Fully updated for Sprite Kit!

PLATFORMER GAME STARTER KIT

By Jake Gundersen

Platformer Game Starter Kit

Jake Gundersen

Table of Contents

Dedication

To my boys, John and Eli.

Introduction

The platformer is one of the most recognizable types of games. If you grew up with a Nintendo, Super Nintendo or Sega Genesis, you surely have nostalgic memories of games like Super Mario Bros., Sonic the Hedgehog and Bionic Commando, just to name a few.

Platform games have the right combination of action and puzzle elements to appeal to almost any video game player. They require dexterity and a keen sense of timing, as well as problem-solving abilities.

Looking back through the annuls of video game history, we can say that there are three essential elements of a great platformer:

1. **Sense of connection between player and character**. The earliest platformers, such as *Super Mario Brothers*, *Sonic the Hedgehog*, *Contra* and *Metroid*, all create a sense of connection between the player and the game's hero. The more responsive the game's controls, the stronger this feeling tends to be. The platformers that lacked this feeling of connection with the game's avatar didn't enjoy the same level of success. Remember *Conan* for NES, anyone?
2. **Complex physics engines**. The second feature of the first generation of 8-bit games that made them so much fun was the complex physics engines behind them. In earlier games, like *Pitfall*, for example, players faced difficulties controlling the character. Movements were pretty static—every jump was the same height, there was no reversing of direction in mid-jump, and so on—and as a result, the game was less enjoyable.
3. **Fluid connection between controller and movement**. The third essential element of a great-feeling platformer is a fluid connection between the controller and the character's movement. The game's reaction to a button press must be instantaneous, and it must be easy for the player to press the buttons as they intend. If it's too easy to press a wrong button or too hard to press the right one, the game can feel disjointed and leave players frustrated. This is an especially important consideration on the mobile platform, where the player doesn't have physical buttons to help orient their fingers.

The goal of the Platformer Game Starter Kit is to teach you the fundamental techniques involved in making a platform game, with these three elements in mind.

You will learn how to create your own physics engine tailored for platformers, implement jump behavior, add enemies and powerups, and much more.

The Platformer Game Starter Kit consists of two parts:

1. **Full source code**. This starter kit includes full source code for a complete side-scrolling platform game for the iPhone, including tile maps, custom physics behavior and enemy AI.
2. **Epic-length tutorials**. With the Platformer Game Starter Kit, not only do you get full source code that you can use in your own games—you also get 10 epic-length tutorials that teach you how to build the entire game!

Whether you're a beginner or an advanced iOS developer, by the time you are done reading this starter kit you will have the knowledge you need to begin creating your own platform game.

Introducing the second edition

It's been a little over 9 months since I first wrote the Platformer 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 Platformer 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 Platformer 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 make use of the Platformer Game Starter Kit.

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

As you look through the code, you can refer to the tutorials to read up on any sections of code that you don't understand. The beginning of this guide has table of contents that can help with that, and the search tool is your friend.

A second way of using the Platformer Game Starter Kit is to go through these tutorials one by one and build the game from scratch. This is the best way to learn because you'll literally write each line of the gameplay code, one small piece at a time.

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

Starter kit overview

The game you will make in this starter kit is called *Pocket Cyclops*—a game about a cyclops having a very bad day! The cyclops lived deep underground with all manner of fantastic creatures, until one fateful day it was kidnapped by evil robots. Now it must battle its way back down to its underworld home.

Here's how you will build this game, chapter by chapter:

Chapter 1

In this chapter, you load the game's level assets, including a TMX tile map and parallax background images. This sets the stage for the animated objects you add later.

Chapter 2

In Chapter 2, you add the main player character, a cyclops, to the game. You also add basic physics rules to the game universe and learn about the class hierarchy.

Chapter 3

In this chapter, you complete the physics engine that governs the universe by adding floor and wall collisions so that the cyclops can explore the world it inhabits. You learn how to query the tiles from the map and you create the logic to make the cyclops move in ways consistent with the platformer experience.

Chapter 4

Chapter 4 introduces the rules that move the cyclops around. You enable the player to jump and to move left and right. You also explore the physics rules that control the speed and momentum of the player character.

Chapter 5

In Chapter 5, you create a HUD node that contains the button controls and the cyclops's life indicator. You learn how to interpret touches in an intelligent way to create an elegant joystick interface on a touch screen.

You also learn about how `SKTextureAtlas` objects work in Sprite Kit and how to preload textures and update images, as well as about some of the pitfalls in Sprite Kit's current implementation.

Chapter 6

In Chapter 6, you add a quality called **state** to the player character. A state machine is a logic structure that has a single value out of many possible values at any given time. The state machine is essential to controlling when the cyclops can jump, is injured or is dying.

The cyclops and its enemies will all have a number of possible states, such as walking, standing, jumping, double-jumping, wall-sliding, attacking and dying. The state of the cyclops determines how fast it can move, what it should be doing and what it should look like.

Chapter 7

You learn to add animations in this chapter. Animations change the appearance of the player character according to whether it's jumping, running or dying. To control which animation is currently running, you use the state machine, which helps you determine which states are valid and which aren't, and which state to transition to next.

Once you animate the cyclops, your game begins to feel like a real boy! Err… I mean game. ☺

Chapter 8

In this chapter, you incorporate enemies into your game. The enemies follow a lot of the same rules—and use the same code—for physics and collision detection as does the cyclops. You add patterns of behavior, from simple to complex, to create enemies that are both intelligent and challenging.

Chapter 9

In this chapter, you learn how to track collisions between the cyclops and its enemies. You write logic that determines which party in the collision takes damage and add logic to trigger the dying state for both the cyclops and the enemy.

You'll also give the cyclops a HUD life indicator and update it when the cyclops is injured.

Chapter 10

In this final chapter, you add the finishing touches, including the logic that moves the player character from one level to the next and determines when the game has been won or lost and how the UI behaves in each case. You also add sound to the game.

And when you're done...

Once you make your way through all of these chapters, not only will you be capable of creating a platformer game on your own—you'll have a robust framework that you can use for many other types of level-based games.

That's because the fundamental concepts, animations, state machines, physics engines, tile maps and so forth that this book teaches are just as applicable to other game genres: adventure, puzzle, beat 'em up and shooter games, to name a few.

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 definitely want to have them on hand as you go through the starter kit.

We've also set up an official forum for the starter kit here:

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

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

Acknowledgements

I would like to thank several people for their assistance making this starter kit possible:

- **Daniel Shiffman**, for his excellent book, *The Nature of Code*.
- **sonicretro.org**, for its great resources on the implementation of *Sonic the Hedgehog*.
- **Everyone on the iOS Games by Tutorials team**, for helping me learn the ins and outs of Sprite Kit.
- 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

Jake Gundersen is a gamer, maker and programmer. He is co-founder of the educational game company, Third Rail Games. He has a particular interest in gaming, image processing and computer graphics. You can find his musings and codings at http://indieambitions.com

About the editors

Matthijs Hollemans is an independent designer and developer who loves to create awesome software for the iPad and iPhone. He also enjoys teaching others to do the same, which is why he wrote The iOS Apprentice series of eBooks. In his spare time, Matthijs is learning to play jazz piano (it's hard!) and likes to go barefoot running when the sun is out. Check out his blog at http://www.hollance.com.

B.C. Phillips is an editor who splits his time between New York City and the Northern Catskills. He has many interests, but particularly loves food, exploring all manner of things, thinking about deep questions and working on his cabin and land in the mountains—even though his iPhone is pretty useless up there.

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

About the artist

Joe McCormick is a visual development artist and animator currently living in New York City. He has studied at the Savannah College of Art and Design as well as the CG Master Academy. To see more of his work, please go to joe-mccormick.com.

Chapter 1: Getting Started

The Platformer Game Starter Kit contains all the resources and instructions you need to make a fun platformer game called *Pocket Cyclops*.

Let's start by taking a look at the files that come with the starter kit. You should see the following folders in the Platformer Game Starter Kit's root directory:

1. Pocket Cyclops Starter

This is the starting point for the project. I'll review this with you in a few pages.

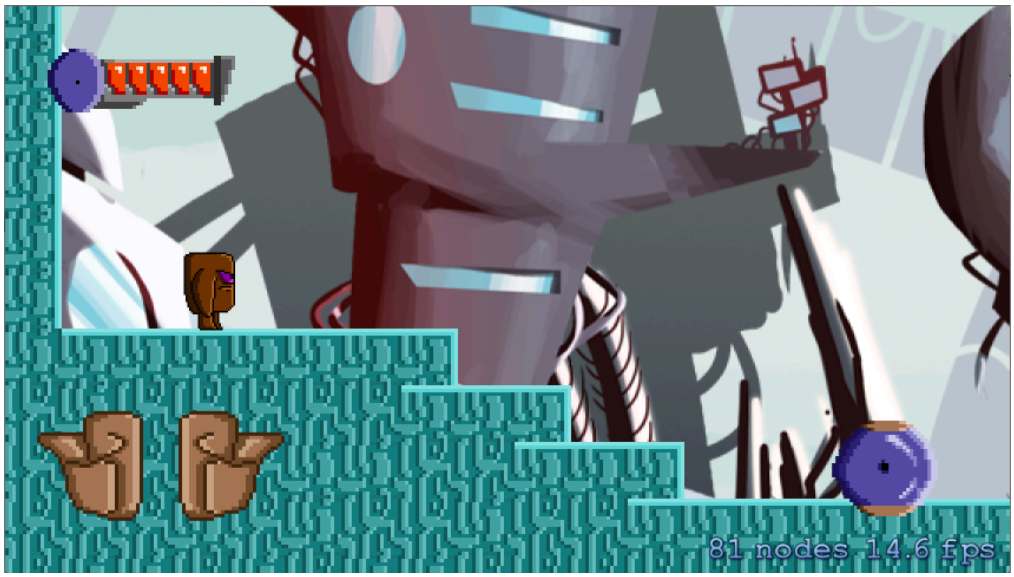
2. Pocket Cyclops Chapter X

For each chapter in the book, I've provided a folder that contains the project in the state it should be at the end of the chapter. This way, if you ever get lost or make a mistake, you can revert back to the correct state by using the project from the end of the previous chapter.

3. Pocket Cyclops Final

This is the game in its completed state.

Let's take a look at the final project. In the folder titled **Pocket Cyclops Final**, you'll see an Xcode project file. Double-click it. Build and run the project on either the Simulator or on an iPhone and take a look at the game in all its glory. This is what you will build as you follow along:



After you've played around with *Pocket Cyclops* a bit, close the Xcode project and open the **Pocket Cyclops Starter** project instead. Build and run, and you'll see that the main menu is functional—but when you start a level, all you get is a screen that's similar to the basic Sprite Kit template app:

In this chapter, you'll spend your time implementing this screen. The rest of the project is a framework to save you time and get you started quickly.

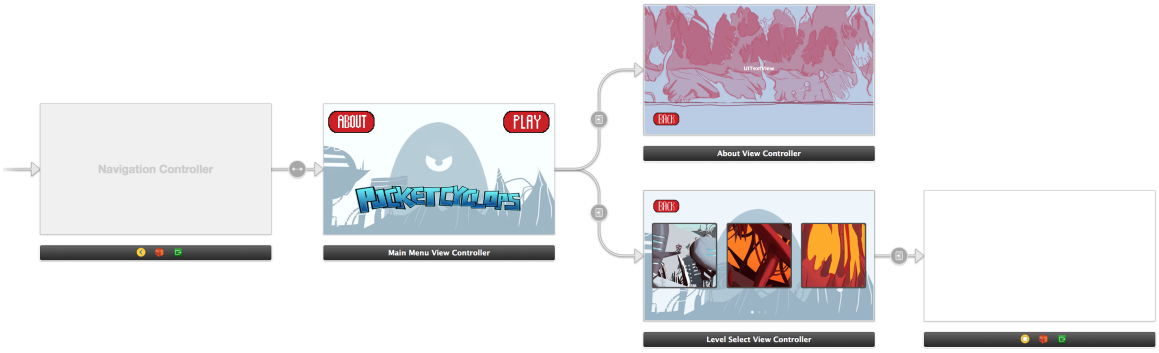
The starter project

This section goes over the different parts of the starter project. For most parts, I've included a small challenge for you to slightly tweak the app. I highly recommend you try each challenge so that you become well acquainted with each part of the code in this project. If you get stuck, you can always compare your work to the solution in the **Pocket Cyclops Chapter 1** folder.

UIKit menu interface

The goal of this starter kit is to show you how to code a platformer's gameplay. To keep the focus on that, I have already provided the code for the game's main menu.

Open **Main_iPhone.storyboard** to see the menu system. There are five view controllers inside.



Because Sprite Kit is already integrated with UIKit, it's easy to create menus and other views, such as a Game Center matchmaker view, and integrate them seamlessly with a Sprite Kit game. This is a big improvement for anyone who used Cocos2D, where it was a bit messy to mix UIKit with the Cocos2D game view.

> **Challenge**: Open **Main_iPhone.storyboard** and change the text on the About View Controller. Rebuild and verify that your new text shows up OK.

Graphic assets

The starter project includes all the images and graphics you need for a game, either in the **sprites.atlas** folder, the **Level Data** folder or the **UI Images** folder.

One of Sprite Kit's unique benefits is its automatic creation of texture atlases. I'll talk more about this later, but here's a quick look at the texture atlas that Sprite Kit created from the individual sprite images (Xcode builds a texture atlas for you automatically).

Sprite Kit has composited every image that is in the project's **sprites.atlas** folder into one giant image file. This has big performance benefits for any game that uses OpenGL to draw its sprites. In other game engines, you'd have to do this yourself or use a third-party tool. Sprite Kit has this feature built-in, and even better, the API to use the images inside the texture atlas is more beginner-friendly than in other game engines.

> **Note**: Joe McCormick created all of the art for the Platformer Game Starter Kit. As you can see, he is awesome! I selected Joe out of a pool of artists because his style really appealed to me.
>
> Joe is accepting contract work, so if you'd like to hire him for your own game, you can contact him here: http://cargocollective.com/joemccormick

Level data

The game includes several pre-built levels. These levels are TMX tile map files that I created using the popular Tiled map editor (http://www.mapeditor.org), and you can find them in the **Level Data** folder as **level1.tmx**, **level2.tmx**, and **level3.tmx**.

A TMX file is a tile-based map system that relies on a grid of tiles and a tileset image (i.e. an image that contains a sub-image for each type of tile) to create a map. It also includes data about the positions of key placemarks, such as the starting position of the player in a layer, and the enemy positions and types.

This tile-based system is similar to early 2D platformers, especially *Super Mario Brothers*. These types of maps can be laid on top of background art that isn't necessarily tile-based.

To follow this tutorial, you need to be familiar with the basics of working with TMX tile maps. There's a great tutorial that covers TMX maps in our Sprite Kit book, *iOS Games By Tutorials*:

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

Sprite Kit doesn't have built-in support for TMX files, but several open source projects exist that allow you to use TMX maps with Sprite Kit. This book uses the one written by Jeremy Stone (@slycrel). The code is already part of the starter project, under the group `JSTileMap`. You can find the original here:

• https://github.com/slycrel/JSTileMap

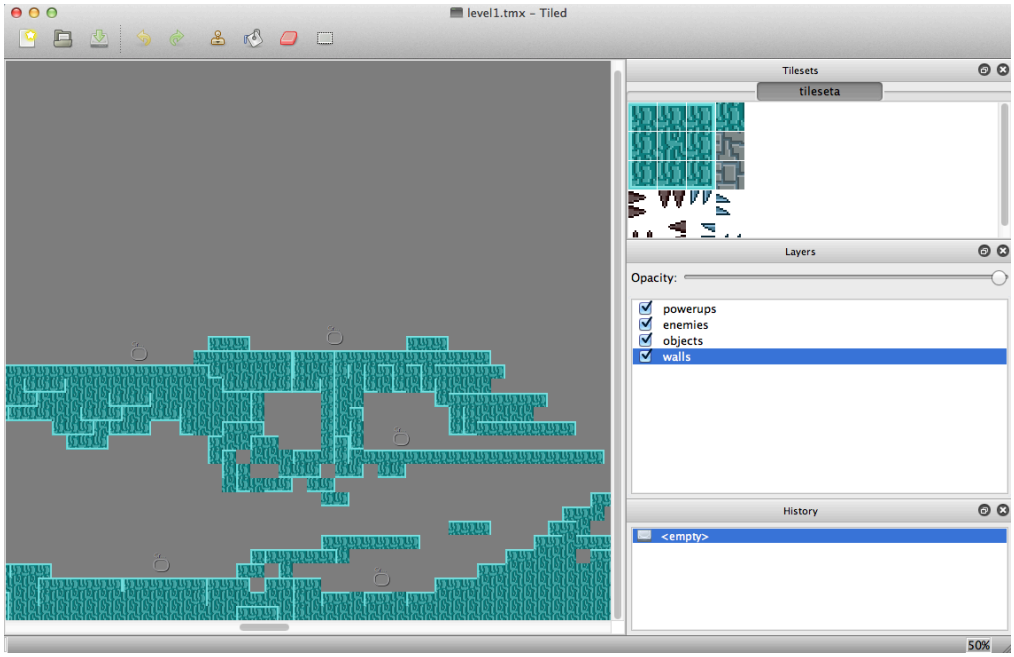
I've added some optimizations to the class to make it perform better on the iPhone 4. It turns out that Sprite Kit will skip drawing nodes that aren't visible on the current scene. However, with large tile maps it takes Sprite Kit quite a bit of time to analyze which nodes don't need to be drawn. Because a tile map is laid out in a grid, it's easy to calculate which nodes should be visible and set all the other tiles `.hidden` property to YES. Rather than relying on Sprite Kit, which has to go through and check every node, we do the work of hiding nodes in the class, only hiding and revealing nodes around the edges of the screen. This is what my additions do. You can find my version here (the version included in the book includes my additions):

• https://github.com/fattjake/JSTileMap

One requirement of `JSTileMap` is the libz.dylib library for reading ZIP files. I have already added that to the project as well.

The way you're going to use `JSTileMap` is very similar to how you would use Cocos2D's `CCTMXTiledMap` class. If you have trouble understanding how TMX tile maps work, you can also look at Cocos2D tutorials to learn more about the underlying concepts.

Here's a screenshot from the free Tiled editor, the tool I used to create and modify the maps in this game.



Feel free to play around with the level design as you build the game!

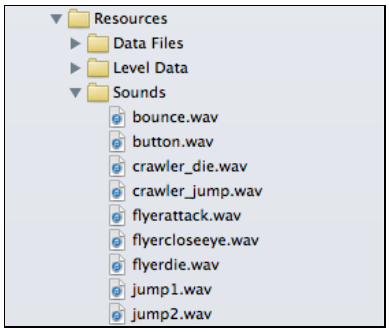
> **Challenge:** Open **level1.tmx** in Tiled and make a small tweak to the lower left corner of the level, like erasing a tile underneath the platform. Save the tile map and later on, when you add this level to the game, look to see if your change made it through!

Another resource included in the kit is a parallax node class called `JLGParallaxNode`. This class is based on the `CCParallaxNode` class in Cocos2D—in fact, much of the code was copied from that class. You can add other nodes to this class and as the parent node moves, it will also move its child nodes, but at a different pace. This is called parallax scrolling and I'll talk more about it soon.

Music and audio

The starter kit includes a number of audio effects and music files for *Pocket Cyclops*, which you can find in the **Sounds** group. I purchased most of these at http://audiojungle.com. That's an affordable place to get high-quality music and sound assets for your game projects.

> **Note:** These sound files were purchased only for use with this starter kit. You cannot reuse the audio assets from this game in your own projects, unless you obtain the rights from the original creators.



> **Challenge**: Replace **lvl1.mp3** with an MP3 of your own choosing. Later on when you add music to the game, this will play!

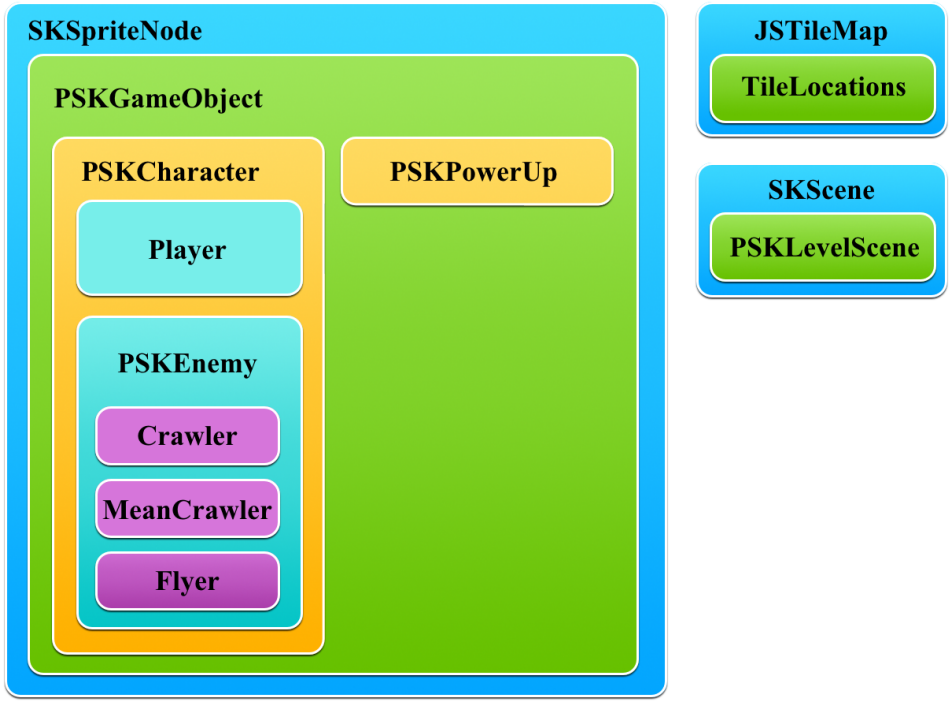
The class hierarchy

Before you dive into coding, let's discuss the class hierarchy you'll use in this game.

As a programmer, it's important to carefully consider the architecture of your code. You want code that's easy to understand, maintain and extend.

In this starter kit, you'll learn how to construct classes and subclasses in ways that minimize redundant code, thus decreasing the amount of time it takes you to make a great platformer game.

This diagram shows how all the classes of this platformer engine fit together:



Here's a quick overview of each class, its major function and the methods and attributes it contains:

1. `SKScene` → `PSKLevelScene`

a. The `PSKLevelScene` object is the parent (or root) node of the entire game node tree. All objects, the map, the background images, the main character (think Mario), the enemies and so forth are child nodes of this parent node. In Sprite Kit, you can have many nodes per scene, but there is only one scene running at a time. You will often use a scene to represent a single level.

b. `PSKLevelScene` has an `initWithLevel` method that takes in an integer that tells the scene which level object to load. For each level, it loads a PLIST file that describes the level. The PLIST contains a dictionary that has data about which other assets—such as sound, TMX map, background images—to load.

c. `PSKLevelScene` loads all of these assets into memory and sets up the level.

d. PSKLevelScene contains the primary methods for the physics engine-based collision detection.

e. PSKLevelScene keeps the camera centered on the cyclops.

2. JSTileMap

a. The JSTileMap object contains all the information about a given TMX map. A map can have one or more TMXLayer objects and zero or more TMXObjectGroup objects. These objects hold the information about the tiles and their positions, as well as data about any other arbitrary objects, such as the player's position at the start of the level, enemy positions and so on.

b. You'll use categories to extend the functionality of this base class so that it can retrieve information about specific tiles to test for collisions—which is how you will keep the player from walking through walls.

3. SKSpriteNode → **PSK**GameObject

a. The PSKGameObject is the parent class for all game sprites. It contains methods to load animations from sprite sheets, something potential relevant to all game sprites.

4. SKSpriteNode → **PSK**GameObject → **PSK**PowerUp

a. A PSKPowerUp is a game object that enhances the cyclops's abilities.

5. SKSpriteNode → **PSK**GameObject → PSKCharacter

a. A PSKCharacter object moves around within the scene and collides with walls, floors and other characters. It contains a number of attributes that the physics engine requires, such as desiredPosition and velocity.

b. Each PSKCharacter instance has a bounding box method called collisionBoundingBox that returns a CGRect you'll use for collision detection. The PSKCharacter superclass contains a default implementation of collisionBoundingBox that uses the size of the sprite texture. Many subclasses of PSKCharacter will override this method to allow for more precise control of collision detection.

c. Each character has a state machine that controls the animations and logic for that entity. The PSKCharacter class contains both the current state attribute and a superclass method for changing the state.

d. Each character has a tookHit method that controls the damage the character receives from colliding with another character. There is an empty superclass implementation in the PSKCharacter class.

6. SKSpriteNode → **PSK**GameObject → PSKCharacter → Player

a. The Player object has a single instance and represents the character that the user controls, the cyclops.

b. The Player object contains the most intricate subclass implementations of the state machine, update logic and animations.

c. The `Player` class queries the state of the HUD's buttons to determine how to move the cyclops around the level.

d. The `tookHit` method controls the player's damage from collisions with enemies and updates the HUD's life meter.

7. `SKSpriteNode` → **`PSK`**`GameObject` → `PSKCharacter` → **`PSK`**`Enemy`

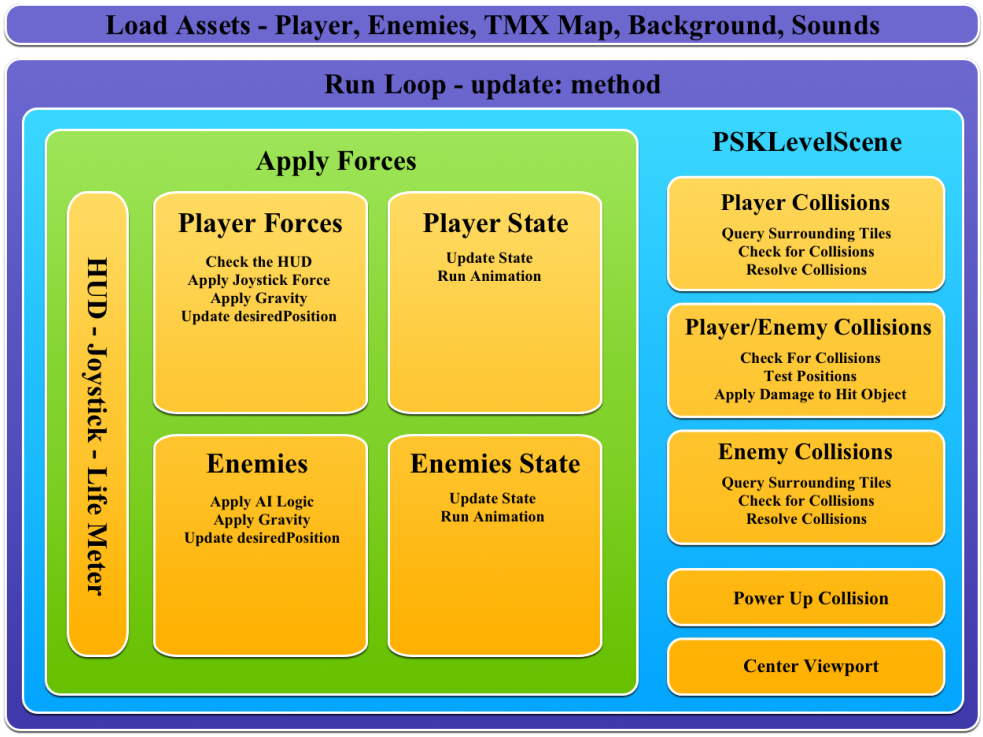
a. In *Pocket Cyclops*, all enemies die after a single hit. You can find this common `tookHit` implementation in the `Enemy` class.

b. Some enemies need to have information about the cyclops's position or information about tiles in the tile map. The `PSKEnemy` class contains these attributes.

8. `SKSpriteNode` → `PSKGameObject` → `PSKCharacter` → `PSKEnemy` → Individual Enemy Subclass

a. An individual `PSKEnemy` subclass contains the information specific to that type of enemy. This includes animations, states and run loop logic.

The above is just a high-level overview of the classes used by the game engine. As you progress from one chapter to another, I'll refer back to this map to help you understand how each piece fits into the whole. This part-to-whole abstraction is key to understanding how to build a complex, functional game.



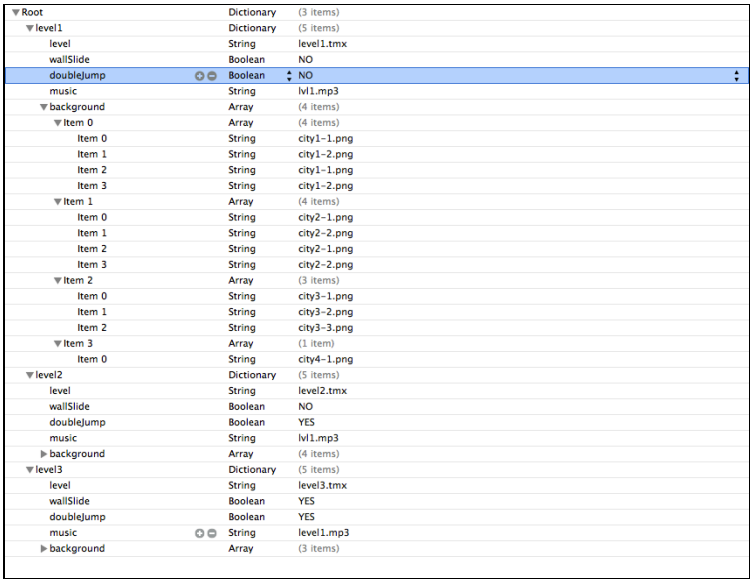
Let's code!

Enough talk—at long last, it's time to code!

You'll start nice and easy by adding some code to load the background and music. That will get your feet wet.

Loading the background

I created a file called **Data Files\levels.plist** that contains information about each level. Open it and take a look:



This file contains information about the parallax background, the music for the level, the TMX tile map and the state of the player's upgrades (**wallSlide** and **doubleJump**) at the beginning of the level. You need to know the player's upgrade state because the levels have been designed so that the player has to find and use a power-up to complete the level.

First, add a custom initialization method to **PSKLevelScene.h**:

```
- (id)initWithSize:(CGSize)size level:(NSUInteger)currentLevel;
```

Next, open **PSKLevelScene.m** and replace the contents of that file with the following:

```
#import "PSKLevelScene.h"

@interface PSKLevelScene ()

@property (nonatomic, assign) NSUInteger currentLevel;

@end
```

```
@implementation PSKLevelScene

- (id)initWithSize:(CGSize)size level:(NSUInteger)currentLevel {
  if ((self = [super initWithSize:size])) {
    self.currentLevel = currentLevel;
  }
  return self;
}

@end
```

This is a custom `init` method that sets the size of the scene—`initWithSize:` is the initializer of the **`SKScene`** superclass—and fills in the `currentLevel` property. The scene needs to know the current level number to transition from one level to another. You'll use the level number to load the appropriate **levels.plist** into an `NSDictionary` object. With the information provided in that `NSDictionary`, you can initialize the level properly.

You also need to change the **`PSKGameViewController`** class to make use of this new initializer method. Change the following line in **PSKGameViewController.m**:

```
PSKLevelScene *scene = [PSKLevelScene
                            sceneWithSize:skView.bounds.size];
```

To this:

```
PSKLevelScene *scene = [[PSKLevelScene alloc]
       initWithSize:skView.bounds.size level:self.currentLevel];
```

This takes the level number from the button that you tapped in the level selection screen—see the code in `LevelSelectViewController` if you're curious how this works—and passes it to the new `PSKLevelScene` object.

Note: The `SKView` class is a special type of `UIView` (or `NSView` on OS X) that deals with an `SKScene` object. It contains methods for loading and presenting a scene, transitioning between scenes, pausing and resuming the game state, converting between coordinate systems, displaying debugging information and more. Once you are comfortable with Sprite Kit, check out the docs to see all of `SKView`'s functionality.

https://developer.apple.com/library/ios/documentation/SpriteKit/Reference/SKView/Reference/Reference.html

The provided starter project has a few properties set on the `SKView` object—`showsFPS` and `showsNodeCount`. These are handy for debugging Sprite Kit games. The bottom right of the screen shows the FPS (frames-per-second) and the number of nodes being rendered. When the game is ready for release

and you want to remove these indicators, just comment out those two lines of code.

Keep in mind that typically you would initialize a view controller inside `viewDidLoad`. That's also where the default Sprite Kit Game template sets up the scene. However, at that time in the view controller's lifecycle, the view doesn't yet have the correct size or orientation. To give the `SKView` object the right size, you need to put the code to initialize the scene inside `viewWillLayoutSubviews` instead. This is especially important when your game is in landscape mode.

Loading the music

Next, you will load the music from the level data. For this you need to import the `SKTAudio` class. Add the following line at the top of **PSKLevelScene.m**, either before or after the other `#import` statements:

```
#import "SKTAudio.h"
```

`SKTAudio` is one of the helper classes that's a part of the `SKTUtils` library of helper methods. The Ray Wenderlich Tutorial Team built these methods for the book *iOS Games by Tutorials* to make common tasks much easier in Sprite Kit. The methods shorten the number of calls to do things like load music and work with `CGPoints`. This library is already included in the starter project.

Now modify `initWithSize:level:` as shown below:

```
//1
NSString *path = [[NSBundle mainBundle]
                  pathForResource:@"levels" ofType:@"plist"];
NSDictionary *allLevelsDict = [NSDictionary
                           dictionaryWithContentsOfFile:path];

//2
NSString *levelString = [NSString stringWithFormat:
                         @"level%ld", (long)self.currentLevel];
NSDictionary *levelDict = allLevelsDict[levelString];

//3
NSString *musicFilename = levelDict[@"music"];

//4
[[SKTAudio sharedInstance] playBackgroundMusic:musicFilename];
```

This code block does the following:

1. First, it loads **levels.plist** into an `NSDictionary` object named `allLevelsDict`.
2. Then it creates a string by taking the level index and appending it to the end of the word "level". This is the key that retrieves the dictionary for the current level. Using that key, the code loads `levelDict` with the data for the current level.

3. Next the code uses the "music" key to retrieve the filename for the MP3 that is the background track for this level.
4. Finally, it gets the `SKTAudio` singleton and starts playing the background music.

That's it! Build and run and select a game level. You get a blank screen—but at least you can hear some groovy tunes playing!

Parallax backgrounds

Next on the agenda: loading a parallax background. You'll use another open source piece of code I created and made available on Github, `JLGParallaxNode`. This is a simple parallax layer, modeled very closely after Cocos2D's `CCParallaxNode` class. `JLGParallaxNode` is included in the starter project.

A parallax node is a node whose children change position at a ratio relative to the parent node. You can add multiple children, all with different ratios. Parallax nodes are most often used as the background for a game. By moving the layers in the back at speeds slower than the foreground layers, you can create an illusion of depth in a 2D world.

Imagine yourself driving down the freeway. The fence next to the road appears to be moving by very quickly, the pasture with cows a little farther away moves by more slowly and the mountains in the background don't appear to move at all. You can simulate this experience using a parallax background with multiple layers moving at different speeds.

Before you can add the new parallax node, you need to create a main node that you can move around. Consider how in a game like *Super Mario Bros.*, the character stays at or close to the center of the screen and the map moves to progress the character through the level. You need to add a node that serves this purpose for both the map and parallax nodes.

First, create a new property in the `@interface` block of **PSKLevelScene.m**:

```
@property (nonatomic, strong) SKNode *gameNode;
```

Then add the following lines to `initWithSize:level:`, after the call to play the background music:

```
self.gameNode = [SKNode node];
[self addChild:self.gameNode];
```

This code creates the `gameNode` object and adds it to the scene. This main node serves as the container for all other nodes in the game.

> **Tip:** `node` is a shortcut for `alloc/init`.

Now you can create the parallax node. Add a new import to the top of the file:

```
#import "JLGParallaxNode.h"
```

And add this new method:

```
- (void)loadParallaxBackground:(NSDictionary *)levelDict {
  //1
  JLGParallaxNode *parallaxNode = [JLGParallaxNode node];

  //2
  NSArray *backgroundArray = levelDict[@"background"];

  //3
  for (NSArray *layerArray in backgroundArray) {

    // 4
    CGFloat indexOfLayer = [backgroundArray
                                indexOfObject:layerArray] + 1.0;
    CGFloat ratio = (4.0 - indexOfLayer) / 8.0;
    if (indexOfLayer == 4.0) {
      ratio = 0.0;
    }

    for (NSString *chunkFilename in layerArray) {

      // 5
      SKSpriteNode *backgroundSprite = [SKSpriteNode
                        spriteNodeWithImageNamed:chunkFilename];

      // 6
      backgroundSprite.anchorPoint = CGPointMake(0.0, 0.0);

      // 7
      NSInteger indexOfChunk = [layerArray
                                    indexOfObject:chunkFilename];

      // 8
      [parallaxNode addChild:backgroundSprite
           z:-indexOfLayer
           parallaxRatio:CGPointMake(ratio, 0.6)
           positionOffset:CGPointMake(indexOfChunk * 1024, 30)];
    }
  }

  // 9
  [self.gameNode addChild:parallaxNode];
  parallaxNode.name = @"parallax";
  parallaxNode.zPosition = -1000;
}
```

1. First, you create a new parallax node.

2. Next, you retrieve the background information from the dictionary for this level and store it in the `backgroundArray` object.

3. The `backgroundArray` object contains an array hierarchy that is two levels deep. The first level is the list of the different layers that make up the background. Each layer in turn is broken into multiple images.

   OpenGL, and therefore Sprite Kit, has a maximum texture size. This code assumes it to be 2048×2048 pixels, which is what the iPhone 4 supports. On newer devices that limit is larger, and on the iPhone 3G and older devices it is lower. Of course, those older iPhones won't run iOS 7 and therefore won't run Sprite Kit.

   The entire background image is much larger than this maximum texture size, so you have to tile the background in 2048-pixel-wide chunks (1024 points wide on Retina screens), which is why you're using a loop to load multiple images into each layer. However, each layer moves at a slower rate than the main tile map for that level, so each parallax layer is smaller than the layer in front of it.

   The background in *Pocket Cyclops* is by far the largest texture and takes up the most memory. The sprite atlas is less than 1024×1024 and contains all the other art assets, including the buttons and HUD. The background images are many times that size—you'll see in Xcode that the memory usage will shoot up to over 100 MB of RAM after loading these textures. Loading all of these textures is also what causes the small delay between tapping the level select button and the game's appearance.

> **Note:** Originally, I included Retina resolution versions of the textures that make up the background layers. However, using these textures pushes the memory usage to nearly 300 MB of RAM. On the iPhone 4 and 4S, you can run only one level at that memory usage. When it's time to transition to a second level or if anything else happens in the app that requires substantial extra memory, the app runs out of memory and crashes.
>
> For this reason, this version of the starter kit includes only the non-retina background textures.

   In your own game you may not have as much memory to spare, so you might find that using fewer textures to create a background makes sense for you. While there are ways you could manage the high memory use by loading and unloading some of these textures as needed, in this game you'll load all these textures at once at the start of the game, so they will be instantly available as your player advances.

4. Each parallax layer moves at its own speed. To calculate this speed, you first find the position of the layer within the parallax background. With this value you can assign a ratio to all the image chunks within that layer; this varies between 1/2 and 1/8th of the speed of the foreground tile map.

5. You load the image for the next chunk in an `SKSpriteNode` object.
6. You set the `anchorpoint` at (0, 0) so that you can easily set the bottom-left corner of each background layer as equal to the bottom-left of the coordinate system. This makes lining up the seams between each image chunk a little easier because you can use the start and end points of the images instead of their centers.
7. You find the index of the filename within the array so that you can position the image in the appropriate place. You set up all the images within an individual layer end-to-end. The `indexOfChunk` variable holds the relative position of the filename within the array and therefore, the position of the image within that set of images.
8. Here you add the image as a child of the parallax node. Using the method `addChild:z:parallaxRatio:positionOffset:`, you align each of the image chunks and give them the appropriate speed and z-position.
9. Once the loop has finished with all the image tiles, you add the parallax node to the main `gameNode` object.

   Finally, you set a `zPosition` for the parallax node. The `zPosition` controls which nodes are drawn in front of other nodes. Setting this property to a high negative value means this node will be drawn first, and everything else will be drawn on top of it. -1000 is just an arbitrary large, negative number. As long as the `zPosition` of the other nodes is greater than that value, they'll be drawn on top of the parallax node.

Now you need to call this new method and pass in the `levelDict` for the current level. Add this line after the line that adds the `gameNode` to the scene:

```
[self loadParallaxBackground:levelDict];
```

You could build and run now, but nothing is moving, so you won't be able to enjoy the visual impact of parallax scrolling. Add the following method to slowly move the `gameNode` backward—effectively moving the camera forward—to see what it looks like. This is optional and you'll overwrite this method later, once you have the camera following the player.

Add this method:

```
- (void)update:(NSTimeInterval)currentTime {
  self.gameNode.position = CGPointMake(
      self.gameNode.position.x - 2.0, self.gameNode.position.y);
}
```

I'll cover `update:` in depth later, but for now know that this code simply moves the `gameNode` object backward 2 pixels every frame, 60 times per second. You should be able to run at full speed so far, even on an iPhone 4.

Build and run now. You can see the parallax layers slowly moving at different speeds.

You've familiarized yourself with the starter project and have written an important bit of code to get you started. It's time for a well-deserved break!

Note: These are quite large textures and piling four layers of full screen moving images on top of one another can tax the performance of a mobile device. Specifically on the iPhone 4, the oldest phone that will run Sprite Kit, you'll find that once you get all the other pieces of *Pocket Cyclops* in place, you'll drop to about 45 FPS. If you take a look at the gauges included in Xcode 5 (also once you get all the other pieces of *Pocket Cyclops* in place), you'll see that the GPU is running at full capacity.

You have a few options to deal with this constraint:

1) You can remove one of the layers on the iPhone 4.

This will bring the FPS back up and the utilization down from 100% to about 80%.

2) You can set the frame rate to 30fps.

The `SKView` has a `frameInterval` property. If you set this to 2.0, it will cause the game to run at 30fps instead of 60fps. Many games run at 30fps and still look fine. This option does have some implications for your game logic, which you'll see later.

Because most games won't include such large background textures, I'm not going to tell you which option you should choose, but by the time you are finished with this book, you'll have enough understanding to make that decision when it comes to your own game.

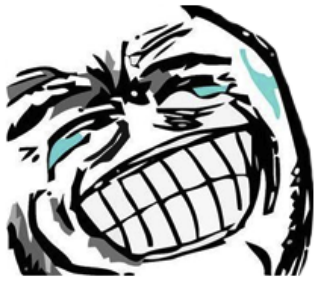
In the next chapter, the real fun begins! You'll add the tile map as well as the cyclops, the star of the game. And you'll start on the core of the platformer: the physics engine!

> **Challenge:** Play around with different parallax ratio numbers on the layers to see how to get the best 3D effect.

Chapter 2: A Budding Physics Engine

In this chapter, you'll begin the process of writing your own physics engine.

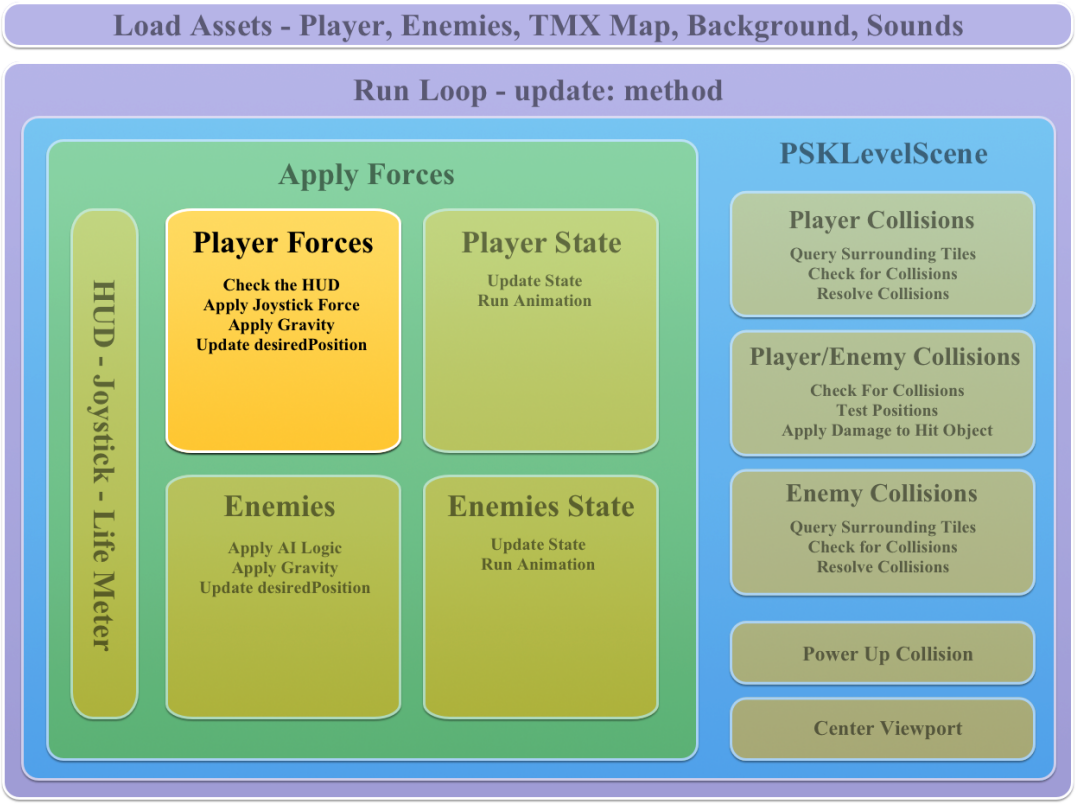
This chapter focuses specifically on applying forces to the `Player` object. You won't check the state of the controls on the HUD quite yet, but you will learn how to effectively store forces in vectors, specifically in `CGPoint` structures, and how to work with and manipulate those forces.



It is all unfolding according to my design!

But first, to get to that point, you'll load some TMX tile map objects from files. These objects contain information about both the tiled level and the positions of the player, enemies and powerup objects. You'll also start to build the class hierarchy that the first chapter discussed.

Here's the portion of the overall plan that this chapter covers—everything except the checking the HUD:



The Tao of physics engines

A platform game revolves around its physics engine, and in this tutorial you'll create your own physics engine from scratch.

There are two main reasons why you'll roll your own rather than use Sprite Kit's built-in physics engine, which is based on the open-source Box2D engine:

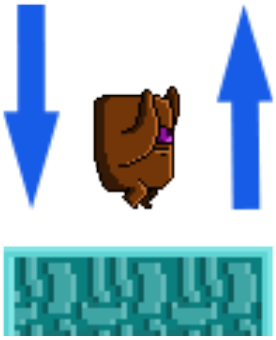
1. **Fine-tuning**. To give a platformer game the right feel, you need to be able to fine-tune the engine's response. In general, platformers created using pre-existing engines don't feel like the classic games you're used to.
2. **Simplicity**. Box2D has a lot of extra capabilities that your game engine doesn't need. So your leaner homebrew engine will end up being less resource-intensive overall.

A physics engine does two important things:

1. **Simulate movement**. A physics engine's first job is to realistically, for certain values of reality, simulate forces like gravity, running, jumping and friction.

   For example, in *Pocket Cyclops* you'll apply an upward force to the cyclops to make it jump. Over time, the force of gravity acts against that initial upward force, which will give you that nice classic parabolic jump pattern.

2. **Detect collisions**. The physics engine's second job is to detect and resolve collisions between objects in the simulation.



For example, when the cyclops hits the ground, collision detection will keep it from falling through. You'll also use collision detection when the cyclops lands on, or bumps into, an enemy.

Let's take a look at the steps required for the physics engine to function.

