitleSceneNotification:(NSNotification *)
notification
{
    SKView *skView = (SKView *)self.view;

    SKScene *titleScene =
    [TitleScene sceneWithSize:skView.bounds.size];

    [skView presentScene:titleScene
        transition:[SKTransition fadeWithDuration:1.0]];
}
```

When ViewController receives the “PresentTitle” notification, it creates a new instance of TitleScene and asks the SKView to present it using a fade-in/fade-out transition.

It’s a simple end to a simple game—though if you wanted to, 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 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 use 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 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 **TitleScene.m** and do the following:

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

//add to initWithSize right after [self setTitle]
[[SKTAudio sharedInstance]
playBackgroundMusic:@"latin_industries.aifc"];
```

When the game creates the **TitleScene**, you play the background music using **SKTAudio**, which is part of the **SKTUtils** package. It creates and uses an **AVAudioPlayer** to play audio files so you don't have to do any of the setup work.

If you do want to change the volume of the background music, you can still do so by editing **SKTAudio.m**.

Open **SKTAudio.m** and do the following:

```
//add this to playBackgroundMusic, right after
self.backgroundMusicPlayer.numberOfLoops = -1
self.backgroundMusicPlayer.volume = 0.5;
```

You set the background music player's volume to half of the original audio file's volume.

Next, switch back to **TitleScene.m** and add the following:

```
//add to the beginning of touchesBegan:
[self runAction:[SKAction playSoundFileNamed:@"blip.caf"
waitForCompletion:NO]];
```

Once the player touches the screen, you play the blip sound effect using an SKAction. For short sound effects, you don't need to use SKTAudio—SKActions are enough.