Your physics algorithm

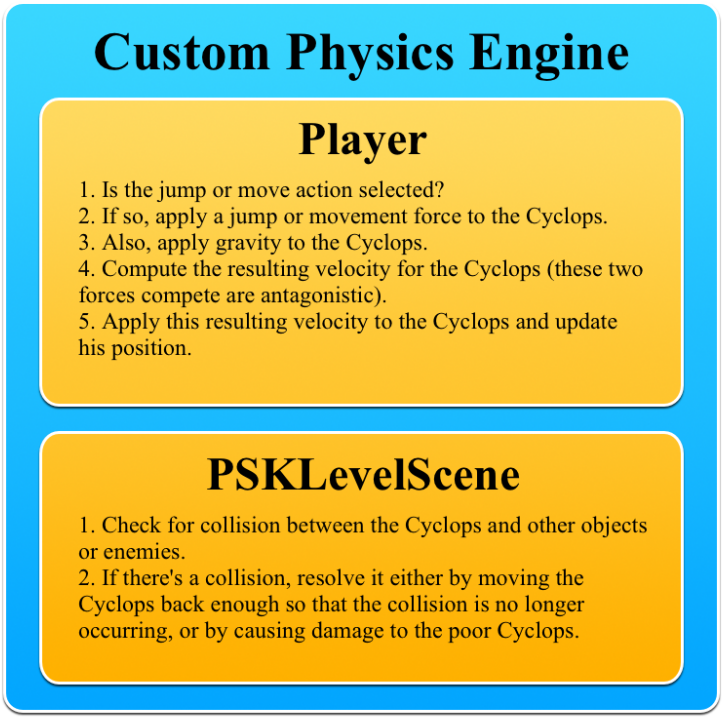
In the physics engine you'll create, the cyclops will have its own movement-describing variables: current velocity (speed and direction), acceleration and position, among others. Using these variables, every movement you apply to the cyclops will follow this algorithm:

1. Is the jump or move action selected?
2. If so, apply a jump or movement force to the cyclops.
3. Apply gravity (another force) to the cyclops.
4. Compute the resulting velocity for the cyclops. The jump or movement force and gravity compete with each other, so you'll have a final resulting force that either pushes upward or pulls downward.
5. Apply this resulting velocity to the cyclops and update its position.
6. Check for a collision between the cyclops and other objects or enemies.
7. If there's a collision, resolve it by either causing damage to the poor cyclops, or moving the cyclops back enough so that the collision no longer occurs.

You'll run these steps for every frame.

In the game world, gravity is constantly pushing the cyclops down and into the ground, but the collision resolution step puts the cyclops back on top of the ground in each frame. You can also use this feature to determine if the cyclops is still touching the ground, and if not, disallow a jump action—because it means the cyclops is already in mid-jump or has just walked off a ledge.

Steps 1-5, as shown in the image below, occur solely within the `Player` class (the cyclops). All the necessary information is contained there and it makes sense to let the cyclops update its own variables.



However, when you reach the sixth step—collision detection—you need to take all of the level features into consideration, such as walls, floors, enemies and other hazards. So you want the `PSKLevelScene` to perform the collision detection in each frame—remember, that's the `SKScene` subclass that will do a lot of the physics engine work.

If you allowed the cyclops to update its own position, it's possible that the cyclops would move itself into a collision with a wall or ground block, and the `PSKLevelScene` would then move him back out, repeatedly. If a draw occurred between these steps, the cyclops would look as if it were vibrating. (Had a little too much coffee, cyclops?)

So you're not going to allow the cyclops to update its own position. Instead, the cyclops will have a new variable, `desiredPosition`, that it will update. The `PSKLevelScene` will check if this `desiredPosition` is valid by detecting and resolving any collisions, and then the `PSKLevelScene` will update the position of the cyclops.

Got it? You'll soon see what this looks like in code, but first, let's do some more game scene set-up.

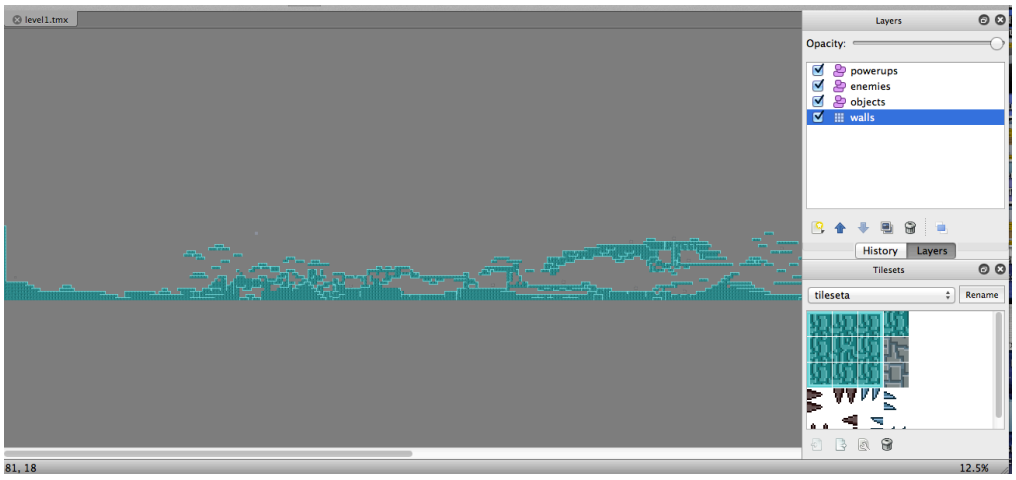
In the beginning, there was a tile map

It's TMX time! Physics wouldn't be much use without geography. The next step in the process is to bring in the tile map.

I'm going to assume you're familiar with how tile maps work. If you aren't, you can learn more about them in this tutorial:

- http://www.raywenderlich.com/1163/how-to-make-a-tile-based-game-with-cocos2d

Take a look at the level by starting the Tiled map editor (download it from http://www.mapeditor.org if you don't have it already). Open **level1.tmx** from the **Level Data** folder in the project directory. You'll see the following:

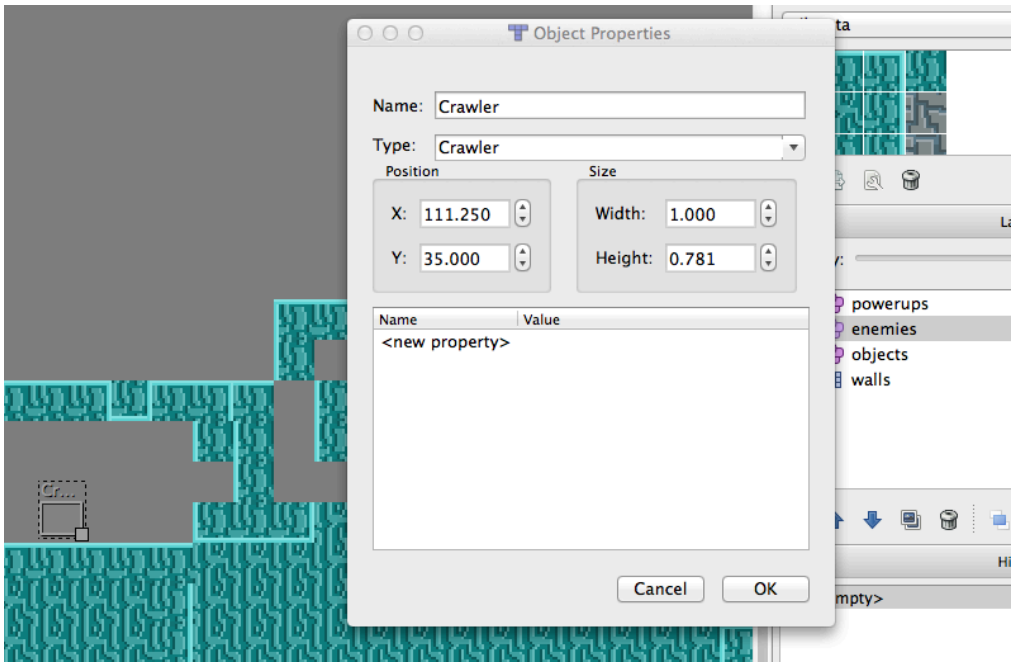


Take a look at the sidebar. There are four kinds of layers in this level file:

1. **Powerups**: This layer contains the objects that grant the cyclops additional powers.
2. **Enemies**: This layer contains the various enemies.

3. **Objects**: This layer contains level features, like the starting point where the cyclops enters the layer and the exit point that triggers the end of the level.
4. **Walls**: This is the main layer, containing all the level's walls—or platforms and ground, if you prefer.

To create enemies, objects and powerups, I added objects (the gray rectangles) to the corresponding layer and gave them particular names, types and properties. For example, if you right-click on one of the enemy objects and choose **Properties**, you will see it has a name and type of Crawler:

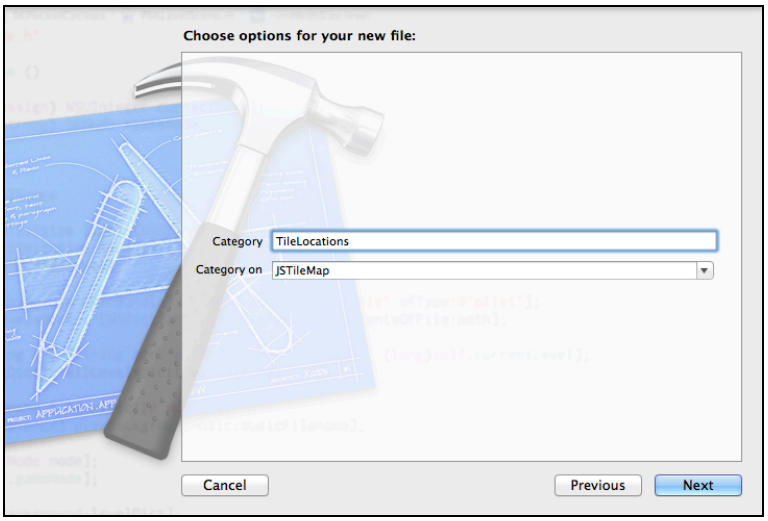


Later on, you will add code to look for objects with these names and types.

It's time to add the tile map to the game.

You need a number of helper methods to facilitate the physics calculations on the tile map. It makes sense to place these methods in a category on `JSTileMap`. Categories are an alternative to subclassing in Objective-C, allowing you to add methods to a class without having to create a subclass. A category is of lighter weight than a subclass and saves you time when you only want to add some logic to a class.

Open the project in Xcode and create a new class by going to **File->New->File**. Choose the **iOS\Cocoa Touch\Objective-C category** class template. Give it the name **TileLocations** and make it a category on `JSTileMap`. The default location for this new class should be the same folder as the other files, and that's fine.



You'll add to this category as you move forward. For now, use it as-is.

Next, import the class into **PSKLevelScene.m**:

```
#import "JSTileMap+TileLocations.h"
```

Add a property for this new object to the class extension (the `@interface` block):

```
@property (nonatomic, strong) JSTileMap *map;
```

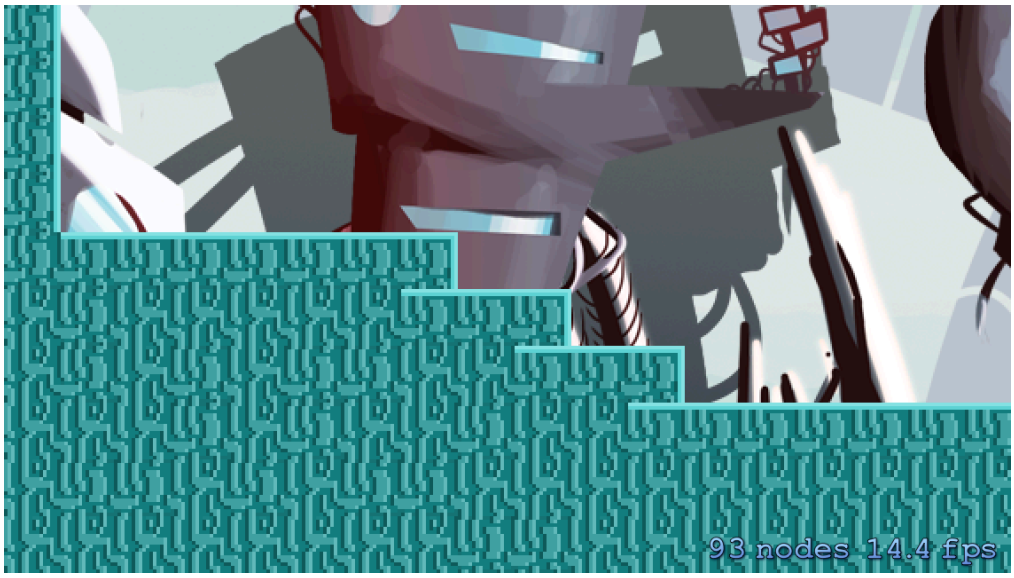
To put the tile map into the game, add the following code to `initWithSize:level:`, right after the code that creates the parallax node:

```
NSString *levelName = [levelDict objectForKey:@"level"];
self.map = [JSTileMap mapNamed:levelName];
[self.gameNode addChild:self.map];
```

In the above code, you retrieve the name of the TMX file from the `levelDict` object. Using that filename, you instantiate the map object with a method called `mapNamed`. Then you add that object to the `gameNode`.

When you begin moving things around, you'll move the `gameNode`, which in turn will move the tile map, all its children—including the enemies and player—and the parallax node, all in one command. This is the power of using a node tree structure. When you apply a position, scale or rotation, all the children of that node inherit these values.

Build and run now, and your game levels should have tiles:



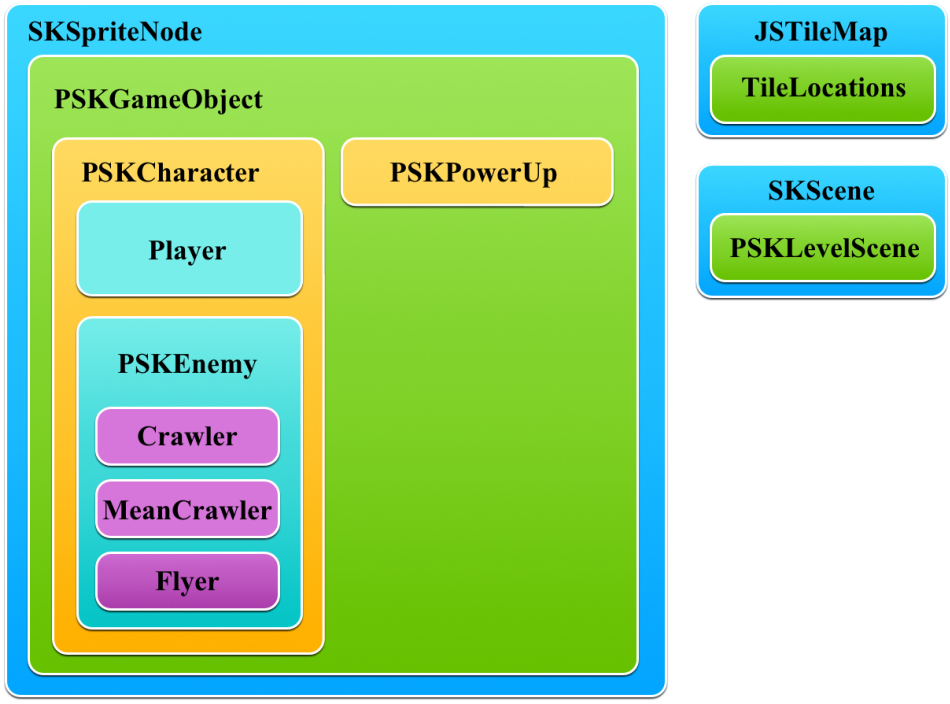
Note: If you see that some of my screenshots have low FPS numbers in the debug readout at the bottom right, it's because I'm running in the Simulator. For OpenGL tasks—things like Sprite Kit and Core Image—performance will often be worse on the Simulator than on the device. This may surprise you—the desktop has so much more power! It's because the Simulator is emulating OpenGL ES 2.0 using a software renderer, instead of using the desktop's GPU to hardware accelerate the drawing.

Building the class heirarchy

Now that you've got a place for the cyclops to stand, there's nothing keeping you from adding your hero to the layer!

Well, except that to accomplish this, you first need to create a hierarchy of classes. It might seem tedious and unnecessary if you aren't used to doing things this way. You'll come to see, though, that in the end you'll have written less code and have a much easier time maintaining that code.

The introductory chapter briefly covered the class hierarchy. You will now create those objects. Once again, here's the diagram showing how all the engine's classes fit together:



As you can see, all of the objects are children of the `SKSpriteNode` class. The highest-level child is `PSKGameObject`.

Create the `PSKGameObject` class now by going to **File->New->File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, set the subclass to `SKSpriteNode` and name it **PSKGameObject**.

This class will contain methods to load animations from XML or PLIST files and sprite sheets, which you'll add in a later chapter. The thing to know about the `PSKGameObject` class is that each of its subclasses will need to load animations, and the methods in the `PSKGameObject` class will help do that.

The next class you need to create is `PSKCharacter`. Both the cyclops and enemy classes will inherit from the `PSKCharacter` class. As such, it will contain methods that are common to both kinds of objects.

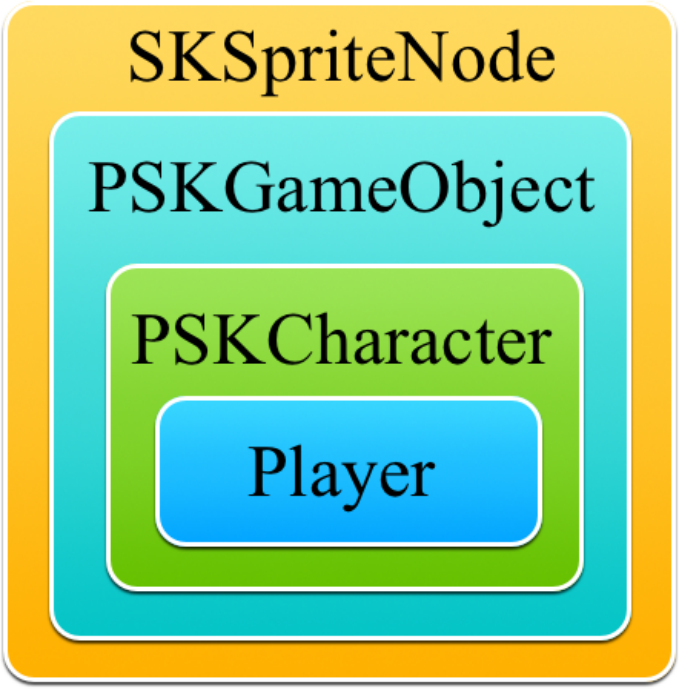
The `PSKCharacter` class will contain methods to:

- Calculate the bounding box of a `PSKCharacter` object.
- Respond to collisions with other `PSKCharacter` objects.
- Control changing the state of the character (attacking vs. resting).

Right now, though, you just need to create an empty subclass to be there when you're ready to start adding code to it.

Add the `PSKCharacter` class by going to **File->New->File**. Select the **iOS\Cocoa Touch\Objective-C class** template, make it a subclass of `PSKGameObject` and call it **PSKCharacter.**

As you progress through this tutorial, you will periodically return to these for-now-empty classes to add attributes and methods. Here's what the subclass tree looks like so far:



Yes, I know—you don't yet have the `Player` class in place. Guess what you're going to do next? ☺

Adding the player character

Are you ready to incarnate your hero?

The `Player` class will be a subclass of `PSKCharacter`, so create a new file by going to **File->New->File**. Select the **iOS\Cocoa Touch\Objective-C class** template, make it a subclass of `PSKCharacter` and call it **Player**. You're dropping the PSK prefix for the lowest-level classes: `Player`, `Crawler`, `Flyer`, `MeanCrawler` and `PowerUp`. These classes won't be subclasses themselves—unless you choose to make them so in your own platformer game!

That's all for the **`Player`** class for now, but you'll give it a bunch of code later on in this very chapter.

Now return to **PSKLevelScene.m** and import the new `Player` class header:

```
#import "Player.h"
```

Next, add a property to the `@interface` section:

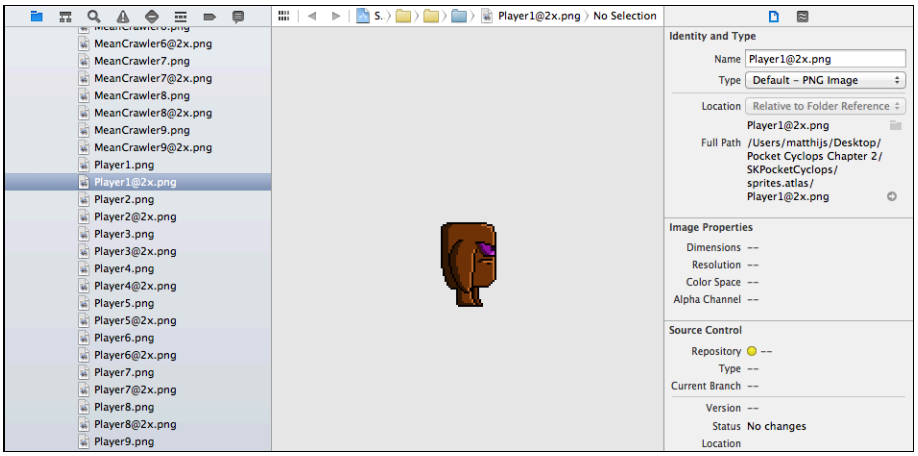
```
@property (nonatomic, strong) Player *player;
```

Now make the player a child of the map by adding the following code after the lines that add the map to the scene:

```
self.player = [[Player alloc] initWithImageNamed:@"Player1"];
self.player.zPosition = 900;
[self.map addChild:self.player];
```

The first line uses a common initializer to create a new sprite node based on a single image file. This code creates a sprite with the **Player1.png** image. The .png extension is assumed and can be omitted in Sprite Kit.

You can navigate to the **sprites.atlas** folder to see what that image looks like:

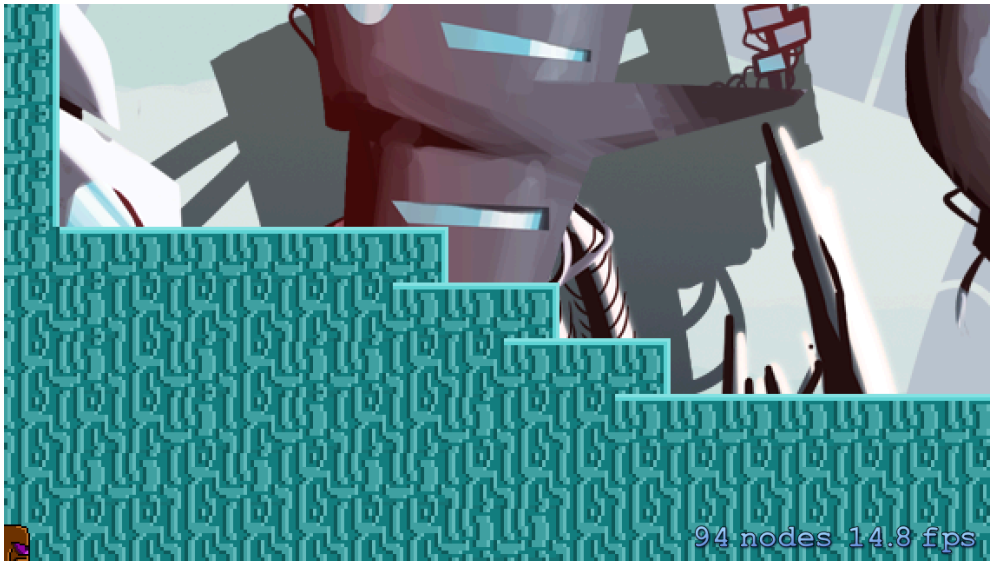


After creating the sprite, you set its `zPosition` and add it to the map node.

> **Note:** Enemies and the player will be children of the map, so you want to make sure they appear in front of all the other layers. In this case it's mostly the wall layer that concerns you. In more complex hierarchies, you'd probably want most of your tile layers behind the objects that move around. `JSTileMap` will print the `zPositions` of each layer to the console so you can see at runtime how they are positioned with respect to each other.
>
> For more information about `zPosition`, see the Sprite Kit documentation.

You are using a very high `zPosition` value to make sure the `Player` shows up on top.

Build and run now, and you should see the player on the screen!



Can you spot him? Hah! The cyclops is hiding in the bottom-left corner.

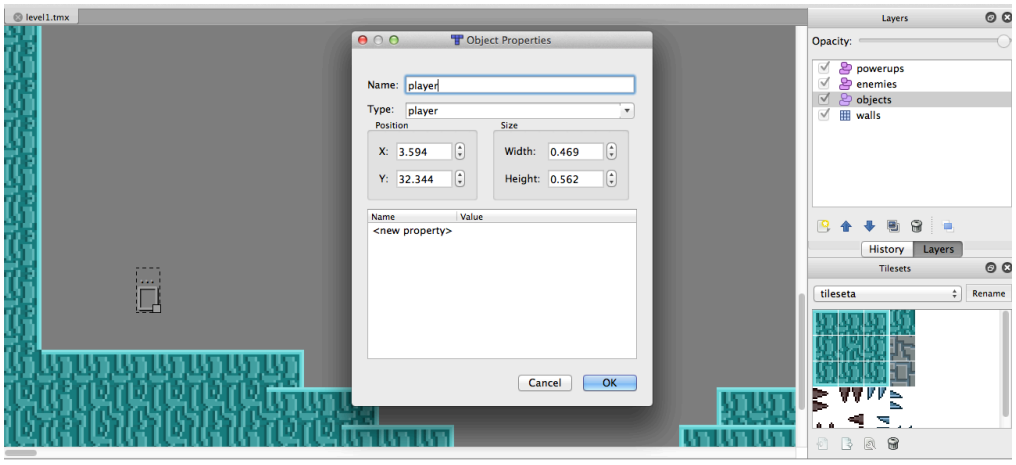
> **Note:** If your parallax background is still moving, then remove the `update:` method from **PSKLevelScene.m** (you added that code earlier for testing purposes and don't need it anymore).

Setting a character's initial position

You need to set the `Player` object's initial position. That's one of the pieces of information stored in the **objects** layer in the tile map. You can retrieve that information from the tile map to get the position.

If you want to understand how that information got in the tile map, load **level1.tmx** in Tiled again. Select the **objects** layer, right-click on the gray box at the beginning of the level and select **Object Properties**. That gray box represents the player's starting position.

Here's a screenshot from Tiled of the data for the gray box within the objects layer:



The gray box has a name and a type, both "player". It also has a position and a size. Right now you only need the name and the position. If you'd like to refresh your understanding of how to create objects like this in tile maps, you can read the following tutorial:

- http://www.raywenderlich.com/1163/how-to-make-a-tile-based-game-with-cocos2d

To retrieve the information about the player character's position, add the following code to `initWithSize:level:`, right after the code that creates the `Player` object:

```
TMXObjectGroup *objects = [self.map groupNamed:@"objects"];
NSDictionary *playerObj = [objects objectNamed:@"player"];
self.player.position = CGPointMake(
    [playerObj[@"x"] floatValue], [playerObj[@"y"] floatValue]);
```

First you retrieve the `TMXObjectGroup` for the objects layer from the tile map. You retrieve a group layer with the `groupNamed:` method.

Then you get the `Player` information as an `NSDictionary` by using the `objectNamed:` method. You can get the data you need by specifying the relevant key, which in this case is the "player" name from the Object Properties for the gray box.

The position values are stored in the x and y keys. You convert these values from `NSNumbers` to floats and use the values to position the `Player`. You can add arbitrary properties in Tiled if you want to specify other information about that particular object.

> **Note:** You're using the subscripting syntax to read values from the dictionary: `playerObj[@"x"]` instead of `[playerObj objectForKey:@"x"]`. This is one of the syntax improvements for Objective-C that was introduced in Xcode 4.5.

Build and run now. That old cyclops is right where it should be!

My, the cyclops doesn't look very happy to be there, does it?

All right, it's time to return to making the world obey the laws of physics! This will be a law-abiding game. ☺

The gravity of your situation

To build a platformer physics environment, you could write a complex set of branching logic that takes the `Player` object's state into account and decides which forces to apply based on that state.

But this would quickly become extremely complicated—and it isn't how physics works. In the real world, gravity is always pulling things toward the earth, so you'll add the constant force of gravity to the `Player` object in every frame.

Other forces don't switch on and off, either. In the real world, a force is applied to an object and the momentum continues to move the object through space until some other force acts on that object to change the momentum.

For example, a vertical force like a jump doesn't switch off gravity. It momentarily overcomes gravity, but over time gravity slows the ascent and ultimately brings the object back to the ground.

Similarly, a force that pushes an object parallel to the earth is countered by friction, which gradually slows down the object until it stops. Friction in the real world can be air friction or contact with a solid; in this game friction will be constant, but some games apply a different friction when the object is touching the ground and when it isn't.

That is how you'll model your physics engine. You won't repeatedly check whether your `Player` is on the ground and decide whether to apply gravity; you will always apply gravity.

The model of cumulative forces

As you just learned, the logic of your physics engine dictates that when a force is applied to an object, that object will continue to move until another force counteracts the original force.

This means that when the cyclops walks off a ledge, it will continue to fall at an accelerating rate until it collides with something else. When you move the cyclops left or right, it won't stop moving as soon as you stop applying force; momentum will carry the cyclops forward, while friction slows it down gradually until there's no momentum remaining and the cyclops stops.

As you continue building your platform game, you'll see that this logic makes it easier to handle complex situations like a slippery floor or a free-fall over a cliff. The model of cumulative forces makes for a fun, dynamic-feeling game.

It also makes implementation easier, because you don't have to constantly query the state of your objects—they will follow the natural laws of your world and their behavior will emerge from the application of those laws!

Sometimes you do get to play God! ☺

Enforcing the laws of physics

Let's define a few terms:

- A **force** is an influence that causes a change in speed or direction.
- **Velocity** describes how fast an object is moving in a given direction.
- **Acceleration** is the rate of change in velocity—how an object's speed and direction change over time.

In a physics simulation, a force applied to an object will accelerate that object to a certain velocity, and that object will continue to move at that velocity until acted upon by another force. Velocity is a value that persists from one frame to the next and only changes when new forces are applied to the object.

CGPoints and forces

You're going to represent three things with `CGPoint` structures: velocity (speed), force/acceleration (change in speed) and position. There are two reasons for using `CGPoint` structures:

- **They're 2D**. Velocity, force/acceleration and position are all 2D values for a 2D game. In other words, they are what mathematicians call vectors. "What?" you might say. "Gravity only acts in one dimension!" However, you could easily imagine a game with changing gravity where you'd need the second dimension. Think *Super Mario Galaxy*!

- **It's convenient**. By using `CGPoints`, you can rely on the various built-in functions provided by `SKTUtils`. You'll make heavy use of functions such as `CGPointAdd` (add two points), `CGPointSubtract` (subtract one point from another) and `CGPointMultiplyScalar` (multiply a point by a float to scale it up or down). This will make your code much easier to write—and debug!

Your `Player` object will have a velocity variable that a number of forces will act upon in each frame, including gravity, walk/jump forces supplied by the user and friction, which will slow (and eventually stop) the cyclops.

In each frame, you'll add all these forces together and then add the resulting cumulative force to the previous velocity of the `Player` object to get the current velocity. You'll scale down the current velocity to match the fractional time amount between each frame, and finally you'll use that scaled value to move the `Player` object's position for that frame.

> **Note:** If any of this sounds confusing, Daniel Shiffman wrote an excellent tutorial on vectors which explains the accumulation of forces structure you'll be using. The tutorial is designed for Processing, a language similar to Java for creative designers, but the concepts are applicable in any programming language. It's a great and accessible read and I highly recommend you check it out!
>
> http://www.processing.org/learning/pvector/

Enforcing gravity

Start with gravity, since it's a constant force that you will always apply to all objects.

First you need to add the velocity attribute. Remember when you created all those classes and didn't put anything into them? You were preparing for this moment. ☺

You'll put the velocity variable into the `PSKCharacter` class, because it is common to all moving game objects. In **PSKCharacter.h**, add this right before the `@end`:

```
@property (nonatomic, assign) CGPoint velocity;
```

The next step is to add a method to update the state and move the objects around each frame. This method will ultimately go in all `PSKCharacter` subclasses.

`SKScene` has a number of methods that are called each frame before the render step, which is when everything gets drawn. One of those methods is `update:`. You'll use this method to update all the other objects.

> **Note:** There are two other callback methods in Sprite Kit. One is called after the built-in Sprite Kit physics engine finishes its physics simulation,

`didSimulatePhysics:`. This game doesn't use the Sprite Kit physics engine, so nothing will have changed at that callback point.

The final callback, `didEvaluateActions:`, happens after all the `SKAction` methods have been processed.

These three methods allow you to control where you handle your game's logic within the constructs of the physics simulation and its actions so that you have what you need to make custom adjustments at each stage of the process.

Move to **PSKLevelScene.m**, where you need to add a new property to keep track of the amount of time that passes between each call to `update:`.

```
@property (nonatomic, assign) NSTimeInterval previousUpdateTime;
```

Now add the new update method:

```
- (void)update:(NSTimeInterval)currentTime {
  //1
  NSTimeInterval delta = currentTime - self.previousUpdateTime;
  self.previousUpdateTime = currentTime;
  //2
  if (delta > 0.1) {
    delta = 0.1;
  }
  //3
  [self.player update:delta];
}
```

Let's go over this piece by piece:

1. The first step is to create a new variable named `delta`. This variable holds the length of time that has passed since the last call to `update:`. This time interval might change, so to have a constant speed and consistent physics, you need to scale the physics engine to the current time interval.
2. In the next step, you check the size of `delta`. If `delta` is too large, then you can have really undesirable behavior, like the cyclops moving through an entire tile before the collision logic is called. That often happens at the very beginning of the game, before anything has even been drawn to the screen. There are several ways to handle this situation. You're handling it by limiting `delta` to a maximum.

   Your game should run at least at 30fps, so any `delta` larger than 0.033 is too large, but for one or two frames at the beginning of the game, you will have these large `delta`s. If there are other reasons you anticipate your game dropping below 30fps, you should not rely on this max `delta`. As I said, using this code, your physics engine will run inconsistently any time `delta` goes above 0.1.
3. Finally, you call `update:` on the player object and pass `delta`. Xcode gives an error telling you that `Player` doesn't have an `update:` method. You can fix that!

Instead of putting the method declaration into the `Player` class, add it to its superclass, `PSKCharacter`. Every `PSKCharacter` subclass will have an `update:` method, though, of course, each of these methods will have a different implementation.

Add this method declaration to **PSKCharacter.h** in the `@interface` section before the `@end` line:

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

To quiet the warning you get when you create a method declaration without an implementation, add this empty implementation of `update:` to **PSKCharacter.m**:

```
- (void)update:(NSTimeInterval)dt {
  //override this method
}
```

You need some of the convenience methods that deal with `CGPoints` in many of these subclasses, so import `SKTUtils` into **PSKCharacter.h** as well:

```
#import "SKTUtils.h"
```

The individual implementations of `update:` will be in the subclasses. `Player` is the first of these subclasses, so add the following methods to **Player.m**:

```
// 1
- (id)initWithImageNamed:(NSString *)name {
  if (self = [super initWithImageNamed:name]) {
    self.velocity = CGPointMake(0.0, 0.0);
  }
  return self;
}

- (void)update:(NSTimeInterval)dt {
  // 2
  CGPoint gravity = CGPointMake(0.0, -450.0);
  // 3
  CGPoint gravityStep = CGPointMultiplyScalar(gravity, dt);
  // 4
  self.velocity = CGPointAdd(self.velocity, gravityStep);
  CGPoint stepVelocity = CGPointMultiplyScalar(
                                          self.velocity, dt);
  // 5
  self.position = CGPointAdd(self.position, stepVelocity);
}
```

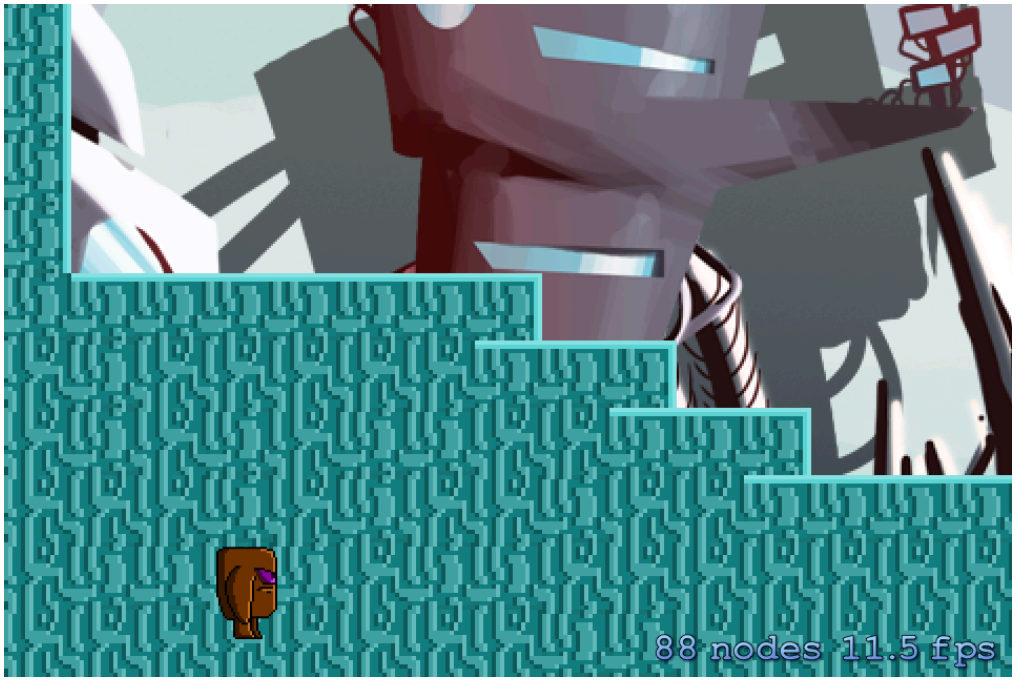
Let's go through the above code section by section:

1. You override the standard `initWithImageNamed:` method from `SKSpriteNode` to set the velocity property to (0,0).
2. Here you declare the value of the gravity vector. Gravity describes acceleration: for every second of time, you accelerate the velocity of the cyclops 450 points

toward the floor. If the cyclops starts from a standstill, it'll be moving at 450 points/second at the one-second mark, 900 points/second at the two-second mark and so forth. That may seem fast, but the cyclops is only going to move a fraction of that each frame.

3. Next, you use `CGPointMultiplyScalar` to scale the acceleration down to the size of the current time step. Recall that `CGPointMultiplyScalar` multiplies a `CGPoint`'s values by a float value and returns the result as a new `CGPoint`. This way, even when you're faced with a variable frame rate, you'll get consistent acceleration.
4. Once you've calculated the gravity for the current step, you add it to your current velocity. What results is the velocity for a single time step. Again, you're doing these calculations in order to get consistent velocity no matter the frame rate.
5. Finally, you use `CGPointAdd` to add the velocity for this time step to the old position to calculate the cyclops's updated position.

With that, you are well on your way to writing a physics engine! Build and run now to see the result:



See how the cyclops falls right through the floor? It won't get very far that way. You need collision detection to put solid ground beneath the cyclops's feet, and that's what you'll implement in the next chapter.

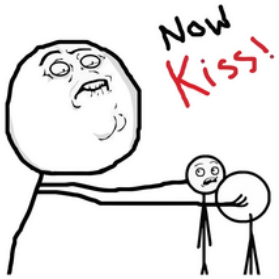
You, on the other hand, have gotten a lot farther than the cyclops. Give yourself a pat on the back. This chapter and the next cover the most difficult concepts in this book and you're already about halfway through it!

> **Challenge:** Variable gravity could make the game a lot more interesting. How would you implement it?

You could use the device’s accelerometer or have a button that would let users alter gravity themselves. What would the code for that look like?

Chapter 3: Collisions

As you learned in the last chapter, a physics engine is responsible for two important things: simulating movement and detecting collisions. The last chapter focused on simulating movement for the cyclops through gravity. This chapter focuses on adding collision detection and resolution into the game!



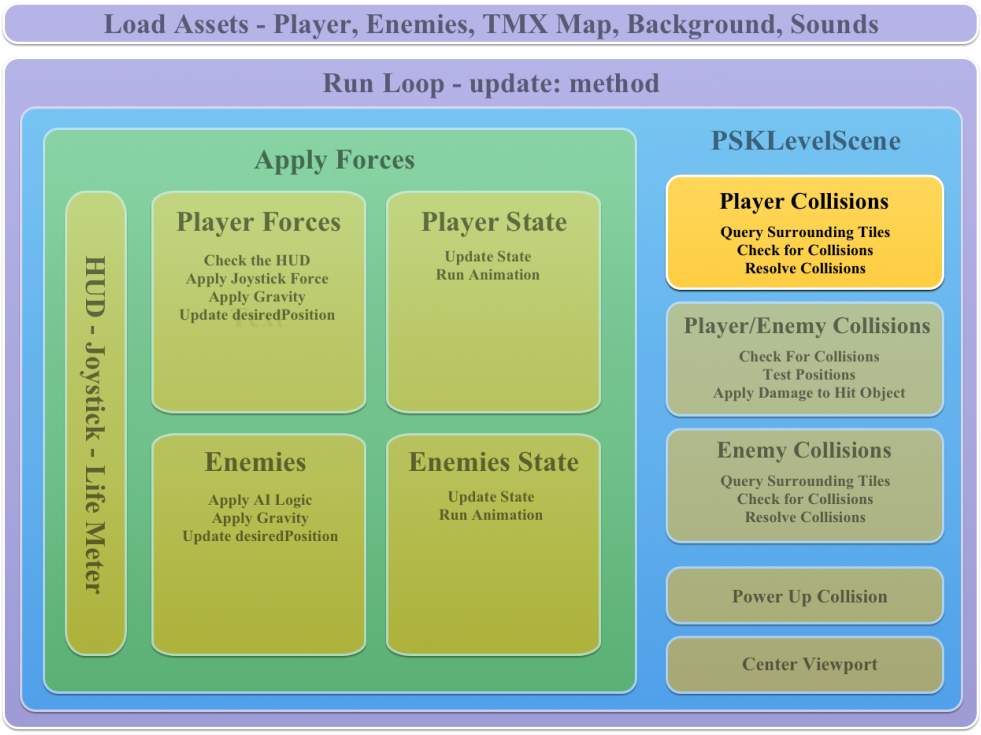
You will add the collision handling code to the `PSKLevelScene`. The `PSKLevelScene` has access to the tile map, the player, the enemies and all the other objects that you want to check for collisions.

You need some methods to query the level for tiles that will generate collisions. You'll add these to `JSTileMap`, in your class category `JSTileMap+TileLocations`.

You'll also examine different ways of resolving collisions and learn which is the most appropriate for your platformer game.

Finally, you'll set up certain flags, like the `onGround` property, that individual classes will use to determine their state and valid transitions to a new state. You can't jump if you aren't on the ground—at least not in *Pocket Cyclops!*

Here's where this chapter falls in your overall game map:



Collision detection mechanics

Collision detection is a fundamental part of any physics engine. There are many different kinds of collision detection, from simple bounding box detection to complex 3D mesh intersects. Lucky for you, a platformer like *Pocket Cyclops* doesn't need a very complicated collision detection engine.

While you need to detect collisions for both the player character and its enemies, let's focus on the cyclops to keep things simple.

To detect collisions for the cyclops, you need to query the tile map for the immediate surrounding tiles. Then you'll use a few built-in iOS functions to test whether the character's bounding box is intersecting with a given tile's bounding box.

> **Note**: What is a bounding box? It's simply the smallest axis-aligned rectangle that a sprite can fit inside. Usually this is the same as the frame of the sprite, including transparent space, but when a sprite is rotated, it gets a little tricky. Don't worry—Sprite Kit has a helper method to calculate this for you. ☺

The functions `CGRectIntersectsRect` and `CGRectIntersection` make these kinds of tests simple:

- `CGRectIntersectsRect` tests if two rectangles intersect.

• `CGRectIntersection` returns the intersecting `CGRect`.

First, though, you need to find the bounding box of your cyclops. Every sprite loaded has a bounding box that is the size of the texture and is accessible via the `frame` property. However, you'll usually want to use a bounding box that is smaller than the default value.

Why? Most textures have some transparent space near the edges. You don't want to register a collision when an object overlaps this transparent space—only when it overlaps visible pixels.

Sometimes you'll even want the pixels of the object and the player sprite to overlap a little bit. Think back: when Mario is unable to move further into a block, is he just barely touching it, or do his arms and nose encroach just a little bit into the block? The correct answer is the latter. ☺.

First, you'll add a method that returns the bounding box of the cyclops. Other objects will need this same method—for instance, enemies also need to collide with the walls and floors of the level. So you want to add this method in a common place.

This is also your first chance to make use of the class hierarchy you set up earlier. Can you guess where the new method should go?

If you guessed the `PSKCharacter` class, you are correct! All classes derived from the `PSKCharacter` class move around and need to collide with the ground and with each other—making it a great place for this method.

Personal space: a bounding box

Add the following method declaration to **PSKCharacter.h**:

```
- (CGRect)collisionBoundingBox;
```

Add the method implementation to **Character.m**:

```
- (CGRect)collisionBoundingBox {
  return self.frame;
}
```

The base version of `collisionBoundingBox` returns the sprite frame's rectangle for the bounding box.

You're going to override this method in subclasses to return a `CGRect` that is smaller than the sprite frame. But it's important to have a common ancestor method, so that even if you don't remember to implement it in a subclass, the subclass will still execute this parent method.

Override `collisionBoundingBox` in **Player.m** as follows:

```
- (CGRect)collisionBoundingBox {
```

```
  CGRect bounding = CGRectMake(
      self.position.x - (kPlayerWidth / 2),
      self.position.y - (kPlayerHeight / 2),
      kPlayerWidth, kPlayerHeight);

  return CGRectOffset(bounding, 0, -3);
}
```

The first line of this new method creates a `CGRect` named `bounding` that starts from the player's origin point, which is the bottom-left corner of the sprite. To get there, you take the player's position, which describes the center of the player sprite, and move left by half the defined width and down by half the defined height. You're using constants for the height and width of the box, and there's a good reason for this.

These values aren't defined yet. To fix that, add the following `#define` statements in **Player.m**, right after the `#import` lines:

```
#define kPlayerWidth 30
#define kPlayerHeight 38
```

When you have a number of different textures that make up animations for a game character, often they don't all have the same dimensions. Some may be taller or wider. If you're using the texture frame to calculate collisions with ground or wall tiles, each time the frame changes you have a slightly different intersection with the tiles. This will cause the sprite to wobble or vibrate.

To rectify this, you need to use a constant height and width to create the bounding rectangle.

> **Note:** Alternatively, you could make sure that all the animation frames are exactly the same size, but that's less reliable. This is especially true if you use Xcode's built-in texture atlas generation because Xcode trims transparent space. You can still get the original frame size, but I settled on using specific width and height values. That way I can be sure I always get the bounding box I want.

The final call to `CGRectOffset` does exactly what you would think: it moves the rectangle by the supplied x- and y-values. You are moving the bounding box down a little from the center.

The resulting bounding box is smaller than the cyclops and oriented at the bottom of its frame. This way the cyclops's feet touch the ground, but when it jumps its head can go through the block a little. Here's what the final bounding box looks like:

If you want to make your game more forgiving, you can shrink this box down a little more so that the player has to get even closer to collide with enemies.

Implementing collision detection

Now that you have a working bounding box, it's time to move to the next step in the physics engine process. It's time to do some heavy lifting. ☺

You need a number of three new methods in your `PSKLevelScene` to accomplish the collision detection:

- A method that returns the coordinates of the eight tiles that surround the cyclops's current location.
- A method to determine which, if any, of these eight tiles is a collision tile. Some of your tiles, such as background tiles, won't have physical properties and therefore your cyclops won't collide with them.
- A method to resolve collisions based on an assigned priority—the first tile is the tile beneath the player, for example.

You also need two methods that will make it easier to accomplish the above goals:

- A method that calculates the tile position of the player. There's already a method in `JSTileMap` that will give you a set of tile coordinates for an input position (in points).

- A method that takes a tile's tile map coordinates and returns the `CGRect` in the coordinate system that the `Player` node occupies.

Tackle the missing helper method first. You'll add this method to the `JSTileMap+TileLocations` class you created earlier.

Add the following method declaration to **JSTileMap+TileLocations.h**, right before the `@end`:

```
- (CGRect)tileRectFromTileCoords:(CGPoint)tileCoords;
```

Now add the method itself to **JSTileMap+TileLocations.m**:

```
- (CGRect)tileRectFromTileCoords:(CGPoint)tileCoords {
  CGFloat levelHeightInPixels = self.mapSize.height *
                                self.tileSize.height;
  CGPoint origin = CGPointMake(
      tileCoords.x * self.tileSize.width, levelHeightInPixels -
                    (tileCoords.y + 1) * self.tileSize.height);

  return CGRectMake(origin.x, origin.y, self.tileSize.width,
                    self.tileSize.height);
}
```

This method calculates the rectangle for a tile. It takes in a set of tile coordinates that describes a position in the tile map and returns the `CGRect` for that tile in points, measured in the coordinate space of the map node.

To do this, the method multiplies the tile coordinates by the tile size. You have to reverse the coordinate systems for the height, so you calculate the total height of the map (`self.mapSize.height * self.tileSize.height`) and then subtract the height of the tiles.

> **Note:** You need to invert the coordinate for the height because the origin point of Sprite Kit/OpenGL's coordinate system is at the bottom left of the world, but the tile map coordinate system starts at the top left of the world. Standards—aren't they great?

Why do you add one to the tile's height coordinate? Remember, the tile coordinate system is zero-based, so the 20th tile has an actual coordinate of 19. If you didn't add one to the coordinate, the method would return a point of `19 * tileHeight`.

What's in a GID

A GID or global ID is a number that represents the tile from the tileset at a given position. The tileset is arranged in a grid and the GID counts from left to right, then starts on the next row, the same way you read text in English.

If there's no tile, then that tile has no GID. You can configure a TMX map to count from 0, but I rely on the fact that the counting starts at 1 to determine which tile map positions contain no tiles. The method returns 0 for empty tiles, so where the GID is 0, there's no tile and no need to resolve a collision.

You need a method to retrieve the GID from a `TMXLayer`. There's a method on `TMXLayer` to retrieve a GID for a pixel position, but you need one for a tile map position. There's already a method for that, too, but it's at a lower level, in the `TMXLayerInfo` class. Rather than drill down to this level every time you want to get a GID, you're going to create another category method that exposes this method directly on `TMXLayer`.

Add this block to **JSTileMap+TileLocations.h** at the very end of the code, after the `@end`:

```
@interface TMXLayer (TileLocations)
- (NSInteger)tileGIDAtTileCoord:(CGPoint)point;
@end
```

This creates a new category on `TMXLayer`—the existing category was connected to `JSTileMap`. Normally you'd create a second set of source files for a new category, but these are so closely related that it makes sense to keep them together in the same file.

Now add the implementation to **JSTileMap+TileLocations.m**, all the way at the bottom:

```
@implementation TMXLayer (TileLocations)

- (NSInteger)tileGIDAtTileCoord:(CGPoint)point {
  return [self.layerInfo tileGidAtCoord:point];
}

@end
```

You can see how simple this is—it just passes the method to the `layerInfo` object and returns the result.

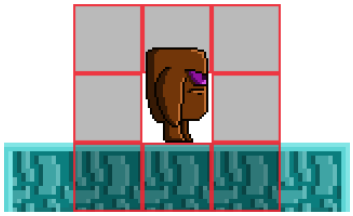
Surrounded by tiles!

Now you'll create the method to iterate through the surrounding tiles. This method will look up the GID of the tile image and information about the `CGRect` for that tile.

You'll rearrange the lookup order by the priority rank that you'll use later to resolve collisions. For example, you want to resolve collisions for the tiles directly left, right, below and above your cyclops before you resolve any collisions along the diagonals. I'll explain why this matters shortly.

Also, when you resolve the collision for a tile below the cyclops, you'll set a flag that tells you whether the cyclops is currently touching the ground. Later on, this flag will let you know whether it's OK to let the cyclops jump!

The physics engine's first step is to collect information about the eight tiles that surround the cyclops.



> **Note:** In this game, the cyclops's bounding box is less than two tiles wide and two tiles high. This is important because if it were larger than that, you'd need to take more than the surrounding eight tiles into account.
>
> If you want to use sprites that are larger than mine, you have two options. First, you could make your tiles larger as well. Or, you could adapt the physics engine to check for tiles farther away than the immediate surrounding ring of tiles.

You'll check with the TMX map specifically for information about the layer called **walls**. This layer contains all the tiles that the player and enemies can't walk, jump or fall through. In this game, this is the only layer that contains collidable tiles, but you'll write the code so that it would be easy to add checks for more layers if you wanted to.

Start by adding the following private property to **PSKLevelScene.m**, in the `@interface` block:

```
@property (nonatomic, strong) TMXLayer *walls;
```

Next, add a line to the `initWithSize:level:` method after the code that adds the map to the `gameNode`:

```
self.walls = [self.map layerNamed:@"walls"];
```

You need a reference to this layer to check it for tiles and positions of tiles.

Now you can begin to add the code that runs the physics engine! This is so exciting!

Add this method to the same class:

```
- (void)checkForAndResolveCollisions:(PSKCharacter *)character
                            forLayer:(TMXLayer *)layer {
  //1
  NSInteger indices[8] = {7, 1, 3, 5, 0, 2, 6, 8};

  //2
  for (NSUInteger i = 0; i < 8; i++) {
    NSInteger tileIndex = indices[i];

    //3
    CGRect characterRect = [character collisionBoundingBox];

    //4
    CGPoint characterCoord = [self.walls
                              coordForPoint:character.position];
    //5
    NSInteger tileColumn = tileIndex % 3;
    NSInteger tileRow = tileIndex / 3;
    CGPoint tileCoord = CGPointMake(
        characterCoord.x + (tileColumn - 1),
        characterCoord.y + (tileRow - 1));

    //6
    NSInteger gid = [self.walls tileGIDAtTileCoord:tileCoord];

    //7 - New code will go after this point
    CGRect tileRect = [self.map
                              tileRectFromTileCoords:tileCoord];
    NSLog(@"Tile Rect %@, tile GID %ld, tile coord %@, character
bounding rect %@", NSStringFromCGRect(tileRect), (long)gid,
        NSStringFromCGPoint(tileCoord),
        NSStringFromCGRect(characterRect));
  }
}
```

Before reviewing this code one section at a time, note that you're passing in a layer object and remember that your tile map has a number of different layers—walls, objects, enemies and powerups—as covered in the last chapter.

Having separate layers allows you to tailor your collision detection to each layer. In this game, I set up the map so that all of the obstacles you need to check for collisions are in a single layer. However, you might want to have other types of

layers in your game, such as a separate layer of obstacles that hurt the player, like spikes.

There are other ways to distinguish between different types of tiles or blocks, but for your needs the layer separation accomplishes this sufficiently.

OK, let's go through the code above section by section:

1. First, you create a simple C array that contains indices for the tiles. This effectively reorders the tiles so that some get checked before others. For example, the tile directly below the cyclops is checked first. The diagram below shows the new order/indices in bold, and the original order/indices in parenthesis.

Tile Order (TileIndex)

5 (0)	**2** (1)	**6** (2)
3 (3)	(4)	**4** (5)
7 (6)	**1** (7)	**8** (8)

   You may already have some intuition about why you're doing this. I'll explain it in detail a little later.

2. Next, you start the loop. It increments from 0 to 7 because you need to look at eight tiles. You use the loop counter to look up the index of the tile in the array you just created. The index of the first tile checked is 7 because it's the first element in the array, which is the tile directly below the player.

> **Note**: You only need information for eight tiles, not nine, because you should never need to resolve a collision with the tile space in the center of the 3×3 grid.
>
> Instead, you should always have caught that collision and resolved it in a surrounding tile position. If there is a collision-susceptible tile in the center of the grid, the cyclops has moved at least half its width in a single frame. It shouldn't move this fast, ever—at least not in this game!

3. Here you get the bounding box, in screen coordinates, of the `PSKCharacter` object that you passed in. For now, that will be the cyclops, but later you'll use this same method to check enemies for collisions as well.
4. Now you get the tile coordinate position for the `PSKCharacter`. All of the tile maps provided with this starter kit have tiles around the sides and bottom, and the

character can't fly to the top of the screen, so this position value will never be on a side, bottom or top edge. This is important for the next step.

5. At this stage, you calculate the relative coordinate position for the current tile index. When you are done, `tileCoord` contains one of the tiles surrounding the `PSKCharacter`.
6. Finally, you make good on the preceding code and get the GID for that tile coordinate. A GID of 0 means an empty space; anything else and it will contain an image from the tileset.
7. Here you get the `CGRect` (a bounding box, really) for the tile in map coordinates. Later, you'll compare this with the character's bounding box to see if you need to resolve a collision.

   The last bit is here temporary. You are logging all of the information this method creates so you can see if it's doing its job.

To see this method in action, you need to call it from the update method. Add the following line to the end of `update`:

```
[self checkForAndResolveCollisions:self.player
                        forLayer:self.walls];
```

Build and run. Now you should have something like the following in your console.

```
2013-11-15 14:19:48.535 SKPocketCyclops[71022:a0b] Tile Rect {{64, 160}, {32, 32}}, tile GID 2, tile coord {2, 34}, character bounding rect {{100, 200.5}, {30, 38}}
2013-11-15 14:19:49.086 SKPocketCyclops[71022:a0b] Tile Rect {{96, 160}, {32, 32}}, tile GID 2, tile coord {3, 34}, character bounding rect {{100, 191.5}, {30, 38}}
2013-11-15 14:19:49.086 SKPocketCyclops[71022:a0b] Tile Rect {{96, 224}, {32, 32}}, tile GID 0, tile coord {3, 32}, character bounding rect {{100, 191.5}, {30, 38}}
2013-11-15 14:19:49.086 SKPocketCyclops[71022:a0b] Tile Rect {{64, 192}, {32, 32}}, tile GID 0, tile coord {2, 33}, character bounding rect {{100, 191.5}, {30, 38}}
2013-11-15 14:19:49.087 SKPocketCyclops[71022:a0b] Tile Rect {{128, 192}, {32, 32}}, tile GID 0, tile coord {4, 33}, character bounding rect {{100, 191.5}, {30, 38}}
2013-11-15 14:19:49.087 SKPocketCyclops[71022:a0b] Tile Rect {{64, 224}, {32, 32}}, tile GID 0, tile coord {2, 32}, character bounding rect {{100, 191.5}, {30, 38}}
2013-11-15 14:19:49.087 SKPocketCyclops[71022:a0b] Tile Rect {{128, 224}, {32, 32}}, tile GID 0, tile coord {4, 32}, character bounding rect {{100, 191.5}, {30, 38}}
2013-11-15 14:19:49.087 SKPocketCyclops[71022:a0b] Tile Rect {{64, 160}, {32, 32}}, tile GID 2, tile coord {2, 34}, character bounding rect {{100, 191.5}, {30, 38}}
2013-11-15 14:19:49.168 SKPocketCyclops[71022:a0b] Tile Rect {{96, 160}, {32, 32}}, tile GID 2, tile coord {3, 34}, character bounding rect {{100, 184.50227}, {30, 38}}
2013-11-15 14:19:49.169 SKPocketCyclops[71022:a0b] Tile Rect {{96, 224}, {32, 32}}, tile GID 0, tile coord {3, 32}, character bounding rect {{100, 184.50227}, {30, 38}}
2013-11-15 14:19:49.169 SKPocketCyclops[71022:a0b] Tile Rect {{64, 192}, {32, 32}}, tile GID 0, tile coord {2, 33}, character bounding rect {{100, 184.50227}, {30, 38}}
2013-11-15 14:19:49.169 SKPocketCyclops[71022:a0b] Tile Rect {{128, 192}, {32, 32}}, tile GID 0, tile coord {4, 33}, character bounding rect {{100, 184.50227}, {30, 38}}
2013-11-15 14:19:49.169 SKPocketCyclops[71022:a0b] Tile Rect {{64, 224}, {32, 32}}, tile GID 0, tile coord {2, 32}, character bounding rect {{100, 184.50227}, {30, 38}}
2013-11-15 14:19:49.170 SKPocketCyclops[71022:a0b] Tile Rect {{128, 224}, {32, 32}}, tile GID 0, tile coord {4, 32}, character bounding rect {{100, 184.50227}, {30, 38}}
2013-11-15 14:19:49.170 SKPocketCyclops[71022:a0b] Tile Rect {{64, 160}, {32, 32}}, tile GID 2, tile coord {2, 34}, character bounding rect {{100, 184.50227}, {30, 38}}
2013-11-15 14:19:49.231 SKPocketCyclops[71022:a0b] Tile Rect {{96, 160}, {32, 32}}, tile GID 2, tile coord {3, 34}, character bounding rect {{100, 171.69781}, {30, 38}}
2013-11-15 14:19:49.231 SKPocketCyclops[71022:a0b] Tile Rect {{96, 224}, {32, 32}}, tile GID 0, tile coord {3, 32}, character bounding rect {{100, 171.69781}, {30, 38}}
2013-11-15 14:19:49.231 SKPocketCyclops[71022:a0b] Tile Rect {{64, 192}, {32, 32}}, tile GID 0, tile coord {2, 33}, character bounding rect {{100, 171.69781}, {30, 38}}
2013-11-15 14:19:49.231 SKPocketCyclops[71022:a0b] Tile Rect {{128, 192}, {32, 32}}, tile GID 0, tile coord {4, 33}, character bounding rect {{100, 171.69781}, {30, 38}}
2013-11-15 14:19:49.232 SKPocketCyclops[71022:a0b] Tile Rect {{64, 224}, {32, 32}}, tile GID 0, tile coord {2, 32}, character bounding rect {{100, 171.69781}, {30, 38}}
2013-11-15 14:19:49.232 SKPocketCyclops[71022:a0b] Tile Rect {{128, 224}, {32, 32}}, tile GID 0, tile coord {4, 32}, character bounding rect {{100, 171.69781}, {30, 38}}
```

That code will crash when the `Player` drifts off the edge of the screen. You'll fix that soon.

Why the order matters

Quite often, resolving the collision for one tile will also resolve collisions for adjacent tiles. When you resolve the tile directly under the character (the cyclops or an enemy), this also resolves the collisions for the lower two diagonal tiles.

Refer to the image below. By resolving the collision beneath the character, shown in red, you also resolve the collision with block 2, because after the cyclops is pushed up he will no longer be colliding with block 2.

Fix middle block First!
This also fixes block 2.

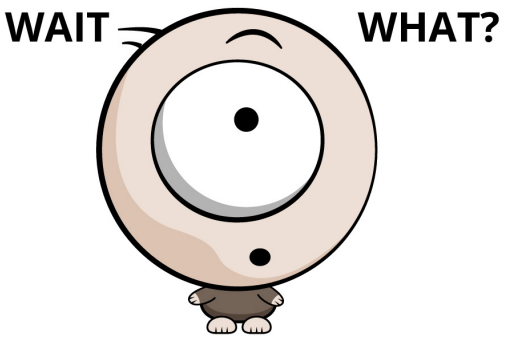
Your collision detection routine makes certain guesses about how to resolve collisions. Those guesses are always valid for adjacent tiles. They are only valid for diagonal tiles most of the time, so you want to avoid collision resolution with diagonal tiles as much as possible. If you're not sure why, keep reading; it will become clear before the end of this chapter.

Below is an image showing the order of the tiles based on the C array. You can see that the bottom, top, left and right tiles are resolved first. This order will also help you know when to set the flag indicating that the cyclops is touching the ground, which determines whether it can jump. You'll set up this flag later in the chapter.

5	2	6
3		4
7	1	8

Grounding the cyclops

Up to this point, the cyclops has had the privilege of setting its own position. But now you're taking that privilege away.



If the `Player` object updates its position and then the `PSKLevelScene` finds a collision, you want to move your cyclops back a little bit so it's no longer colliding with the obstacle.

So the cyclops needs a new variable that it can update, but one that will stay a secret between itself and the `PSKLevelScene`—call it `desiredPosition`.

You want the `PSKCharacter` class, which applies to both the cyclops and enemies, to calculate and store its desired position. But the `PSKLevelScene` will update the character's position after that position is validated (and modified, if necessary) for collisions. The same applies to the collision tile detection loop—you don't want the collision detector to update the actual sprite until you've checked all the tiles for collisions and resolved those collisions.

To get there, you need to change a few things. First add this new property to **PSKCharacter.h**:

```
@property (nonatomic, assign) CGPoint desiredPosition;
```

Now modify `collisionBoundingBox` in **PSKCharacter.m** to:

```
- (CGRect)collisionBoundingBox {
  CGPoint diff = CGPointSubtract(self.desiredPosition,
                                 self.position);
  return CGRectOffset(self.frame, diff.x, diff.y);
}
```

This computes a bounding box based on the desired position. The layer will use this bounding box for collision detection.

Finally, modify `collisionBoundingBox` in **Player.m**—you overrode it, if you remember—so that it looks like this:

```
- (CGRect)collisionBoundingBox {
  CGRect bounding = CGRectMake(
      self.desiredPosition.x - (kPlayerWidth / 2),
      self.desiredPosition.y - (kPlayerHeight / 2),
      kPlayerWidth, kPlayerHeight);

  return CGRectOffset(bounding, 0, -3);
}
```

The method is the same, except that you've changed all the original references to `position` to `desiredPosition`.

Still in **Player.m**, make the following change to `update:` so that it's updating the `desiredPosition` property instead of the `position` property. Replace this line:

```
self.position = CGPointAdd(self.position, stepVelocity);
```

With this:

```
self.desiredPosition = CGPointAdd(self.position, stepVelocity);
```

Let's resolve some collisions!

It's time for the real deal! This is where you'll tie it all together.

What follows is one of the longest blocks of code you'll ever see in a Ray Wenderlich tutorial. Sorry about that. I suggest you copy and paste it into your project and read it there (it will make it a bit easier).

Alter the `checkForAndResolveCollisions:forLayer:` method in **PSKLevelScene.m** to include the following lines. Delete everything after the **//New code will go after this point** line and add this:

```
//1
if (gid != 0) {
  //2
  CGRect tileRect = [self.map tileRectFromTileCoords:tileCoord];
  //3
  if (CGRectIntersectsRect(characterRect, tileRect)) {
    CGRect intersection = CGRectIntersection(characterRect,
                                             tileRect);
    //4
    if (tileIndex == 7) {
      //tile is directly below the character
      character.desiredPosition = CGPointMake(
        character.desiredPosition.x,
        character.desiredPosition.y + intersection.size.height);
    } else if (tileIndex == 1) {
      //tile is directly above the character
      character.desiredPosition = CGPointMake(
        character.desiredPosition.x,
        character.desiredPosition.y - intersection.size.height);
    } else if (tileIndex == 3) {
      //tile is left of the character
      character.desiredPosition = CGPointMake(
          character.desiredPosition.x + intersection.size.width,
          character.desiredPosition.y);
    } else if (tileIndex == 5) {
      //tile is right of the character
      character.desiredPosition = CGPointMake(
          character.desiredPosition.x - intersection.size.width,
          character.desiredPosition.y);
    } else {
      if (intersection.size.width > intersection.size.height) {
        //5
        //tile is diagonal, but resolving collision vertically
        CGFloat resolutionHeight;
        if (tileIndex > 4) {
          resolutionHeight = intersection.size.height;
```

```
          } else {
            resolutionHeight = -intersection.size.height;
          }
          character.desiredPosition = CGPointMake(
              character.desiredPosition.x,
              character.desiredPosition.y + resolutionHeight);
        } else {
          //tile is diagonal, but resolving horizontally
          CGFloat resolutionWidth;
          if (tileIndex == 6 || tileIndex == 0) {
            resolutionWidth = intersection.size.width;
          } else {
            resolutionWidth = -intersection.size.width;
          }
          character.desiredPosition = CGPointMake(
              character.desiredPosition.x + resolutionWidth,
              character.desiredPosition.y);
        }
      }
    }
  }
  //6
  character.position = character.desiredPosition;
```

You'll review this code section by section, but first stop to consider why you're using the `PSKCharacter` class here instead of the `Player` class (or later, the `Enemy` class).

Can you think of why? Because the `PSKCharacter` class has the `desiredPosition` attribute, you can now use this single method to handle all the different kinds of characters in your game.

The collision system uses the `desiredPosition` and the `collisionBoundingBox` methods to detect and resolve collisions with tiles in the level. The `PSKCharacter` class handles these attributes, so you don't need to know if you are resolving an enemy collision or a player collision—the logic is exactly the same for both.

Since both enemies and the player are derived from the same base `PSKCharacter` class, it would make no sense to create a dedicated method to resolve collisions for different types of character objects.

When this tutorial refers to a character moving forward, it means either a `Player` or an `Enemy` object. By using the character abstraction, you write less code—and if you need to modify the physics system in the future, you only have to make changes in one place.

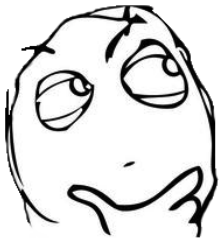
OK! Let's look at the code you just added:

1. You start about where you left off the last time you added code. Once you have retrieved the GID, you want to check it. There may not be an actual tile in that position. If there isn't, you'll have stored a zero in the variable. In that case, the current loop iteration is over and the code moves on to the next tile.

2. If there is a tile at that position, you need to get the `CGRect` for that tile and store it in the `tileRect` variable. This `CGRect` is in screen coordinates (in points). Now that you have a `CGRect` for the character—you stored it in `characterRect`, remember?—and a `CGRect` for the tile, you can use them to check for collisions.
3. To check for the collision, use `CGRectIntersectsRect`. If there is a collision, then use `CGRectIntersection` to get the rectangle that describes the overlapping section of the two `CGRects`.

Pausing to consider a dilemma...

Here's the tricky bit – you need to determine how to resolve this collision.



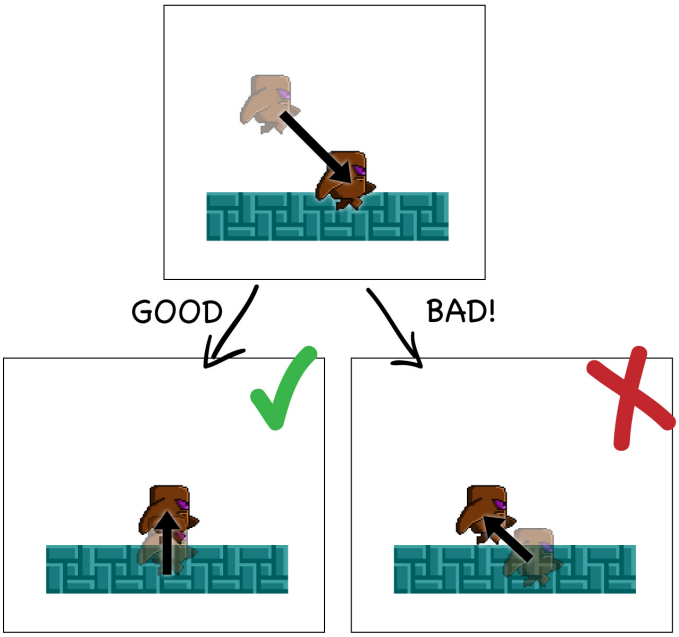
Hmm... not as easy as it appears.

You might think the best way to do so is to move the character backward and out of the collision. Or in other words, to reverse the last move (`self.velocity * -1`) until a collision no longer exists with a tile. That's the way some physics engines work, but you're going to implement a better solution.

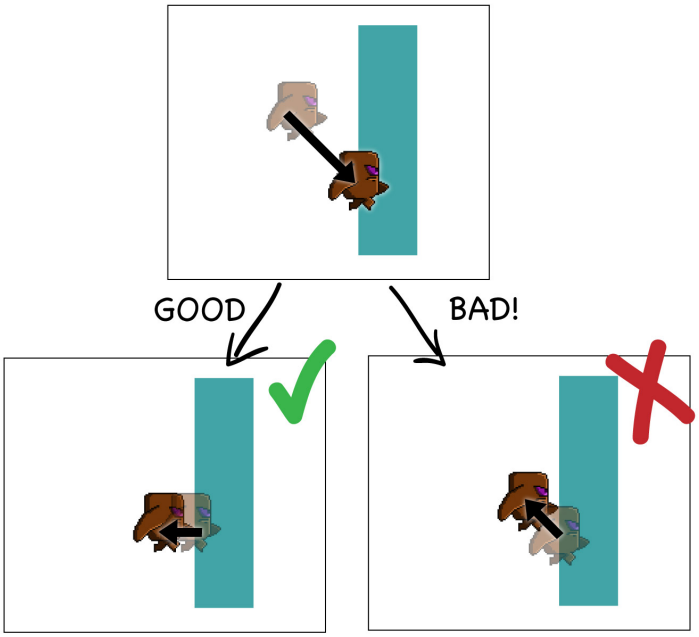
Consider this: gravity is constantly pulling the character down onto the tiles underneath it, and your physics engine is constantly resolving those collisions.

Imagine that the character is moving forward: it is also going to be moving downward at the same time due to gravity. If you choose to resolve that collision by reversing the last move (forward and down), you would need to move the character upward and backward—but that's not what you want!

The character needs to move up enough to stay on top of those tiles, but continue to move forward at the same pace.



This same problem also presents itself if the player or enemy is sliding down a wall. If the user is pressing the cyclops into the wall, for example, then its desired trajectory is diagonally downward and into the wall. Reversing this trajectory would move it upward and away from the wall—again, not the motion you want! You want the cyclops to stay on the outside of the wall without slowing or reversing its downward speed.

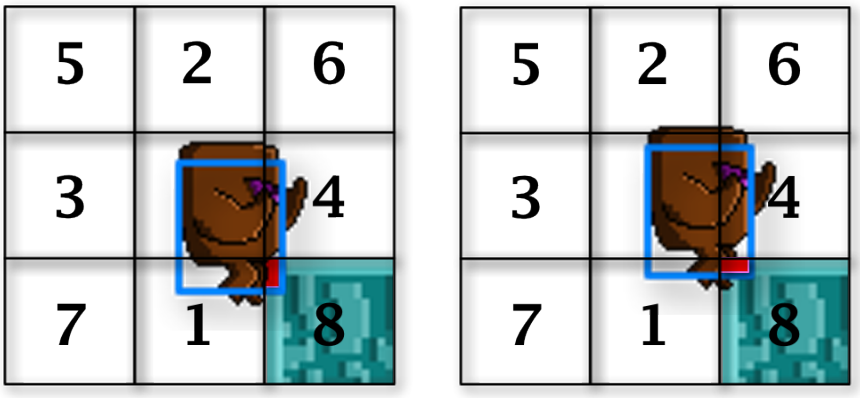


Therefore, you need to decide when to resolve collisions vertically and when to resolve them horizontally, and handle both events as mutually exclusive cases.

Some physics engines always resolve one type first, but you really want to make the decision based on the location of the tile relative to the character. So, for example, when the tile is directly beneath the cyclops, you will always resolve that collision by moving it upward.

What about when the tile is diagonal to the character's position? In this case, you'll use the intersecting `CGRect` to guess at how you should move it. If the intersection of the two `CGRects` is wider than it is tall, you'll assume that the correct resolution is vertical. If the intersecting `CGRect` is taller than it is wide, you'll resolve the collision horizontally.

Does this sound confusing? The image below provides a visual explanation. The intersecting `CGRect` is highlighted in red:



This process will work reliably as long as the character's velocity stays within certain bounds and your game runs at a reasonable frame rate. Later on, you'll include some clamping code for the `PSKCharacter` class so that the character doesn't fall too quickly, which could cause problems such as moving the character through an entire tile in one step.

Once you've determined whether you need a horizontal or vertical collision resolution, you will use the intersecting `CGRect`'s size to move the character back out of a collision state. Basically, you'll look at the height or width, as appropriate, of the collision `CGRect` and use that value as the distance to move the character.

Remember why you want to resolve collisions according to a specific order. These diagonal resolutions are sometimes not exactly what you want; avoiding them is preferable.

Fix middle block First!
This also fixes block 2.

The tiny blue area to the right of the red bar is tall and skinny because that collision intersection represents only a small portion of the whole collision. If you tried to resolve the collision with this diagonal tile first using the `CGRect` size assumption, it would get resolved horizontally, pushing the character backwards!

However, if you've already resolved the collision for the tile directly beneath the character, then the character will no longer be in a collision state with the diagonal tile on the right—thereby avoiding the unreliable choice of how to resolve the diagonal tile's collision.

Back to the code!

And now we return you to the regularly scheduled dissection of the monster `checkForAndResolveCollisions:forLayer:` method!

4. Here you look at the index of the current tile. You use the index to determine the position of the tile. You are going to deal with the adjacent tiles individually, moving the character by subtracting or adding the width or height of the collision, as appropriate. That's simple enough. Once you get to the diagonal tiles, though, you'll implement the logic described in the section above.
5. This is where you handle the diagonal collisions. First you determine whether the collision is wide or tall. If it's wide, you resolve vertically. In that case, you'll move the character either up or down, which you determine next by seeing if the tile index is greater than five, since tiles six and seven are beneath the character. Based on that, you know whether you need to add or subtract the collision height. You use the same logic to resolve horizontal collisions.
6. Finally, you set the character's position to the final desired position you calculated during collision detection.

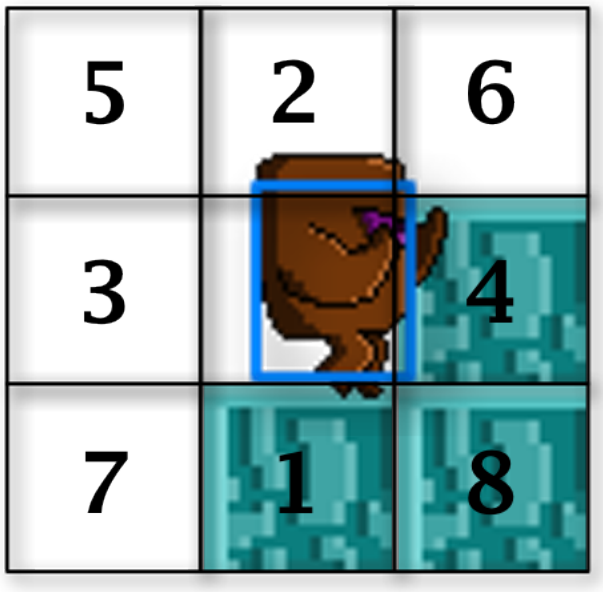
This method is the guts of your collision detection system. It's a basic system, and you may find that if your game moves very quickly or has other goals, you need to alter it to get consistent results. But for this particular game, it works well. ☺

If you're like me, pictures can sometimes be easier to understand than words. If the above doesn't make a lot of sense, or if you're still a bit confused, here is description of the whole process with accompanying images, showing the character's position along the way:

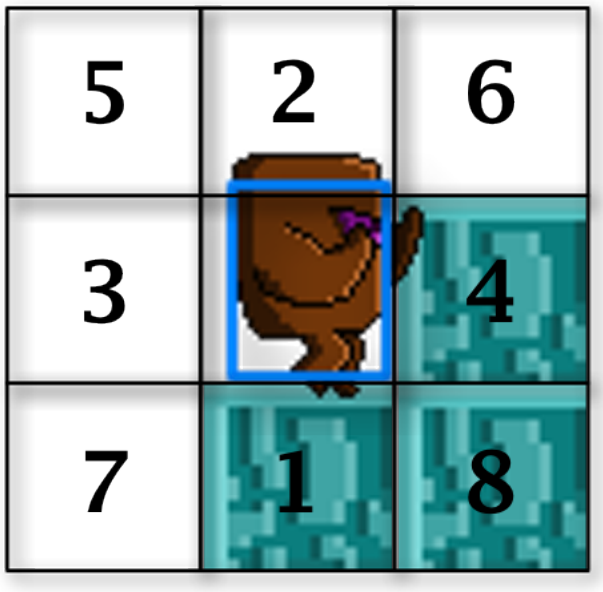
1. **Get tiles in order**: You've got collisions with three tiles 1, 4 and 8.

2. **Iterate through the tiles**: Look at each tile in order of importance, starting at the tile directly below the character (index 1).
3. **Check for GID**: If you have a tile at a given position, you will get a GID representing the tile image. If the GID is 0, there's no tile there and you can move on to the next tile position. The GID of tile 1 is a wall image, so there is a collision.
4. **Resolve collision**: You resolve the collision with tile 1 by moving the character's `desiredPosition` up by the height of the collision, which you retrieve using `CGRectIntersection`. Notice that resolving the collision with tile 1 also resolves the collision with tile 8.

   Keep in mind that your `collisionBoundingBox` is based on `desiredPosition` and you are changing `desiredPosition` as you go. So the collisions that exist during the first loop (tile 1) may no longer exist after you resolve the collision.



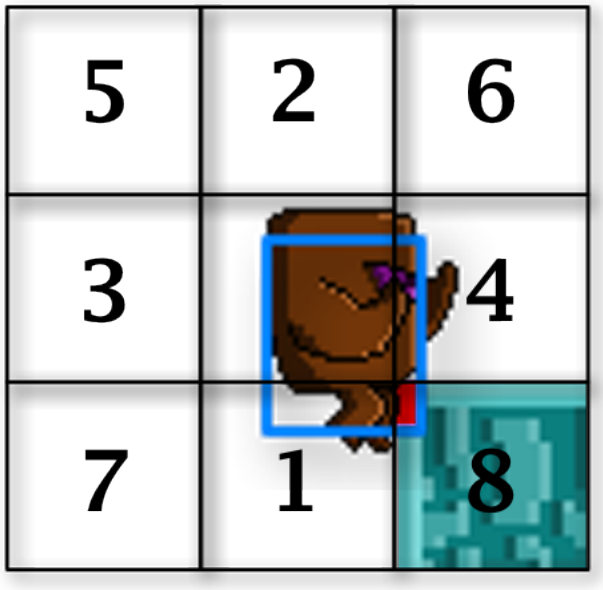
5. **Loop through tiles with GID 0**: Tiles 2 and 3 return a GID of 0, so the loop moves to the next iteration in both cases without testing for intersection or resolving collisions.
6. **Resolve collision with tile 4**: Because tile 4 is an adjacent tile and to the right, you resolve by subtracting the width of the collision from the `desiredPosition`.



7. **Loop through tiles with GID 0**: Tiles 5 through 7 return GID 0, so skip over them.

8. **Skip tiles with no collision**: Tile 8 has a GID, so test it for `CGRectIntersectsRect`. That will return false, because resolving the collisions with tiles 1 and 4 made it so there's no longer a collision with tile 8.

Usually an adjacent collision will resolve a diagonal one, as with tile 8 above, but what about this case:



If there's a collision with a tile that's diagonal to the player, you need to decide whether to resolve it horizontally or vertically. In the above image, the collision is tall and skinny, so the algorithm will choose to resolve it by moving the cyclops to the left and keeping the vertical position the same.

This case occurs often enough—think of when you are jumping up and over a tall obstacle, like a pipe in *Super Mario Bros*. The player character will probably push into the side of the object all the way up. So long as that collision rectangle is taller than it is wide, the character will move up smoothly.

However, in the last frame before the character clears the pipe, it's possible that the collision will be wider than tall, in which case the resolution would put the character on top of the pipe.

In *Pocket Cyclops*, this last frame boost allows you to "roll" over the edge of a corner that you might not have quite cleared otherwise, but the effect is imperceptible. For me, this behavior is not undesirable. You will want to pay close attention to your game and decide if it's desirable for you.

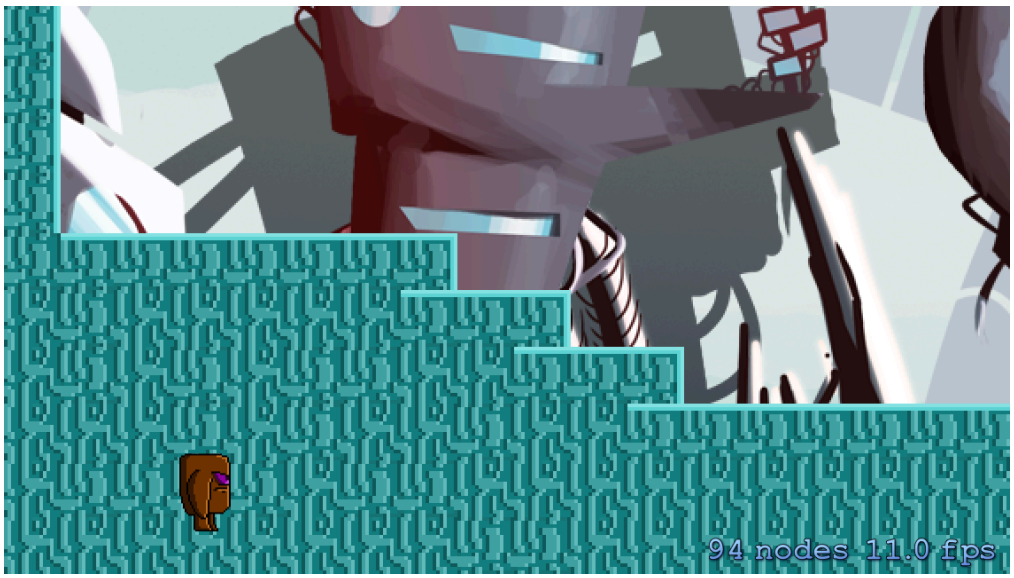
If you'd like to learn more about collision detection, here are some resources:

- The *Sonic the Hedgehog* Wiki has a great section describing how Sonic interacts with solid tiles: http://info.sonicretro.org/SPG:Solid_Tiles

- Perhaps the best guide to implementing platformers is from Higher-Order Fun: http://higherorderfun.com/blog/2012/05/20/the-guide-to-implementing-2d-platformers/
- The creators of *N* have a great tutorial that goes well beyond the basics of tile-based collision systems: http://www.metanetsoftware.com/technique/tutorialA.html

That's it! You've completed your collision detection method and you are using it to check for collisions with the player character every frame.

Build and run now:



The cyclops is stopped by the floor, but sinks into it eventually! And then the game crashes. ☹ What gives?

Be sure to reset the velocity

Can you guess what's causing this? What happens to the gravity force each frame?

Every frame, you are adding the force of gravity to `self.velocity`. This means that the character is constantly accelerating downward, pushing harder and harder into the floor until the speed of the character's downward trajectory is greater than the size of the tile. You're moving through an entire tile in a single frame, which is a problem I mentioned earlier.

When you resolve a collision, you also need to reset the velocity of the character to zero for that dimension! If the cyclops has stopped moving, the velocity value should reflect it.

If you don't do this, you'll get weird behaviors, where your character does things like move through tiles as you saw above or jump against a low ceiling and float against it longer than s/he should. This is the kind of unrealism you generally want to avoid in your game.

I also mentioned before that you need a good way to determine when the character is on the ground so you can make sure s/he can't jump in midair. You'll set up that flag now.

First add an `onGround` property to the `PSKCharacter` class. Add the following line to **PSKCharacter.h**:

```
@property (nonatomic, assign) BOOL onGround;
```

Now add the highlighted lines to `checkForAndResolveCollisions:forLayer:` in **GameLevelLayer.m** (as if it wasn't big enough already!):

```
- (void)checkForAndResolveCollisions:(PSKCharacter *)character
                            forLayer:(TMXLayer *)layer {
  character.onGround = NO;

  NSInteger indices[8] = {7, 1, 3, 5, 0, 2, 6, 8};

  . . .

  if (tileIndex == 7) {
    //tile is directly below the character
    character.desiredPosition = CGPointMake(
        character.desiredPosition.x,
        character.desiredPosition.y + intersection.size.height);
    character.velocity = CGPointMake(character.velocity.x, 0.0);
    character.onGround = YES;
  } else if (tileIndex == 1) {
    //tile is directly above the character
    character.desiredPosition = CGPointMake(
        character.desiredPosition.x,
        character.desiredPosition.y - intersection.size.height);
    character.velocity = CGPointMake(character.velocity.x, 0.0);
  } else if (tileIndex == 3) {
    //tile is left of the character
    character.desiredPosition = CGPointMake(
        character.desiredPosition.x + intersection.size.width,
        character.desiredPosition.y);
  } else if (tileIndex == 5) {
    //tile is right of the character
    character.desiredPosition = CGPointMake(
        character.desiredPosition.x - intersection.size.width,
        character.desiredPosition.y);
  } else {
    if (intersection.size.width > intersection.size.height) {
      //tile is diagonal, but resolving collision vertically
      CGFloat resolutionHeight;
      if (tileIndex > 4) {
        resolutionHeight = intersection.size.height;
        if (character.velocity.y < 0) {
          character.velocity = CGPointMake(
                                        character.velocity.x, 0.0);
```

```
            character.onGround = YES;
          }
        } else {
          resolutionHeight = -intersection.size.height;
          if (character.velocity.y > 0) {
            character.velocity = CGPointMake(
                                        character.velocity.x, 0.0);
          }
        }
        . . .
```

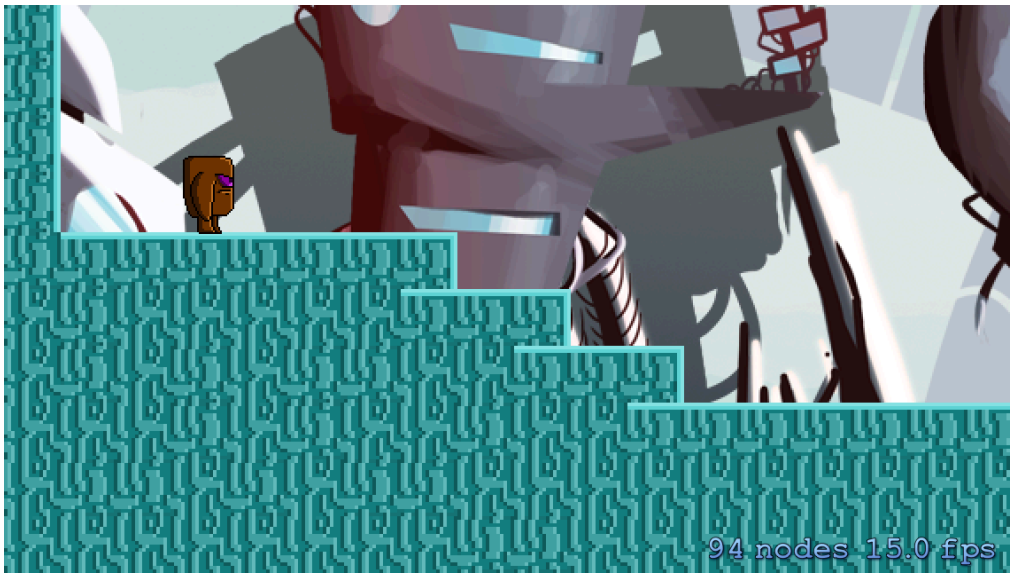
You begin by setting the character's `onGround` property to `NO`. Then, each time the character has a tile under it either adjacent or diagonally, you set `onGround` to `YES` and the velocity to zero.

Also, if the character has an adjacent tile above them, you set the character's velocity to zero. This makes the velocity variable properly reflect the character's actual movement and speed.

You could also set the horizontal velocity to zero when a character hits a wall, but you'll do that in another way later on, and it's unnecessary for the time being.

When setting the vertical velocity to zero for a diagonal tile, there's one more thing you need to do. In the last section I mentioned that in some cases, the cyclops will "roll" over the corner of a tile. If you set the vertical velocity to zero in that case, the cyclops will lose its upward momentum. This feels wrong and isn't what you want. So, to avoid that, you check for the velocity. If the cyclops is still jumping, you resolve the collision in the same way, but you don't change the cyclops's velocity or set the `onGround` flag.

Build and run now, and you should finally have a cyclops on solid ground!



Your game's collision detection engine is now in place. You've completed the hardest part of making a platformer game. Congratulations! That a big accomplishment and a lot more fun awaits you.

Challenge: The current iteration of the physics engine gets the coordinates for a single tile at the center point of the player character and then gets the surrounding eight tiles. It will be the same for enemies.

This structure works as long as the collision bounding box is less than 2×2 tiles (64×64 points). If it were larger than that, you'd potentially have a collision in a tile that is farther than one tile width away.

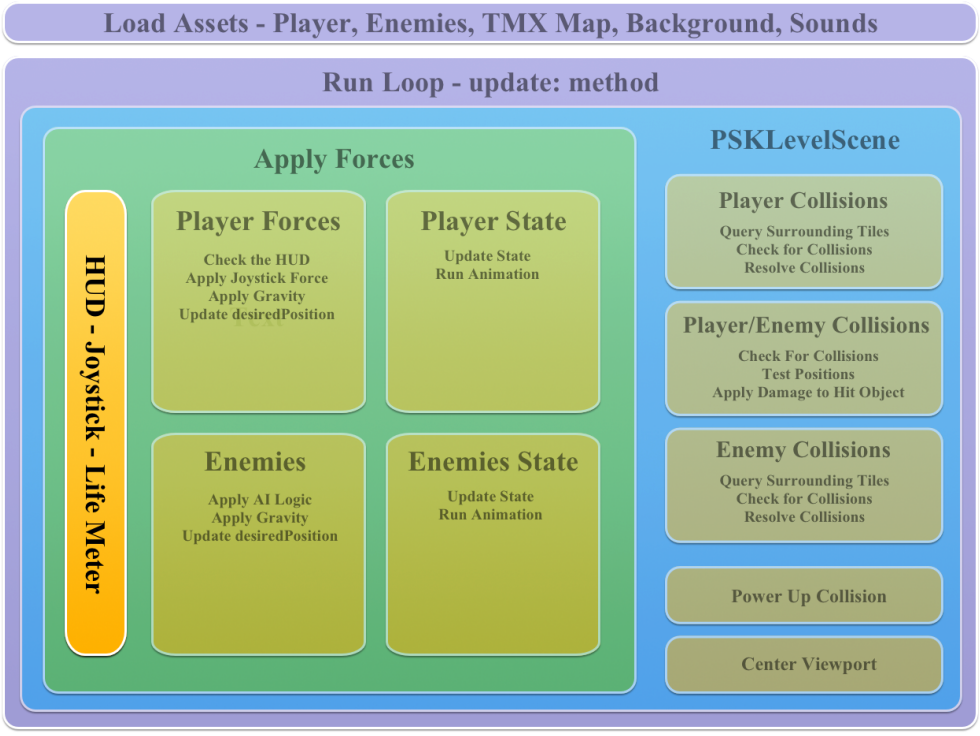
As a challenge, think through the following question: How would you construct a method to look for collision tiles for sprites larger than 2×2 tiles?

Here's a hint: Currently the collidable tiles surround the center tile. In the case of a sprite larger than 2×2, there would be multiple tiles in the center, surrounded by a ring, however large. Figure out which tiles comprise the outermost ring, alter the method to retrieve all of those tiles and think through the various collision resolution options.

Chapter 4: Creating a HUD

In this chapter, you will create a heads-up display or HUD node that contains both the game's controls and a life indicator for the cyclops.

Here's where this chapter falls in your overall game plan:



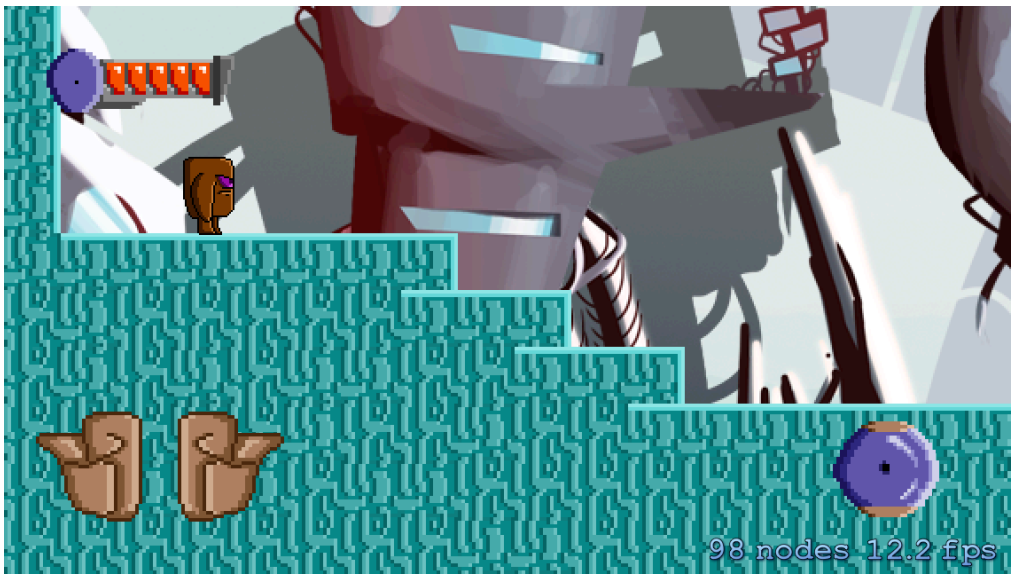
A HUD is just an `SKNode` subclass that contains `SKSpriteNodes` for the buttons and the life indicator.