Switch to **GameLayer.m** and do the following (new code is highlighted):

```
//Modify actionSpriteDidAttack: (new code is highlighted)
- (BOOL)actionSpriteDidAttack:(ActionSprite *)actionSprite
{
    BOOL didHit = NO;
    if (actionSprite == self.hero) {

        CGPoint attackPosition;
        Robot *robot;
        for (robot in self.robots) {

            if (robot.actionState < kActionStateKnockedOut &&
                robot.actionState != kActionStateNone) {

                if ([self collisionBetweenAttacker:self.hero
                    andTarget:robot
                    atPosition:&attackPosition]) {

                    BOOL showEffect = YES;

                    DamageNumber *damageNumber = [self getDamageNumber];

                    damageNumber.zPosition =
                        MAX(robot.zPosition, self.hero.zPosition) + 1;

                    if (self.hero.actionState == kActionStateJumpAttack) {

                        //add this
                    }
                }
            }
        }
    }
}
```

```
[self runAction:[SKAction playSoundFileNamed:@"hit1.caf"
waitForCompletion:NO]];

[robot knockoutWithDamage:self.hero.jumpAttackDamage
direction:CGPointMake(self.hero.directionX,
0)];

[damageNumber showWithValue:self.hero.jumpAttackDamage
fromOrigin:robot.position];

showEffect = NO;

} else if (self.hero.actionState ==
kActionStateRunAttack) {

    //add this
    [self runAction:[SKAction playSoundFileNamed:@"hit0.caf"
waitForCompletion:NO]];

[robot knockoutWithDamage:self.hero.runAttackDamage
direction:CGPointMake(self.hero.directionX,
0)];

[damageNumber showWithValue:self.hero.runAttackDamage
fromOrigin:robot.position];

} else if (self.hero.actionState ==
kActionStateAttackThree)  {

    //add this
    [self runAction:[SKAction playSoundFileNamed:@"hit1.caf"
waitForCompletion:NO]];

[robot knockoutWithDamage:self.hero.attackThreeDamage
direction:CGPointMake(self.hero.directionX,
0)];

[damageNumber showWithValue:self.hero.attackThreeDamage
fromOrigin:robot.position];

showEffect = NO;

} else if (self.hero.actionState ==
kActionStateAttackTwo)  {

    //add this
    [self runAction:[SKAction playSoundFileNamed:@"hit0.caf"
waitForCompletion:NO]];

[robot hurtWithDamage:self.hero.attackTwoDamage
force:self.hero.attackForce
direction:CGPointMake(self.hero.directionX,
0.0)];
```

```
[damageNumber showWithValue:self.hero.attackTwoDamage
    fromOrigin:robot.position];

} else {

    //add this
    [self runAction:[SKAction playSoundFileNamed:@"hit0.caf"
waitForCompletion:NO]];

    [robot hurtWithDamage:self.hero.attackDamage
        force:self.hero.attackForce
        direction:CGPointMake(self.hero.directionX,
0.0)];

    [damageNumber showWithValue:self.hero.attackDamage
        fromOrigin:robot.position];
}

didHit = YES;

if (showEffect) {
    HitEffect *hitEffect = [self getHitEffect];
    hitEffect.zPosition = damageNumber.zPosition + 1;
    [hitEffect showEffectAtPosition:attackPosition];
}
}

}

if (self.boss &&
    self.boss.actionState < kActionStateKnockedOut &&
    self.boss.actionState != kActionStateNone) {

    if ([self collisionBetweenAttacker:self.hero
        andTarget:self.boss
        atPosition:&attackPosition]) {

        BOOL showEffect = YES;
        DamageNumber *damageNumber = [self getDamageNumber];
        damageNumber.zPosition = MAX(self.boss.zPosition,
            self.hero.zPosition) + 1;

        if (self.hero.actionState == kActionStateJumpAttack) {

            //add this
            [self runAction:[SKAction playSoundFileNamed:@"hit1.caf"
waitForCompletion:NO]];

            [self.boss hurtWithDamage:self.hero.jumpAttackDamage
force:self.hero.attackForce direction:CGPointMake(self.hero.directionX,
0)];
        }
    }
}
```

```
[damageNumber showWithValue:self.hero.jumpAttackDamage
fromOrigin:self.boss.position];

showEffect = NO;

} else if (self.hero.actionState ==
kActionStateRunAttack) {

    //add this
    [self runAction:[SKAction playSoundFileNamed:@"hit0.caf"
waitForCompletion:NO]];

    [self.boss hurtWithDamage:self.hero.runAttackDamage
force:self.hero.attackForce direction:CGPointMake(self.hero.directionX,
0)];

    [damageNumber showWithValue:self.hero.runAttackDamage
fromOrigin:self.boss.position];

} else if (self.hero.actionState ==
kActionStateAttackThree) {

    //add this
    [self runAction:[SKAction playSoundFileNamed:@"hit1.caf"
waitForCompletion:NO]];

    [self.boss hurtWithDamage:self.hero.attackThreeDamage
force:self.hero.attackForce direction:CGPointMake(self.hero.directionX,
0)];

    [damageNumber showWithValue:self.hero.attackThreeDamage
fromOrigin:self.boss.position];

    showEffect = NO;

} else if (self.hero.actionState ==
kActionStateAttackTwo) {

    //add this
    [self runAction:[SKAction playSoundFileNamed:@"hit0.caf"
waitForCompletion:NO]];

    [self.boss hurtWithDamage:self.hero.attackTwoDamage
force:self.hero.attackForce/2
direction:CGPointMake(self.hero.directionX, 0.0)];

    [damageNumber showWithValue:self.hero.attackTwoDamage
fromOrigin:self.boss.position];

} else {

    //add this
```

```
[self runAction:[SKAction playSoundFileNamed:@"hit0.caf"
waitForCompletion:NO]];

        [self.boss hurtWithDamage:self.hero.attackDamage force:0
direction:CGPointMake(self.hero.directionX, 0.0)];

        [damageNumber showWithValue:self.hero.attackDamage
fromOrigin:self.boss.position];
    }

    didHit = YES;

    if (showEffect) {
        HitEffect *hitEffect = [self getHitEffect];
        hitEffect.zPosition = damageNumber.zPosition + 1;
        [hitEffect showEffectAtPosition:attackPosition];
    }
}

MapObject *mapObject;
for (mapObject in self.mapObjects) {

    if ([self collisionBetweenAttacker:self.hero
                                andObject:mapObject
                           atPosition:&attackPosition]) {

        HitEffect *hitEffect = [self getHitEffect];
        hitEffect.zPosition = MAX(mapObject.zPosition,
                                   self.hero.zPosition) + 1;

        [hitEffect showEffectAtPosition:attackPosition];

        if (mapObject.objectState != kObjectStateDestroyed) {

            //add this
            [self runAction:[SKAction playSoundFileNamed:@"hit1.caf"
waitForCompletion:NO]];

            [mapObject destroyed];
            Weapon *weapon = [self getWeapon];
            [weapon droppedFrom:mapObject.size.height/2
to:CGPointMake(mapObject.position.x, mapObject.position.y -
mapObject.size.height/2)];
            weapon.hidden = NO;
        }
        //add this else clause
    } else {
        [self runAction:[SKAction playSoundFileNamed:@"hit0.caf"
waitForCompletion:NO]];
    }
}
}
```

```
        return didHit;

    }

    else if (actionSprite == self.boss) {

        if (self.hero.actionState < kActionStateKnockedOut &&
            self.hero.actionState != kActionStateNone) {

            CGPoint attackPosition;

            if ([self collisionBetweenAttacker:self.boss
                                         andTarget:self.hero
                                         atPosition:&attackPosition]) {

                //add this
                [self runAction:[SKAction playSoundFileNamed:@"hit1.caf"
                                              waitForCompletion:NO]];

                [self.hero knockoutWithDamage:self.boss.attackDamage
direction:CGPointMake(actionSprite.directionX, 0.0)];

                [self.hud setHitPoints:self.hero.hitPoints
fromMaxHP:self.hero.maxHitPoints];

                didHit = YES;

                DamageNumber *damageNumber = [self getDamageNumber];

                [damageNumber showWithValue:self.boss.attackDamage
                                fromOrigin:self.hero.position];

                damageNumber.zPosition = MAX(self.boss.zPosition,
                                             self.hero.zPosition) + 1;

                HitEffect *hitEffect = [self getHitEffect];
                hitEffect.zPosition = damageNumber.zPosition + 1;
                [hitEffect showEffectAtPosition:attackPosition];
            }
        }
    }

    else {

        if (self.hero.actionState < kActionStateKnockedOut &&
            self.hero.actionState != kActionStateNone) {

            CGPoint attackPosition;
            if ([self collisionBetweenAttacker:actionSprite
                                         andTarget:self.hero
                                         atPosition:&attackPosition]) {
```

```

    //add this
    [self runAction:[SKAction playSoundFileNamed:@"hit0.caf"
                                             waitForCompletion:NO]];

    [self.hero hurtWithDamage:actionSprite.attackDamage
                      force:actionSprite.attackForce
                     direction:CGPointMake(actionSprite.directionX,
0.0)];

    [self.hud setHitPoints:self.hero.hitPoints
                  fromMaxHP:self.hero.maxHitPoints];

    didHit = YES;

    DamageNumber *damageNumber = [self getDamageNumber];

    damageNumber.zPosition =
        MAX(actionSprite.zPosition, self.hero.zPosition) + 1;

    [damageNumber showWithValue:actionSprite.attackDamage
                      fromOrigin:self.hero.position];

    HitEffect *hitEffect = [self getHitEffect];
    hitEffect.zPosition = damageNumber.zPosition + 1;
    [hitEffect showEffectAtPosition:attackPosition];
}
}

return didHit;
}