The only tricky thing about the HUD layer is the touch handling logic, which the layer needs so it can respond to a player's fingers pressing and sliding onto and off of the buttons.

Normal buttons—`UIButton` objects—won't quite do the job. The HUD needs to handle multi-touch so that the player can press certain buttons simultaneously—in this case, the jump and left/right buttons.

Finally, the HUD needs an interface that other classes can access to read the state of the controls and update the life indicator.

Here's what the HUD will look like when you're done.



Adding the sprites

Create a new class by going to **File->New->File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, name the class **PSKHUDNode** and make it a subclass of **SKNode.** You can place this class in the same folder with the rest of the class files.

The first step is to load all the sprites that will exist in the node. These include the left and right buttons, the jump button and the life indicator. You will create properties for all these objects in order to change them in response to touch interaction, as well as keep track of them for touch handling.

> **Note:** I recently changed my personal coding style to use properties for most of my variables. This is a habit I formed after working with some seasoned Objective-C developers. While accessing properties instead of instance variables directly does take slightly longer, the benefits outweigh that cost.
>
> Using properties means that you can easily debug by overriding the setter or getter, you get free KVO and various other benefits of modern Objective-C. You'll see properties a lot more often in apps than in games, but it's

considered by Apple to be "best practice" and I've grown accustomed to doing it this way.

Add the following `@interface` declaration to **PSKHUDNode.m** after the `#import` line:

```
@interface PSKHUDNode ()
//1
@property (nonatomic, strong) SKSpriteNode *lifeBarImage;
//2
@property (nonatomic, strong) SKSpriteNode *leftButton;
@property (nonatomic, strong) SKSpriteNode *rightButton;
@property (nonatomic, strong) SKSpriteNode *jumpButton;
//3
@property (nonatomic, strong) NSArray *buttons;
@end
```

Let's break this down:

1. You declare the sprite for the life indicator. You will change the texture of this sprite as the player's life changes.
2. You also declare sprites for the three buttons.
3. Finally, you declare a buttons array that you'll use to easily track and iterate through the buttons.

The next step is to initialize all of these sprites and place them on the screen. First, add the new method to the header file, **PSKHudNode.h**:

```
- (instancetype)initWithSize:(CGSize)size;
```

Now implement the method in **PSKHudNode.m**:

```
- (instancetype)initWithSize:(CGSize)size {
  //1
  if ((self = [super init])) {
    //2
    self.userInteractionEnabled = YES;
    //3
    self.lifeBarImage = [SKSpriteNode
                         spriteNodeWithImageNamed:@"Life_Bar_5_5"];
    self.lifeBarImage.position = CGPointMake(
                                          80, size.height - 40);
    [self addChild:self.lifeBarImage];
    //4
    self.leftButton = [SKSpriteNode
                       spriteNodeWithImageNamed:@"leftButton"];
    self.leftButton.position = CGPointMake(50, 90);
    [self addChild:self.leftButton];
    //5
    self.rightButton = [SKSpriteNode
```

```
                        spriteNodeWithImageNamed:@"rightButton"];
    self.rightButton.position = CGPointMake(130, 90);
    [self addChild:self.rightButton];
    //6
    self.jumpButton = [SKSpriteNode
                       spriteNodeWithImageNamed:@"jumpButton"];
    self.jumpButton.position = CGPointMake(size.width - 70, 60);
    [self addChild:self.jumpButton];
    //7
    self.buttons = @[self.leftButton, self.rightButton,
                     self.jumpButton];
  }
  return self;
}
```

Let's look at this in more detail:

1. Initialize the superclass. **SKNode** doesn't have many fancy initializers, so you don't have a lot of options here.
2. Each `SKNode` has a `userInteractionEnabled` property. This property makes the node receive touch events so you can implement the standard touch callback methods, such as `touchesBegan` and `touchesMoved`. An `SKScene` object receives touches by default—you just have to implement the touch methods—but an `SKNode` doesn't, so you must set this property first.
3. The first sprite you create is the `lifeBarImage`. You name the images for the six states of life according to a convention so that you can easily retrieve them in code later. Each one has the name **Life_Bar_x_5.png**, where **x** is a number between 0 and 5 representing the number of red life squares that image depicts. You can omit the .png extension in Sprite Kit.
4. Here you create the left button using the familiar `spriteNodeWithImageNamed:` initializer. You will position all of these sprites relative to the edges of the screen using the 'size' argument in the `initWithSize:` method. In this case, you want the sprite's center 50 pixels from the left and 90 pixels from the bottom.
5. Next, you do the same thing with the right button, placing it 80 pixels to the right of the left button.
6. The jump button follows the same pattern, except you place it 70 pixels from the right edge of the screen. This way the button is in the correct place whether the player is running your game on 3.5 or 4-inch iPhone screen or an iPad, as long as you pass in the screen size to the initializer.
7. Finally, you create the buttons array and give it the three buttons. Later you will use this array to iterate through the buttons and check for touch locations within the bounds of each.

To see your work so far, you need to add the HUD to the main game scene. Add the following import statement to **PSKLevelScene.m**:

```
#import "PSKHUDNode.h"
```

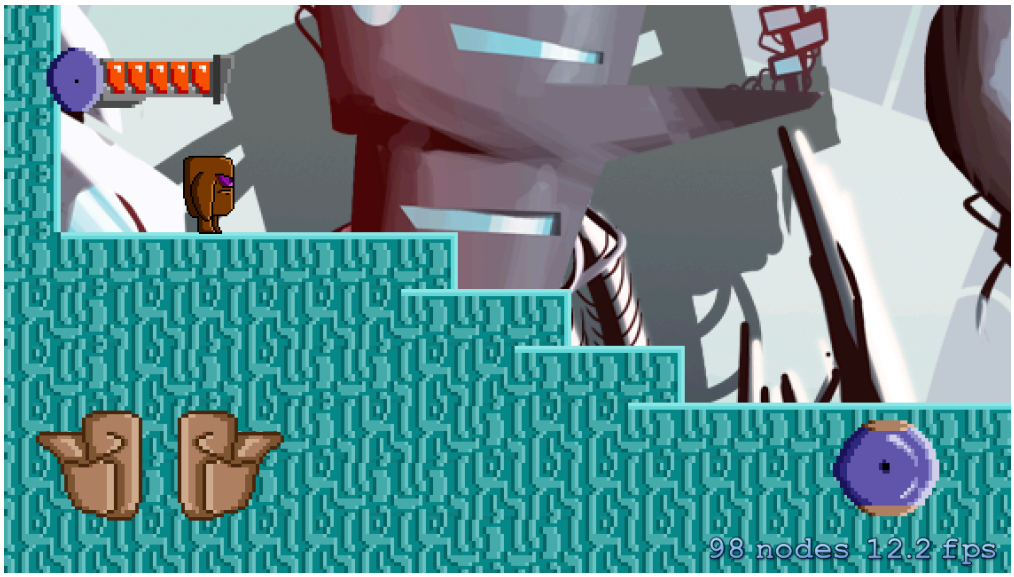
Then add a property to the `@interface` block:

```
@property (nonatomic, strong) PSKHUDNode *hud;
```

Finally, add this code to the last line of `initWithSize:level:`

```
self.hud = [[PSKHUDNode alloc] initWithSize:size];
[self addChild:self.hud];
self.hud.zPosition = 1000;
```

Build and run. You should see your new HUD images nicely placed on top of the map and background.

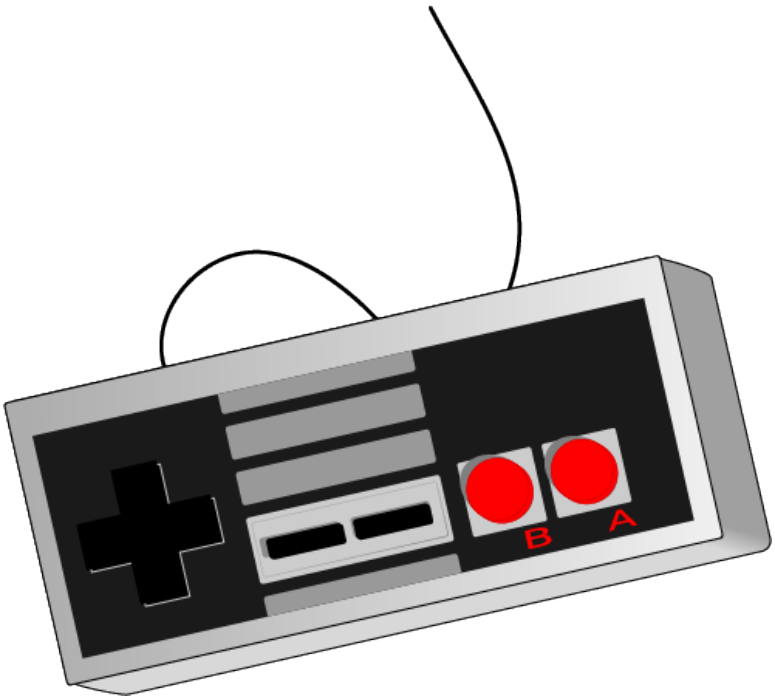


Everything looks great! But now it's time for the hard work: making these sprites do something. This brings you to the next step, which is to set up the touch handling so the buttons react to user touches.

Responsive touch handling

The most important part of a platformer game is the physical connection between the player and the game character. The input from the controller is a critical part of this connection. Finicky or unresponsive controls compromise the core fun of the platformer-style game and can frustrate the player.

It's worth taking a minute to talk about how you would use a physical controller. Imagine an original NES controller or the equivalent. Remember how you could press more than one button simultaneously? Many games required you to run and jump at the same time, or move forward, press the sprint button and then jump—all in quick succession. Developing these skills was part of the fun. That's what you want to replicate in your touch screen controller.



One limitation of an NES controller was that the player could not press both forward and backward at the same time. The D-pad allows you to press down and forward simultaneously (diagonal down), but not down and up. This makes it easier to switch directions quickly, which is essential in a platformer game. In your game, you will want to prevent the player from pressing the forward and back buttons at the same time and make it easy to switch between them.

One last thing that's worth mentioning, and is perhaps the most problematic issue, is that touch screens don't have any physical features beyond a flat surface. It is hard for you to know where your finger is on the screen—you can look, but usually your finger obscures the exact position. You don't get any tactile confirmation that you've moved from one button to another.

Many touch screen games try to build around this limitation. One way is to increase the size of the hit regions so the player doesn't have to hit the button exactly to register a button press. Another strategy is to create a floating button set so that the player doesn't have to hit a region perfectly, only move his or her finger in a certain direction. The relative position of the touch is interpreted as the new direction. This second solution is mostly for D-pads.

A third common touch screen interface strategy is to limit the number of buttons so that there are fewer ways to accidently miss or press the wrong button. The most successful platformers on iOS do this.

In this game, you'll use all three of these strategies to make the touch screen work as a controller: use only three buttons, enlarge the hit regions so that a close touch registers as a button press, and prevent player from pressing the front and back buttons simultaneously.

First implement the `touchesBegan:` method as follows in **PSKHUDNode.m**:

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  //1
  for (UITouch *touch in touches) {
    //2
    CGPoint touchLocation = [touch locationInNode:self];
    //3
    for (SKSpriteNode *button in self.buttons) {
      if (CGRectContainsPoint(CGRectInset(
                    button.frame, -25, -50), touchLocation)) {
        //4
        if (button == self.jumpButton) {
          [self sendJump:YES];
          //5
        } else if (button == self.rightButton) {
          [self sendDirection:kJoyDirectionRight];
        } else if (button == self.leftButton) {
          [self sendDirection:kJoyDirectionLeft];
        }
      }
    }
  }
}
```

1. The first step is to iterate through the `touches`. The `NSSet` includes all touches so you can handle multiple touches at once.
2. Sprite Kit adds a new method to `UITouch` called `locationInNode` that gives the location of the touch within the provided node.
3. Next, you iterate though the array of buttons you created earlier. You could also have hard-coded the buttons and checked them one at a time. Since there are only three, it wouldn't be much more code, but I find this structure more convenient.

   For each button, you check whether the touch location is within a `CGRect`. The `CGRect` starts with the frame (bounding box) of the button, but you are enlarging it with the convenience function `CGRectInset`. This function shrinks the rectangle by the provided height and width. By passing a negative height and width, you expand the `CGRect` instead of shrink it. This makes it easier to hit the button so the player doesn't have to touch exactly within the image's bounds.
4. At this point, you compare the `button` variable—the sprite in the current iteration of the array—with your properties to figure out which one the player has touched. If it's the jump button, you call `sendJump:YES`. You haven't implemented this method yet; I'll explain it in further detail below.
5. Here you check the left and right buttons for a match and send the appropriate direction `enum` to the `sendDirection` method, which you also have yet to implement.

Because `touchesEnded:` follows the same pattern, add it now:

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
```

```
    //1
    for (UITouch *touch in touches) {
      CGPoint touchLocation = [touch locationInNode:self];
      for (SKSpriteNode *button in self.buttons) {
        if (CGRectContainsPoint(CGRectInset(
                        button.frame, -25, -50), touchLocation)) {
          if (button == self.jumpButton) {
            [self sendJump:NO];
          //2
          } else {
            [self sendDirection:kJoyDirectionNone];
          }
        }
      }
    }
  }
```

1. You can see that this is nearly identical to `touchesBegan:`. In fact, it follows the same logic, but calls the inverse methods—`sendJump:NO` instead of `YES`.
2. It doesn't matter whether the player has released the right or left button. Remember, you are enforcing that only one direction can be pressed at a time. So if the player removes her finger and it was on the left button, you know for sure that the right button wasn't also in a pressed state. You can send `kJoyDirectionNone` either way.

That's it for this block of code. It's time to talk a little about why you are using these `sendDirection` and `sendJump` methods.

Here's why. You want an extra layer of abstraction. There are a couple of reasons for this:

- First, you will be calling these methods in all of the touch handling methods, so you don't want to repeat the same code three times.
- Second, you may want to send the direction or jump commands from something other than the onscreen controls. Using abstract methods allows you to connect an external controller, like another device via Bluetooth, an iCade or even Apple's new external controller—when we get it.

It's time to add empty implementations of these methods so you can continue without the warnings and errors. Add this code to **PSKHUDNode.h**:

```
typedef NS_ENUM(NSInteger, JoystickDirection) {
  kJoyDirectionNone,
  kJoyDirectionLeft,
  kJoyDirectionRight
};

typedef NS_ENUM(NSInteger, JumpButtonState) {
  kJumpButtonOn,
  kJumpButtonOff
};
```

These are `enum` values that you will use to communicate the state of the buttons both internally and externally. An `enum` or enumeration is a set of constants that you define to represent something.

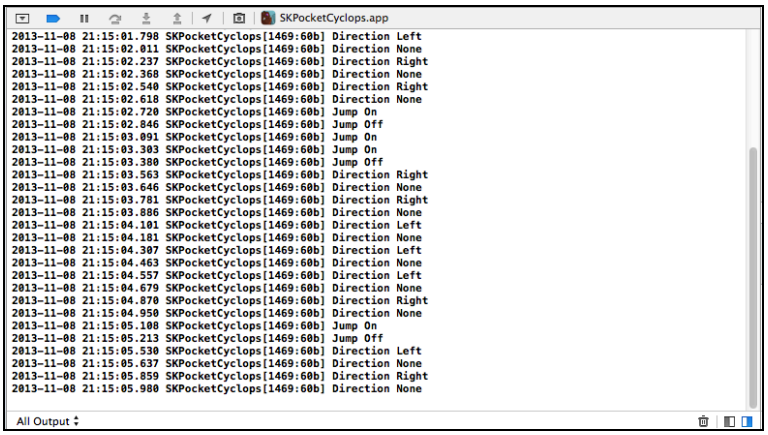
Next, add partial implementations of `sendDirection:` and `sendJump:` to **PSKHUDNode.m**. First you'll simply log the state to the console to validate that `touchesBegan` and `touchesEnded` are working as you'd expect.

```
- (void)sendJump:(BOOL)jumpOn {
  //1
  if (jumpOn) {
    //2
    NSLog(@"Jump On");
  } else {
    NSLog(@"Jump Off");
  }
}

- (void)sendDirection:(JoystickDirection)direction {
  //3
  if (direction == kJoyDirectionLeft) {
    NSLog(@"Direction Left");
  } else if (direction == kJoyDirectionRight) {
    NSLog(@"Direction Right");
  } else {
    NSLog(@"Direction None");
  }
}
```

1. You check the Boolean sent into the jump method to determine if the message is jump on or jump off. This is an example of a state machine, which I'll talk about later.
2. Then you log the appropriate message.
3. In this third method, you aren't using a Boolean, but one of three `enum`s: Left, Right, or None. These three states are mutually exclusive.

Build and run now. As you press the buttons, you should see the console logging the state.



Let's update the method to do something more than print the state to the console. For now, you are just going to turn the button images on and off. After you finish with the touch handling code, you can add the code to communicate the state of the buttons to the rest of the app.

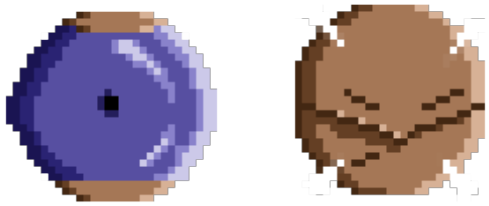
Add this code to **PSKHUDNode.m**:

```
- (void)sendJump:(BOOL)jumpOn {
  if (jumpOn) {
    [self.jumpButton setTexture:[[PSKSharedTextureCache
              sharedCache] textureNamed:@"jumpButtonPressed"]];
  } else {
    [self.jumpButton setTexture:[[PSKSharedTextureCache
              sharedCache] textureNamed:@"jumpButton"]];
  }
}

- (void)sendDirection:(JoystickDirection)direction {
  if (direction == kJoyDirectionLeft) {
    [self.leftButton setTexture:[[PSKSharedTextureCache
               sharedCache] textureNamed:@"leftButtonPressed"]];
    [self.rightButton setTexture:[[PSKSharedTextureCache
                     sharedCache] textureNamed:@"rightButton"]];
  } else if (direction == kJoyDirectionRight) {
    [self.rightButton setTexture:[[PSKSharedTextureCache
              sharedCache] textureNamed:@"rightButtonPressed"]];
    [self.leftButton setTexture:[[PSKSharedTextureCache
                      sharedCache] textureNamed:@"leftButton"]];
  } else {
    [self.rightButton setTexture:[[PSKSharedTextureCache
                     sharedCache] textureNamed:@"rightButton"]];
    [self.leftButton setTexture:[[PSKSharedTextureCache
                      sharedCache] textureNamed:@"leftButton"]];
  }
}
```

These two methods are logically pretty simple at this point. All you do is set a new texture on the button based on the input Boolean or `enum` value.

The **sprites.atlas** contains two images for each button: one normal image and another with the word "Pressed" tagged onto the end of the name that is slightly dimmed, or closed in the case of the eye/jump button. Here's what the **jumpButton** and **jumpButtonPressed** images look like:



`SKSpriteNode` has a method to change the image the sprite displays. The underlying image data is called a texture, and the `setTexture:` method allows you to set a new `SKTexture` object for that sprite. Normally Sprite Kit loads the `SKTexture` object automatically when you create a new sprite. But if you want to change the image a sprite displays, you can load a new `SKTexture` object and set it on an existing sprite.

The direction logic does one extra thing that the jump logic doesn't have to worry about. If a direction is sent, then the opposite direction has to be turned off. In other words, if the player pressed the right button, you need to set the right button to the pressed image *and* set the left button to the normal image. Sometimes this will be a redundant step. Perhaps the left button wasn't in a pressed state. That's OK. It doesn't take much extra work to set it to the normal image again.

In *Pocket Cyclops*, you will use a custom texture management object called `PSKSharedTextureCache`. This object preloads and keeps track of texture objects for your game. In games, preloading textures is a common practice. It takes a significant amount of time to load a texture into memory, but once loaded it's very quick to retrieve and display it in your scene. This is why most games have loading periods before the game starts.

You will create this texture management object next.

Note: There are different ways to create an `SKTexture`. One of the benefits of Sprite Kit over other open source gaming engines, like Cocos2D, is how easy it makes managing texture objects and atlases.

Sprite Kit has an API for managing texture atlases. Xcode automatically creates a texture atlas when you add a folder with the extension **.atlas**. `[SKTextureAtlas atlasNamed:]` will get you a reference to that texture atlas. From there you can call `textureNamed:` on the `SKTextureAtlas` to get an individual `SKTexture` object from that atlas.

Another function of Sprite Kit texture atlases is the `preloadTextures` method. This call loads all the textures from that atlas into memory. Once this is done,

your game shouldn't need to load them again, resulting in big performance benefits. However, Sprite Kit may unload and reload textures behind the scenes to maximize memory and performance.

There's just one problem. At the time of writing, there is a bug in Sprite Kit that loads these texture atlases way more often than necessary and at the wrong times. This negatively affects performance—it is dramatic on older devices, like the iPhone 4. The solution is to preload the textures individually yourself and keep a reference to them. If you have a reference to the texture you need, then Sprite Kit shouldn't unload and reload when you don't want it to.

In this book, you'll create the `PSKSharedTextureCache` object to preload and keep references to all the textures in an atlas so that you can use them later without having to load them at that time. Without this object, *Pocket Cyclops* will reload the texture atlas every time a button is changed. The atlas contains all the images for all the moving elements in the game—talk about inefficient!

When this bug is fixed, the `PSKSharedTextureCache` will become obsolete and you will want to use the built-in calls to `atlasNamed`, `textureNamed` and `preloadTextures` to load textures.

The shared texture cache

You will use your shared texture cache keep track of texture objects. By keeping a reference to an `SKTexture`, you can take control of when Sprite Kit loads and unloads it. For the time being, you'll want to do this, as Sprite Kit doesn't always make the right decisions about when might be the best time to load an `SKTexture` from a disk.

The class is simple. It holds an `NSMutableDictionary` that has a reference to each texture loaded with that texture's name as a key without the .png extension. All it needs are methods to add and remove these objects. Really, the class is little more than a wrapper around the dictionary that lets you treat it as a singleton for easy access from any class.

Create a new class by going to **File->New->File**. Choose the **iOS\Cocoa Touch\Objective-C class** template, make it a subclass of **NSObject** and give it the name **PSKSharedTextureCache**. The default location for this new class should be in the same folder as the other files, and that's fine.

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

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

@interface PSKSharedTextureCache : NSObject
```

```
+ (instancetype)sharedCache;

- (SKTexture *)textureNamed:(NSString *)name;

- (void)addTexture:(SKTexture *)texture
              name:(NSString *)textureName;

@end
```

There are three methods you need to implement. The first is a standard singleton class method that will always return the same instance of the object. You will only ever need to use the one instance. The singleton pattern makes it easier to keep track of the right instance that holds all your texture references.

The second method returns an `SKTexture` for a given name and the third method adds a texture with a supplied name.

Replace the contents of **PSKSharedTextureCache.m** with the following code:

```
#import "PSKSharedTextureCache.h"

//1
@interface PSKSharedTextureCache ()
@property (strong, nonatomic) NSMutableDictionary *textures;
@end

@implementation PSKSharedTextureCache

//2
+ (instancetype)sharedCache {
  static dispatch_once_t pred;
  static PSKSharedTextureCache *sharedCache;
  dispatch_once(&pred, ^{
    sharedCache = [PSKSharedTextureCache new];
  });
  return sharedCache;
}

//3
- (id)init {
  if ((self = [super init])) {
    self.textures = [NSMutableDictionary dictionary];
  }
  return self;
}

//4
- (SKTexture *)textureNamed:(NSString *)textureName {
  if ([textureName pathExtension]) {
    textureName = [textureName stringByDeletingPathExtension];
  }
```

```
    return self.textures[textureName];
}

//5
- (void)addTexture:(SKTexture *)texture
              name:(NSString *)textureName {
    if ([textureName pathExtension]) {
        textureName = [textureName stringByDeletingPathExtension];
    }
    self.textures[textureName] = texture;
}

@end
```

1. The new `@interface` block adds a property for the mutable dictionary that keeps track of the textures. Because you've added this property here instead of in the header, it will only be accessible internally.
2. This is the way you access a singleton instance of the class under ARC. This is a very common way to implement the singleton pattern. `dispatch_once()` is a GCD call that ensures the initialization code only gets called once, and the static `sharedCache` variable means that it will always point to the same object.

> **Note:** For more information about the singleton pattern or its use under ARC, see this Stack Overflow thread:
>
> http://stackoverflow.com/questions/7568935/how-do-i-implement-an-objective-c-singleton-that-is-compatible-with-arc

3. This is the initializer for the class and its only purpose is to create the dictionary.
4. This first method is how you retrieve an `SKTexture` from the dictionary by name. You add a little bit of code here to strip the .png extension if it has been provided. This way, you can use or not use the file extension and it will work.
5. Finally, you create a method that adds an `SKTexture` to the dictionary. The dictionary key will be the filename without the extension.

That's it for the shared texture cache. Now that you've got the singleton class, it's time to load the textures when you load your scene so that they'll be available when you need them.

First, import the class into **PSKLevelScene.m:**

```
#import "PSKSharedTextureCache.h"
```

Next, add a few lines of code to `initWithSize:level:`, right before the line that initializes the `self.player` object:

```
SKTextureAtlas *atlas = [SKTextureAtlas atlasNamed:@"sprites"];
```

```
for (NSString *textureName in [atlas textureNames]) {
  SKTexture *texture = [atlas textureNamed:textureName];
  [[PSKSharedTextureCache sharedCache] addTexture:texture
                                             name:textureName];
}
```

You get a reference to the `SKTextureAtlas` object created for the images in the **sprites.atlas** folder. `SKTextureAtlas` has a method called `textureNames` that you can use to get all the names of the individual image files in the folder.

Next, you iterate through the texture names from this method and create an `SKTexture` object for each name. Finally, you add that object to your `PSKSharedTextureCache` singleton. Now you have a reference to those texture objects and Sprite Kit shouldn't unload them—and need to reload them—later.

You want to make sure you've preloaded all your textures with this method before you try to access them in any other class. In this case, you will access the textures in the HUD, the `Player` class and in all the `PSKEnemy` subclasses. Make sure you've preloaded before you access any of those classes.

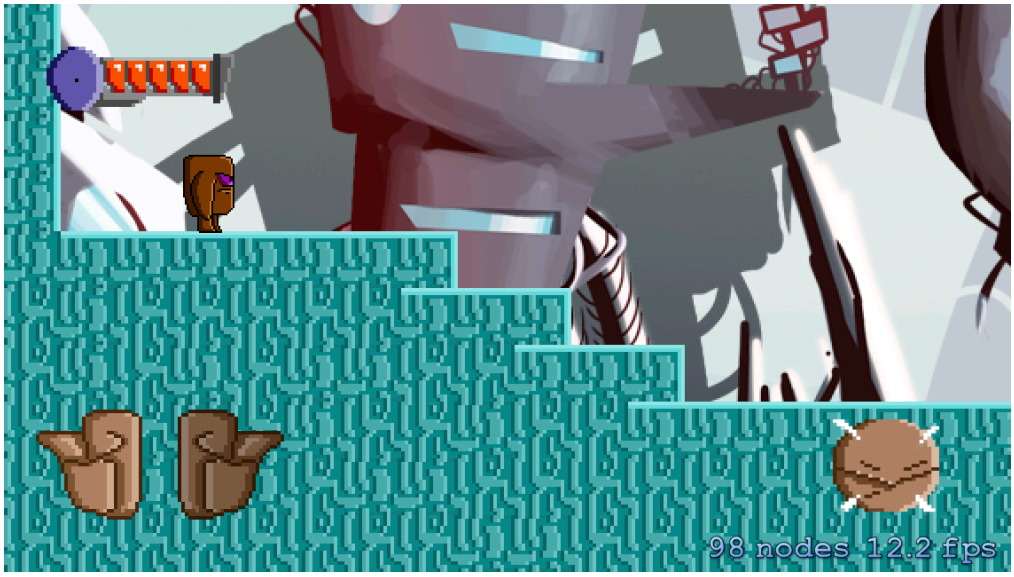
Responsive buttons

Now that you have a better way to access your textures, you can return to the HUD and do something with all that touch handling code you put in place earlier.

The only thing left to do is import the `PSKSharedTextureCache` into **PSKHUDNode.m**:

```
#import "PSKSharedTextureCache.h"
```

Build and run now. You should be able to change the images of the buttons by pressing them.



At this point you've only implemented `touchesBegan:` and `touchedEnded:`, so you can press a button and remove your finger with the result that it will change the image and then change it back.

However, if you press a button and then slide your finger off, the button remains changed. Also, if you press down outside of the button and then slide onto the button, it doesn't recognize the press. You can't slide your finger from the left to right buttons to change direction. But that's the behavior that will make the onscreen controls feel easy to use, so how do you implement it?

For this to work, you need the most complex method, `touchesMoved:`. Don't worry—it's not *too* complex. Add this code to **PSKHUDNode.m**:

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event {
  for (UITouch *touch in touches) {
    //1
    CGPoint touchLocation = [touch locationInNode:self];
    CGPoint previousTouchLocation = [touch
                                     previousLocationInNode:self];

    for (SKSpriteNode *button in self.buttons) {
      //2
      if (CGRectContainsPoint(
              button.frame, previousTouchLocation) &&
          !CGRectContainsPoint(button.frame, touchLocation)) {
        //3
        if (button == self.jumpButton) {
          [self sendJump:NO];
        } else  {
          [self sendDirection:kJoyDirectionNone];
        }
      }
                                     }
    //4
    for (SKSpriteNode *button in self.buttons) {
      if (!CGRectContainsPoint(
                  button.frame, previousTouchLocation) &&
          CGRectContainsPoint(button.frame, touchLocation)) {
        //5
        //We don't get another jump on a slide-on, we want the
        //player to let go of the button for another jump
        if (button == self.rightButton) {
          [self sendDirection:kJoyDirectionRight];
        } else if (button == self.leftButton) {
          [self sendDirection:kJoyDirectionLeft];
        }
      }
    }
  }
}
```

This code executes in two phases, iterating through the buttons array twice. The first time, you look for instances where the touch point was previously within a

button's region, but isn't now—a slide release. In the second pass, you look for touches that weren't initially within a touch region, but now are—a slide press.

1. To help figure this out, you'll use a new `UITouch` method from Sprite Kit called `previousLocationInNode`. This gives you the last touch point before the current one; `locationInNode`, as you've seen, gives you the current touch location. With these two touch points you can figure out if the player has pressed or released any new buttons.

2. You test both touch points to see whether the state of a button should change. In this first test, you are looking for the previous touch location to be inside the touch region of a button and for the current touch to be outside the touch region of that same button—the slide release. If the last location and current location are both inside or both outside of the button's region, you don't need to do anything because this code keeps track of state, and you'll implement more of that in a minute. You only need to worry about things changing.

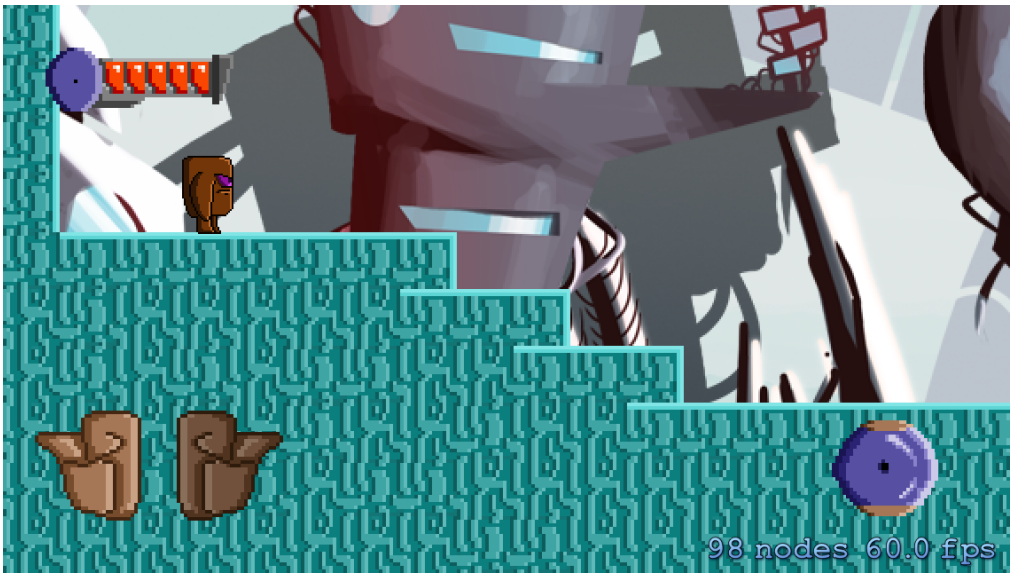
   You might wonder what happens if you slide off of one button (release) and onto another button (press) in the same `touchesMoved` method. That is automatically handled in the two iterations of the loop. Each time the test runs, you are only looking at a single button. One iteration of the loop handles the release and another iteration handles the press action.

3. Once you know the player has released a button, you simply check each button for a match and send the appropriate button change message. Like `touchesEnded`, you don't need to check whether it's the left or right button; once you know the player's released a direction button, you send the `kJoyDirectionNone`.

4. The second time you run through the buttons array, you simply invert the operation. You look for a last location that was outside of a button's touch region and a current location inside the button's touch region. This means the player pressed the button. Once you know that's true, you check whether it's the left or right button.

5. Notice that in this loop, you don't check for the jump button. In my game, I don't want a slide-on to trigger the jump button. To jump, the player needs to release and press again. This is a design choice and you should feel free to make a different choice in your game.

That's it for the touch methods. Build and run now, and test the direction-switching and jump-pressing to trigger the button states.



> **Note:** From this point on, it may be better to do most of your builds on a device. The Simulator doesn't give you an easy way to use the multi-touch screen to simultaneously press buttons, which will make it difficult to navigate through the levels.

Communicating with the HUD node

There's only one small thing left to do. You need a way for other classes to communicate with the HUD node. This will take two forms: properties that will reveal the state of the buttons and a method to update the life indicator.

Add the following code to **PSKHUDNode.h** inside the `@interface` block:

```
@property (nonatomic, assign) JoystickDirection joyDirection;
@property (nonatomic, assign) JumpButtonState jumpState;
```

You will query the state of these properties to determine how to move the cyclops around in the next few chapters.

Now add some new properties to the `@interface` block in **PSKHUDNode.m**:

```
@property (nonatomic, assign) BOOL jumpButtonPressed;
@property (nonatomic, assign) BOOL leftButtonPressed;
@property (nonatomic, assign) BOOL rightButtonPressed;
```

These new Booleans keep track of the state of the buttons. You will use them to set the state of the properties above.

Now change `sendDirection` and `sendJump` to set the state of those Booleans, as well as the image states, like this:

```
- (void)sendJump:(BOOL)jumpOn {
  if (jumpOn) {
    self.jumpButtonPressed = YES;
    [self.jumpButton setTexture:[[PSKSharedTextureCache
                sharedCache] textureNamed:@"jumpButtonPressed"]];
  } else {
    self.jumpButtonPressed = NO;
    [self.jumpButton setTexture:[[PSKSharedTextureCache
                     sharedCache] textureNamed:@"jumpButton"]];
  }
}

- (void)sendDirection:(JoystickDirection)direction {
  if (direction == kJoyDirectionLeft) {
    self.leftButtonPressed = YES;
    self.rightButtonPressed = NO;
    [self.leftButton setTexture:[[PSKSharedTextureCache
                sharedCache] textureNamed:@"leftButtonPressed"]];
    [self.rightButton setTexture:[[PSKSharedTextureCache
                       sharedCache] textureNamed:@"rightButton"]];
  } else if (direction == kJoyDirectionRight) {
    self.rightButtonPressed = YES;
    self.leftButtonPressed = NO;
    [self.rightButton setTexture:[[PSKSharedTextureCache
              sharedCache] textureNamed:@"rightButtonPressed"]];
    [self.leftButton setTexture:[[PSKSharedTextureCache
                     sharedCache] textureNamed:@"leftButton"]];
  } else {
    self.rightButtonPressed = NO;
    self.leftButtonPressed = NO;
    [self.rightButton setTexture:[[PSKSharedTextureCache
                       sharedCache] textureNamed:@"rightButton"]];
    [self.leftButton setTexture:[[PSKSharedTextureCache
                     sharedCache] textureNamed:@"leftButton"]];
  }
}
```

At each point, you've simply added a line of code to set the appropriate Boolean to match the state of the button.

Finally, add the getters for the properties you added to the header file. The getters won't return the underlying instance variables but will calculate the values based on the Boolean properties.

```
#pragma mark - Properties

- (JumpButtonState)jumpState {
  if (self.jumpButtonPressed) {
    return kJumpButtonOn;
  }
  return kJumpButtonOff;
}
```

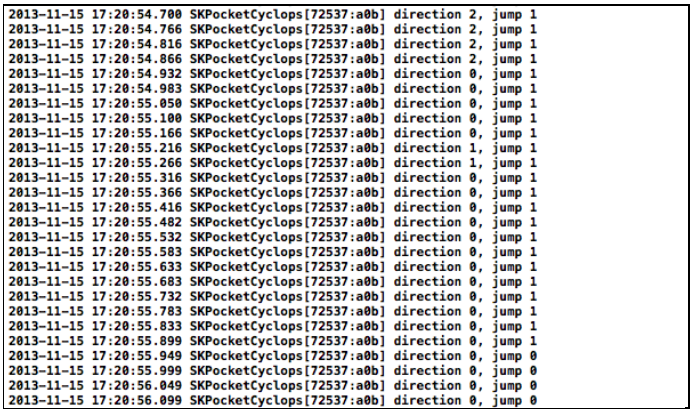
```
- (JoystickDirection)joyDirection {
  if (self.leftButtonPressed) {
    return kJoyDirectionLeft;
  } else if (self.rightButtonPressed) {
    return kJoyDirectionRight;
  }
  return kJoyDirectionNone;
}
```

This code should be straightforward. Each method looks at the states of the Booleans, which were set by the player pressing or releasing the buttons, and returns the appropriate `enum` value that abstracts the intention of those buttons.

To test this, add the following `NSLog` statement to your `update:` method in **PSKLevelScene.m**:

```
NSLog(@"direction %ld, jump %ld", (long)self.hud.joyDirection,
                                  (long)self.hud.jumpState);
```

Build and run now. As you press the buttons, you should see the console logging the button state changes.



Remove that log statement from the `update:` method. Now that you've verified that the buttons are working, you don't need it.

Lastly, you need a method that allows the game to update the state of the life indicator.

Add the method declaration to **PSKHUDNode.h**:

```
- (void)setLife:(CGFloat)life;
```

Now add the implementation to **PSKHUDNode.m**:

```
- (void)setLife:(CGFloat)life {
  //1
  int num = (int)(life * 5);
  //2
```

```
    NSString *lifeFrame = [NSString stringWithFormat:
                                          @"Life_Bar_%d_5", num];
    //3
    [self.lifeBarImage setTexture:[[PSKSharedTextureCache
                            sharedCache] textureNamed:lifeFrame]];
    //4
    [self.lifeBarImage setSize:self.lifeBarImage.texture.size];
}
```

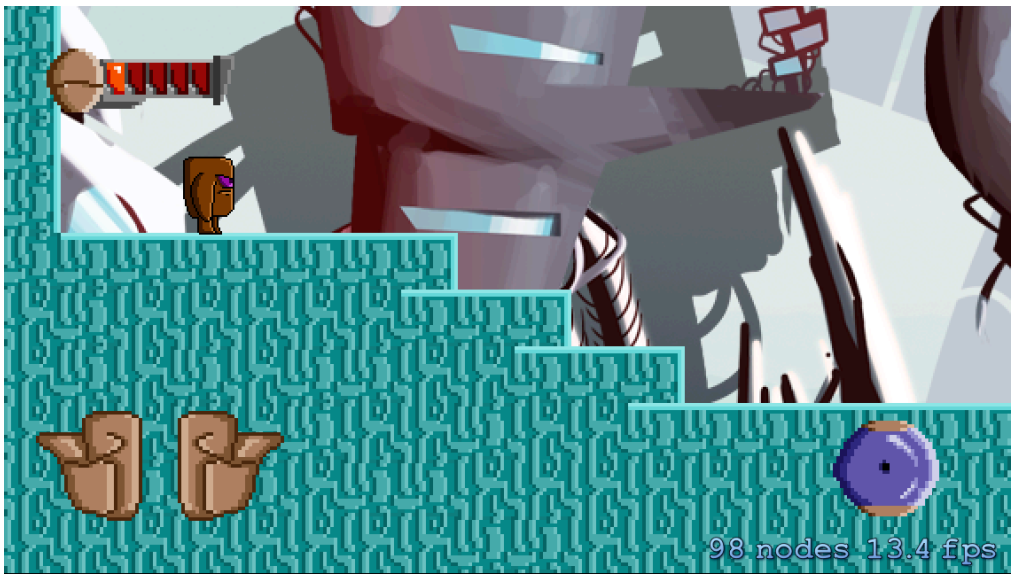
1. First, you call this method with any value between 0.0 and 1.0. However, there are only six life state images—zero through five bars. This first line of code converts the input float into a number between 0 and 5.
2. Each image that represents a life state has a filename that follows a convention, **Life_Bar_x_5.png**, where x is a value between 0 and 5. This line constructs a filename based on the value in `num`.
3. Next, you retrieve the `SKTexture` object from the `PSKSharedTextureCache` and set it on the `lifeBarImage` sprite.
4. This line of code comes with a slightly longer explanation (see below). At the time of writing, there's a bug in Sprite Kit. This is the workaround.

To test this code, add this line to the `update:` method of **PSKLevelScene.m**:

```
[self.hud setLife:fmod(currentTime, 1.2)];
```

This is simply a quick and dirty way to change the setting of the life value so you can see the HUD updating. It does an `fmod` on the current time, which is constantly incrementing, to find the remainder when dividing by 1.2.

Build and run now, and you can see the life bar rolling through its states.



Remove that line of code—later you'll update the life indicator from the `Player` class based on interactions with enemy objects.

Note: Let's discuss the problem with sprite sizes. The sprite size is the frame of the displayed image on the screen, measured in points. Texture size is the size in points of the underlying image data. When these two values are the same, the scale of the sprite is equal to 1.0 (or 100%). Normally when you call `setTexture:` on a sprite, the sprite's size should be updated to match the texture size. Scale should be preserved, but this doesn't always happen.

If all the textures you set on a sprite are the same size, you don't have to worry about this, yet there is another complication. Sprite Kit trims transparent pixels when it creates the texture atlas, so even though your input images have the same dimensions on the way in, they may not have the same dimensions in the atlas. Sprite Kit should handle this seamlessly, but it doesn't.

To demonstrate this, comment out line 4 in the code above, build and run, and watch how the life indicator changes size. What's actually happening? The sprite's dimensions stay the same, but the underlying texture size is changing. You have to call `setSize:` and pass in the size of the texture to force the sprite's size to update.

This can cause complications if you are changing the sprite's `xScale` and `yScale` properties, because setting the size of the sprite to the size of the texture resets the scale to 1.0. I have heard that you can use the third-party app Texture Packer to create texture atlases, instead of Xcode, to work around this problem.

This problem only appears to occur when you call `setTexture:`. Changing a texture using animation or other methods is fine, as you'll see later.

That's all for this chapter. Now that you have a HUD in place, you can get on to the good stuff—making the cyclops move!

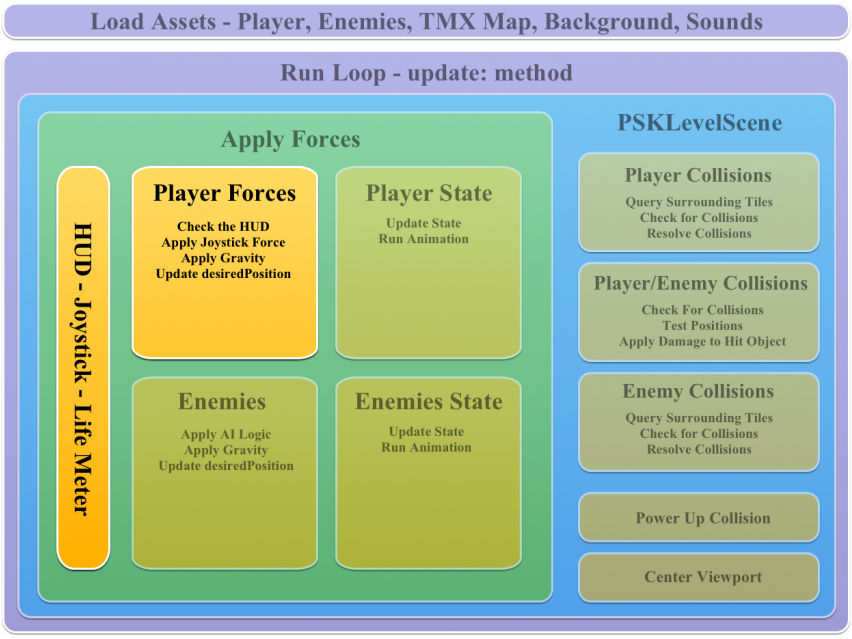
Challenge: Recall how in *Super Mario Bros.*, pressing and holding the B button makes Mario run. How would you implement a dash button? This is a particularly difficult task on a touch screen. Some apps use a tap to activate and keep the dash button activated until the player removes his or her finger from the screen, whether or not the finger stays on the button. Does that make sense to you?

Chapter 5: Moving the Player

In this chapter, you'll explore the interaction between the controls on the heads-up display (HUD) and the cyclops. You will also learn about using forces to move the player character around and the mechanics of the all-important platformer jump.

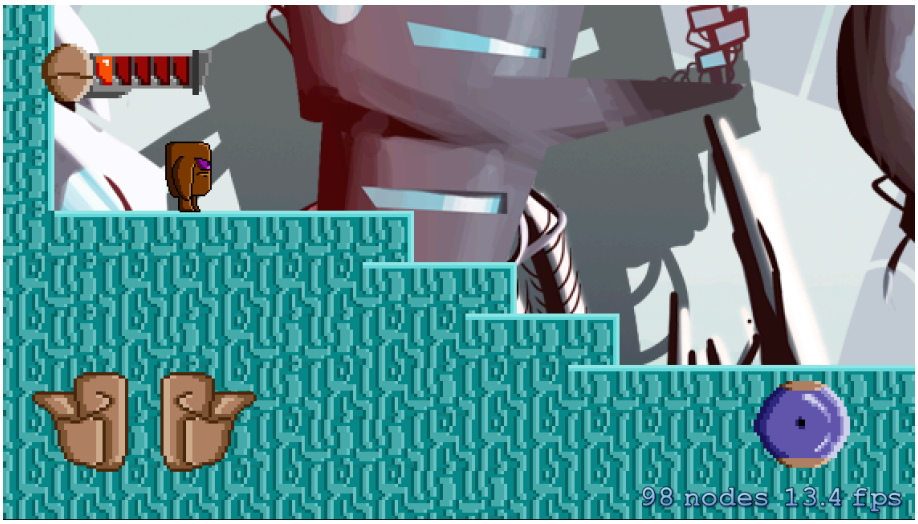
This will be a short and fun chapter, so limber up and get ready to do some coding!

As far as how this chapter fits into your overall game map, it completes the HUD portion, plus the section of the `Player` class that queries the HUD state and applies physics forces:



Talking to the HUD

In the last chapter, you created the HUD layer and made it interpret touches and set the button states, as well as communicate updates to the life indicator.



Of course, your goal is that pressing the buttons with move the cyclops around, not just log to the console. But how do you make the `Player` object do something, considering it's in a completely separate node? Similarly, you'll sometimes need the `Player` object to tell the HUD to do something, such as update the life indicator when the cyclops is harmed.

There are many ways to get the HUD and the `Player` object talking to each other. One way is to create delegates and set up the HUD to communicate button presses by sending messages to either the `Player` or the `PSKLevelScene` class. You could also set the HUD as a delegate of those classes and have them communicate back to the HUD when the player's life score changes.

Another approach is to use the observer pattern by setting up `NSNotifications` to communicate back and forth. To learn more about `NSNotifications`, check out this tutorial:

- http://www.raywenderlich.com/4295/multithreading-and-grand-central-dispatch-on-ios-for-beginners-tutorial

For now, let's put that question aside and focus on implementing movement. To do this, you will query the state of the buttons from the `Player`'s `update:` method. If the buttons are in the pressed state, you know it's time to move the cyclops!

Why have the `Player` object poll the state of the controls rather than have the controls notify the `Player` object? In the next chapter, you'll explore the `Player` object's **state machine**, the system that will determine whether the player character is running or jumping. The animations that the cyclops displays at any point will depend on changes to the state machine.

> **Note**: With a state machine, the character can have one—and only one—state at a time. The state dictates the current visual representation of the player, whether single frame or animation, and provides information about which states are valid when transitioning between states. The logic for all characters will rely heavily on the current state.

Because a state machine will drive the player, and eventually the enemies too, it will be conceptually easier to handle the user input from the HUD in the state machine as well. Every time the `Player`'s `update:` method is called, it will query the state of the HUD buttons and handle them according to their state.

Because you've already written the HUD code, you'll do most of this chapter's work within the `Player` class. You need a way for the `Player` object to access the state of the HUD, so you're going to create a `@property` for the HUD in the `Player` class.

First, add an import for the `PSKHUDNode` class in **Player.h**:

```
#import "PSKHUDNode.h"
```

Then, add an `@property` declaration for the HUD in the `@interface` block:

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

Next, you need to assign the HUD object to this new `hud` property. You need to do this after you've created both the HUD and the `Player` objects, so add this line to the bottom of `initWithSize:level:` in **PSKLevelScene.m**, right after the line that sets the `zPosition` of the HUD:

```
self.player.hud = self.hud;
```

Controlling the player

Now that you have a reference to the HUD in the `Player` class, you can talk to the HUD from within `Player` to get the HUD's state information and update it if necessary. That means you can add the code that controls the `Player` object.

Move to `update:` in **Player.m** and add the following code before the existing code:

```
//1
CGPoint joyForce = CGPointZero;
//2
if (self.hud.joyDirection == kJoyDirectionLeft) {
  //3
  self.flipX = YES;
  joyForce = CGPointMake(-kWalkingAcceleration, 0);
} else if (self.hud.joyDirection == kJoyDirectionRight) {
  self.flipX = NO;
```

```
    joyForce = CGPointMake(kWalkingAcceleration, 0);
  }
  //4
  CGPoint joyForceStep = CGPointMultiplyScalar(joyForce, dt);
  //5
  self.velocity = CGPointAdd(self.velocity, joyForceStep);
```

Here's what's happening in the code above:

1. First, you introduce a new `joyForce` variable and set it to zero. This variable will store a force value in the x-dimension. You could use a simple `float` here, but to easily add the force later using `CGPointAdd`, you use a `CGPoint` instead.
2. Next, the `if` statement looks at the `joyDirection` property of the HUD to see if has a state of left or right. If it finds a direction, you set `joyForce` to positive or negative `kWalkingAcceleration`. If there is no direction—meaning neither button is in the pressed state—you simply leave `joyForce` at zero. `kWalkingAcceleration` is a constant that you'll add in just a minute.
3. This section also sets the `flipX` property. `flipX` is a new property that you can use to horizontally flip the image. This will make coding certain parts of the movement logic easier because you won't need both right- and left-facing images. Here you set `flipX` depending on the direction, so that the player character is facing the correct way. You'll add `flipX` in a minute as well.
4. You scale the `joyForce` value to the size of the current time step. Once again, this is to keep things moving at a constant speed, irrespective of the frame rate. If the value is zero, scaling it has no effect.
5. Finally, you add the scaled value to the current velocity.

It's time to add the `flipX` property. You might want to flip any of the sprites in your game, so you should put this variable in the `PSKGameObject` class. Add this to **PSKGameObject.h** in the `@interface` block:

```
@property (nonatomic, assign) BOOL flipX;
```

Now, add the following custom setter code to the implementation file:

```
- (void)setFlipX:(BOOL)flipX {
  if (flipX) {
    //1
    self.xScale = -fabs(self.xScale);
  } else {
    self.xScale = fabs(self.xScale);
  }
  //2
  _flipX = flipX;
}
```

1. In Sprite Kit, you can flip a sprite horizontally or vertically by setting `xScale` or `yScale` to negative values. When you set the `flipX` property to YES, it's going to set the `xScale` to the negative of the current scale value.

2. Then you set the instance variable, as well.

This code has interactions with a number of other `SKSpriteNode` properties. Calling `setScale:`, or setting the individual `xScale` or `yScale`, changes the value of the sprite node's `size` property. On the other hand, changing `size` doesn't change `xScale` or `yScale`, but it does revert the `xScale` property back to a positive value. If you've set `flipX` to YES, which makes `xScale` negative, and then calling `setSize:` will undo the flip.

Scale, size, frame, position and rotation all have interacting relationships. They don't always follow the expected rules, so pay attention and don't rely on the idea that you can set one thing and then read another in a predictable way. Also, when you set one of these properties, make sure you check that it hasn't changed another without you knowing about it.

In this game, you are using `setSize:` to fix the instances where the texture size doesn't match the sprite size. Add a custom setter for `size` that deals with the `flipX` property appropriately so that it won't undo your flip when you call `setSize:`.

Add this code to **PSKGameObject.m**:

```
- (void)setSize:(CGSize)size {
  if (!self.flipX) {
      [super setSize:size];
  } else {
      [super setSize:CGSizeMake(-size.width, size.height)];
  }
}
```

This code preserves the state of `xScale` when you call `setSize:`. You won't need this in place until a little later, but now seemed like a reasonable time to add it.

All that's left is to define `kWalkingAcceleration`. Add this line to **Player.m**:

```
#define kWalkingAcceleration 1600
```

Note: Does that number seem huge? It did to me at first. For this to make sense, remember how the physics engine performs its simulation. This value, like gravity, is a measure of acceleration. It represents how much the player character's velocity *increases* in a second. This value will get scaled by 1/60 of a second—in a perfect world, at least—to about 27. In one time step, this increases the velocity of the player by 27 points per second.

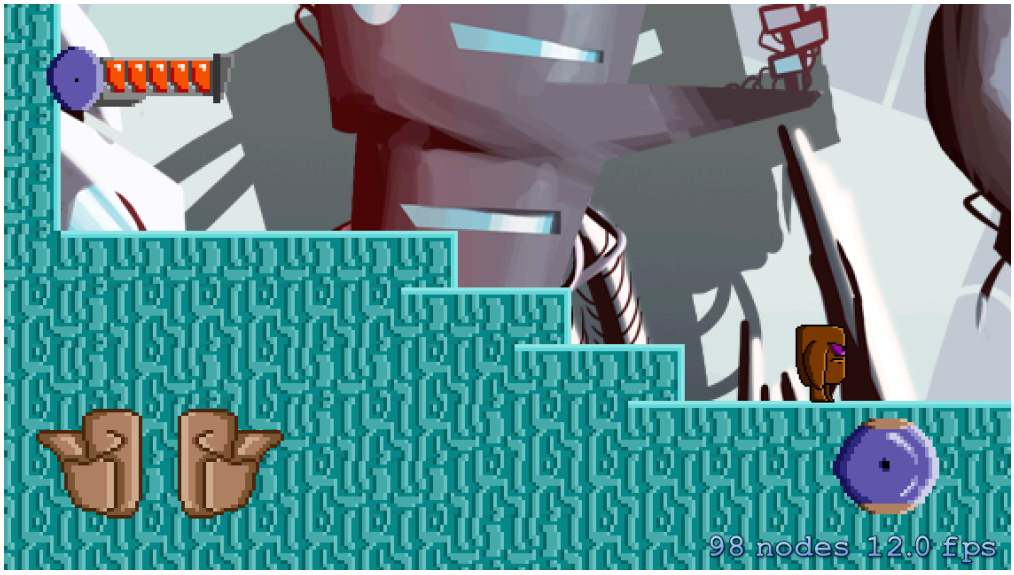
If the cyclops is at a standstill and starts walking, after 1/60th of a second its speed is 27 points per second. After the second time step, 54 points per second. After the third, 71 points per second. That speed value is then scaled again by 1/60 to determine how much the cyclops actually moved during that time step. The first time step only moves it half a pixel!

Increasing this acceleration value decreases the time it takes the cyclops to get to its top speed. You want your character to reach top speed in less than a second, so the acceleration needs to be a pretty big value.

The resulting time until top speed is long enough to make it feel like you're controlling a physical object, because real objects don't go from zero to full speed instantly. But in a fast-moving game, you don't want it to take too long to reach that full speed, or the game is too slow to be fun. The same applies to gravity.

Later, you'll see that you won't scale the jump in the same way. For your purposes, you pretend that the jump force is applied instantaneously. Why? It just feels better!

Build and run, then tap the right button a couple of times and watch what happens:



The cyclops looks like it's ice skating!

Introducing friction

This is a bit weird because the cyclops keeps moving even if you're no longer holding down the button. This is because nothing is slowing it down. Recall that your physics simulation requires the application of force to change momentum. Once there's force involved, the momentum causes continued movement until a countering force is applied.

Unless you're setting your game in an ice world, you need to introduce friction into the equation to make things more realistic. This is a value that you'll call **damping**, something less than 1.0 that you'll multiply by the resulting velocity every frame.

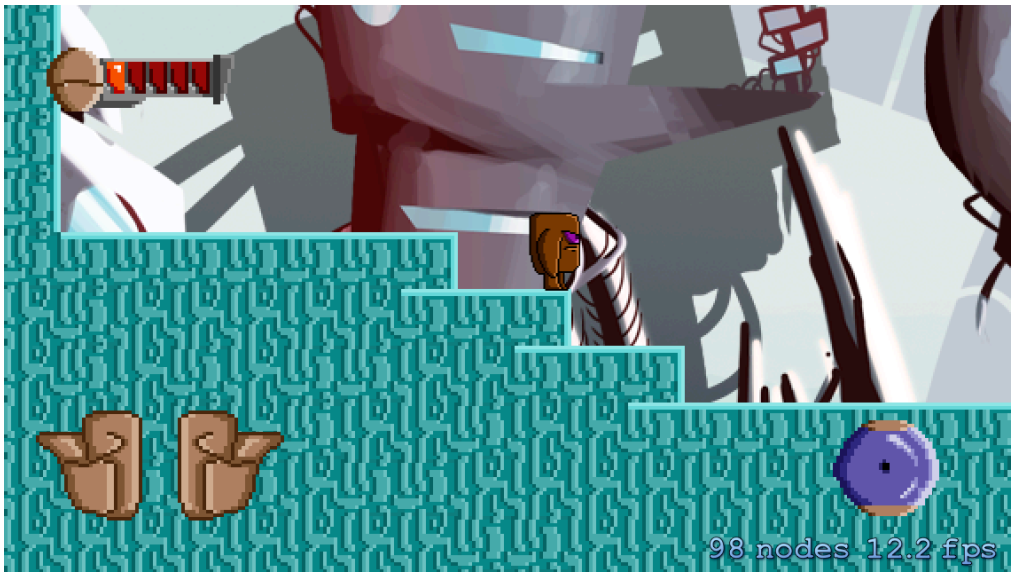
Add the following `#define` to **Player.m**:

```
#define kDamping 0.85
```

Now add the following line to `update:` in **Player.m**, right before the line at the end of the method that calculates the `stepVelocity` and sets `desiredPosition`:

```
self.velocity = CGPointMake(self.velocity.x * kDamping,
                            self.velocity.y);
```

Build and run. Now when you tap the controller, the cyclops moves and then has a short slide to a stop. Hey, this is beginning to feel like a platformer!



Hold down the button and watch what happens. The cyclops still accelerates, but at a much more moderate pace. You can play with the damping and walking speed values to customize game behavior to suit your tastes.

A damping value closer to 1.0 would make it take longer for the cyclops to come to a stop on its own. This might be appropriate for slippery surfaces, but remember that the acceleration will accumulate much more quickly with a higher `kWalkingAcceleration` value.

You also want to set a maximum speed. These three values— `kMaxSpeed`, `kWalkingAcceleration` and `kDamping`—will determine the top speed and control how quickly the cyclops reaches that speed and how quickly it stops on its own, assuming the user doesn't press the opposite button.

Add this new `#define` to **Player.m**:

```
#define kMaxSpeed 250
```

Then add this line in `update:`, just before the line that calculates `stepVelocity`:

```
self.velocity = CGPointMake(
    Clamp(self.velocity.x, -kMaxSpeed, kMaxSpeed),
    Clamp(self.velocity.y, -kMaxSpeed, kMaxSpeed));
```

The `Clamp` function ensures that an input point, `self.velocity` in this case, is always within a specified minimum and maximum value range. Here you set a minimum of -250 and a maximum of 250 in both the x- and y-directions.

So, why is `kWalkingAcceleration` 1600 if `kMaxSpeed` is 250? Remember that `kWalkingAcceleration` is scaled to the time step twice. For each step it adds 1/60th (about 27) to the velocity value and then the velocity value is scaled by 1/60 as well. Think back to your physics classes in school: acceleration is measured in distance per second *squared*, while velocity is distance per second. It's the same principle.

This means the cyclops's speed will reach the maximum in less than a second, which is what you want. This does not take into account the damping, which slows it down a bit. Damping creates a practical upper speed limit—add 27 to 250 and multiply by .85 to end up with 235. So when you build and run, the cyclops won't actually reach the max values.

The best way to tweak the feel of your physics engine is to play with these values—and their equivalents when later you deal with gravity and jumping force.

As you tweak the feel of your game, it's good to include this upper bound to retain as much control as possible. Also, remember that your update method isn't guaranteed to run at 1/60 a second. Sometimes your update method might be called after as much as a second or more, if the phone is busy dealing with something else. If this happens, your delta time value could be very large, which would greatly increase the value added to the previous velocity. Without a maximum value, you'd have a sudden burst of speed.

And, of course, that sort of erratic behavior is what you are trying to avoid with your finely-tuned physics engine. ☺

Following with the camera

This game is shaping up nicely, but it's a bit disappointing how quickly the cyclops disappears off the right side of the screen.

Since the cyclops is the hero of the game, you want the camera to follow it as it moves around the screen. To do this, add the following method to **PSKLevelScene.m**:

```
- (void)setViewpointCenter:(CGPoint)position {
  NSInteger x = MAX(position.x, self.size.width / 2);
  NSInteger y = MAX(position.y, self.size.height / 2);
  x = MIN(x, (self.map.mapSize.width * self.map.tileSize.width)
                                        - self.size.width / 2);
  y = MIN(y, (self.map.mapSize.height *
              self.map.tileSize.height) - self.size.height / 2);
  CGPoint actualPosition = CGPointMake(x, y);
  CGPoint centerOfView = CGPointMake(self.size.width/2,
                                     self.size.height/2);
  CGPoint viewPoint = CGPointSubtract(centerOfView,
                                      actualPosition);
  self.gameNode.position = viewPoint;
}
```

I've adapted this code from an early Ray Wenderlich tutorial, so I'm not going to go through it line by line. You can refer to the original tutorial if you want to learn more about the method:

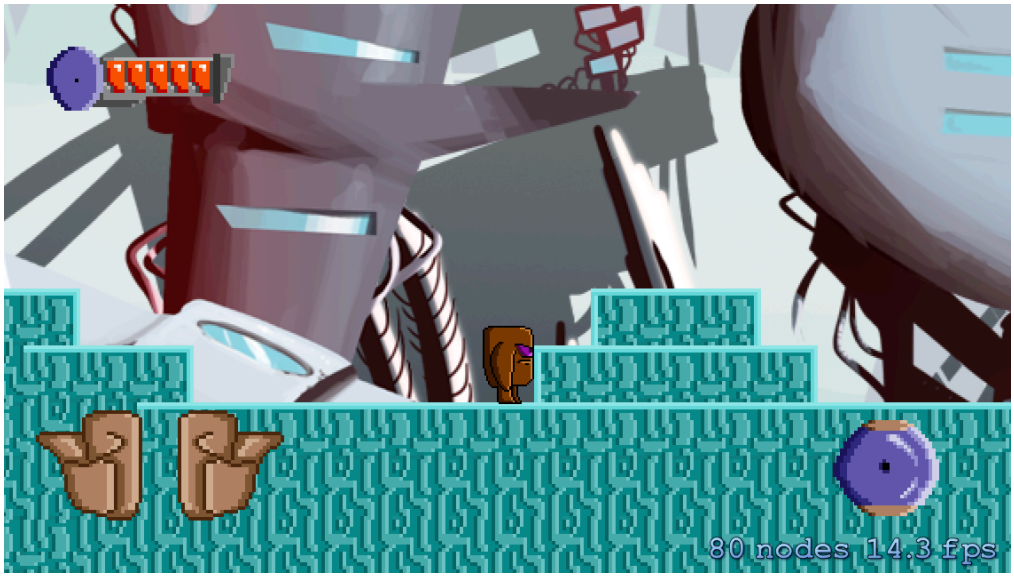
- http://www.raywenderlich.com/1163/how-to-make-a-tile-based-game-with-cocos2d

Essentially, this code moves the `gameNode`, which contains the map and the parallax layer, so that the player sprite stays centered on the screen, except when the player reaches the edges of the game world.

The only thing left to do is add this line to the end of `update:` in **PSKLevelScene.m**:

```
  [self setViewpointCenter:self.player.position];
```

This will be a big payoff for a tiny amount of work, so build and run for the prize!



Now the camera follows the cyclops, which means you get to enjoy the parallax scrolling action you set up in Chapter 1. Excellent work!

The joy of jumping

The cyclops can now move around quite well. But once it hits a valley, it's stuck since it can't get up those peaks. ☹

It's time to make your cyclops jump!

The jump is the distinguishing feature of the platformer and the one that leads to the most fun. You want to make sure your jumping movement is fluid and feels good. In this tutorial, you'll implement the jump algorithm used in *Sonic the Hedgehog*, as described here:

• http://info.sonicretro.org/Sonic_Physics_Guide

First, add the following constant to **Player.m**:

```
#define kJumpForce 400
```

Next, add this code to `update:`, immediately after the line `self.velocity = CGPointAdd(self.velocity, joyForceStep);:`

```
if (self.hud.jumpState == kJumpButtonOn) {
  if (self.onGround) {
    self.velocity = CGPointMake(self.velocity.x, kJumpForce);
  }
}
```

Build and run to try this out:

At this point, you have old school Atari-style jumping. Every jump is the same height. You apply a force to the `Player` object and wait until gravity pulls it back down.

Fine-tuning a jump

In modern platformer games, users expect much finer control over the jump action. You want controllable, completely unrealistic (but fun as hell) jumping à la *Super Mario Bros.* or *Sonic the Hedgehog*, where the user can change direction in mid-air and even stop a jump short.

To accomplish this, you need to add a variable component. There are a couple of ways to do this, but you're going to do it the *Sonic* way: you'll reduce the upward force of the jump once the user stops pressing the jump button.

Replace the code you just added with the following:

```
if (self.hud.jumpState == kJumpButtonOn) {
  if (self.onGround) {
    self.velocity = CGPointMake(self.velocity.x, kJumpForce);
  }
} else {
  if (self.velocity.y > kJumpCutoff) {
    self.velocity = CGPointMake(self.velocity.x, kJumpCutoff);
  }
}
```

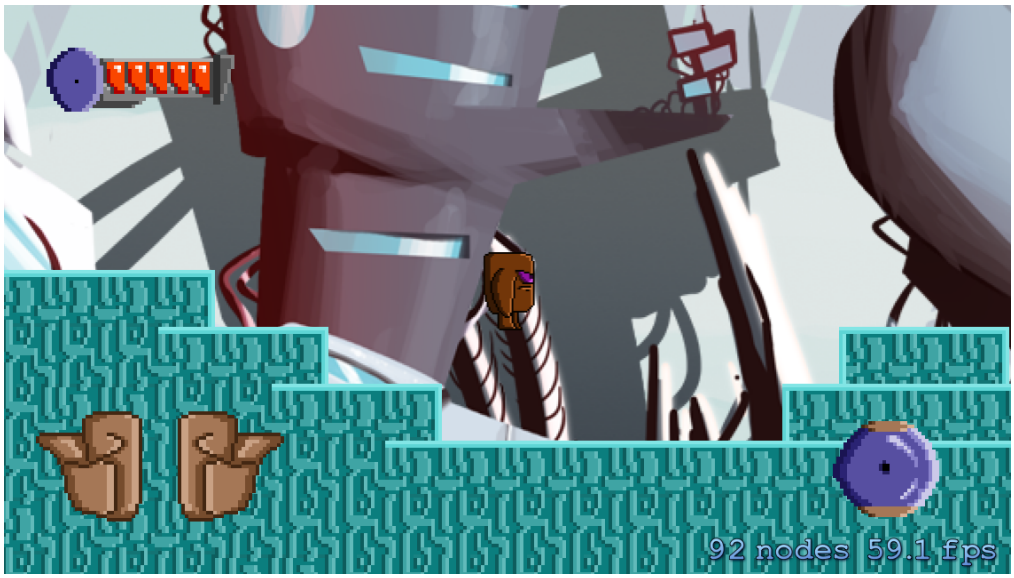
Since you're relying on a new constant, `kJumpCutoff`, you need to add it to **Player.m**. Do so as follows:

```
#define kJumpCutoff 150
```

This code performs an extra step. In the event the user stops pressing the jump button, it checks the upward velocity of the player. If that value is greater than the cutoff, it sets the velocity to the cutoff value.

This effectively reduces the force of the jump. This way, you'll always get a minimum jump at least as high as `kJumpCutoff`, but if you continue to hold, you'll get the full available jump force.

Build and run. You now have more control over your jump height.



This is starting to feel like a real game! You've done enough so that all the buttons are connected and you can navigate through all the levels.

Preventing the kangaroo jump

Here's something you may not have considered: The only precondition for the jump action, besides that the jump button is pressed, is that `onGround` is set to `YES`. This means if you hold down the jump button, the cyclops jumps as soon as it hits the ground, continuously. This is known as the kangaroo jump. ☺

If you play the game as you would normally, you may not even notice this behavior, and some very successful games do include the kangaroo jump. But, being the perfectionist you are, you're going to fix it. This is no kangaroo—it's a cyclops! ☺

The typical user's expectation is that to complete a new jump, s/he needs to release the button before pressing it again. To implement this, you need a new Boolean property that records the state of the jump button. Call it `jumpReset`. This property will tell you whether the jump button was in a released state before a jump, letting you prevent a new jump until the user releases the button.

Add a class extension to **Player.m**, immediately above the `@implementation` line:

```
@interface Player ()
```

```
@property (nonatomic, assign) BOOL jumpReset;
@end
```

This simply adds the `jumpReset` Boolean to the class.

Now set the `jumpReset` flag in `initWithImageNamed:` after the line that sets `velocity`:

```
self.jumpReset = YES;
```

Finally, alter the code in `update:` that handles the jump action. Make it look like the following:

```
if (self.hud.jumpState == kJumpButtonOn) {
  if (self.onGround && self.jumpReset) {
    self.velocity = CGPointMake(self.velocity.x, kJumpForce);
    self.jumpReset = NO;
  }
} else {
  if (self.velocity.y > kJumpCutoff) {
    self.velocity = CGPointMake(self.velocity.x, kJumpCutoff);
  }
  self.jumpReset = YES;
}
```

You've made three changes to the original code:

1. You now check that both `onGround` and `jumpReset` are `YES`.
2. After you apply the jump force to `velocity`, you set `jumpReset` to `NO`. This prevents you from applying the jump force again until you've set `jumpReset` back to `YES`. You only set `jumpReset` to `NO` when you apply the jump force. If the user continuously presses the jump button, the game sets `jumpReset` to `NO` only in the very first frame, when the cyclops first jumps off the ground.
3. In the `else` block, which is triggered if the jump button isn't in a pressed state, you set `jumpReset` back to `YES`.

Build and run again. The kangaroo jump behavior is gone, giving users better control of the cyclops.

Your cyclops now runs and jumps! And you can explore all the levels. You are well on your way to creating a complete platformer game. Give yourself a break—you deserve it. ☺

Challenge: The *Sonic*-based jump that you implement here uses linear forces to push the cyclops up at a decelerating rate and bring it back down at an accelerating rate.

But some games allow the player to "float" at the top of a jump for just a moment. Games with that sort of jump behavior feel more forgiving and make it easier for the player to get up and around platforms.

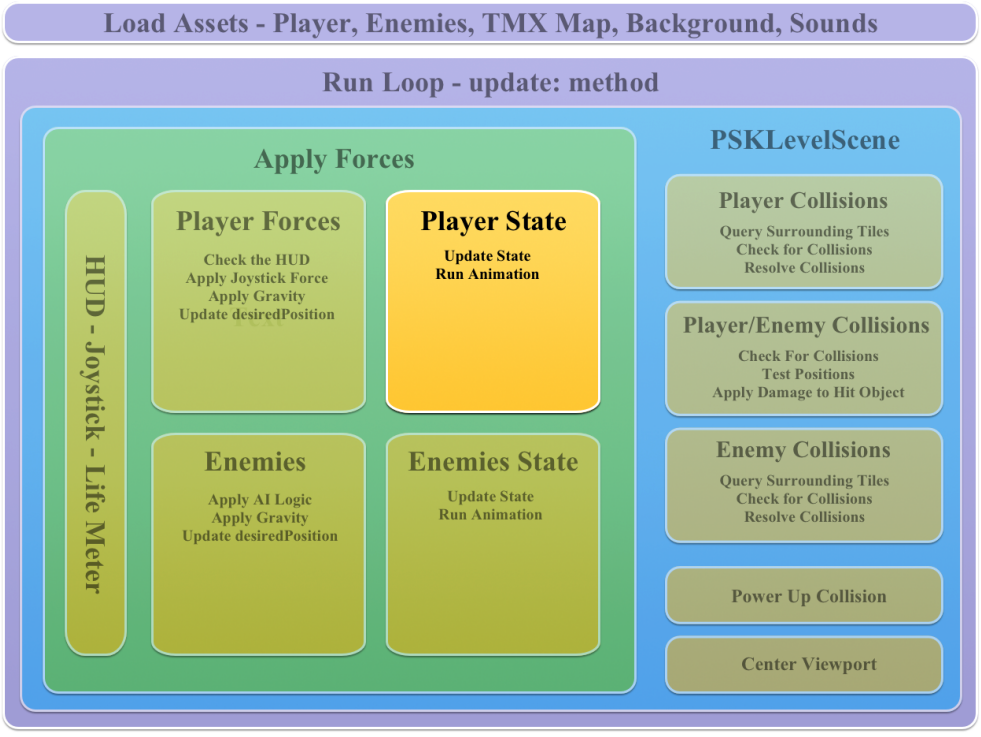
How would you go about implementing that behavior in this game?

Chapter 6: The State Machine

This chapter focuses on the all-important state machine. Many games, including this one, use a state machine to control the appearance and behavior of the game's characters.

Your state machine will control which character animation the game displays at any given time—like jumping or walking—and help the game logic determine which character actions are valid, and when. For example, the cyclops can't jump from a falling state and can't wall-slide when touching the ground. You will soon learn more about that wall slide. ☺

Here's where this chapter falls in your overall game plan:



What is a state machine?

A state machine is simply a mechanism to track of the state of an object, perform actions based on the current state and switch between states when appropriate. Specifically, for this game's state machine:

- The character will be in one (and only one) state at a time.
- In each frame, you'll run a series of tests to determine if the character's state should change.

The state machine serves two purposes. First, you need to check for a number of conditions to transition into certain states. The current state will help determine which tests you should run to change states.

For example, one of the upgrades the cyclops will obtain during the course of the game is the ability to wall slide, or slowly slide down along a vertical wall instead of dropping as rapidly as gravity would usually require.

To transition into the wall-slide state, there are three conditions:

1. The cyclops is falling;
2. The user is pressing the direction button towards the wall;
3. The cyclops is colliding with the wall.

Thus, the cyclops needs to be in a falling state, or the descent part of a jumping or double-jumping state, and it needs to be pushing into a wall. What's a double-jump? Once again, you'll learn more soon. ☺

The state machine's second purpose is to determine the current animation for a character. Your game will have different animations for jumping, running, wall sliding and other states. When something triggers a state change, the game will load and run a new animation or a single frame, and stick with it until the next transition.

This is a simple structure, and some games don't use a state machine because their animations are too complex for a state machine to accommodate easily. But the state machine is a straightforward way to organize and track character state, and it'll serve *Pocket Cyclops* just fine.

In this chapter, you'll first set up all of the individual states for the player character. Then you'll learn how to load and run animations.

Calling all states!

Because both the cyclops and its enemies will rely on a state machine, you'll add the code that handles state change and definition to the `PSKCharacter` superclass. The logic that processes the individual states as well as the animations will be different for each character, so you'll put that code into the appropriate character subclasses.

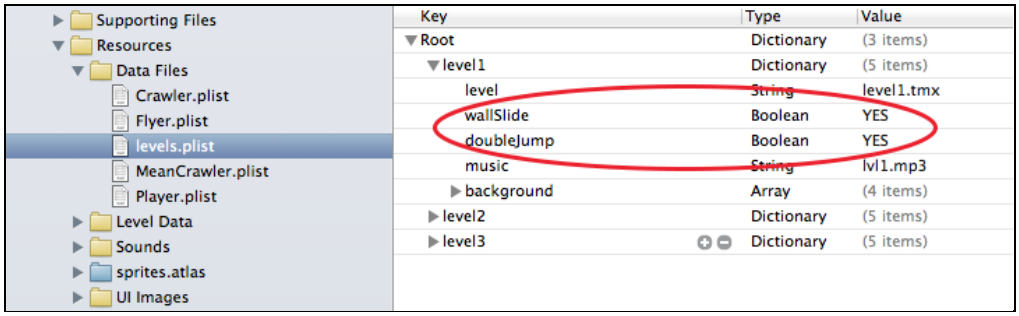
You need at least eight states for the `Player` class, and the enemies add three more. Add them as an `enum` by inserting the following code after the `#import` statement in **PSKCharacter.h**:

```
typedef NS_ENUM(NSInteger, CharacterState) {
  kStateJumping,
  kStateDoubleJumping,
  kStateWalking,
  kStateStanding,
  kStateDying,
  kStateFalling,
  kStateDead,
  kStateWallSliding,
  kStateAttacking,
  kStateSeeking,
  kStateHiding
};
```

The last three states—attacking, hiding and seeking—pertain to enemies. At any given time, the cyclops will be in one and only one of the remaining eight states. You can use the present state to determine if you can and should transition to another state. For example, the cyclops needs to be in the jumping state before it can enter the double-jumping state.

To make it easy to test and write the state machine, you are going to turn on both of the upgrades for the first level. There is a **levels.plist** file that sets the initial state of the player in the levels. Inside this file, there are two Booleans for the upgrades.

Use the Project Navigator to find **levels.plist** inside the **Resources/Data Files** folder and select it. Expand the root and then the **level1** object. You should see entries for `wallSlide` and `doubleJump`. Change them both from `NO` to `YES`. It should look like this:



You may be enabling these upgrades now for testing, but you're not giving the cyclops a free pass! In a later chapter, you'll make the cyclops earn these abilities by obtaining the relevant power-up objects.

Add the following code to **PSKCharacter.h**:

```
@property (nonatomic, assign) CharacterState characterState;
```

```
- (void)changeState:(CharacterState)newState;
```

The first line declares a property to contain the current `CharacterState`. The second line declares the method that handles changing the state.

Now add a method, which will be overridden in subclasses, to **PSKCharacter.m**:

```
- (void)changeState:(CharacterState)newState {
  //override this method
}
```

This quiets Xcode's warning about not having a method implementation.

Next, add the first override for `changeState:` to **Player.m**:

```
- (void)changeState:(CharacterState)newState {
  if (newState == self.characterState) return;
  NSLog(@"Change State %ld", (long)newState);
  self.characterState = newState;
}
```

The method first checks to see if the `newState` is equal to the current state of the player character, since there's nothing to do if the state is not changing. Then it logs the new state and sets the player character's state to `newState`.

That's all for now. Later, you'll change the character's animation or the displayed sprite, depending on the state.

Testing for state

The meat of the logic will occur in the `update:` method. That's where you'll do a series of tests to determine the new state that you'll pass to `changeState:`. To start, you'll implement tests that set the walking, standing and jumping states.

I'm going to give you three code chunks to add to your `update:` method, and then I'll explain what they are. First, still in **Player.m**, add the following line to the beginning of `update:`:

```
CharacterState newState = self.characterState;
```

Now, in the block that sets the jump force, find the line that sets `jumpReset` to `NO` and add this first code chunk immediately after it:

```
newState = kStateJumping;
self.onGround = NO;
```

Then add these two code chunks before the line that declares the `gravity` `CGPoint`:

```
if (self.onGround && self.hud.joyDirection ==
                                        kJoyDirectionNone) {
  newState = kStateStanding;
```

```
  } else if (self.onGround && self.hud.joyDirection !=
                                          kJoyDirectionNone) {
    newState = kStateWalking;
  }
  [self changeState:newState];
```

There will be occasions when you will change the `newState` variable multiple times in the course of a single update loop. To avoid calling `changeState:` multiple times, you've created this variable so you can run through all of your tests and change the variable as you go. Once you're done with the tests, you call `changeState:` to set the final state value.

These three state tests are all mutually exclusive and the value of `newState` won't change more than once, but it's not a bad idea to set it up this way regardless.

First is the jumping test. It's less a test, and more a matter of simply setting the state when you first apply the jump force. You begin the jumping state by applying a jump force and continue until the cyclops hits the ground again. Later tests won't override the jumping state as long as `onGround` is `NO`.

Why do you set `self.onGround` to `NO` after you've applied a jump force? Won't the collision detection method set this variable to `NO`? What do you think?

Remember your physics engine. The collision detection system applies the forces, resolves them and sets `onGround` to `YES` if it has to resolve a collision with a tile underneath the character.

After you apply the jump force to the cyclops, you set its `velocity` property, ensuring that the cyclops will move upward. But this won't actually happen until the next loop of the `GameLevelLayer`'s `update:` method. So, immediately after you set the jump within the `Player`'s `update:` method, `onGround` is still `YES`.

This means when you get to the last tests, which look for `onGround` and set the walking or standing states, `onGround` is still `YES` even if the cyclops is in the middle of a jump. That's why you set `onGround` to `NO`—to ensure that the standing and walking states aren't triggered when the cyclops is actually in the air.

The later tests check `onGround` and if the user is directing the character left or right, and choose either the walking or standing states. The way it's set up, the cyclops will transition from walking to standing when the user stops pressing the directional buttons.

However, because of the momentum the physics engine creates, this may or may not be the actual moment the cyclops stops moving. Depending on your physics values, the game might replace the walking animation with the standing animation while the cyclops is still sliding left or right.

This is what you want in most cases. The alternative is that, because of momentum, the walking animation continues to play after the user stops pressing a button. Not only would this look strange—it would detract from the user's sense of connection with the character.

Build and run your game. Walk or jump around and observe the console reporting the changing state value. These values are `enums`, so they are represented as integer values that indicate their location in the `enum` list. For example, the first state, `kStateJumping`, is zero.

```
2013-11-16 11:06:07.413 SKPocketCyclops[75335:a0b] Layer walls has zPosition -80.000000
2013-11-16 11:06:07.438 SKPocketCyclops[75335:a0b] Object Group objects has zPosition -60.000000
2013-11-16 11:06:07.438 SKPocketCyclops[75335:a0b] Object Group enemies has zPosition -40.000000
2013-11-16 11:06:07.439 SKPocketCyclops[75335:a0b] Object Group powerups has zPosition -20.000000
2013-11-16 11:06:07.969 SKPocketCyclops[75335:a0b] Change State 3
2013-11-16 11:06:08.722 SKPocketCyclops[75335:a0b] Change State 2
2013-11-16 11:06:09.188 SKPocketCyclops[75335:a0b] Change State 3
2013-11-16 11:06:10.121 SKPocketCyclops[75335:a0b] Change State 0
2013-11-16 11:06:10.888 SKPocketCyclops[75335:a0b] Change State 3
2013-11-16 11:06:11.654 SKPocketCyclops[75335:a0b] Change State 2
2013-11-16 11:06:12.355 SKPocketCyclops[75335:a0b] Change State 3
2013-11-16 11:06:12.605 SKPocketCyclops[75335:a0b] Change State 2
2013-11-16 11:06:13.205 SKPocketCyclops[75335:a0b] Change State 3
2013-11-16 11:06:13.755 SKPocketCyclops[75335:a0b] Change State 0
2013-11-16 11:06:14.455 SKPocketCyclops[75335:a0b] Change State 3
```

The wall slide

At long last, it's time to add the states for wall sliding and the double jump! These special abilities require state machine tests that are significantly trickier. First you'll tackle the test for wall sliding.

Here are the conditions under which wall sliding can occur:

1. **The cyclops is falling.** You don't want to trigger wall sliding when the player is jumping up, only when they're falling. Nor do you want wall sliding to occur when the player is on the ground.
2. **There is a collision between the player and the wall.** For your purposes, this means that the user is pressing the cyclops into a wall. Falling downward next to a wall shouldn't be enough.

Let's give this a shot!

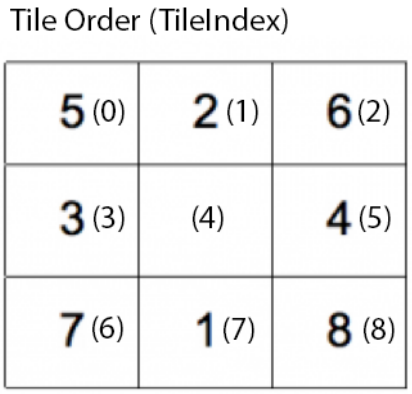
Implementing the wall slide

First you need a new property in **PSKCharacter.h**, because you will use this property in some enemy classes as well. Add it as follows:

```
@property (nonatomic, assign) BOOL onWall;
```

This property is set to `YES` if there is a collision with a wall. It operates much like the `onGround` Boolean, which means you need to set it up in `checkForAndResolveCollisions:forLayer:` in **PSKLevelScene.m**.

Do you remember this diagram from the chapter on collision detection?



The diagram shows the order of the tiles in the collision detection array, along with their index in the original array in parentheses. First you check the tile below the character for collisions, then the tile above, then the tile to the right and so on.

Now look at tile 3 (left-middle), tile 4 (right-middle), tile 7 (left-bottom) and tile 8 (right-bottom). Those are the places where the character might collide with a wall. You are going to set `onWall` to `YES` when you resolve a collision in one of these tiles.

You won't count collisions resolved with tiles 5 and 6, the ones at the top. Why not? You don't want the wall slide to occur if the cyclops is only hanging on by the corner of the diagonal tile above it. The cyclops needs more of a surface for a successful slide. This choice is my design preference. You can include those tiles in a wall slide if you like. ☺

Your task is to find the places in `checkForAndResolveCollisions:forLayer:` that resolve collisions, and if you find a collision with one of the tiles listed, add a line that sets `onWall` to `YES`. First, however, you need to set `onWall` to `NO` at the beginning of the method, so that it doesn't remember its state from a previous frame.

Add this line at the beginning of `checkForAndResolveCollisions:forLayer:` in **PSKLevelScene.m**:

```
character.onWall = NO;
```

Now you'll find all the places in the code where any of the tiles with a `tileIndex` value of 3, 5, 6 or 8 are resolved—these are the numbers inside the parentheses in the above diagram—and add the line that sets `onWall` to `YES` and `velocity.x` to `0.0`.

To do this, add the following code to the `if` blocks where the `tileIndex` is 3 or 5. Add it after the line that sets `character.desiredPosition`:

```
character.onWall = YES;
character.velocity = CGPointMake(0.0, character.velocity.y);
```

In the case of a `tileIndex` of 6 or 8, there are two scenarios: a horizontal resolution and a vertical one. You only want to set `onWall` if the collision resolution is horizontal, so you need to find the place in the code where this happens.

Right after the last instance of this line:

```
character.desiredPosition = CGPointMake(
        character.desiredPosition.x + resolutionWidth,
        character.desiredPosition.y);
```

Add this:

```
if (tileIndex == 6 || tileIndex == 8) {
  character.onWall = YES;
}
character.velocity = CGPointMake(0.0, character.velocity.y);
```

In case the above directions were hard to follow, here's the whole method with the modifications highlighted:

```
- (void)checkForAndResolveCollisions:(PSKCharacter *)character
                            forLayer:(TMXLayer *)layer {
  character.onGround = NO;
  character.onWall = NO;

  NSInteger indices[8] = {7, 1, 3, 5, 0, 2, 6, 8};

  for (NSUInteger i = 0; i < 8; i++) {
    NSInteger tileIndex = indices[i];

    CGRect characterRect = [character collisionBoundingBox];
    CGPoint characterCoord = [self.walls
                              coordForPoint:character.position];

    NSInteger tileColumn = tileIndex % 3;
    NSInteger tileRow = tileIndex / 3;
    CGPoint tileCoord = CGPointMake(characterCoord.x +
            (tileColumn - 1), characterCoord.y + (tileRow - 1));

    NSInteger gid = [self.walls tileGIDAtTileCoord:tileCoord];
    if (gid != 0) {
      CGRect tileRect = [self.map
                              tileRectFromTileCoords:tileCoord];
      if (CGRectIntersectsRect(characterRect, tileRect)) {
        CGRect intersection = CGRectIntersection(
                                        characterRect, tileRect);
        if (tileIndex == 7) {
          //tile is directly below the character
          character.desiredPosition = CGPointMake(
              character.desiredPosition.x,
              character.desiredPosition.y +
                                        intersection.size.height);
```

```
            character.velocity = CGPointMake(
                                        character.velocity.x, 0.0);
            character.onGround = YES;
          } else if (tileIndex == 1) {
            //tile is directly above the character
            character.desiredPosition = CGPointMake(
                character.desiredPosition.x,
                character.desiredPosition.y -
                                         intersection.size.height);
            character.velocity = CGPointMake(
                                        character.velocity.x, 0.0);
          } else if (tileIndex == 3) {
            //tile is left of the character
            character.desiredPosition = CGPointMake(
                character.desiredPosition.x +
                                           intersection.size.width,
                character.desiredPosition.y);
            character.onWall = YES;
            character.velocity = CGPointMake(
                                        0.0, character.velocity.y);
          } else if (tileIndex == 5) {
            //tile is right of the character
            character.desiredPosition = CGPointMake(
                character.desiredPosition.x -
                                           intersection.size.width,
                character.desiredPosition.y);
            character.onWall = YES;
            character.velocity = CGPointMake(
                                        0.0, character.velocity.y);
          } else {
            if (intersection.size.width >
                                        intersection.size.height) {
              //tile is diagonal, but resolving collision
              //vertically
              CGFloat resolutionHeight;
              if (tileIndex > 4) {
                resolutionHeight = intersection.size.height;
                if (character.velocity.y < 0) {
                  character.velocity =
                          CGPointMake(character.velocity.x, 0.0);
                  character.onGround = YES;
                }
              } else {
                resolutionHeight = -intersection.size.height;
                if (character.velocity.y > 0) {
                  character.velocity =
                          CGPointMake(character.velocity.x, 0.0);
                }
              }
              character.desiredPosition = CGPointMake(
                  character.desiredPosition.x,
                  character.desiredPosition.y + resolutionHeight);
```

```
            } else {
              //tile is diagonal, but resolving horizontally
              CGFloat resolutionWidth;
              if (tileIndex == 6 || tileIndex == 0) {
                resolutionWidth = intersection.size.width;
              } else {
                resolutionWidth = -intersection.size.width;
              }
              character.desiredPosition = CGPointMake(
                  character.desiredPosition.x + resolutionWidth,
                  character.desiredPosition.y);

              if (tileIndex == 6 || tileIndex == 8) {
                character.onWall = YES;
              }
              character.velocity = CGPointMake(
                                          0.0, character.velocity.y);
            }
          }
        }
      }
      character.position = character.desiredPosition;
    }
  }
```

Notice that you set the `Player` object's x-velocity to zero. The character has a small amount of momentum as the x-velocity decays. The character won't move perceptibly, but it'll still be colliding with the block next to it. This can last almost a full second, depending on the physics values you chose for acceleration and friction.

This means that for a short time after the user stops pressing toward the wall, the cyclops will still be colliding with the wall and triggering a wall slide. This is unacceptable, as the user will expect wall sliding to stop immediately after s/he stops pressing towards the wall. Zeroing the x-velocity fixes this and is more accurate, as the cyclops has stopped moving.

Now you can return to the `Player` class's `update:` method and take advantage of this new information that tells you whether a wall collision is occurring.

First dip your toe in the water by testing for the value. Add this log statement to the end of `update:` in **Player.m**:

```
if (self.onWall) {
  NSLog(@"On a wall");
}
```

Build and run, and then run the cyclops into a wall. The Xcode console should show that the cyclops is colliding with a wall, but only for as long as you press it into one.

```
2013-11-16 11:26:54.624 SKPocketCyclops[75491:a0b] Change State 3
2013-11-16 11:26:56.027 SKPocketCyclops[75491:a0b] Change State 2
2013-11-16 11:26:57.242 SKPocketCyclops[75491:a0b] Change State 3
2013-11-16 11:26:57.842 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.193 SKPocketCyclops[75491:a0b] Change State 2
2013-11-16 11:26:58.193 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.242 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.292 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.342 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.392 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.442 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.492 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.542 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.593 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.642 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.692 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.742 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:58.792 SKPocketCyclops[75491:a0b] Change State 3
2013-11-16 11:26:58.793 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:26:59.493 SKPocketCyclops[75491:a0b] Change State 2
2013-11-16 11:26:59.959 SKPocketCyclops[75491:a0b] Change State 3
2013-11-16 11:27:00.676 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:27:03.726 SKPocketCyclops[75491:a0b] Change State 2
2013-11-16 11:27:03.776 SKPocketCyclops[75491:a0b] On a wall
2013-11-16 11:27:03.825 SKPocketCyclops[75491:a0b] On a wall
```

If, when you press the cyclops into a wall, you have state change and "On a wall" notifications in your console, you have done everything right. Make sure that the wall notifications stop immediately when you release the button. You are ready to move on to the state change test.

Return to `update:` in **Player.m** and modify the code that tests for the walking and standing states to the following (add the highlighted lines):

```
if (self.onGround && self.hud.joyDirection ==
                                          kJoyDirectionNone) {
  newState = kStateStanding;
} else if (self.onGround && self.hud.joyDirection !=
                                          kJoyDirectionNone) {
  newState = kStateWalking;
} else if (self.onWall && self.velocity.y < 0) {
  newState = kStateWallSliding;
}
```

The test that you just added transitions the cyclops to a wall-sliding state if `onWall` is `YES` and if the cyclops's velocity is negative, which means it is falling.

You don't need to test for `self.onGround`. If `onGround` is `YES`, the execution will never reach that third test—it will trigger a true condition and execute one of the other `if` cases. The fact that the third `if else` is running at all means that `self.onGround` is not `YES`.

The benefits of wall sliding

So what are the benefits of wall sliding? There are two:

1. The character can jump off a wall when in the wall-sliding state. This means you need to alter the jump test to take that into account—now the cyclops can jump when either `onGround` or `onWall` is true.
2. The character will have a reduced falling speed in a wall slide state. That's going to take a little more doing, so you'll get to it in a minute.