```

All you do here is add some lines in the appropriate spots to play some sound effects.

Switch to **Hero.m** and do the following:

```

//add inside knockoutWithDamage:direction:, inside if (self.actionState
== kActionStateKnockedOut)
if (self.hitPoints <= 0) {
    [self runAction:[SKAction playSoundFileNamed:@"herodeath.caf"
                                             waitForCompletion:NO]];
}

```

Finally, do the following for both **Robot.m** and **Boss.m**:

```

//add this method
- (void)knockoutWithDamage:(CGFloat)damage
                      direction:(CGPoint)direction
{

```

```
[super knockoutWithDamage:damage direction:direction];

if (self.actionState == kActionStateKnockedOut &&
    self.hitPoints <= 0) {

    [self runAction:[SKAction playSoundFileNamed:@"enemydeath.caf"
waitForCompletion:NO]];
}
```

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 Sprite Kit's diagnostics found in the lower-right corner of the game screen.

To do this, open **ViewController.m** and do the following:

```
//in viewWillLayoutSubviews, replace these
skView.showsFPS = YES;
skView.showsNodeCount = YES;
skView.showsDrawCount = YES;

//with these
skView.showsFPS = NO;
skView.showsNodeCount = NO;
skView.showsDrawCount = NO;
```

And that's really 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 first beat 'em up game!

It's been a long journey, but you made it.

Perhaps the best thing you've done is start 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 will take 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.

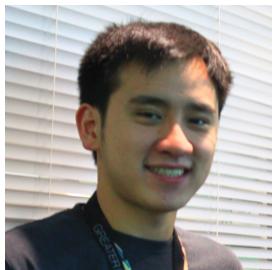
And of course, there are optional things you might want to add to improve the game:

- Create more actions—for example, 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.
- Add music and sound effects of your choice.

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 and everything our team works on there.

I appreciate each and every one of you for taking the time to try out this starter kit. If you have an extra few minutes, I would really love to hear what you thought of it!

Please leave a comment on the official private forums for the Beat 'Em Up Game Starter Kit at 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 do 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