Still in **Player.m**'s `update:`, find the following line,

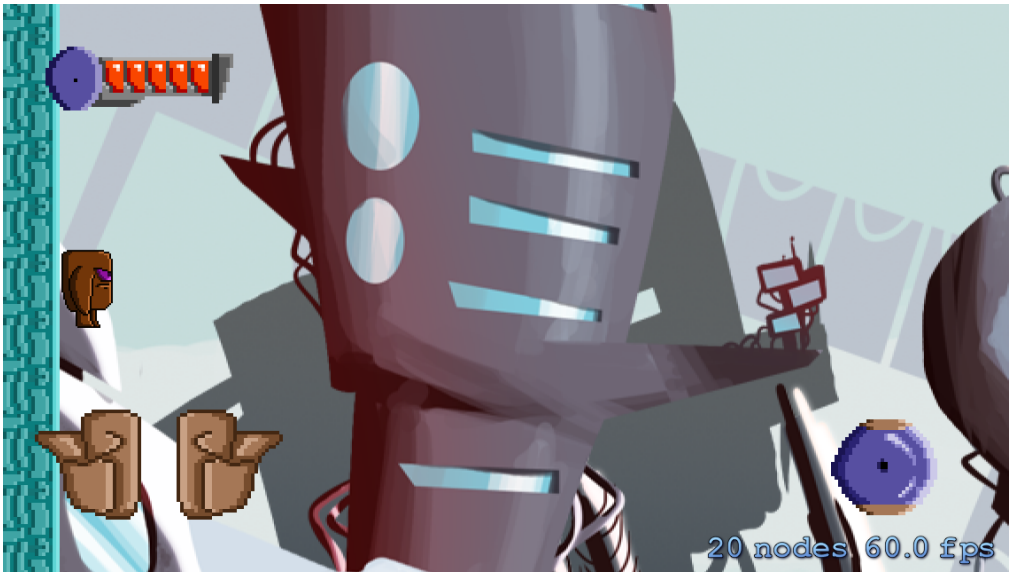
```
if (self.onGround && self.jumpReset) {
```

And alter it thusly:

```
if ((self.onGround || self.characterState == kStateWallSliding)
    && self.jumpReset) {
```

This updates the jump code so the cyclops can jump off the wall from a wall-sliding state as well as from the ground. Because you're checking the state and not the `onWall` property, you don't also have to check whether the cyclops is falling.

Build and run. You should now be able to make the cyclops jump off a wall by pressing against the wall and jumping.



Does it seem like something's missing? In most games, the jump from a wall slide also propels the character out and away from the wall. Add that behavior now!

Still in the jump block you modified above, add this code immediately after `self.jumpReset = NO;`:

```
if (self.characterState == kStateWallSliding) {
   NSInteger direction = -1;
   if (self.flipX) {
      direction = 1;
   }
   self.velocity = CGPointMake(direction * kJumpOut,
                               self.velocity.y);
}
```

You check to see if the character is jumping off of the wall. If it is, you add some force to push the character away from the wall.

The direction of the force depends on the the value of `flipX`, which determines whether the character is against a right or left wall. Since the cyclops must be

pressing against a wall in order to wall slide, the direction that the cyclops is facing tells you which way to direct the jump—the opposite one!

Now add the `#define` statement for the new constant to **Player.m**:

```
#define kJumpOut 360
```

Build and run again. The game should propel the cyclops away from the wall when you jump from a wall slide, just like any decent wall-sliding jumper.

Wall sliding's other benefit is slowing your descent. To make this happen, you want to check the velocity of the player character after you've completed all the other velocity calculations.

Add a new `#define` to **Player.m**:

```
#define kWallSlideSpeed -30
```

This value is the maximum speed per second that the cyclops will fall during a wall slide.

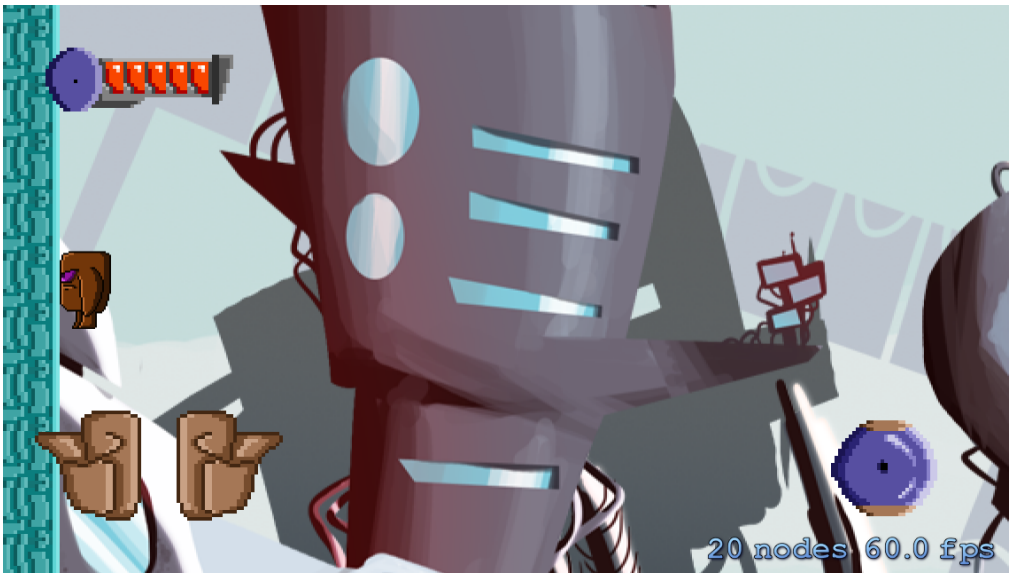
Now add this code to the bottom of `update:` in **Player.m**, right before the line that sets `stepVelocity`:

```
if (self.characterState == kStateWallSliding) {
  CGFloat fallingSpeed = Clamp(self.velocity.y,
                               kWallSlideSpeed, 0);
  self.velocity = CGPointMake(self.velocity.x, fallingSpeed);
}
```

Also, if you still have the `onWall` logging as the last bit of code in `update:`, remove it.

The code above checks if the player character is wall sliding and if so, applies a clamp to the y-component of the velocity variable, setting the lower bound to `kWallSlideSpeed` and the upper bound to zero. If the player character is wall sliding, velocity along y shouldn't be greater than zero—that's part of the test that puts the player into a wall-sliding state—so clamping it to zero should have no effect. The `Clamp()` function simply requires a maximum, and that's why I selected zero!

Build and run now, and you should have a cyclops that can slowly slide down walls:



You still have one little problem. If the cyclops starts wall sliding and then moves in the other direction, letting go of the wall, it still falls slowly. What's going on here?

It's because there's nothing in your code to take the cyclops out of its wall-sliding state. Once a wall slider, always a wall slider, as they say… What, they don't say that?

Transitioning out of special states

You need to beef up the test at the end of `update:` in **Player.m**, so that if the user stops pressing the cyclops into a wall, the cyclops transitions back to the falling state. At the same time, you are going to add tests for whether the player character is in the jumping or double-jumping states. You need to cover all of the states in order to handle coming out of the wall-sliding state and entering a falling state—the state of last resort!

In `update:`, add some `else if`s to the end of the section that sets the `newState` variable, so that it looks like this:

```
if (self.onGround && self.hud.joyDirection ==
                                          kJoyDirectionNone) {
  newState = kStateStanding;
} else if (self.onGround && self.hud.joyDirection !=
                                          kJoyDirectionNone) {
```

```
    newState = kStateWalking;
  } else if (self.onWall && self.velocity.y < 0) {
    newState = kStateWallSliding;
  } else if (self.characterState == kStateDoubleJumping ||
                                    newState == kStateDoubleJumping) {
    newState = kStateDoubleJumping;
  } else if (self.characterState == kStateJumping
                                       || newState == kStateJumping) {
    newState = kStateJumping;
  } else {
    newState = kStateFalling;
  }
  [self changeState:newState];
```

Now, no matter what else has happened, this section will set `newState` to one of the six states for a `Player` object: standing, walking, wall sliding, double jumping, jumping or falling.

You've added handling for `kStateDoubleJumping` and `kStateJumping` in this block. I'll show you how to enter into those states in the very next section.

Let's double-check the logic. What conditions do these new `if` statements manage?

Obviously, the cyclops is not on the ground because you've handled that already, nor is it `onWall` and sliding down. However, the cyclops could be `onWall` and moving upward, in which case jumping is the correct state. At this point, you check if the state is either jumping or double jumping. You need to check the current state, set in the last update loop, as well as the new state, which comes from handling user input from the jump button.

There are only three things that take the cyclops out of the jumping or double jumping states:

- The cyclops lands on the ground;
- The user presses the cyclops against a wall; or
- The user presses the jump button again to move the cyclops from jumping to double jumping.

If the cyclops was in one of those states previously, or if the user has pressed the jump button, then a jumping or double jumping state is appropriate.

> **Note:** In some games, you might want to switch from jumping to falling when the player character's velocity changes—that is, when it stops moving upward. You could easily add that test here. For my purposes, I don't need to determine that change.

If all of the previous tests fail, the only state left is the falling state. This state occurs when either:

- The cyclops walks of a ledge, or

• The cyclops exits the wall-sliding state.

That second option is the bug you're trying to fix!

This stuff gets complex quickly!

Build and run now, and the cyclops should return to normal falling speed when you release from a wall slide. Reverting back to `kStateJumping` now works.

The double jump

Now it's time to enable the other power-up, the double jump, which gives the cyclops the ability to jump a second time from midair.

The double jump requires two conditions:

1. The current state of the player character should be `kStateJumping`.

2. The jump button should have been pressed, released and pressed again.

You're going to add an `if` statement to the current jumping code that checks if the cyclops is already in the jumping state. If it is, the new code will put it into a double-jump state.

This new test will come first, before the current test that puts the `Player` into the jumping state, and the existing code will become an `else if`. Add the following in `update:` in **Player.m** (new lines highlighted):

```
if (self.hud.jumpState == kJumpButtonOn) {
  if ((self.characterState == kStateJumping ||
       self.characterState == kStateFalling) && self.jumpReset) {
    self.velocity = CGPointMake(self.velocity.x, kJumpForce);
    self.jumpReset = NO;
    newState = kStateDoubleJumping;
```

```
    } else if ((self.onGround || self.characterState ==
                            kStateWallSliding) && self.jumpReset) {
      . . .
```

This code is identical to the other jumping code. The only difference is that you are setting the state to double jump. You can see that the player character is able to enter the double-jumping state from either a jumping or a falling sate. This means that if the cyclops walks off a ledge, it can still do a midair jump.

It's a little tricky to determine whether the cyclops is in a jumping or double-jumping state, because unlike all the other states that rely only on the present condition of things, this state takes a previous state into account. So you'll determine it here and test for it again later.

Refer back to the section that tests for state and sets the `newState` variable. It should currently look like this:

```
if (self.onGround && self.hud.joyDirection ==
                                              kJoyDirectionNone) {
  newState = kStateStanding;
} else if (self.onGround && self.hud.joyDirection !=
                                              kJoyDirectionNone) {
  newState = kStateWalking;
} else if (self.onWall && self.velocity.y < 0) {
  newState = kStateWallSliding;
} else if (self.characterState == kStateDoubleJumping ||
                               newState == kStateDoubleJumping) {
  newState = kStateDoubleJumping;
} else if (self.characterState == kStateJumping ||
                                    newState == kStateJumping) {
  newState = kStateJumping;
} else {
  newState = kStateFalling;
}
```

The code needs to check both the `self.characterState` and the `newState` variables because while you set the `newState` variable the first time the `Player` enters the double-jumping state, every time after that it's `self.characterState` that has the double-jump state.

There's no way to infer the difference between a single-jump and a double-jump state—`onGround`, `onWall`, `self.velocity` and so on won't tell you if the cyclops has jumped once or twice. That's why this test requires a little more maintenance. You could achieve the same result by creating a previous state variable that you set when the state changes.

One side effect of this test—and it's a desired effect—is that if the cyclops transitions to a wall-sliding state, and then back to a jumping state, it can then double-jump again. So by touching a wall, the player gets back their second jump.

> **Note:** If you didn't want the game to behave this way, you would need to create a Boolean to keep track of whether or not another double jump is available. You could, for example, give the double jump back to the cyclops only after it touches the ground.

Here's the entire `update:` method in its current form. Check to make sure you've placed all the code snippets in the correct places:

```
- (void)update:(NSTimeInterval)dt {
  CharacterState newState = self.characterState;

  CGPoint joyForce = CGPointZero;
  if (self.hud.joyDirection == kJoyDirectionLeft) {
    self.flipX = YES;
    joyForce = CGPointMake(-kWalkingAcceleration, 0);
  } else if (self.hud.joyDirection == kJoyDirectionRight) {
    self.flipX = NO;
    joyForce = CGPointMake(kWalkingAcceleration, 0);
  }

  CGPoint joyForceStep = CGPointMultiplyScalar(joyForce, dt);
  self.velocity = CGPointAdd(self.velocity, joyForceStep);

  if (self.hud.jumpState == kJumpButtonOn) {
    if ((self.characterState == kStateJumping ||
         self.characterState == kStateFalling)
     && self.jumpReset) {
      self.velocity = CGPointMake(self.velocity.x, kJumpForce);
      self.jumpReset = NO;
      newState = kStateDoubleJumping;
    } else if ((self.onGround || self.characterState ==
                        kStateWallSliding) && self.jumpReset) {
      self.velocity = CGPointMake(self.velocity.x, kJumpForce);
      self.jumpReset = NO;

      if (self.characterState == kStateWallSliding) {
         NSInteger direction = -1;
         if (self.flipX) {
            direction = 1;
         }
         self.velocity = CGPointMake(direction * kJumpOut,
                                     self.velocity.y);
      }

      newState = kStateJumping;
      self.onGround = NO;
    }
  } else {
    if (self.velocity.y > kJumpCutoff) {
      self.velocity = CGPointMake(self.velocity.x, kJumpCutoff);
```

```
    }
    self.jumpReset = YES;
  }

  if (self.onGround && self.hud.joyDirection ==
                                          kJoyDirectionNone) {
    newState = kStateStanding;
  } else if (self.onGround && self.hud.joyDirection !=
                                          kJoyDirectionNone) {
    newState = kStateWalking;
  } else if (self.onWall && self.velocity.y < 0) {
    newState = kStateWallSliding;
  } else if (self.characterState == kStateDoubleJumping ||
                              newState == kStateDoubleJumping) {
    newState = kStateDoubleJumping;
  } else if (self.characterState == kStateJumping ||
                                   newState == kStateJumping) {
    newState = kStateJumping;
  } else {
    newState = kStateFalling;
  }
  [self changeState:newState];

  CGPoint gravity = CGPointMake(0.0, -450.0);
  CGPoint gravityStep = CGPointMultiplyScalar(gravity, dt);
  self.velocity = CGPointAdd(self.velocity, gravityStep);

  self.velocity = CGPointMake(self.velocity.x * kDamping,
                              self.velocity.y);

  self.velocity = CGPointMake(
      Clamp(self.velocity.x, -kMaxSpeed, kMaxSpeed),
      Clamp(self.velocity.y, -kMaxSpeed, kMaxSpeed));

  if (self.characterState == kStateWallSliding) {
    CGFloat fallingSpeed = Clamp(self.velocity.y,
                                 kWallSlideSpeed, 0);
    self.velocity = CGPointMake(self.velocity.x, fallingSpeed);
  }

  CGPoint stepVelocity = CGPointMultiplyScalar(
                                          self.velocity, dt);
  self.desiredPosition = CGPointAdd(self.position,
                                    stepVelocity);
}
```

That's the state machine! You can see that it gets very complicated as you progress—it starts out simple and easy, but can turn hairy in a jiffy. ☺

Build and run. For the moment, your cyclops is fully powered-up. You can make it wall slide and double jump to your heart's content. In a later chapter, you'll take away the free power-ups and make the player earn them.

Congratulations on coming this far! Between the custom physics engine and the state machine, you've tackled the hardest parts of this tutorial. That's some stellar work. ☺

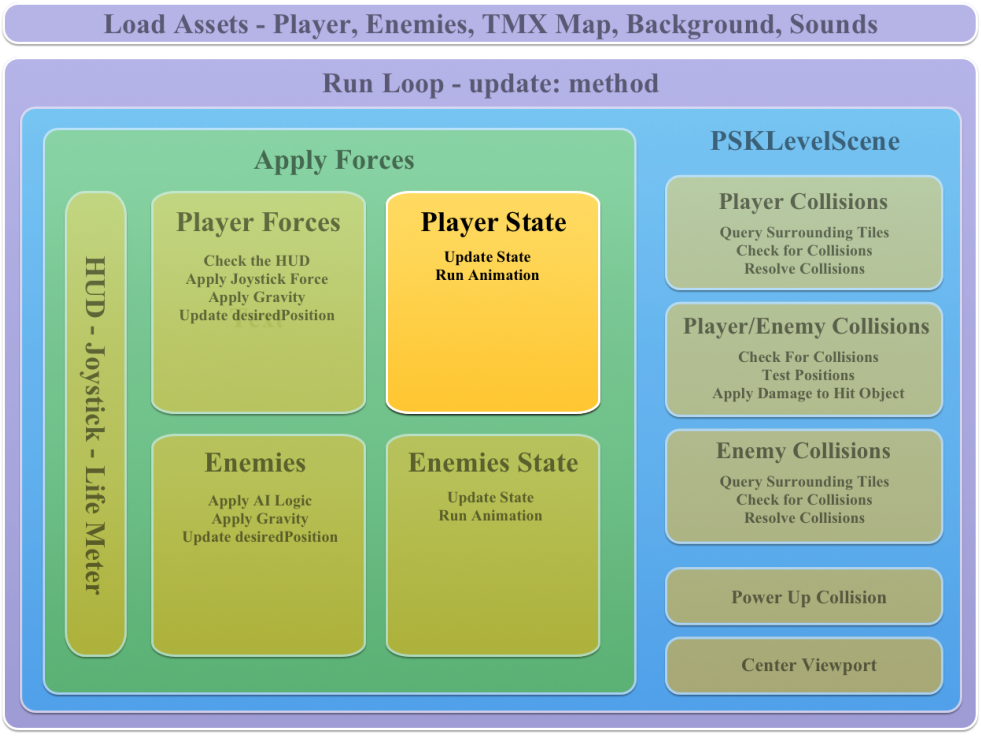
But don't think you're done with the state machine! It will make plenty of appearances in later chapters as you add character animations, enemies and power-ups to the game.

> **Challenge:** In the game's current implementation, you can enter the double-jump state from midair after walking off a ledge or doing a wall slide. What if you wanted to allow a double jump after a wall slide, but not after walking off a ledge? What additional states and checks would you need to add?

Chapter 7: Animation

Animations add life and personality to your game's characters. In this short chapter, you'll take advantage of all the hard work you did on the state machine to deploy the animations for the cyclops and lay the groundwork for enemy and power-up animations. Afterward, your game will look a lot more like something you would enjoy playing.

Here's where this chapter fits in the scheme of things:



Animations and state

Now that the state machine is working, you can use state transitions to switch between animations for your characters. You don't want to start the animation over

every frame, but you do want to switch animations when the state changes. That's why you return from `changeState:` right at the beginning if the `newState` is equal to `self.characterState`—you bail out if the state hasn't changed.

Open **Player.m** and take a look at `changeState:`. Currently, if there's a change in character state, the method logs the new state. It's time to replace that log statement with code that does something awesome—namely, animates the character. This is the chapter where your game begins to look fun.

But first, you need to import the animations into your game scene. The next section will show you a nifty way to do it.

Loading animations from a PLIST

This technique is adapted from the book *Learning Cocos2D* by Rod Strougo and Ray Wenderlich. It involves setting up a property list (PLIST) object, provided as a part of the starter project, that lists all of the sprite frames that make up an animation. The advantage of putting this information into a PLIST is that it makes it easy to tweak.

You will write a method to load these animations for the cyclops and its enemies. After that, you'll only need to run the animations at the appropriate time, depending on the character state as passed to `changeState:`.

You will add these methods to `PSKGameObject` because both the cyclops and the enemies derive from that. Open **PSKGameObject.h** and add the following method declarations:

```
- (SKAction *)loadAnimationFromPlist:(NSString *)animationName
                            forClass:(NSString *)className;
- (void)loadAnimations;
```

You will override `loadAnimations` in subclasses of `PSKGameObject`, so you just need an empty implementation here. Add the following code to **PSKGameObject.m**:

```
- (void)loadAnimations {
  //override this method
}
```

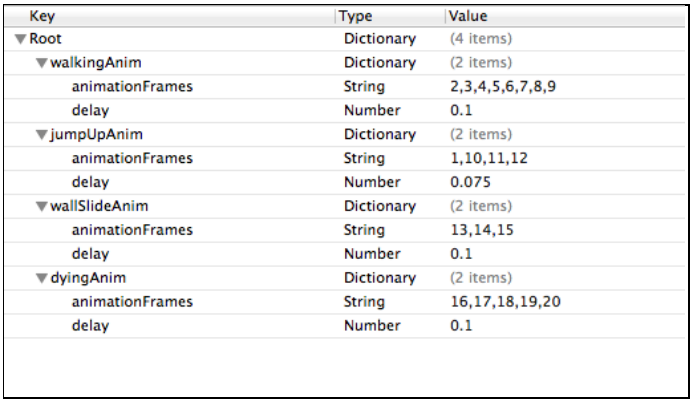
Next create an initialization method that calls `loadAnimations`:

```
- (id)initWithImageNamed:(NSString *)name {
  if ((self = [super initWithImageNamed:name])) {
    [self loadAnimations];
  }
  return self;
}
```

Now any subclass of **PSKGameObject** calls `loadAnimations` on startup.

The other method definition is for a method that loads a set of animations from a PLIST file. The children of this class will be the player, enemies and power-up objects. What they all have in common is that they need sets of frames loaded into an animation object (`SKAction`) from a PLIST file.

Let's take a look at the format of the PLIST file you will use. Open **Resources\Data Files\Player.plist** and you will see the following:



Key	Type	Value
▼Root	Dictionary	(4 items)
▼walkingAnim	Dictionary	(2 items)
animationFrames	String	2,3,4,5,6,7,8,9
delay	Number	0.1
▼jumpUpAnim	Dictionary	(2 items)
animationFrames	String	1,10,11,12
delay	Number	0.075
▼wallSlideAnim	Dictionary	(2 items)
animationFrames	String	13,14,15
delay	Number	0.1
▼dyingAnim	Dictionary	(2 items)
animationFrames	String	16,17,18,19,20
delay	Number	0.1

This file is an `NSDictionary` that contains several other `NSDictionary` objects, one for each animation. The key for each child dictionary object is the name of the animation. Within the individual dictionaries, there are a few pieces of information that describe the animation.

The first is a string that contains a comma-separated list of frame numbers. You will write code to parse this list and use the numbers to load the frames from the sprite sheet. I'll say more about this in a moment.

The second piece of information is the delay, a float that gives the amount of time in seconds between each frame change. For example, if there are five frames in an animation sequence and the delay value is 0.1, then during one second the animation will play completely twice—because 5 frames × 0.1 seconds is 0.5, and that means the animation takes half a second to complete.

Add this method to **PSKGameObject.m**:

```
- (SKAction *)loadAnimationFromPlist:(NSString *)animationName
                            forClass:(NSString *)className {
  //1
  NSString *path = [[NSBundle mainBundle]
                    pathForResource:className ofType:@"plist"];
  NSDictionary *plistDictionary = [NSDictionary
                             dictionaryWithContentsOfFile:path];
  //2
  NSDictionary *animationSettings =
                                 plistDictionary[animationName];
  //3
  float delayPerUnit = [animationSettings[@"delay"] floatValue];
  //4
  NSString *animationFrames =
```

```
                              animationSettings[@"animationFrames"];
  NSArray *animationFrameNumbers = [animationFrames
                                componentsSeparatedByString:@","];
  //5
  NSMutableArray *frames = [NSMutableArray array];
  for (NSString *frameNumber in animationFrameNumbers) {
    //6
    NSString *frameName = [NSString
                 stringWithFormat:@"%@%@",className,frameNumber];
    SKTexture *frame = [[PSKSharedTextureCache sharedCache]
                                           textureNamed:frameName];
    [frames addObject:frame];
  }
  //7
  return [SKAction animateWithTextures:frames
                 timePerFrame:delayPerUnit resize:YES restore:NO];
}
```

Here's a step by step breakdown of the above method:

1. The code in the first section loads the PLIST file into `plistDictionary`. The convention for naming these files is ***className*.plist**. So **Player.plist** is the name of the file for the `Player` class.
2. In the second section, you retrieve the specific animation dictionary from `plistDictionary`. Since the PLIST file contains all the animations for the given class, you call this method once for each animation, passing the name of the class and the name of the animation into the method. Each time you call the method, the result is assigned to a new `SKAction` object.
3. This retrieves the delay value that is the interval between each frame change in the animation sequence.
4. Then you retrieve the list of animation frames, a comma-separated list of frame numbers. You parse it into an `NSArray` of strings, one for each frame number, with a call to `componentsSeparatedByString`. This method takes a string and separates it using the specified character, returning an array of the individual string components. It's handy-dandy and I like it. ☺
5. Next, you create a mutable array to contain the individual `SKTexture` frames of the animation. Then you start a `for` loop that iterates over the array of frame numbers.
6. At this stage, you create an `NSString` that contains the name of the frame based on the current number in the array loop. This is easy because every image in the texture atlas is named after the class it pertains to with a number appended at the end.

   Then, you retrieve the texture from the `PSKSharedTextureCache` object based on its name and add that texture to the frames array.

   For all of this to work, your `PSKSharedTextureCache` must have already loaded the textures. If you ask for a texture that hasn't been loaded, you'll get `nil` back for

the texture object. You can't put a `nil` object into an array, so you'll get an exception.

You need to make sure you create and load the `PSKSharedTextureCache` first. Next I'll have you make a change that will fix a potential problem.

7. Finally, you create the `SKAction` object with the `animateWithTextures:timePerFrame:resize:restore:` class initializer. This method creates an animation `SKAction` using the provided array of `SKTextures` (frames). The `timePerFrame` parameter is the length of time between each frame change. Setting `resize` to `YES` resizes the `SKSpriteNode` to the size of the `SKTexture`. The `restore` parameter restores the frame to the same `SKTexture` it was before the animation started.

All the game classes that have animations to load will use the above method. Again, be sure that all the filenames of the animation images in the texture atlas (in the sprites.atlas folder) follow the proper naming conventions, where each sprite frame's name is based on the class name and the frame number.

You need to import the `PSKSharedTextureCache` into **PSKGameObject.h**. You want it in the header, because the subclasses of `PSKGameObject` will be using it as well. Add it as follows:

```
#import "PSKSharedTextureCache.h"
```

You want the code that preloads the textures into the `PSKSharedTextureCache` to execute before the `Player` object is initialized. Go to `initWithSize:level:` in **PSKLevelScene.m** and make sure the following code is above the call to `self.player = [[Player alloc] initWithImageNamed:@"Player1"]`, and not below:

```
SKTextureAtlas *atlas = [SKTextureAtlas atlasNamed:@"sprites"];
for (NSString *textureName in [atlas textureNames]) {
  SKTexture *texture = [atlas textureNamed:textureName];
  [[PSKSharedTextureCache sharedCache] addTexture:texture
                                             name:textureName];
}
```

Now that you have the animation loader in place, you can load the animations for the player in the `Player` class.

First, you need some properties to keep track of the various animations. Add the following to the `@interface` section at the top of **Player.m**:

```
@property (nonatomic, strong) SKAction *walkingAnim;
@property (nonatomic, strong) SKAction *jumpUpAnim;
@property (nonatomic, strong) SKAction *wallSlideAnim;
@property (nonatomic, strong) SKAction *dyingAnim;
```

The names of these animations tell you what they are. Besides walking, jumping and wall sliding, the provided PLIST files have another animation—one for dying.

Implement the `loadAnimations` method in **Player.m** as follows:

```
- (void)loadAnimations {
  self.wallSlideAnim = [self
                        loadAnimationFromPlist:@"wallSlideAnim"
                                      forClass:@"Player"];
  self.walkingAnim = [self loadAnimationFromPlist:@"walkingAnim"
                                         forClass:@"Player"];
  self.jumpUpAnim = [self loadAnimationFromPlist:@"jumpUpAnim"
                                        forClass:@"Player"];
  self.dyingAnim = [self loadAnimationFromPlist:@"dyingAnim"
                                       forClass:@"Player"];
}
```

You can see that each animation name in the calls to `loadAnimationFromPlist:forClass:` corresponds to the animation name in the PLIST. While you are creating the `dyingAnim` now, you won't be implementing that until later.

Animations deploy!

Now that you have your animations loaded into properties, it's time to run the animations when you switch states. To do this, replace `changeState:` in **Player.m** with the following:

```
- (void)changeState:(CharacterState)newState {
  if (newState == self.characterState) return;
  self.characterState = newState;
  //1
  [self removeAllActions];
  //2
  SKAction *action = nil;
  //3
  switch (newState) {
    case kStateStanding: {
      //4
      [self setTexture:[[PSKSharedTextureCache sharedCache]
                                      textureNamed:@"Player1"]];
      [self setSize:self.texture.size];
      break;
    }
    case kStateFalling: {
      [self setTexture:[[PSKSharedTextureCache sharedCache]
                                     textureNamed:@"Player10"]];
      [self setSize:self.texture.size];
      break;
    }
    case kStateWalking: {
      //5
      action = [SKAction repeatActionForever:self.walkingAnim];
      break;
    }
```

```
      case kStateWallSliding: {
        action = [SKAction
                        repeatActionForever:self.wallSlideAnim];
        break;
      }
      case kStateJumping: {
        action = self.jumpUpAnim;
        break;
      }
      case kStateDoubleJumping: {
        [self setTexture:[[PSKSharedTextureCache sharedCache]
                                        textureNamed:@"Player10"]];
        [self setSize:self.texture.size];
        break;
      }
      //6
      default: {
        [self setTexture:[[PSKSharedTextureCache sharedCache]
                                         textureNamed:@"Player1"]];
        break;
      }
    }
    //7
    if (action) {
      [self runAction:action];
    }
  }
```

Everything before section #1 was already in the method, but you've removed the `NSLog` statement since you no longer need it. Let's go through the rest section by section:

1. You call `removeAllActions` to cancel any currently running animations.

   Some states, like `kStateStanding`, don't get an animation—instead, they get a single frame. If you don't stop the currently running animation, it will continue to run even after you change the frame, and the new frame will only display briefly before the old animation takes over again.

   If there isn't an animation running, calling `removeAllActions` won't do anything. No harm, no foul. ☺

2. You create a new `SKAction` variable and initialize it to `nil`. This local variable can contain an animation after all the state checks or it can still be `nil`, depending on the state.

3. The state checks begin here. You simply check the current state and execute a different code block for each state. Depending on the state, you either start an animation or set the character to a specific sprite frame. If the cyclops is standing, for example, it doesn't have an animation.

4. This sets the standing state. You retrieve an `SKTexture` object from the cache using the name of the original image file and call `setTexture:`. This is similar to what you did when you changed the textures of the sprites on the HUD layer.

   The falling state is also a static frame and so uses the `setTexture:` call as well.

5. This puts the animation from the `walkingAnim` property into the local `action` variable you declared earlier. The animation is simply information about a series of frames and the timing for those frames. The code wraps that animation `SKAction` in another `SKAction` with the initializer `repeatActionForever` so that the animation runs indefinitely—until you stop it via the call to `removeAllActions` in section #1, that is.

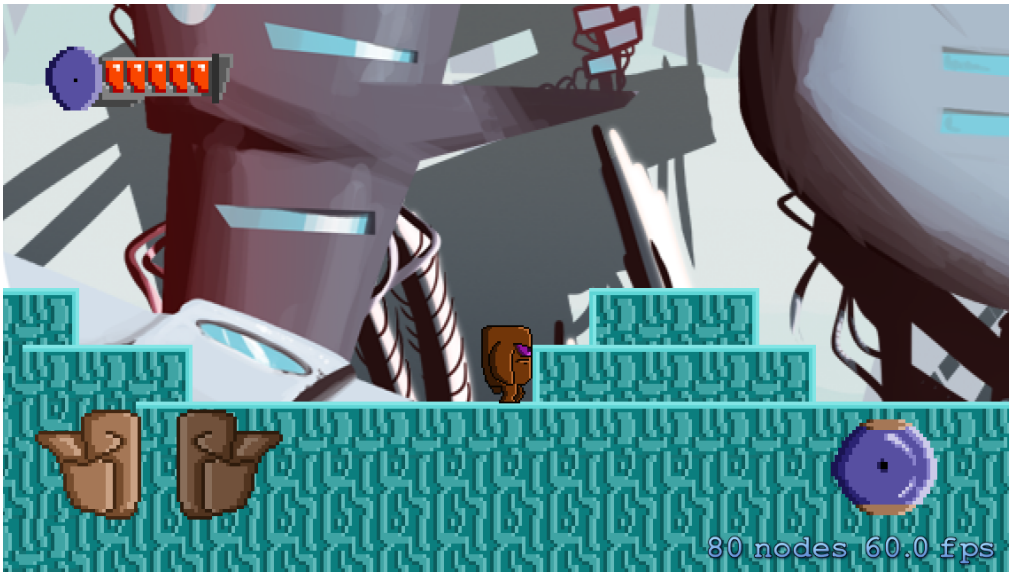
   The walking and wall sliding animations repeat for as long as the user continues that action. The jump animation runs just once at the beginning of the jump behavior, which is why you don't wrap the jump animation in a `repeatActionForever`.

   Every `Player` state does one of these two things: either sets the frame directly or starts an animation.

6. This section sets the default character animation. If the state variable is not equal to any of your enumerated states, the method will choose this option, which is the same as the standing state. This should never execute because you should always have a state that is one of the enumerated states. You could also have skipped `kStateStanding` and simply used this default case for that. You're doing it this way to be thorough.

7. Finally, you run a test to see if the method has set an `action`—otherwise `action` will still be `nil`. You can't run a `nil` action, so you don't want to call `runAction:` until you're sure that there's an action there. It is the call to `runAction:` that starts the animation.

That's it! Build and run the game, and you should have a cyclops that is running, wall sliding and jumping like he means it!



Now that you have a fully working player character, you can use these same techniques to create enemies. They will rely on the same physics engine methods, state machine methods and animation methods the cyclops uses, though they will be a bit simpler.

That's all for this chapter. It was pretty painless, no? Congratulations, you've now completed well over half of the starter kit!

> **Challenge:** In this game, the `characterState` variable drives the animations. There's exactly one animation for every state. But what if you had a game where the player character had different suits/costumes? Think of Mario with his white suit or raccoon outfit. How would you incorporate that into this game structure?

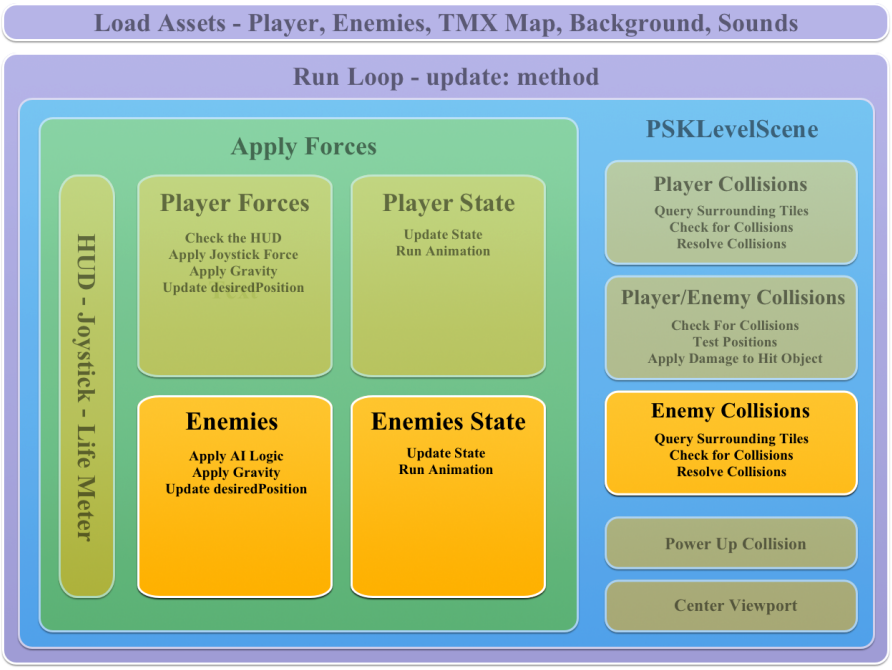
Chapter 8: Enemies

It's time to raise the stakes. The game is now playable, but the cyclops has it way too easy without anyone to oppose it!

Sounds like this is going to be a fun chapter!

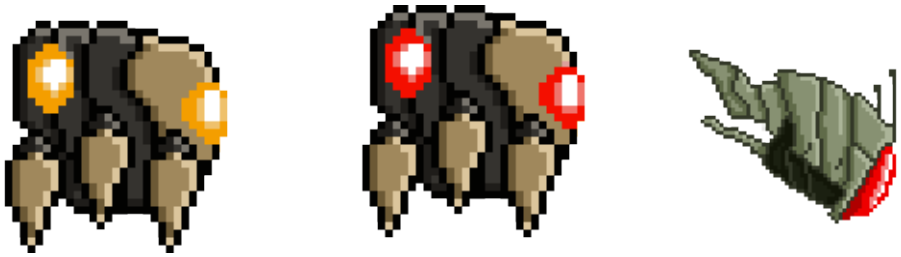
In this chapter, you'll add enemy characters that will move about the game level under their own power. You'll have to think about how to make them smart or dumb, difficult or easy in how they attack the player. Most games, especially platformers, have enemies behave in discernable patterns so that the player can learn to anticipate their moves.

Here's the part of the game you'll cover in this chapter:



Introducing the opposition

In this game you'll create three enemies, as shown below:



The first is the simplest: call it the **simple crawler**. It moves in a specific direction and the only stimulus that changes its action is a bump into a wall, which causes it to switch direction.

The second enemy is much like the first, but more intelligent. Call it the **mean crawler.** This one pursues the player. When necessary, it can jump over things in its pursuit of the cyclops.

The third enemy is the meanest of them all, fearsome for its ability to fly. Call it the **two-faced flyer**. To give the cyclops a chance of beating it, this enemy can only purse the cyclops when the cyclops has its back to it, like the ghost in *Super Mario World*. The two-faced flyer is forced to freeze when the cyclops is looking at it—unless the cyclops gets too close, so watch out!

You will build these enemies to use the same logic as the player character, including the same collision detection method. You will also use a state machine to determine their current animation and expected behavior.

While the enemies use the same collision detection and physics logic as the Player, the enemies will operate based on predetermined rules that you write. These rules can be very simple, like run until you hit a wall then turn around. Or, they can be more complex with different movement patterns based on different enemy states, the player's attributes (position, state, etc), or you could even change the enemies behavior based on some other property (like how much life the enemy has left, if you make the enemy take multiple hits before dying.

Adding enemies to your levels

It's time to get started assembling your enemy army!

Create a new file with the **iOS\Cocoa Touch\Objective-C class** template. Name the class **PSKEnemy** and make it a subclass of **PSKCharacter**.

Repeat the same process as above to create a new class named **Crawler**, except make it a subclass of `PSKEnemy` rather than `PSKCharacter`.

Now that you have the `Crawler` class, empty though it may be, you can add some simple crawlers to the layer.

The first step is to add an array to `PSKLevelScene` to keep track of all the enemies, not just the crawlers.

Switch to **PSKLevelScene.m** and add this to the `@interface` block:

```
@property (nonatomic, strong) NSMutableArray *enemies;
```

Now add the following to the `#import` section:

```
#import "Crawler.h"
```

Next you're going to add the code that creates the enemies. Since it's a bit long and complex, you'll create a separate method that you'll call from `initWithSize:level:`.

First, add the following code to `initWithSize:level:`, right before the `self.hud = [[PSKHUDNode alloc] initWithSize:size]` line:

```
[self loadEnemies];
```

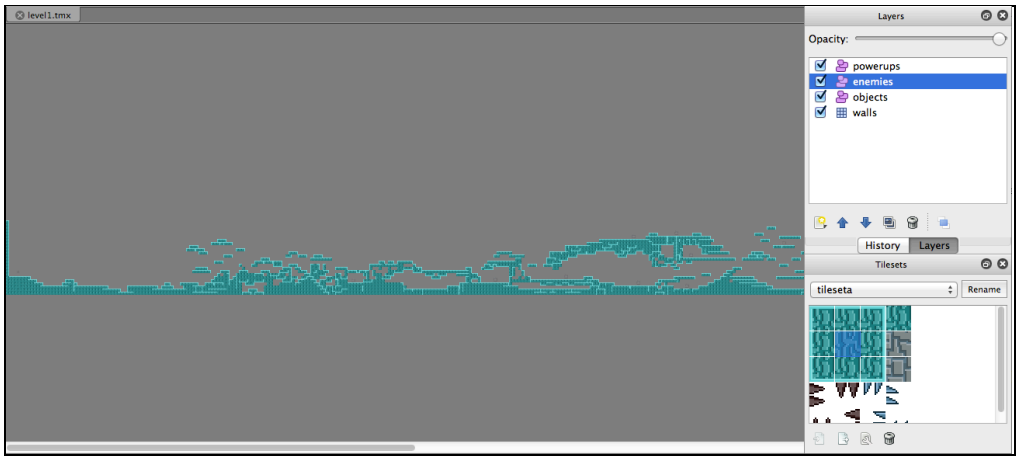
Now add the method:

```
- (void)loadEnemies {
  //1
  self.enemies = [NSMutableArray array];
  //2
  TMXObjectGroup *enemiesGroup = [self.map
                                    groupNamed:@"enemies"];
  for (NSDictionary *enemyDict in enemiesGroup.objects) {
    //3
    NSString *firstFrameName = @"Crawler1.png";
    //4
    PSKEnemy *enemy = [[Crawler alloc]
                       initWithImageNamed:firstFrameName];
    //5
    enemy.position = CGPointMake(
        [enemyDict[@"x"] floatValue],
        [enemyDict[@"y"] floatValue]);
    //6
    [self.map addChild:enemy];
    enemy.zPosition = 900;
    //7
    [self.enemies addObject:enemy];
  }
}
```

It's a big chunk of code, so here's a line-by-line explanation:

1. First, you initialize the `enemies` property with a new `NSMutableArray` object.

2. Now you need to retrieve data about the enemies from the TMX tile map layer. This comes in the form of a `TMXObjectGroup`. If you look again at the provided tile map files in the Tiled editor, you'll see an object layer (or group) called **enemies**, which contains a bunch of objects specifying where the enemies should spawn. This is what you're loading here.



   You then loop through the `objects` array inside the `TMXObjectGroup`. Each object in this array is an `NSDictionary` that describes an enemy as defined in the tile map.

3. You set the name of the first frame image on the `Crawler` object.

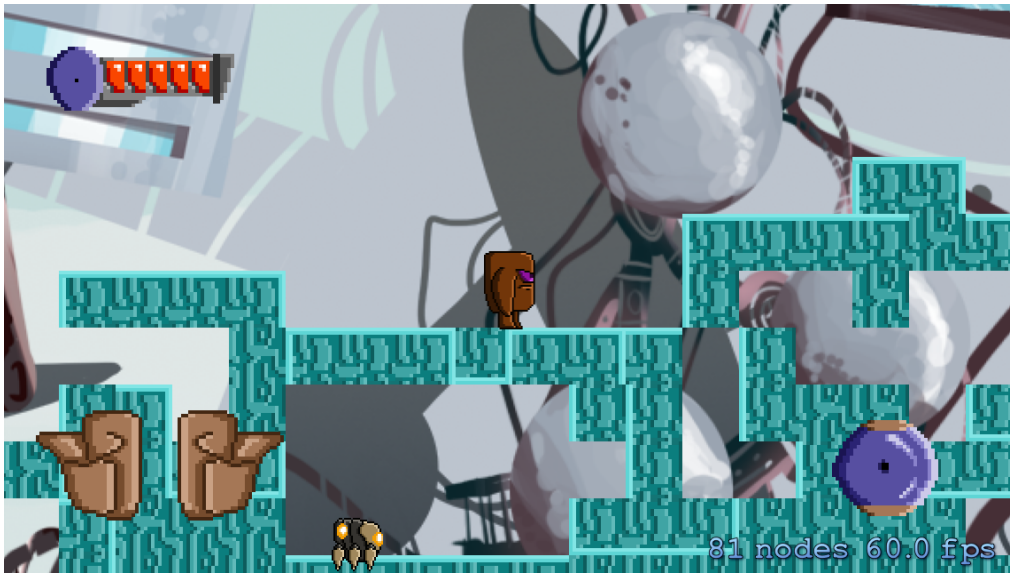
4. Now you use the value from step 3 to initialize the crawler object. Note that `enemy` has the type `PSKEnemy` instead of `Crawler`. This is so you can use this same loop later to initialize other types of enemies. In fact, the whole code block is designed for use with all the different enemy types, even if you don't know what types you'll be getting.

5. You set the position of the enemy using the x and y keys in the `NSDictionary`.

6. You add the enemy to the map, giving the object a `zPosition` of 900 to make sure it shows up in front of the map and parallax layer.

7. Finally, you add the enemy to the `enemies` array.

Build and run, and walk through the level until you find the first enemy—it might take a little while to get there.



The crawler sits motionless at the very bottom of the screen, not unlike the cyclops in the beginning. Now you can apply physics to the enemy, just as you did for the cyclops!

Applying physics to enemies

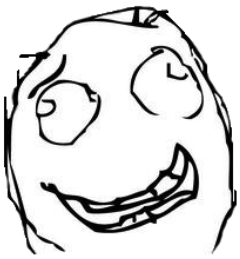
Just as with the cyclops, you'll apply the physics rules for enemies in two places. The enemy's `update:` method will apply gravity and movement and the `PSKLevelScene` will resolve collisions.

The `update:` method will also contain the enemy AI. This first crawler is very simple, so let's start there. Add the following method to **Crawler.m**:

```
- (void)update:(NSTimeInterval)dt {
  CGPoint gravity = CGPointMake(0.0, -450.0);
  CGPoint gravityStep = CGPointMultiplyScalar(gravity, dt);

  self.velocity = CGPointAdd(self.velocity, gravityStep);
  self.desiredPosition = CGPointAdd(self.position,
                       CGPointMultiplyScalar(self.velocity, dt));
}
```

For now, this is simply gravity. You'll return to this method later and add all the code for movement. But first you need to add the code that resolves collisions—otherwise this enemy will fall right through the floor!



You also need a `collisionBoundingBox` method for each of the enemies. The default implementation of that method in **PSKCharacter.m** might work in some cases, but only when all the animation frames are the same size. To be sure you don't get jittery physics, it makes more sense to use a custom size.

Add the following code to **Crawler.m**:

```
- (CGRect)collisionBoundingBox {
  return CGRectMake(
    self.desiredPosition.x - (kCrawlerWidth / 2),
    self.desiredPosition.y - (kCrawlerHeight / 2),
    kCrawlerWidth, kCrawlerHeight);
}
```

This should look familiar—it's similar to the code you used for the `Player` object. Now add the `#defines` for the height and width values after the `#import` statement:

```
#define kCrawlerWidth 32
#define kCrawlerHeight 32
```

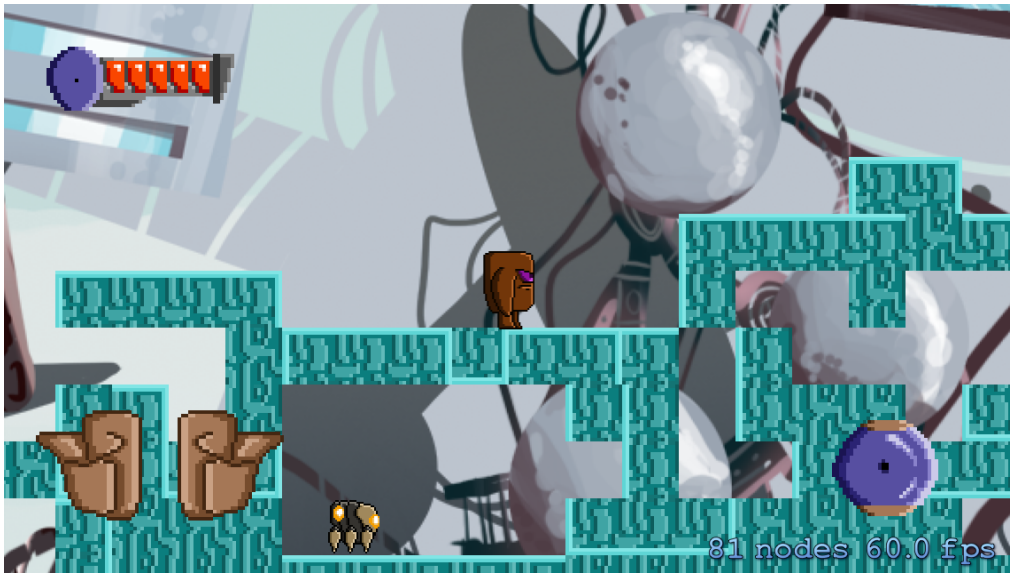
Switch to **PSKLevelScene.m** and add the following to `update:`, before the line that calls `setViewpointCenter:`:

```
for (PSKEnemy *enemy in self.enemies) {
  [enemy update:delta];
  [self checkForAndResolveCollisions:enemy forLayer:self.walls];
}
```

This `for` loop iterates over every enemy in the `enemies` array. For each one, it first calls `update:`, which, as you're aware, currently does nothing but apply gravity.

Once `update:` has applied the appropriate forces, you then call `checkForAndResolveCollisions:forLayer:` on each enemy. This exactly mirrors the process applied to the player.

Build and run, and you'll see that your enemies are now affected by gravity and that collisions are also active. You can tell physics are working because the Crawler is resting atop the tiles beneath it rather than intersecting, or floating above it.



Your first taste of AI

Now that the enemies are subject to the laws of physics, you can begin to give them behaviors. These behaviors are patterns of movement that you will invent to make contending with them interesting, challenging and fun.

This first enemy, the simple crawler, will live up to its name. It will walk in one direction until it collides with a wall and then it will reverse its direction. Once you have this enemy behavior in place, you'll move on to more complex behaviors.

First, add the following to **Crawler.m** after the `#import` line:

```
#define kMovementSpeed 60
```

This is a constant to control the speed of the simple crawler's movement.

Now alter `update:` in **Crawler.m** by adding the following code to the beginning of the method (leave the existing code in place):

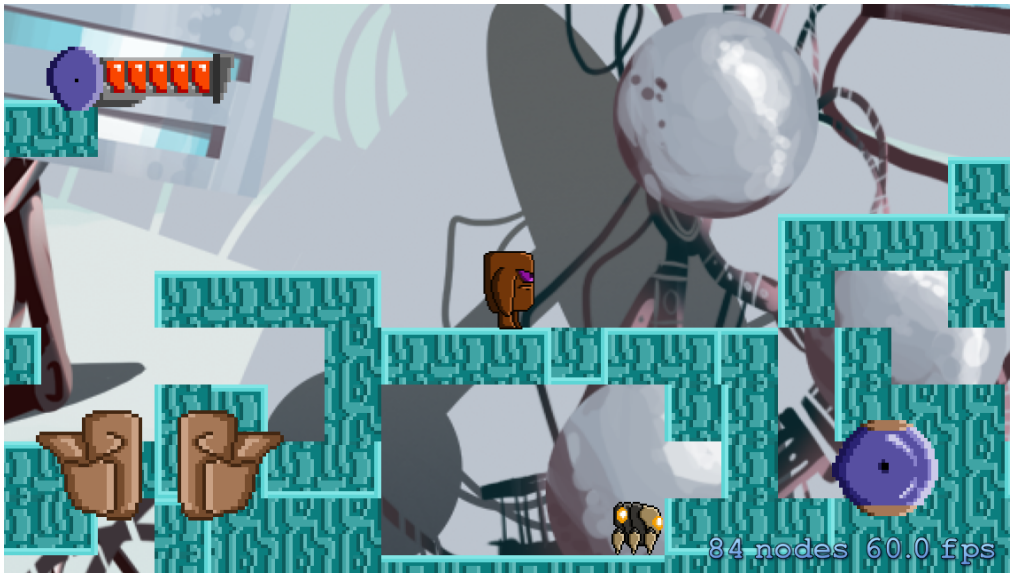
```
//1
if (self.onGround) {
  //2
  if (self.flipX) {
    self.velocity = CGPointMake(-kMovementSpeed, 0);
  } else {
    self.velocity = CGPointMake(kMovementSpeed, 0);
  }
  //3
} else {
  self.velocity = CGPointMake(self.velocity.x * 0.98,
                              self.velocity.y);
}
```

```
//4
if (self.onWall) {
  self.velocity = CGPointMake(-self.velocity.x,
                              self.velocity.y);
  //5
  if (self.velocity.x > 0) {
    self.flipX = NO;
  } else {
    self.flipX = YES;
  }
}
```

Here's what's happening one section at a time:

1. You detect whether or not the crawler is on the ground. If it's on the ground, then it's crawling, otherwise it has crawled off a ledge and is falling.
2. If the crawler's on the ground, you use the `flipX` property to determine which direction the crawler should be going—forward, of course! Based on this value, you set the velocity to `kMovementSpeed`. You make the velocity negative if the crawler is facing and crawling toward the left side of the screen.
3. If the crawler has gone off a ledge, then you want to slow its velocity along the x-axis so that it mostly falls straight down, instead of floating diagonally down. The Crawler still has a velocity force in the x dimension, so without damping he continues to move sideways, while falling down. This doesn't look right, once he's in mid air, he shouldn't continue to move forward at the same rate.
4. If the crawler has hit a wall, then the `onWall` property is `YES`. You use this property to determine when it's time to switch direction. Switching direction is as easy as negating the x property of `self.velocity`. If it's already negative, then this switches it back to positive, but you knew that already!
5. Finally, when you switch direction, you want to flip the sprite. So you check whether velocity is positive or negative and set the `flipX` property appropriately.

That's all it takes to create a simple crawler! Build and run now. You should have them crawling all about your level.



Don't worry; they won't hurt the cyclops—at least for now! ☺

Enemy animations

Now it's time to liven up these crawlers with some animations, using the same technique you used in the `Player` class. You'll load the animation objects when you initialize the crawler and then you'll keep track of the crawler's state, switching animations when it switches state. Do you remember the first step?

Load those animations! Create an instance variable to keep track of the `SKAction` object by adding the following to **Crawler.m**, right before the `@implementation` line:

```
@interface Crawler ()
@property (nonatomic, strong) SKAction *walkingAnim;
@end
```

For now, you have just one animation. Later you'll add others for the simple crawler and the more complex enemies as well.

Now add the `loadAnimations` method:

```
- (void)loadAnimations {
  self.walkingAnim = [self loadAnimationFromPlist:@"walkingAnim"
                                         forClass:@"Crawler"];
}
```

This is nothing to break a sweat over. The crawler has two main states: walking and falling. The falling state has just a single frame and no animation.

The next step is to put together the state machine that controls the transitions to go along with the `update:` method.

Add the following method:

```
- (void)changeState:(CharacterState)newState {
  if (newState == self.characterState) return;
  [self removeAllActions];
  self.characterState = newState;

  SKAction *action = nil;
  switch (newState) {
    case kStateWalking: {
      action = [SKAction repeatActionForever:self.walkingAnim];
      break;
    }
    case kStateFalling: {
      [self setTexture:[[PSKSharedTextureCache sharedCache]
                                    textureNamed:@"Crawler1.png"]];
      [self setSize:self.texture.size];
      break;
    }
    default:
      break;
  }
  if (action != nil) {
    [self runAction:action];
  }
}
```

This should look familiar! The only difference here from `Player`'s `changeState:` method is that the crawler has only two states.

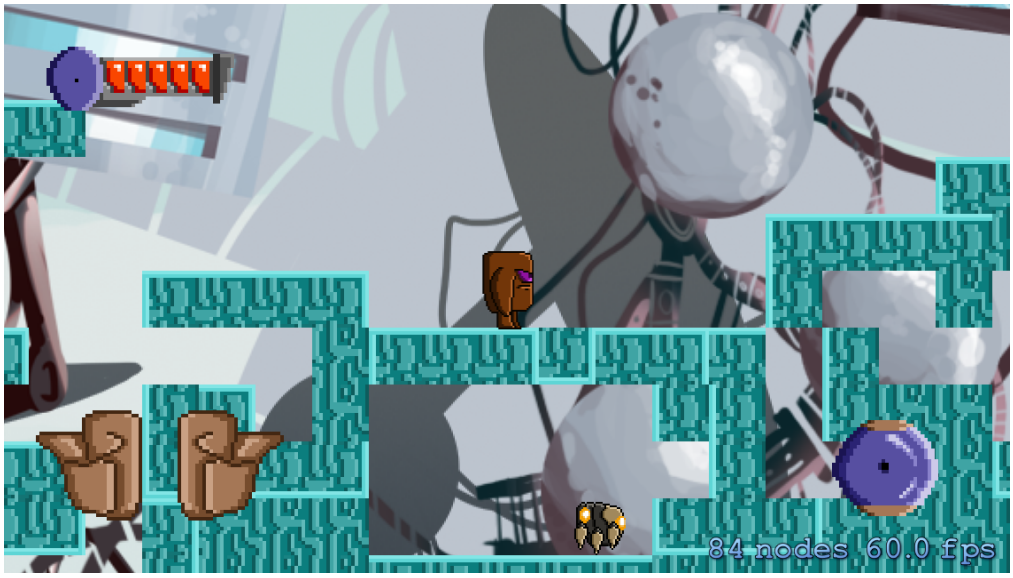
The final step is to trigger `changeState:` at the appropriate time. So in `update:`, after the test for `self.onGround`, change the state to `kStateWalking`:

```
if (self.onGround) {
  [self changeState:kStateWalking];
```

Then, after the `else` statement in that same `if` block but before the line that sets `self.velocity`, change the state to `kStateFalling`:

```
} else {
  [self changeState:kStateFalling];
```

Build and run again, and go find an enemy. Your crawler is much more spirited:



It's ALIVE!

More complex AI

That's it for the simple crawler. If you're wondering how this crawler and the enemies to come will exchange blows with the cyclops, the next chapter takes up that subject.

I think this crawler is a good start, but it's easy to avoid. You need a crawler that's a little more obnoxious and hence, the mean crawler. ☺

Create a new file with the **iOS\Cocoa Touch\Objective-C class** template. Name the class **MeanCrawler** and make it a subclass of `PSKEnemy`.

This mean crawler does a few things that the simple crawler can't. For one, it will pursue the player instead of just blindly walking back and forth. Second, the mean crawler can jump—both to climb stairs to get to the cyclops and to pounce on the cyclops if the cyclops tries to jump over it.

This new crawler will be much harder to avoid!

> **Note**: Why did you create `MeanCrawler` as a subclass of `PSKEnemy` instead of `Crawler`? Even though they look very similar, the two types of crawlers actually don't share enough code to make it worthwhile to make `MeanCrawler` a subclass of `Crawler`.

Adding new enemy types

First you need to add two new properties that together will tell the enemy the location of the cyclops. Add the following imports to **PSKEnemy.h**:

```
#import "Player.h"
#import "JSTileMap+TileLocations.h"
```

Then add the new properties:

```
@property (nonatomic, weak) Player *player;
@property (nonatomic, weak) JSTileMap *map;
```

These two properties give the enemies access to information they need to act more intelligently. Some enemies will chase the player, or otherwise react to his actions. Other enemies will react to the features of the level (by jumping over obstacles in its path, for example).

Now you need to set these properties when you initialize an enemy instance and add it to the layer. In fact, you also need to make some changes to the code that loads the enemies so that you can load different types.

Update `loadEnemies` in **PSKLevelScene.m** so it looks like this:

```
- (void)loadEnemies {
  self.enemies = [NSMutableArray array];

  TMXObjectGroup *enemiesGroup = [self.map
                                        groupNamed:@"enemies"];
  for (NSDictionary *enemyDict in enemiesGroup.objects) {
    //1
    NSString *enemyType = enemyDict[@"type"];
    NSString *firstFrameName = [NSString
                          stringWithFormat:@"%@1.png", enemyType];
    // 2
    PSKEnemy *enemy = [[NSClassFromString(enemyType) alloc]
                               initWithImageNamed:firstFrameName];

    enemy.position = CGPointMake([enemyDict[@"x"] floatValue],
                                 [enemyDict[@"y"] floatValue]);

    //3
    enemy.player = self.player;
    enemy.map = self.map;

    [self.map addChild:enemy];
    enemy.zPosition = 900;
    [self.enemies addObject:enemy];
  }
}
```

I've marked the changes/additions with numbers:

1. Instead of hardcoding "Crawler", you now use the `type` key in the dictionary to look up the enemy's class. You construct the frame name using this same class name.

2. Instead of using the `Crawler` class to initialize the enemy object type, you now `NSClassFromString()`, a nice feature of Objective-C. This function gets a reference to the class based on the passed-in string. You use this to initialize whatever type of enemy you want to create.
3. Here, you set the `player` and `map` properties of the enemy object. Not all enemies will make use of the information (the basic crawler won't), but that's OK. It's convenient to use this same loop to instantiate all your enemy types.

The rest of the code is the same as before.

It's time build and run to test the code, but since you're now loading the enemies from the tile map, you'll get a crash if you don't have a class declared in your code for every enemy type in the map file. To avoid this, you need to create one more class.

Do that by creating a new file with the **iOS\Cocoa Touch\Objective-C class** template. Make it a subclass of `PSKEnemy` and name it **Flyer**.

Now add import statements for the new classes to **PSKLevelScene.m**:

```
#import "Flyer.h"
#import "MeanCrawler.h"
```

One final thing you must do is include a minimal `update:` method in each of these classes. Recall that your logic for collision resolution relies on the `desiredPosition` property. If you don't set this value, it will probably be zero. When your collision detection logic asks for the surrounding tiles, a `desiredPosition` of (0,0) will result in a `TMXLayer` error—a tile outside of the tile map's boundaries.

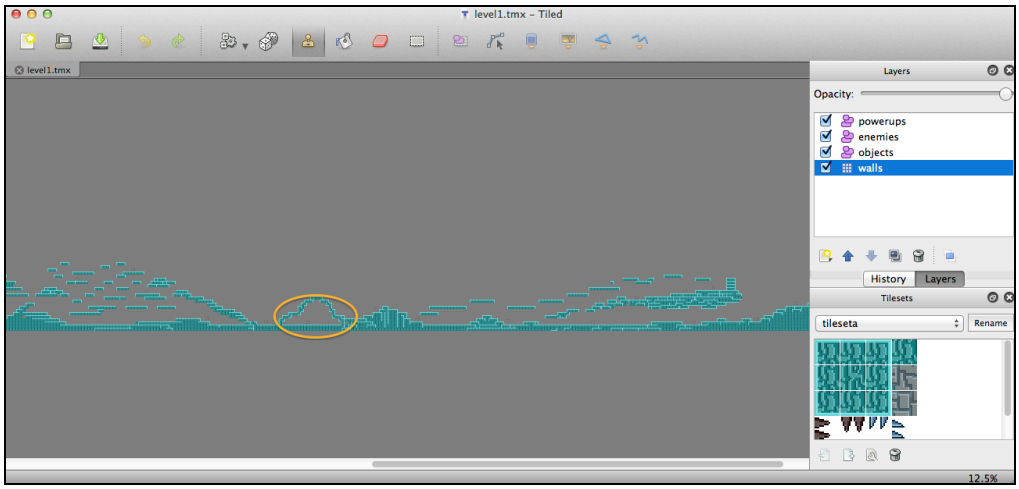
So add the following code to both **MeanCrawler.m** and **Flyer.m**:

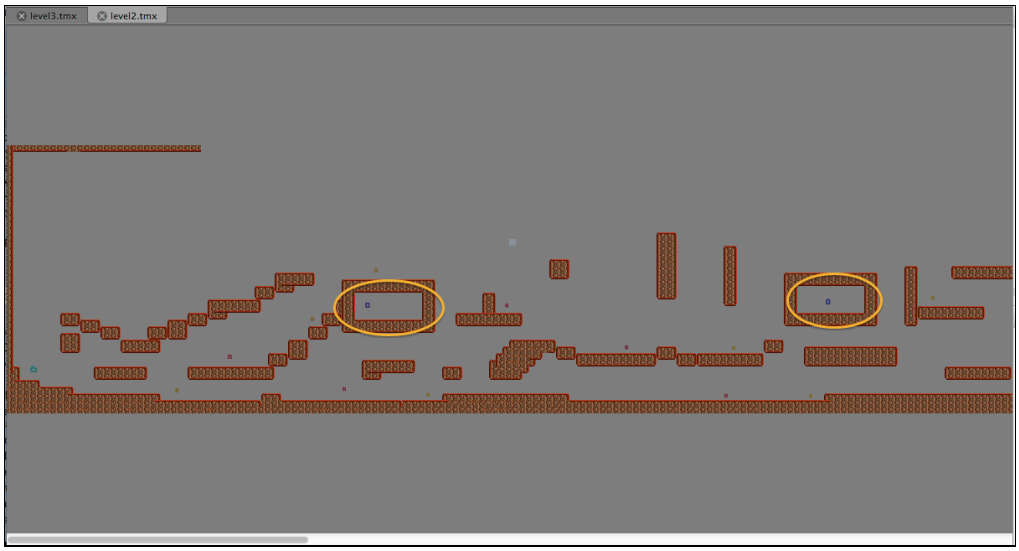
```
- (void)update:(NSTimeInterval)dt {
  self.desiredPosition = self.position;
}
```

Now you can build and run without errors—so do it! Mean crawlers and flyers appear in your levels, though they float idly in the air and will until you modify `update:` to give them some purpose.

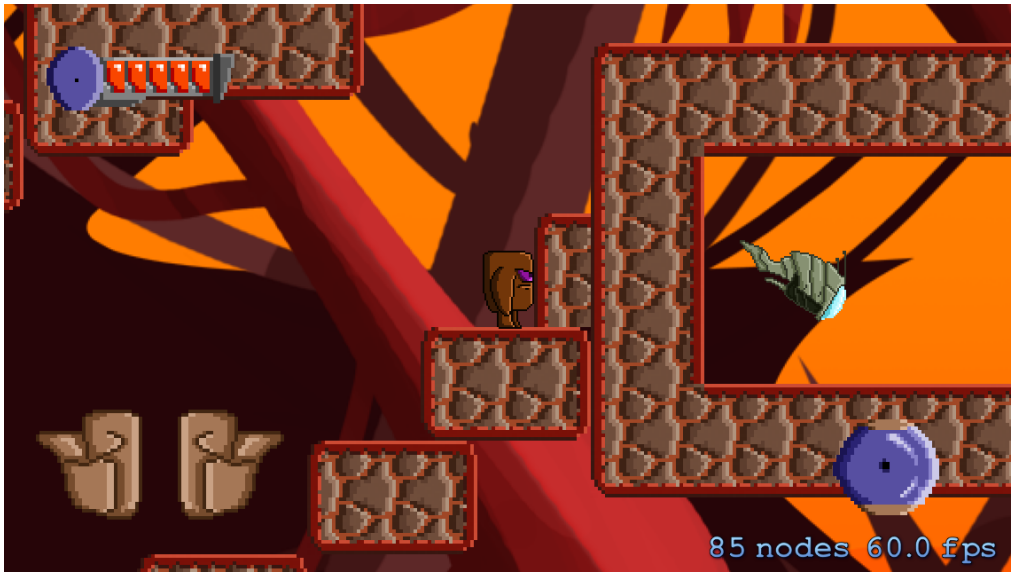
The first level has mean crawlers toward the end. To see a flyer, you need to go to Level 2.

Here are the locations in Levels 1 and 2 where you can find these enemies:





Here's what you should see:



As you know, this civilized behavior between the cyclops and its enemies won't last for long. ☺

The mean crawler

Now that your game successfully loads the new enemy types, the only thing left to do is to add the logic and animations for each class.

Add the following class extension to **MeanCrawler.m**, above the `@implementation`:

```
@interface MeanCrawler ()
@property (nonatomic, strong) SKAction *walkingAnim;
@property (nonatomic, strong) SKAction *jumpUpAnim;
@end
```

Then add the following new method inside the `@implementation`:

```
- (void)loadAnimations {
  self.walkingAnim = [self loadAnimationFromPlist:@"walkingAnim"
                                         forClass:@"MeanCrawler"];
  self.jumpUpAnim = [self loadAnimationFromPlist:@"jumpUpAnim"
                                         forClass:@"MeanCrawler"];
}
```

By now, this code should make sense to you. Eventually all characters will also have a death animation, but those will come after you add the damage logic in the next chapter.

Next add `changeState:`. This should also need no explanation:

```
- (void)changeState:(CharacterState)newState {
  if (newState == self.characterState) return;
  [self removeAllActions];
```

```
  self.characterState = newState;

  SKAction *action = nil;
  switch (newState) {
    case kStateWalking: {
      action = [SKAction repeatActionForever:self.walkingAnim];
      break;
    }
    case kStateJumping: {
      action = self.jumpUpAnim;
      break;
    }
    case kStateFalling: {
      [self setTexture:[[PSKSharedTextureCache sharedCache]
                              textureNamed:@"MeanCrawler1.png"]];
      [self setSize:self.texture.size];
      break;
    }
    default:
      break;
  }

  if (action != nil) {
    [self runAction:action];
  }
}
```

Now for the interesting part—adding the logic. ☺ But first, add the `kMovementSpeed` constant to **MeanCrawler.m**, after the `#import` line:

```
#define kMovementSpeed 60
```

With that done, replace the placeholder `update:` you added earlier with the following:

```
- (void)update:(NSTimeInterval)dt {
  //1
  CGFloat distance = CGPointDistance(self.position,
                                     self.player.position);
  if (distance > 1000) {
    self.desiredPosition = self.position;
    return;
  }
  //2
  CGPoint gravity = CGPointMake(0.0, -450.0);
  CGPoint gravityStep = CGPointMultiplyScalar(gravity, dt);
  //3
  if (self.player.position.x > self.position.x) {
    self.velocity = CGPointMake(kMovementSpeed,
                                self.velocity.y);
  } else {
    self.velocity = CGPointMake(-kMovementSpeed,
```

```
                                    self.velocity.y);
    }
    //4
    self.velocity = CGPointAdd(self.velocity, gravityStep);
    //5
    if (self.velocity.x > 0) {
      self.flipX = NO;
    } else {
      self.flipX = YES;
    }
    //6
    CGPoint velocityStep = CGPointMultiplyScalar(
                                              self.velocity, dt);
    self.desiredPosition = CGPointAdd(self.position,
                                      velocityStep);
}
```

This is the first iteration of `update:` for the mean crawler. You'll add the jump behavior soon. Here's what's happening:

1. The first step is something new. If the cyclops isn't within at least 1000 points of the enemy, you don't want the enemy to pursue the cyclops. You want the enemy to wait until the cyclops gets closer. This code bails out of `update:`—but only after setting the `desiredPosition`—until the cyclops gets within 1000 points of the enemy.
2. This is standard by now. You set the gravity vector and create a `gravityStep` variable for the current loop iteration.
3. You test whether the cyclops is left or right of the mean crawler and set the crawler's velocity accordingly, based on `kMovementSpeed`.
4. You add the current speed to the current velocity.
5. You set `flipX` according to the mean crawler's direction of movement.
6. Finally, you set the `desiredPosition` by creating a scaled `velocityStep` and adding it to the current position.

Build and run. Your mean crawlers now pursue the cyclops.

While you're at it, add the same proximity check to the `Crawler` class so that it will remain dormant until the player gets within 1000 points. This helps your level design by preventing your enemies from straying before the cyclops gets to them, and it saves memory resources as well.

Add the following code to **Crawler.m** at the beginning of the crawler's `update:` method, before all of the existing code:

```
CGFloat distance = CGPointDistance(self.position,
                                   self.player.position);

if (distance > 1000) {
  self.desiredPosition = self.position;
  return;
}
```

Making the mean crawler jump

Getting back to the mean crawler**,** what if there happens to be a step in the mean crawler's way? In this next section, you'll add logic to detect a wall tile that's two tiles in front of the mean crawler. If there is a wall tile, then the mean crawler will jump.

You wouldn't want to encounter one of these in a dark room.

This code will also include logic that changes the mean crawler's state.

Replace `update:` in **MeanCrawler.m** with the following improved version:

```
- (void)update:(NSTimeInterval)dt {
  //1
  CGFloat distance = CGPointDistance(self.position,
                                     self.player.position);
  if (distance > 1000) {
    self.desiredPosition = self.position;
    return;
  }
  //2
  if (self.onGround) {
    [self changeState:kStateWalking];
  }
  //3
  TMXLayer *layer = [self.map layerNamed:@"walls"];
  CGPoint myTileCoord = [layer coordForPoint:self.position];
  //4
  CGPoint twoTilesAhead = CGPointZero;
  //5
  NSInteger playerDirection = (signbit(
            self.position.x - self.player.position.x)) ? 1 : -1;
  //6
  twoTilesAhead = CGPointMake(
          myTileCoord.x + (playerDirection * 2), myTileCoord.y);
  //7
  twoTilesAhead = CGPointMake(
      Clamp(twoTilesAhead.x, 0, self.map.mapSize.width),
      Clamp(twoTilesAhead.y, 0, self.map.mapSize.height));
  //8
  self.velocity = CGPointMake(
           (playerDirection * kMovementSpeed), self.velocity.y);
  //9
  if ([self.map isWallAtTileCoord:twoTilesAhead] ||
         (distance < 100 && (self.player.position.y -
                             self.position.y) > 50)) {
    //10
    if (self.onGround) {
```

```
        //11
        self.velocity = CGPointMake(self.velocity.x, kJumpForce);
        [self changeState:kStateJumping];
      }
    }
    //12
    CGPoint gravity = CGPointMake(0.0, -450.0);
    CGPoint gravityStep = CGPointMultiplyScalar(gravity, dt);
    self.velocity = CGPointAdd(self.velocity, gravityStep);
    if (self.velocity.x > 0) {
      self.flipX = NO;
    } else {
      self.flipX = YES;
    }
    CGPoint velocityStep = CGPointMultiplyScalar(
                                                 self.velocity, dt);
    self.desiredPosition = CGPointAdd(self.position,
                                      velocityStep);
}
```

Here's a full explanation of this improved method:

1. You check the cyclops's proximity to the crawler and bail out if it's not close enough.
2. You check to see if the mean crawler is on the ground and if it is, you set its state to walking.
3. You want to find out if the `MeanCrawler` needs to jump over anything to get to the cyclops. You do this by looking two tiles horizontally in front of the `MeanCrawler`'s current position. The first step of this process is to get the `MeanCrawler`'s current tile coordinates. You can do this with the `coordForPoint` method on the "walls" `TMXLayer`.
4. Then, you create a new `CGPoint` variable that will eventually hold the coordinates of the tile that is two tiles in front of the `MeanCrawler`.
5. You know the `MeanCrawler` will always be pursuing the cyclops, so by finding out where the cyclops is relative to the `MeanCrawler`, you can decide the direction you want the crawler to face/travel.

   The `signbit()` function returns a non-zero value, interpreted by your test as true, if the passed-in value is negative. Otherwise `signbit()` returns a zero, interpreted as false. In this case you subtract the x-position of the cyclops from the x-position of the `MeanCrawler`.

   If the cyclops is to the right of the `MeanCrawler`, the cyclops's x-position is larger and subtracting that larger value from the `MeanCrawler`'s smaller value results in a negative number. This means that the test returns false and `playerDirection` equals 1, so the `MeanCrawler` faces and travels to the right.
6. Now you use `playerDirection` to find the coordinates of the tile that is two tiles in front of the `MeanCrawler` and store them in `twoTilesAhead`.

7. You need to make sure `twoTilesAhead` doesn't contain coordinates that are beyond the size of the tile map. If you ask for a negative tile or a tile larger than the map, it will fail an `NSAssert` statement and your game will halt.
8. You also use `playerDirection` to set the direction of the `MeanCrawler`'s velocity
9. This `if` statement contains two tests. The first tests whether there is a wall at the position in `twoTilesAhead`. If there is, you want to make your `MeanCrawler` jump. You haven't written this `isWallAtTileCoord` method yet, but you will in a minute.

   The second test is whether the `MeanCrawler` is within 100 points of the cyclops, and if the player is significantly above the `MeanCrawler`. When you try to jump over a `MeanCrawler`, it is going to try to jump up and catch you in the air. This test looks for that.
10. If either of the above conditions is true, you also need to make sure the `MeanCrawler` is on the ground, as it can't jump from midair.
11. Finally, you set the velocity on the y-axis to some jump force and set the state to jumping.
12. The rest of this code was covered previously.

You need to set the value of `kJumpForce` for this code to work, so add the following line at the top of **MeanCrawler.m**:

```
#define kJumpForce 250
```

The other piece required is the method that checks whether there is a wall tile at a certain set of coordinates. To implement it, first add the following method declaration to **JSTileMap+TileLocations.h**:

```
- (BOOL)isWallAtTileCoord:(CGPoint)tileCoord;
```

Next add the method to **JSTileMap+TileLocations.m**:

```
- (BOOL)isWallAtTileCoord:(CGPoint)tileCoord {
  TMXLayer *layer = [self layerNamed:@"walls"];
  NSInteger gid = [layer tileGIDAtTileCoord:tileCoord];
  return (gid != 0);
}
```

The logic here should remind you of testing for a GID in the collision resolution function. You simply check for a GID and return `YES` if there is one and `NO` if there isn't.

Build and run. Your mean crawler is now much meaner!

The meanest of them all

Your game has acquired new life, but there's one last type of enemy with logic missing: the flyer.

It lives up to its name. The flyer's behavior will follow this pattern:

1. If the cyclops is more than 1,000 points away, stay dormant.
2. If the cyclops is facing the flyer, go to sleep.
3. If the cyclops is facing away, pursue it.
4. If the cyclops gets close, attack!

You'll tackle `changeState:` first, since it's become routine. The flyer has two animations. Add the properties for those via a class extension at the top of **Flyer.m**:

```
@interface Flyer ()
@property (nonatomic, strong) SKAction *seekingAnim;
@property (nonatomic, strong) SKAction *attackingAnim;
@end
```

Now implement `loadAnimations`:

```
- (void)loadAnimations {
  self.seekingAnim = [self loadAnimationFromPlist:@"seekingAnim"
                                         forClass:@"Flyer"];
  self.attackingAnim = [self
              loadAnimationFromPlist:@"attackingAnim"
                            forClass:@"Flyer"];
}
```

And then `changeState:`:

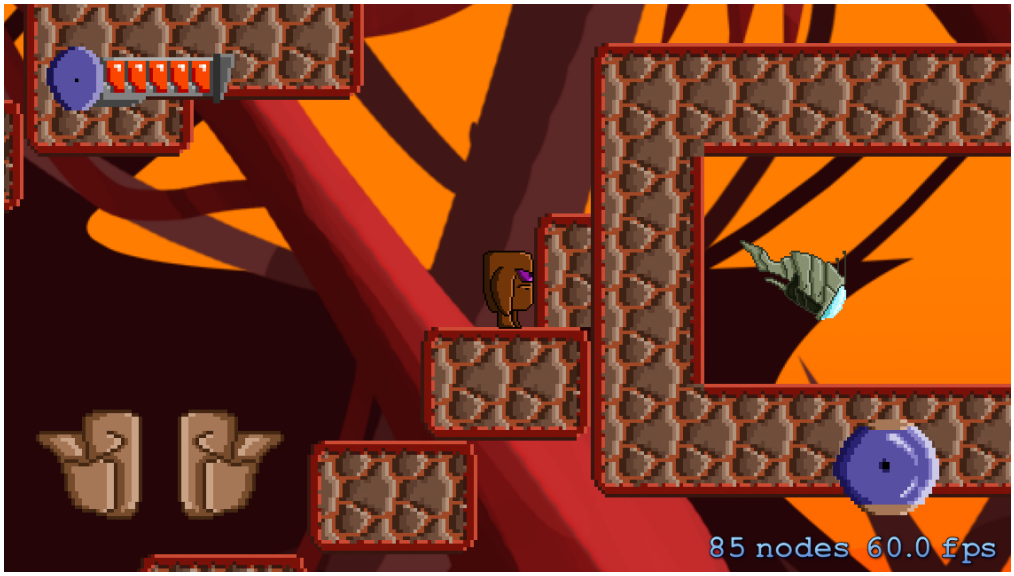
```
- (void)changeState:(CharacterState)newState {
  if (newState == self.characterState) return;
  [self removeAllActions];
  self.characterState = newState;

  SKAction *action = nil;
  switch (newState) {
    case kStateSeeking: {
      action = [SKAction repeatActionForever:self.seekingAnim];
      break;
    }
    case kStateHiding: {
      [self setTexture:[[PSKSharedTextureCache sharedCache]
                                    textureNamed:@"Flyer4.png"]];
      [self setSize:self.texture.size];
      break;
    }
    case kStateAttacking: {
      action = [SKAction
                      repeatActionForever:self.attackingAnim];
      break;
    }
    default:
      break;
  }
  if (action != nil) {
    [self runAction:action];
  }
}
```

The third time's the charm, right? All of this logic mirrors what you've done before. The three states are:

- **Seeking** when the cyclops's back is turned to the flyer;
- **Hiding** when the cyclops is facing the flyer;
- **Attacking** when the cyclops gets too close.

Build and run now. You won't see any changes in the behavior of the flyer, because you haven't yet modified the `Flyer` class's `update:` method to include state changes, but you can verify that all this code is compiling without errors.



Finally, replace `update:` with this new version:

```
- (void)update:(NSTimeInterval)delta {
  //1
  CGFloat distance = CGPointDistance(self.position,
                                     self.player.position);
  if (distance > 1000) {
    self.desiredPosition = self.position;
    return;
  }
  //2
  CGFloat speed;
  //3
  if (distance < 100) {
    [self changeState:kStateAttacking];
    speed = 100;
  //4
  } else if ((!self.player.flipX && self.player.position.x <
                                    self.position.x)
          || (self.player.flipX && self.player.position.x >
                                    self.position.x)) {
    [self changeState:kStateHiding];
    speed = 0;
  //5
  } else {
    [self changeState:kStateSeeking];
    speed = 60;
  }
  //6
  CGPoint v = CGPointNormalize(
          CGPointSubtract(self.player.position, self.position));
  self.velocity = CGPointMultiplyScalar(v, speed);
  //7
  if (self.position.x < self.player.position.x) {
```

```
      self.flipX = NO;
    } else {
      self.flipX = YES;
    }
    //8
    CGPoint stepVelocity = CGPointMultiplyScalar(
                                          self.velocity, delta);
    self.desiredPosition = CGPointAdd(self.position,
                                      stepVelocity);
  }
```

Here's your last method breakdown for this chapter:

1. This is similar to the other enemy `update:` methods. It looks to see if the cyclops is in the vicinity and returns if it is not.
2. Here you create a variable that will be populated by the `if` statement in the next section. There is a speed for each of the flyer's three possible states.
3. If the cyclops is really close, less than 100 points away, then the flyer attacks, regardless of whether the cyclops is facing it. This first block sets up this action by setting the flyer's speed to 100 and changing its state to attacking.
4. Here, you check whether the cyclops is looking in the direction of the flyer. You know that the cyclops is more than 100 points away, or the first block would have executed and this test wouldn't be running. If the cyclops is looking at the flyer, then you change the flyer's state to hiding and set its speed to zero.
5. If neither of the above tests are true, then you can infer that the cyclops is more than 100 points away and has its back to the flyer. In this case, you set the flyer's speed to a moderate pace of 60 and its state to seeking.
6. Next, you retrieve a normalized vector, first by calculating a `CGPoint` that is the distance between the flyer and the cyclops. You then normalize the point by scaling its length to one. So, for example, if you had a distance that was 100 points high and 50 points wide, normalizing it would change it to (0.89,0.45), a vector with a length of one. This is called a **unit vector**.

   You can then scale up the vector to give each time step a constant speed in whichever direction the flyer needs to move to seek the cyclops. The second line does just that.
7. You determine the direction in which the flyer should be moving—whichever is towards the cyclops—and set the `flipX` value accordingly.
8. Finally, you scale the current velocity to the size of the time step and then calculate the `desiredPosition` based on the current position of the flyer and the velocity for the step.

There's one last thing you need to do: add a custom bounding box method. If you run this code now, it will work because you can rely on the `collisionBoundingBox` method in `PSKCharacter`. A custom method is better, however, in case any of the frames are of a different size.

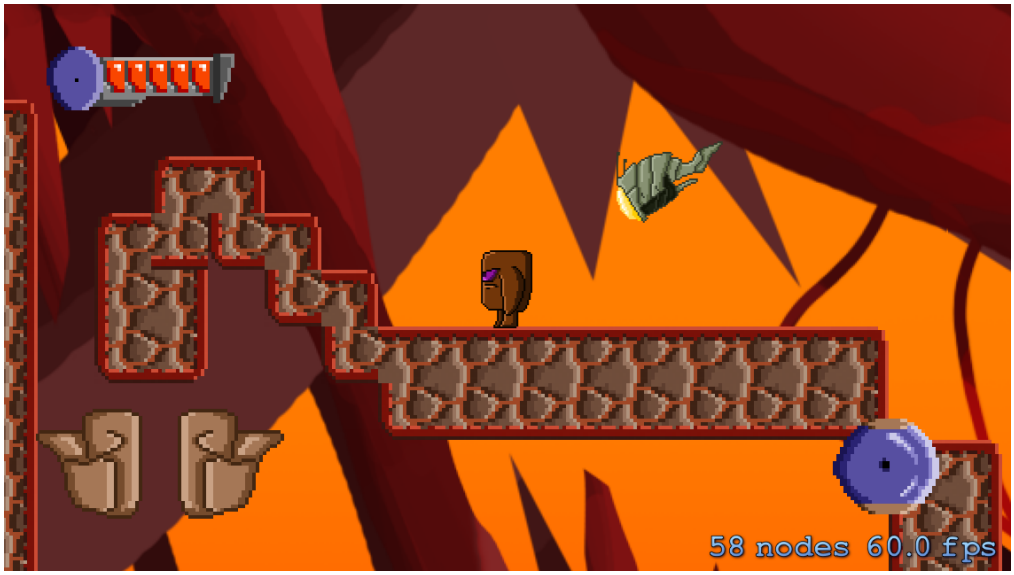
Add this method (anywhere in the implementation):

```
- (CGRect)collisionBoundingBox {
  return CGRectMake(
      self.desiredPosition.x - (kFlyerWidth / 2),
      self.desiredPosition.y - (kFlyerHeight / 2),
      kFlyerWidth, kFlyerHeight);
}
```

Then add the defines after the `#import` line:

```
#define kFlyerWidth 64
#define kFlyerHeight 64
```

That's it! Build and run again. Your flyers should seek, attack and hide. But remember that you have to move to Level 2 to encounter flyers!



You now have three working enemy types and a framework for creating enemy AI. You should now understand how you could create additional enemies, or enemies with more complex behaviors, by increasing the number of states and expanding the logic in `update:`.

In fact, this technique can accommodate fairly complicated enemies. And while this game has only three enemies, your games will probably have many more.

Get ready for the next chapter, where the game will get a lot more dangerous as you make those enemies deal the cyclops some serious damage!

No fear, no fun!

Challenge: Create a new enemy type of your choosing and add it to the game! If you need an idea, consider adding an enemy that has different attack states that you cycle through.

Chapter 9: Damage & Death-Dealing

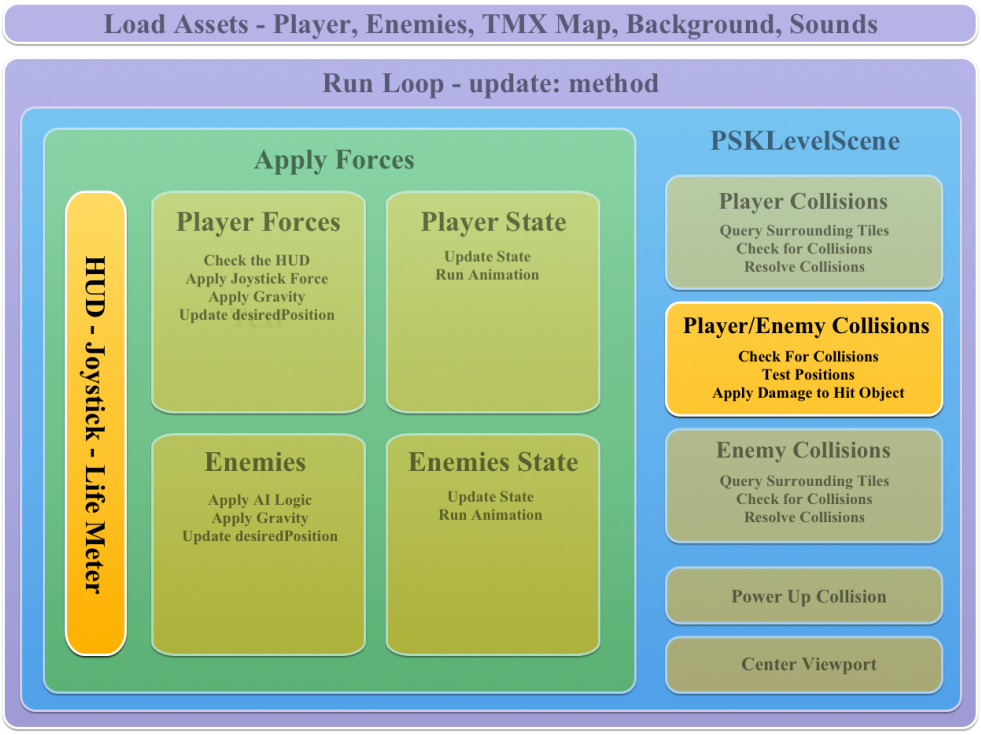
Now that you've got a fully functioning army of enemies, it is time to end the armistice and bring them into battle with the cyclops.

Here are the terms of engagement:

- If the cyclops lands on top of an enemy, it will damage the enemy. You'll add logic that will allow multiple hits for a kill, but in this game all of your enemies will die after one hit.
- Any other collision with an enemy will damage the cyclops.
- The cyclops has enough life to take five hits before it dies.

You also need to create conditions under which the characters can be killed. This requires death animations and state change logic for the cyclops and each of the enemies.

Here are the sections of the overall game plan you'll be tackling in this chapter:



Player/enemy collisions

The first step is to create a method that determines when the cyclops has collided with an enemy.

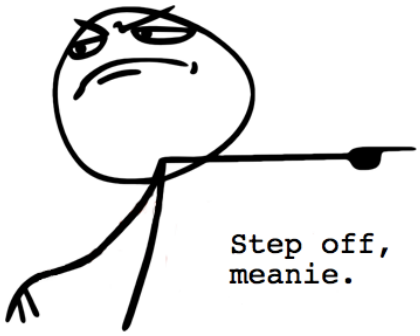
To make this method more efficient, you'll add a new property to enemies to tracks whether they are active. If an enemy is farther than 1,000 points away from the cyclops, you'll consider it to be in a dormant state and won't include it in collision testing.

Add the following property to **PSKCharacter.h**:

```
@property (nonatomic, assign) BOOL isActive;
```

This property will do two things. For `PSKEnemy` subclasses, you'll set it to `NO` if the distance between the cyclops and the enemy is too great. Then, when checking for collisions, the `update:` method will use this property to skip over enemies that are in distant sections of the level.

In the case of the cyclops, you'll use this property to give it some temporary protection after taking damage from an enemy. There will be a cool-down period after every hit during which the cyclops is invulnerable. If not for this, the cyclops could be hit too many times in quick succession during a single collision.



Modify `update:` in each of the `PSKEnemy` subclasses: **Crawler.m**, **MeanCrawler.m** and **Flyer.m**. Each of them has the following code block at the very beginning of `update::`

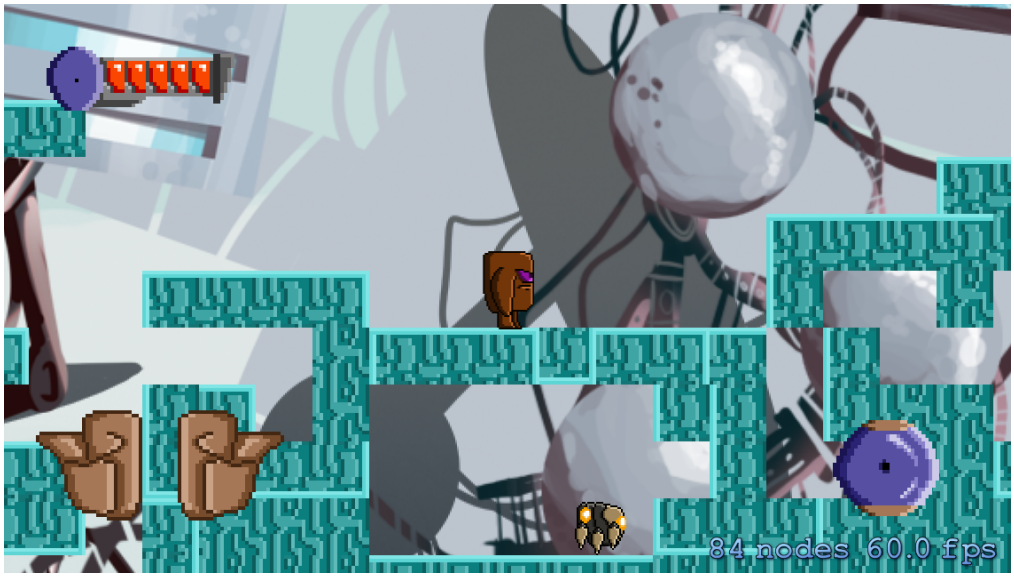
```
CGFloat distance = CGPointDistance(self.position,
                                   self.player.position);
if (distance > 1000) {
  self.desiredPosition = self.position;
  return;
}
```

Modify the above so that it matches the following (changes highlighted):

```
CGFloat distance = CGPointDistance(self.position,
                                   self.player.position);
if (distance > 1000) {
  self.desiredPosition = self.position;
  self.isActive = NO;
  return;
} else {
  self.isActive = YES;
}
```

All you've done is alter the original code so that if the cyclops isn't close to the enemy, you set `isActive` to `NO` before bailing out. Otherwise, you set `isActive` to `YES` and continue processing.

Build and run now. Then find some enemies and make sure they're still moving—that is, that you haven't broken anything.



Now that you've got a valid `isActive` property, you can turn to the method that does the actual collision detection.

Add the following method to **PSKLevelScene.m**:

```
- (void)checkForEnemyCollisions:(PSKEnemy *)enemy {
  //1
  if (enemy.isActive && self.player.isActive) {
    //2
    if (CGRectIntersectsRect(self.player.collisionBoundingBox,
                             enemy.collisionBoundingBox)) {
      //3
      CGPoint playerFootPoint = CGPointMake(
          self.player.position.x, self.player.position.y -
              self.player.collisionBoundingBox.size.height / 2);
      //4
      if (self.player.velocity.y < 0 && playerFootPoint.y >
                                              enemy.position.y) {
        //5
        [self.player bounce];
        [enemy tookHit:self.player];
      } else {
        [self.player tookHit:enemy];
      }
    }
  }
}
```

You're going to call this method from the `update:` loop. Here's what each line does:

1. You check if the current enemy is active. Inactive enemies couldn't possibly collide with the cyclops.
2. Then you check for a collision between the bounding boxes of the enemy and the cyclops.

3. Next you find a point that represents the bottom center of the cyclops's bounding box. This is the foot point. You'll use it to determine whether the cyclops is hitting the top of the enemy.
4. This tests whether the foot point of the cyclops is directly above the enemy and whether the cyclops is falling. That's enough to determine whether the cyclops is coming down on top of the enemy.
5. Finally, you call `tookHit:` for the cyclops or the enemy, depending on who should be damaged by the collision.

Now implement `tookHit` in the `PSKCharacter` class. Each subclass can have its own implementation of `tookHit`, but since you want it to be common to both `Player` and `PSKEnemy` subclasses, it makes sense to add it to the `PSKCharacter` class.

First add the declaration to **PSKCharacter.h**:

```
- (void)tookHit:(PSKCharacter *)character;
```

Then add the method to **PSKCharacter.m**:

```
- (void)tookHit:(PSKCharacter *)character {
  NSLog(@"Took hit %@, %@", character, self);
}
```

The above code simply logs the two objects involved in the collision. You're going to use this to test your code before implementing the full method.

Now add the following line to **PSKLevelScene.m**'s `update:` method, inside the loop that iterates through the `enemies` array, after the call to `checkForAndResolveCollisions:forLayer:`:

```
[self checkForEnemyCollisions:enemy];
```

You'll implement `bounce` a little later, but for now you need a method so your code will compile. Add the following to **Player.h:**

```
- (void)bounce;
```

And add the empty method implementation in **Player.m**:

```
- (void)bounce {
  // nothing here yet
}
```

There is one last thing to do before you test if this works. The `Player` needs to have their `isActive` flag set to `YES` during initialization. Add the following line to `initWithImageNamed:` in **Player.m**, after the line that initializes `self.velocity`:

```
self.isActive = YES;
```

Build and run. You should see the Xcode console logging collisions. The first object is dealing the hit and the second is receiving damage.

Experiment with this by running directly into crawlers and by jumping on them. You should see the difference in the console, based on the situation.

Of course, the cyclops and its enemies still seem to be on civil terms because neither gives or takes damage. You're about to change that!

Applying damage to enemies

All the enemies in the game will die after one hit, so their logic is easy to implement and is a good place to start. It's time to squash the meanies!

When an enemy takes a hit, it will transition to the death state and trigger the death animation.

You can use the same `tookHit:` implementation for all enemies, so implement it as follows in **PSKEnemy.m**:

```
- (void)tookHit:(PSKCharacter *)character {
  self.life = self.life - 100;
  if (self.life <= 0) {
    [self changeState:kStateDead];
  }
}
```

This code references a new property, `life`. Every enemy will have a `life` value of 100 and so will be finished by a single hit.

> **Note**: Although the code's not using it, `tookHit:` takes a `PSKCharacter` instance as a parameter. If you ever wanted to have different characters deal different amounts of damage, you could implement that by inspecting the class of the character.

You need to add the `life` property to the `Character` class, since the player will also have life—though more of it. Do so by adding this line to **PSKCharacter.h**:

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

Now initialize the property for all enemies by adding the following initializer to **PSKEnemy.m**:

```
- (id)initWithImageNamed:(NSString *)name {
  if (self = [super initWithImageNamed:name]) {
    self.life = 100;
  }
  return self;
}
```

Each `PSKEnemy` subclass now needs to have an entry in the `changeState:` method for `kStateDead`. They also need to have an animation available for that state. I've called that animation `dyingAnim`.

Start with the animations. Because every `PSKCharacter` subclass has a `dyingAnim`, even though the frames are different, you can add this to **PSKCharacter.h**:

```
@property (nonatomic, strong) SKAction *dyingAnim;
```

Now change `loadAnimations` for all three subclasses by adding this line to **Crawler.m**:

```
self.dyingAnim = [self loadAnimationFromPlist:@"dyingAnim"
                                     forClass:@"Crawler"];
```

Add this line to **MeanCrawler.m**:

```
self.dyingAnim = [self loadAnimationFromPlist:@"dyingAnim"
                                     forClass:@"MeanCrawler"];
```

And add this line to **Flyer.m**:

```
self.dyingAnim = [self loadAnimationFromPlist:@"dyingAnim"
                                     forClass:@"Flyer"];
```

Now add the `kStateDead` branch to each of the `changeState:` methods of each of the `PSKEnemy` subclasses. Add it at the end of the method, before the `default` case:

```
case kStateDead: {
  action = [SKAction sequence:@[self.dyingAnim,
                [SKAction performSelector:@selector(removeSelf)
                                 onTarget:self]]];
  break;
}
```

This code creates an `SKAction` with a sequence, which allows you to chain a number of animations or other actions together. In this case, you are first running the animation and then calling `SKAction performSelector:onTarget:`, an action that calls a user-defined method. Next you need to set up that method, `removeSelf`.

Add the method declaration for `removeSelf` to **PSKEnemy.h**:

```
- (void)removeSelf;
```

And add the implementation to **PSKEnemy.m**:

```
- (void)removeSelf {
  self.isActive = NO;
}
```

When you want to remove an `SKNode` from its parent node, there's a method called `removeFromParent:` that you can use. In the case of enemies, though, you also need to remove them from the `enemies` array in `PSKLevelScene`. The `PSKLevelScene` needs to know that it's time to remove the object. But how will it know when to remove them?

Removing enemies from the scene

Well, you're going to create a check that looks in the `enemies` array for enemies that have both their state set to `kStateDead` and `isActive` set to `NO`. `kStateDead` itself

isn't sufficient, because you need to give the `dyingAnim` some time to run all the way through. If you were to remove the enemy as soon as its state became `kStateDead`, it would just disappear, wasting your artist's hard work!

You need to alter the `update:` method of `PSKLevelScene` to accomplish this. Replace the current `update:` implementation in **PSKLevelScene.m** with this modified version:

```
- (void)update:(NSTimeInterval)currentTime {
  NSTimeInterval delta = currentTime - self.previousUpdateTime;
  self.previousUpdateTime = currentTime;
  if (delta > 0.1) {
    delta = 0.1;
  }

  [self.player update:delta];
  [self checkForAndResolveCollisions:self.player
                             forLayer:self.walls];

  //1
  NSMutableArray *enemiesThatNeedDeleting;

  for (PSKEnemy *enemy in self.enemies) {
    [enemy update:delta];
    //2
    if (enemy.isActive) {
      [self checkForAndResolveCollisions:enemy forLayer:self.walls];
      [self checkForEnemyCollisions:enemy];
    }
    //3
    if (!enemy.isActive && enemy.characterState == kStateDead) {
      if (enemiesThatNeedDeleting == nil) {
        enemiesThatNeedDeleting = [NSMutableArray array];
      }
      [enemiesThatNeedDeleting addObject:enemy];
    }
  }

  //4
  for (PSKEnemy *enemyToDelete in enemiesThatNeedDeleting) {
    [self.enemies removeObject:enemyToDelete];
    [enemyToDelete removeFromParent];
  }

  [self setViewpointCenter:self.player.position];
}
```

Here's a blow-by-blow of the above changes:

1. You make a new array to hold those enemies that you need to delete. This is necessary because you can't remove objects from an array while iterating through its contents. Note that this array object isn't allocated unless there are

actually objects to delete. Most of the time there won't be, and there's no point to allocating a new array if you're not going to use it.

2. This is a good time to make a small optimization. Most of the enemies that are scattered around the layer are in the `isActive = NO` state. They aren't moving and forces like gravity don't affect them—remember, you bail out of their `update:` methods—so there's no reason to check for collisions with the walls or the cyclops. Here you enclose those two lines in an `if` statement to prevent those parts of the code from running unless `isActive` is `YES`. This significantly reduces the amount of CPU cycles your game needs.
3. Within the loop that iterates through the enemies, you check for your two conditions. If both conditions are met, then you add the enemy to the array that will contain all the deleted enemies, creating the array if necessary.
4. Finally, you iterate through the delete array. You first remove the dead enemies from the `enemies` array. Then you call `removeFromParent` to remove them from the `PSKLevelScene` and clean up any used memory.

There's one last thing to be done. Once an enemy starts its dying animation and gets into the removal process, you don't want it to continue to run its normal update loop—pursuing the cyclops, jumping and so forth. This isn't a zombie game! ☺

So add this test to the beginning of `update:` in each `PSKEnemy` subclass—**Crawler.m**, **Flyer.m** and **MeanCrawler.m**—right at the beginning, before the other existing code:

```
if (self.characterState == kStateDead) {
  self.desiredPosition = self.position;
  return;
}
```

This tests if the enemy's state is `kStateDead`. If it is, the code sets the `desiredPosition` to the current position. Then it returns from the method, skipping all the rest of the logic.

Build and run. You can now jump on enemies to trigger their dying animation and remove them from the level. STOMP! Enjoy your invincibility while it lasts.

Applying damage to the player

Handling damage to the `Player` is a bit more complicated. First, the cyclops has enough life to withstand five hits and you need to account for that. Second, you need to add code to update the life meter on the HUD layer to reflect the cyclops's remaining life.

What's more, multiple collisions with an enemy can happen in a fraction of a second. While this might be realistic, in practice virtually all platformer games give the player a short cool-down period during which they are impervious to further harm. You'll increase the cyclops's transparency to indicate to the user when they are invulnerable, and restore the cyclops to normal opacity when that period ends.

You'll also add a knockback, a nice feature that increases the penalty for being hit and gives the user an obvious visual cue that the cyclops has taken damage.

Finally, wouldn't it be nice if the cyclops got a little bounce after landing on top of an enemy, as in other platformer games such as *Super Mario Bros.*? You'll add that, too.

First, initialize the value of the `life` property in `initWithImageNamed:` in **Player.m**:

```
self.life = 500;
```

Now add the `tookHit:` implementation to **Player.m**:

```
- (void)tookHit:(PSKCharacter *)character {
  //1
  self.life = self.life - 100;
  if (self.life < 0) {
    self.life = 0;
  }
```

```
    //2
    [self.hud setLife:self.life / 500.0];
    //3
    if (self.life <= 0) {
      //4
      [self changeState:kStateDead];
    } else {
      //5
      self.alpha = 0.5;
      self.isActive = NO;
      //6
      if (self.position.x < character.position.x) {
        self.velocity = CGPointMake(-kKnockback / 2, kKnockback);
      } else {
        self.velocity = CGPointMake(kKnockback / 2, kKnockback);
      }
      //7
      [self performSelector:@selector(coolDownFinished)
                 withObject:nil afterDelay:kCoolDown];
    }
  }
```

Here's a step-by-step explanation of what's happening in the above method:

1. The cyclops starts with 500 life points and you reduce its life by 100 per hit, meaning the cyclops can take five hits before it dies. This coincides with the number of segments in the life bar on the `PSKHUDNode`.
2. You have a reference to the HUD, so you can set the life score directly from the `Player` class.
3. Next, you check to see if the value of the `life` property is zero.
4. If it is, the cyclops is dead and you change their state to `kStateDead`.
5. If the cyclops is still alive after the hit, then you need to do a few things. You change the cyclops's opacity to half to indicate that it is in the cool-down state. You set the `isActive` property to `NO`, temporarily stopping collision detection.
6. Here you test whether the enemy is to the right or left of the cyclops and knock the cyclops back in the direction opposite the collision side.
7. Finally, you reverse those modifications to the cyclops to resume collisions after the cool-down period ends. This line calls a method that does this after a delay.

You need to set the two new constants referenced in the above code. One is the force of the knockback and the other is the length of the cool-down period. Add the following at the top of **Player.m**:

```
#define kKnockback 100
#define kCoolDown 1.5
```

Now implement the method to end the cool-down period:

```
- (void)coolDownFinished {
```

```
    self.alpha = 1.0;
    self.isActive = YES;
}
```

This simply resets the values of the `alpha` and `isActive` properties so that the cyclops is again fully visible and collision-enabled.

Now handle `kStateDead` in `Player's changeState:` method by inserting the following right before the `default` case:

```
case kStateDead: {
    action = [SKAction sequence:@[
        self.dyingAnim,
        [SKAction waitForDuration:0.5],
        [SKAction performSelector:@selector(endGame)
                         onTarget:self]]];
    break;
}
```

This new state performs the `dyingAnim` animation that you added in chapter 7 and then the `endGame` method.

You need to add a check to `update:` so that you can no longer move the cyclops after its state changes to `kStateDead`. Add this to the beginning of `update:`, before all the existing code:

```
if (self.characterState == kStateDead) {
    self.desiredPosition = self.position;
    return;
}
```

Add an empty implementation for `endGame`. For now, this method doesn't do much of anything. You'll implement it in the next chapter.

```
- (void)endGame {
    NSLog(@"Game should end");
}
```

Build and run the game. You should see the knockback and the cyclops's transparency increase after taking a hit. Time out!

If you take enough hits, you should see the dying animation for the cyclops:

Great! The cyclops has joined us mortals. The game is almost complete!

The platformer bounce

Remember how I mentioned giving the cyclops a little bounce after it lands on top of an enemy? That's what you'll do in this section. In the process, you'll fix another issue.

Testing the game just now, did you notice that when the cyclops lands on top of an enemy, it takes damage as well? That isn't supposed to happen. If the cyclops jumps on an enemy, only the enemy should take a hit. You'll see how to ensure this when you implement the bounce.

You already added an empty implementation for the `bounce` method, so you just need to add the meat to the bone. Implement the full method in **Player.m** as follows:

```
- (void)bounce {
  //1
  self.velocity = CGPointMake(self.velocity.x, kJumpForce / 3);
  //2
  self.isActive = NO;
  //3
  [self performSelector:@selector(coolDownFinished)
             withObject:nil afterDelay:.5];
}
```

Here's what's happening:

1. You add an upward force to the cyclops that is 1/3rd the strength of a jump.
2. Sometimes when the cyclops jumps on top of an enemy, it'll be in collision for multiple loops. To avoid the `Player` getting hit when they deliver a blow, you set

the cyclops's `isActive` property to `NO`. This prevents the next loop from triggering another collision.

3. You call the method that turns `isActive` back to `YES`.

Build and run again. This time, the cyclops doesn't take a hit when damaging an enemy. And of course, the cyclops gets a little bounce after inflicting its blow:

You have traveled far, but your journey hasn't been in vain. You have learned a lot and have almost developed a complete platformer game! The next and final chapter will fill in the last few pieces of the puzzle.

> **Challenge:** What do you think it would take to inflict different amounts of damage, depending on the enemy? That is, some enemies do more damage to the player than others, and/or hits by the player affect enemy types in different ways? I've already given you a hint earlier in the chapter!

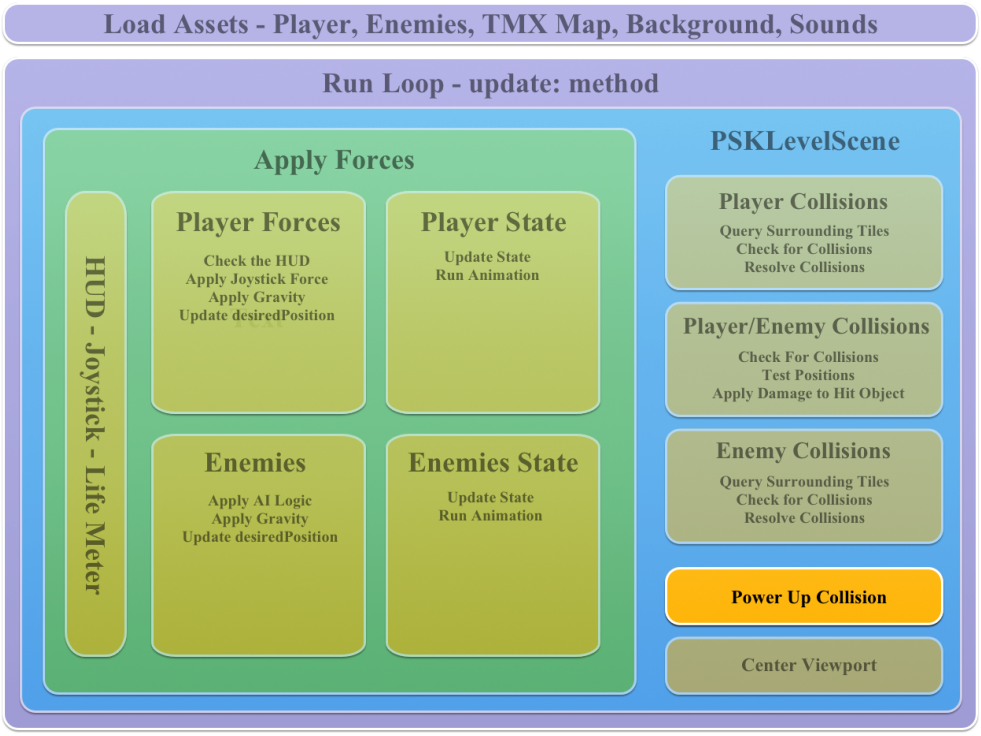
Chapter 10: Finishing Touches

Your platform game has all the fundamentals in place, but some final stitches remain:

- You need to add the power-up objects so that the cyclops has to work to earn those upgrades!
- At the moment, there is no way to beat a level! You need to enable smooth transitions from one level to the next.
- Right now, when the cyclops dies, nothing happens. Why don't you set up more encouraging win and lose scenarios?
- This game could use some smart sound effects.

Stitch it up, youngster,
so your game will pwn!

Here's all that's left to complete from the game map. You can see just how far you've come:



Adding power-ups

Recall that a few chapters back, you turned on both power-ups so that you could see their effect on the character state and animations. Since then, Super Cyclops has been wall sliding and double jumping through the levels scot-free. Now it's time to make the cyclops earn those powers.

You'll add new objects to the levels and when a collision happens between the cyclops and one of these objects, it will activate either the wall slide or the double jump. But you've got to take these abilities away from the cyclops before you can give them back!

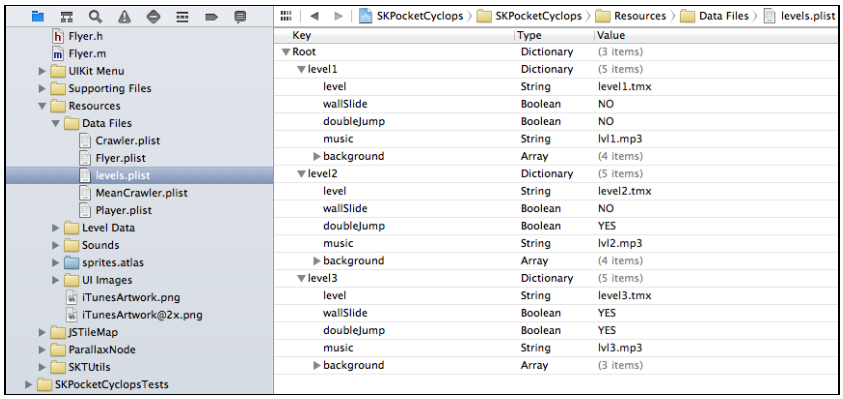
Demoting the cyclops

Go into **levels.plist** (in the **Data Files** folder) and expand the dictionaries within it.

- In the **level1** dictionary, both the **doubleJump** and **wallSlide** attributes should be `NO`
- In the **level2** dictionary, **wallSlide** should be `NO` and **doubleJump** should be `YES`.
- In **level3**, both should be `YES`.

Yep, you guessed it: the cyclops will obtain the double jump in the first level and the wall slide in the second. But at the beginning of those levels, the `Player` won't yet possess those powers.

The **levels.plist** should look like this:



The `Player` now needs two new properties to keep track of the power-ups. Add the following to **Player.h**:

```
@property (nonatomic, assign) BOOL canDoubleJump;
@property (nonatomic, assign) BOOL canWallSlide;
```

When you load each level, in the initialize method, you need to check the contents of **level.plist** for that level and set these properties.

Go to `initWithSize:level:` in **PSKLevelScene.m** and find the line that sets the cyclops's position. Immediately after that line, add the following code:

```
self.player.canDoubleJump =
                    [levelDict[@"doubleJump"] boolValue];
self.player.canWallSlide = [levelDict[@"wallSlide"] boolValue];
```

This sets the cyclops's power-up values based on the information in **levels.plist**.

Now you need to alter the `Player`'s `update:` method so that wall sliding and double jumping aren't available unless the relevant properties are set to `YES`.

First, fix the double jump. Find this line in `update:` in **Player.m**:

```
if ((self.characterState == kStateJumping || self.characterState
    == kStateFalling) && self.jumpReset) {
```

And change it to this:

```
if ((self.characterState == kStateJumping || self.characterState
    == kStateFalling) && self.jumpReset && self.canDoubleJump) {
```

This is the point in your logic where you determine if the cyclops should enter the double-jump state. Now, if the `canDoubleJump` property is `NO`, the double-jump state will never be triggered. You don't affect any of the other logic with this change.

The wall slide is next. Find this line:

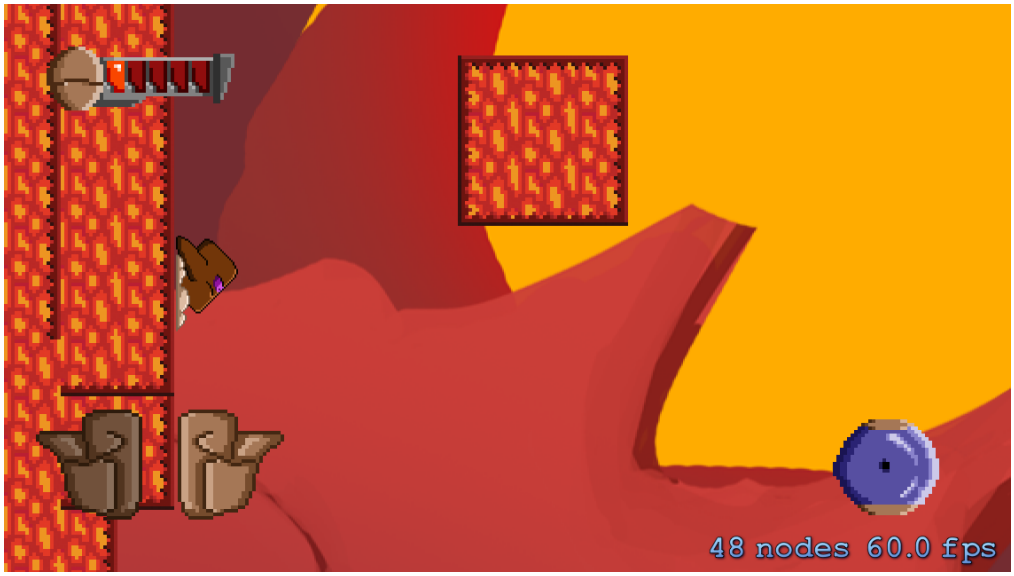
```
} else if (self.onWall && self.velocity.y < 0) {
```

And change it to this:

```
} else if (self.onWall && self.velocity.y < 0
                                        && self.canWallSlide) {
```

This follows the same logic as for the double jump: you check to see if the power-up is enabled and if not, the cyclops never enters the wall-slide state.

Build and run now. Test different levels to verify that double jumping and wall sliding are no longer available in Level 1, but still work in Level 3.



Adding power-up objects

Good job! Now for some fun—it's time to add the power-up objects that will give the cyclops the ability to wall slide or double jump.

They look a bit like a shiny red berries, the kind that might make you sick! But these are what the cyclops eats for breakfast.

Code-wise, you're going to create a new `PSKPowerUp` object that will be a subclass of `PSKGameObject`. This way, `PSKPowerUp` objects will be initialized by `initWithImageNamed:` and have access to the `loadAnimations` logic you've used before.

In *Pocket Cyclops*, your power-up berries will be single frames, but you could animate them if you wanted to—in your own hit game. ☺

Create a new file with the **iOS\Cocoa Touch\Objective-C class** template. Make it a subclass of `PSKGameObject` and name the object **PSKPowerUp**.

Open the new **PSKPowerUp.h** file. Add a new property:

```
@property (nonatomic, strong) NSString *powerUpType;
```

This is to let you query the power-up object later to determine which of the Booleans to change in the `Player` class.

That's all you need there. Return to **PSKLevelScene.m** and import the new class:

```
#import "PSKPowerUp.h"
```

Next, add a new property to the class extension:

```
@property (nonatomic, strong) NSMutableArray *powerUps;
```

This array will keep track of the power-up objects in the level. You'll have only one power-up per level in this game, and once the cyclops gets a power-up it will always have it. When you create your own game, though, it's likely you'll want more than one power-up in a single level—hence, the array.

Adding power-ups to a level and checking for collisions is similar to what you did with the enemies, but a bit less complicated.

To load the power-up objects, add a new method in **PSKLevelScene.m**:

```
- (void)loadPowerUps {
  //1
  TMXObjectGroup *powerUpsGroup = [self.map
                                   groupNamed:@"powerups"];
  //2
  self.powerUps = [NSMutableArray array];
  //3
  for (NSDictionary *powerUpDict in powerUpsGroup.objects) {
    //4
    NSString *powerUpType = powerUpDict[@"type"];
    //5
    PSKPowerUp *powerUp = [[PSKPowerUp alloc]
                           initWithImageNamed:powerUpType];
```

```
    //6
    powerUp.position = CGPointMake(
          [powerUpDict[@"x"] floatValue],
          [powerUpDict[@"y"] floatValue]);
    //7
    powerUp.powerUpType = powerUpType;
    //8
    [self.powerUps addObject:powerUp];
    powerUp.zPosition = 850;
    [self.map addChild:powerUp];
  }
}
```

This code should all look fairly familiar. Here's a step-by-step walkthrough:

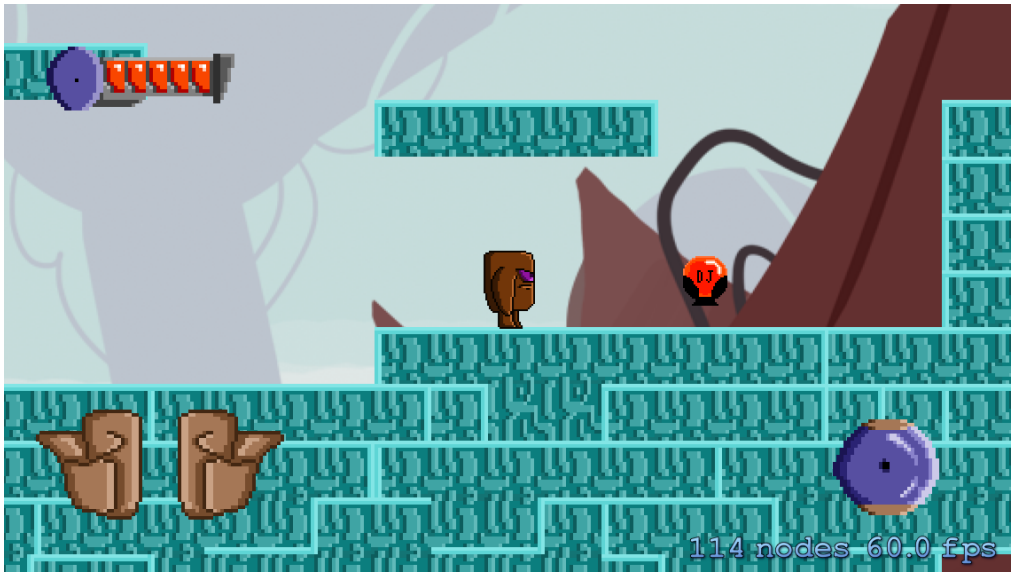
1. You load the `TMXObjectGroup` from the tile map file. There's a layer named **powerups** that contains all the power-up objects for each level.
2. You create the array that you'll use to manage the power-ups.
3. You iterate through the objects in the `powerUpsGroup` layer.
4. First, you get the name of the power-up type from the object. This is set up in Tiled.
5. The type string and the name of the file are the same. This makes it easy to retrieve the texture from the sprite atlas.
6. Next, you set the position the same way you've done before, from the information contained in the object dictionary.
7. You then set the `powerUpType` attribute using the value from step 5. When the cyclops collides with a power-up object, the game will look at this to figure out what type of power-up to bestow.
8. Finally, you add the sprite to both the map node and the array, setting the `zPosition` so that it's in front of the map and background, but behind the player and enemy nodes.

Now add the following line to `initWithSize:level:`, right after the `[self loadEnemies]` line:

```
[self loadPowerUps];
```

Build and run. Now you should be able to find the double-jump power-up berry in the first level.

See if you can find it. It's almost at the end of the level, up on a ledge:



Collisions with power-ups

As you may have predicted, touching the power-up berry doesn't do anything yet. It's time to fix that!

Add the following method to **PSKLevelScene.m**:

```
- (void)checkForPowerUpsCollisions {
  // 1
  PSKPowerUp *powerUpToRemove;
  // 2
  for (PSKPowerUp *powerUp in self.powerUps) {
    // 3
    if (CGRectIntersectsRect([self.player collisionBoundingBox],
                             powerUp.frame)) {
     // 4
      if ([powerUp.powerUpType isEqualToString:@"DoubleJump"]) {
        self.player.canDoubleJump = YES;
      } else {
        self.player.canWallSlide = YES;
      }
      // 5
      powerUpToRemove = powerUp;
      break;
    }
  }
  // 6
  if (powerUpToRemove != nil) {
    [self.powerUps removeObject:powerUpToRemove];
    [powerUpToRemove removeFromParent];
  }
}
```

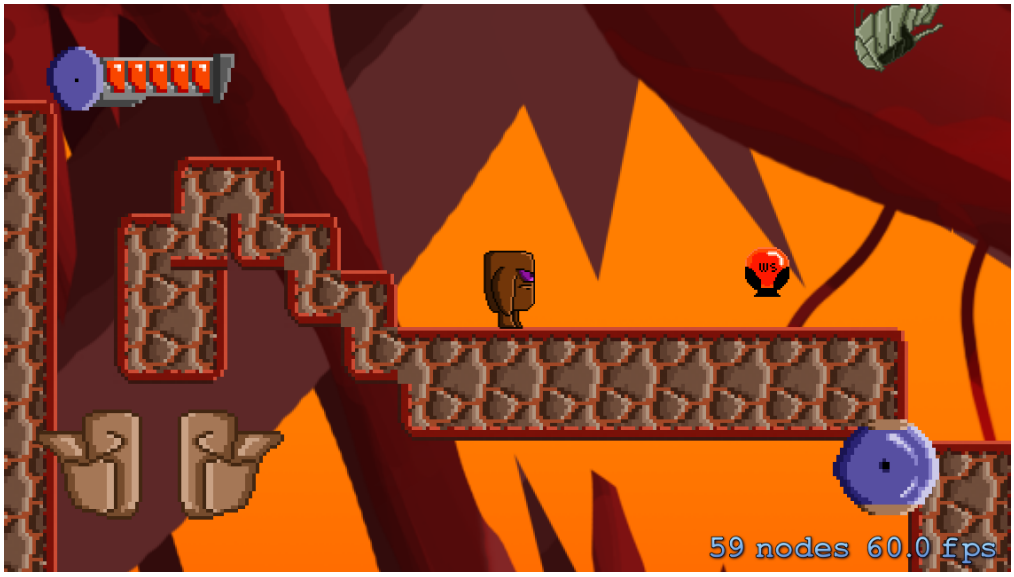
Here's what's happening:

1. As mentioned before, you can't remove an object from an array while you are iterating through that array. So if the cyclops did collect a power-up, you store it in this variable so you can delete after the loop.
2. You start iterating through all the power-up objects in the `powerUps` array.
3. You test whether the bounding box of the power-up object and the collision bounding box of the cyclops intersect. If they do, then you need to trigger the logic that will give the cyclops the power-up ability and remove the power-up from the level.
4. Then you test whether the `powerUpType` string is "DoubleJump". If it is, you set `canDoubleJump` to `YES`. If it's not, then the string must be "WallSlide". You only have two power-up types, so this logic is very simple.
5. Remember to delete this power-up and break out of the loop. Realistically speaking, the player can only pick up one power-up at a time (unless you put them all next to each other), so there's no need to look at the other ones.
6. Removal entails both removing the `PSKPowerUp` object from the map node by calling `removeFromParent` and removing it from the `powerUps` array.

The only thing left is to call this new method each frame via `update:`. Add the following line before the call to `setViewpointCenter:`:

```
  [self checkForPowerUpsCollisions];
```

Build and run. You should now be able to find and obtain the double-jump power-up in the first level, and the wall-slide power-up in the second level.

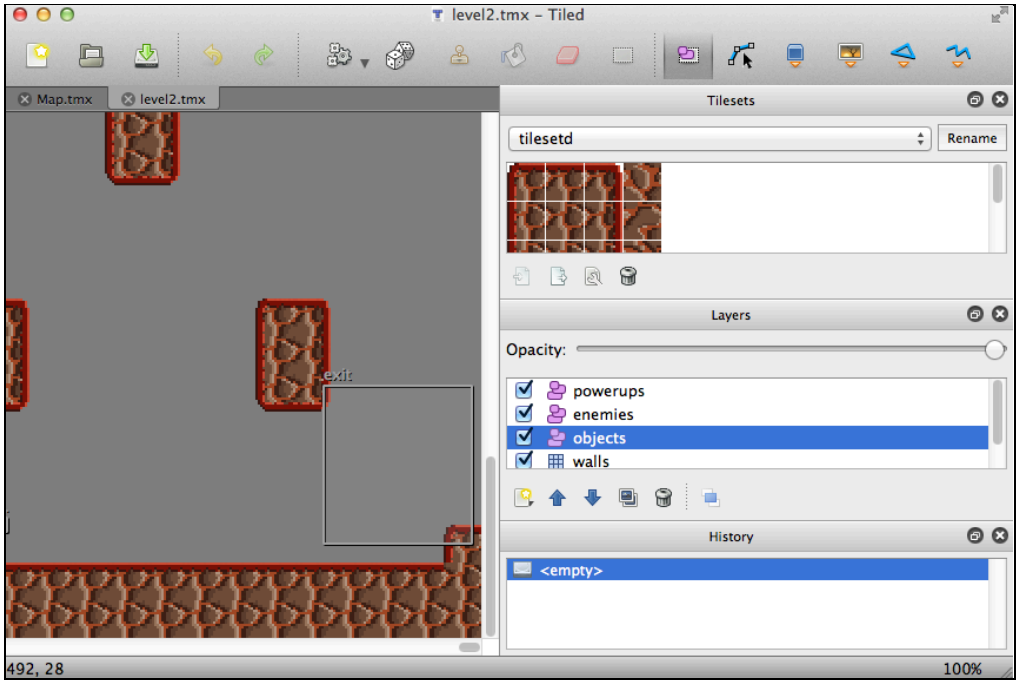


At this time, there's no visual cue when the player nabs the power-up, but that's a good thing to practice adding on your own! You could make the character blink, fire a particle system, pause the game à la *Metroid* or anything else you can imagine.

Note: If you get tired of playing through the level each time you want to test the power-up, you can simply edit the level map and either move the power-up object to the beginning of the level or add a second copy to the beginning of the level. ☺

Forward progress to victory!

Just as there's an object on the tile map indicating the starting point for the `Player`, there's also an object for the location of the exit.



There are different ways to detect this exit, but you'll use a simple method: if the cyclops is within 100 points of the center point of the exit object, then you'll trigger either the end of the level or the end of the game.

First you need a new property for the exit point. Add this to the class extension in **PSKLevelScene.m**:

```
@property (nonatomic, assign) CGPoint exitPoint;
```

Then in `initWithSize:level:`, after the line that loads the `PSKPowerUp` objects, add the code to set up the exit point:

```
NSDictionary *exit = [objects objectNamed:@"exit"];
self.exitPoint = CGPointMake(
  [exit[@"x"] floatValue] + ([exit[@"width"] floatValue] / 2),
  [exit[@"y"] floatValue] + ([exit[@"height"] floatValue] / 2));
```

This loads the information about the exit (a rectangle) into a dictionary. Then it calculates the center point for the exit and puts it into the `exitPoint` property.

To finish off a level, you'll want to pause the execution of the update loop. If you don't, the loop will call certain methods multiple times after the player has already reached the exit—like `checkForExit`, which you will implement next. This behavior can cause problems, like trying to allocate more than one instance of the next level.

Add a Boolean that will interrupt the update loop to the class extension in **PSKLevelScene.m**:

```
@property (nonatomic, assign) BOOL gameRunning;
```

Next, initialize `gameRunning` to `YES` at the end of the `init` statement:

```
self.gameRunning = YES;
```

Finally, add this line to very beginning of the update statement:

```
if (!self.gameRunning) return;
```

This essentially puts all the game's logic on hold when `self.gameRunning` is set to `NO`. Sprite Kit will still run, so music, animations and other game engine functions will continue. But the physics engine, enemy AI and player controls will stop.

Now that you have the exit point set up, you can create a method that checks for the level end conditions. Add this new method to **PSKLevelScene.m**:

```
- (void)checkForExit {
  //1
  CGFloat distanceToExit = CGPointDistance(self.player.position,
                                           self.exitPoint);
  if (distanceToExit < 100) {
    self.gameRunning = NO;
    //2
    if (self.currentLevel < 3) {
      //3
      [[self.gameNode childNodeWithName:@"parallax"] removeFromParent];
      //4
      NSUInteger nextLevel = self.currentLevel + 1;
      SKScene *nextLevelScene = [[PSKLevelScene alloc]
                         initWithSize:self.size level:nextLevel];
      //5
      [self.view presentScene:nextLevelScene
                   transition:[SKTransition
                               flipHorizontalWithDuration:0.5]];
    } else {
      //6
      SKLabelNode *win = [SKLabelNode
                          labelNodeWithFontNamed:@"Marker Felt"];
      win.text = @"You Win";
      win.position = CGPointMake(self.size.width / 2.0,
                                 self.size.height / 2.0);
```

```
        win.fontSize = 60.0;
        [self addChild:win];
        //7
        [self performSelector:@selector(restart) withObject:nil
                    afterDelay:3.0];
      }
    }
  }
```

This method is simple enough:

1. First, you determine if the cyclops is close enough to the exit to end the level. `CGPointDistance` returns a `CGFloat` value that is the distance between two points, which in this case are the cyclops and the exit.

   Also, at this stage you set the `gameRunning` Boolean to `NO`. This prevents the update loop from calling `checkForExit` and other game end methods multiple times in a row.

2. You check the current level. If it's any level before the third, then you progress to the next level.

3. To progress to the next level, you first call `removeFromParent` on the parallax node. This subclass of `SKNode` has a custom `removeFromParent` method that cleans up the memory used by the parallax node. It stops the actions from running and removes some other references to the internal resources and image files in the parallax node to prevent a memory leak.

4. Next, create a new `PSKLevelScene` object with a level number that is one larger than the current level.

> **Note:** When you initialize a second instance of `SKScene`, you must have enough memory to hold two of these objects in RAM: the currently running scene and the new scene to which you'll transition. If you are running low on memory, this can cause problems when you try to initialize the second scene.
>
> If that happens to you, you'll need to figure out a way to unload some of the texture objects before creating the new scene. Texture objects are usually what eat up the most memory in a game. One option is to include code in the parallax layer to remove some of the background tiles that are no longer onscreen. This would make for a great challenge!

5. Once you have the new scene object, you perform a transition to dismiss the current scene and switch to the new scene. You call `presentScene:transition:` to make that switch. The transition argument takes an `SKTransition` object. There are a bunch of standard transition animations built into that class, or you can create your own using Core Image.

6. If the player has reached the end of Level 3, they've won the game, so let them know! You center the label on the screen.

7. Finally, dismiss the view and go back to the level selection screen. The `restart` method takes care of that, but here you call the method after a three-second delay so that the user can see the win label first.

Now you need make `checkForExit` as the last call in `update:`. Add that code:

```
[self checkForExit];
```

Next, still in **PSKLevelScene.m**, implement the `restart` method:

```
- (void)restart {
  [[SKTAudio sharedInstance] pauseBackgroundMusic];

  [[self.gameNode childNodeWithName:@"parallax"]
                                          removeFromParent];
  [self.sceneDelegate dismissScene];
}
```

When dismissing the scene, the first thing you want to do is turn off the music.

Second, to make sure the `JLGParallaxNode` object is destroyed, you have to tell it to stop any actions it is currently running. If you don't, the parallax node object is never deallocated and you have a memory leak.

Lastly, because you don't have a direct way to talk to the `PSKGameViewController`—the controller of the Sprite Kit view—you need a way to pass a message. You are going to create a delegate protocol and make the `PSKGameViewController` act as the delegate.

Creating the delegate protocol is easy, but you have to put code in several places. First, add the following to **PSKLevelScene.h**, after the `#import`:

```
@protocol SceneDelegate <NSObject>
- (void)dismissScene;
@end
```

Then add a new property to the same file, for the delegate:

```
@property (nonatomic, weak) id <SceneDelegate> sceneDelegate;
```

Now, switch to **PSKGameViewController.m** and in the `@interface` line, add the highlighted bit to make that class conform to your new delegate protocol:

```
@interface PSKGameViewController () <SceneDelegate>
@property (nonatomic, strong) NSMutableArray *observers;
@end
```

Then, right before `[skView presentScene:scene]`, set the delegate of the `PSKLevelScene` object you create in `viewWillLayoutSubviews`:

```
scene.sceneDelegate = self;
```

Finally, implement the `dismissScene` method in that class:

```
- (void)dismissScene {
  for (id observer in self.observers) {
    [[NSNotificationCenter defaultCenter]
                                    removeObserver:observer];
  }
  self.observers = nil;

  [self.navigationController
                          popToRootViewControllerAnimated:YES];
}
```

This tells the `navigationController` to pop the current view controller off the navigation stack. Doing this will put you back onto the intro screen. You also need to remove the observers for the background notifications and set the array to `nil`, to break an ownership cycle that would otherwise keep the game view controller and the `PSKLevelScene` objects alive, causing a memory leak.

Build and run. You can now move from level to level and win the game!

There is one small thing I skipped over that you must now do before continuing. You are setting the `sceneDelegate` when you load the scene from the level picker `UIKit` view. But, what about when you create a scene and transition to it using a `SKTransition`? In that case you need to set the `sceneDelegate` on that newly instantiated scene as well.

Add this line to `checkForExit`, immediately before the call to `presentScene:transition::`

```
  nextLevelScene.sceneDelegate = self.sceneDelegate;
```

Now, you maintain the reference to the delegate so you can pop the `SKView` from the navigation controller.

Note: There might be a small delay before the label appears. That's because Sprite Kit needs to load the Marker Felt font and turn it into a texture. This only needs to happen once but it still breaks the flow of the game. For better performance, you might want to use bitmap fonts. You can create such fonts with the Glyph Designer tool:

http://www.71squared.com/en/glyphdesigner

In the event of defeat

Winning is no fun if you can't lose, right? It's time to add a lose scenario!

You want to trigger the same restart conditions when the player loses as when they win, but this time you'll display a "You Lose" message. The `PSKLevelScene` is responsible for showing this message, too, meaning that's where you need to implement the code.

However, the `Player` class is the first to know when the cyclops's life is fully depleted—you already implemented the `endGame` method in that class. So the trick is to pass a message from the `Player` class to the `PSKLevelScene` about the end of the game.

How will the `Player` class identify the object to which it needs to send the message? In Sprite Kit, every `SKNode` has a `.scene` property that identifies the current scene in which it's running. You can pass the message to that.

Replace `endGame` in **Player.m** with the following:

```
- (void)endGame {
  PSKLevelScene *levelScene = (PSKLevelScene *)self.scene;
  [levelScene loseGame];
}
```

The `Player` class doesn't know anything about `PSKLevelScene`, however. You need to import that class. Add this import at the top of **Player.m**:

```
#import "PSKLevelScene.h"
```

`PSKLevelScene` currently doesn't have a `loseGame` method. First, add the method declaration to **PSKLevelScene.h**:

```
- (void)loseGame;
```

Then implement the method in **PSKLevelScene.m**. It looks a lot like the code for the win scenario:

```
- (void)loseGame {
  self.gameRunning = NO;
```

```
    SKLabelNode *lose = [SKLabelNode
                         labelNodeWithFontNamed:@"Marker Felt"];
    lose.text = @"You Lose";
    lose.position = CGPointMake(self.size.width / 2.0,
                                self.size.height / 2.0);
    lose.fontSize = 60.0;
    [self addChild:lose];
    [self performSelector:@selector(restart) withObject:nil
                afterDelay:3.0];
}
```

Build and run. Find an enemy and let it take the cyclops's life. You should see this screen:

Gratuitous sound effects

There's only one thing left to do before your game is a wrap: add sound effects so that your animated sprites seem that much more alive. The trick here is putting the code in the right place.

You used the `SKTAudio` class to load the background music. To play sound effects, you're going to use Sprite Kit's build in audio file playing methods. Sprite Kit it pretty good about keeping the audio files cached in memory to avoid expensive disk reads when you play a sound effect.

Let's start with the sounds for the cyclops.

You can use `SKAction`'s `playSoundEffect` method to play a sound at any time. However, the first time you play a sound file, the method will have to load it from disk. If that happens in the middle of your game, you will notice a definite hiccup in game play and frame rate. For this reason, you want to pre-load the sounds for each `PSKCharacter` class.

You have one sound for jumping, another for double jumping, one for dying and one for when the cyclops bounces off the top of an enemy.

To preload the audio files, you'll create an `SKAction` property for each one. Add the following to the `@interface` block in Player.m:

```
@property (nonatomic, strong) SKAction *playJumpSound;
@property (nonatomic, strong) SKAction *playDoubleJumpSound;
@property (nonatomic, strong) SKAction *playDyingSound;
@property (nonatomic, strong) SKAction *playBounceSound;
```

Next, initialize all four in `initWithImageNamed`:

```
self.playJumpSound = [SKAction playSoundFileNamed:@"jump1.wav"
waitForCompletion:NO];
self.playDoubleJumpSound = [SKAction playSoundFileNamed:@"jump2.wav"
waitForCompletion:NO];
self.playDyingSound = [SKAction playSoundFileNamed:@"player_die.wav"
waitForCompletion:NO];
self.playBounceSound = [SKAction playSoundFileNamed:@"bounce.wav"
waitForCompletion:NO];
```

The `playSoundFileNamed:waitForCompletion:` method takes a sound file and creates an `SKAction` that you can use as you would any other `SKAction`—you can put it into a series, play it on a node and so forth. The `waitForCompletion` Boolean tells the `SKAction` whether to consider the action completed after the file has finished playing.

In this instance, you will only be running the action that plays the sound, so whether you set `waitForCompletion` to `YES` or `NO` doesn't make any difference. But, if you wanted to chain the action to play a sound file in a sequence, then you could decide whether the next action in the sequence should wait until the sound file finishes playing (`waitForCompletion = YES`) or execute immediately after the file begins playing (`waitForCompletion = NO`).

Now you can add the first three sounds to `changeState:`.

Alter the switch statement in `changeState:` in **Player.m** as follows:

```
case kStateJumping: {
  [self runAction:self.playJumpSound];
  action = self.jumpUpAnim;
  break;
}
case kStateDoubleJumping: {
  [self runAction:self.playDoubleJumpSound];
  [self setTexture:[[PSKSharedTextureCache sharedCache]
                                    textureNamed:@"Player10"]];
  [self setSize:self.texture.size];
  break;
}
case kStateDead: {
```

```
    [self runAction:self.playDyingSound];
    action = [SKAction sequence:@[
        self.dyingAnim,
        [SKAction waitForDuration:0.5],
        [SKAction performSelector:@selector(endGame)
                         onTarget:self]]];
    break;
  }
```

In the cases of the jump, double jump and dying states, you've added a line that calls `playSoundEffect:` on the `SKTAudio` singleton. `playSoundEffect:` takes a string that is the name of an audio file. It's best to use a WAV.

Because MP3 files are compressed, WAV files are much larger than MP3 files. However, in order to play an MP3 you have to decompress it. The choice between WAV and MP3 files is a choice between speed (takes resources to decompress) and file size (MP3 files are smaller). For sound effects, it makes sense to use WAV because they are short (so not very big to begin with) and you want to play them as quickly as possible. With background music, it makes sense to use MP3, because you have time when the level is loading to decompress the MP3 and they are much longer sound files.

For the fourth sound effect, add the following call to the beginning of the `bounce` method, before all the existing code:

```
  [self runAction:self.playBounceSound];
```

Now you'll do the same thing to the `PSKEnemy` subclasses. All the enemies will play a sound when they die, for a satisfyingly good riddance.

Add the new property to **PSKEnemy.h**, because all subclasses will have a dying sound effect:

```
  @property (nonatomic, strong) SKAction *playDyingSound;
```

Begin with the `Crawler` class. First, because you don't have an initialization method in this class—it uses the superclass initialization—you need to add one:

```
  - (id)initWithImageNamed:(NSString *)name
  {
    if (self = [super initWithImageNamed:name]) {
      self.playDyingSound = [SKAction
  playSoundFileNamed:@"crawler_die.wav" waitForCompletion:NO];
    }
    return self;
  }
```

Inside `changeState:`, add this line to the `kStateDead` case:

```
  [self runAction:self.playDyingSound];
```

That's it for the crawler—just one sound effect.

`MeanCrawler` will use the same dying sound effect, so add that very same initializer in the `MeanCrawler` class:

```
- (id)initWithImageNamed:(NSString *)name
{
  if (self = [super initWithImageNamed:name]) {
    self.playDyingSound = [SKAction
playSoundFileNamed:@"crawler_die.wav" waitForCompletion:NO];
  }
  return self;
}
```

And add the same line to `changeState:`, again to the `kStateDead` case:

```
[self runAction:self.playDyingSound];
```

In addition to that, add a new property for the jumping sound in the `@interface` of **MeanCrawler.m**:

```
@property (nonatomic, strong) SKAction *playJumpSound;
```

Add this to `initWithImageNamed`:

```
self.playJumpSound = [SKAction playSoundFileNamed:@"crawler_jump.wav"
waitForCompletion:NO];
```

And add this line for `kStateJumping`:

```
[self runAction:self.playJumpSound];
```

The `Flyer` class also needs sounds. It will have the dying sound plus two more. First, add properties for the two new sounds to **Flyer.m**:

```
@property (nonatomic, strong) SKAction *playAttackSound;
@property (nonatomic, strong) SKAction *playCloseEyeSound;
```

Then add the same initializer with the three sounds:

```
- (id)initWithImageNamed:(NSString *)name
{
  if (self = [super initWithImageNamed:name]) {
    self.playDyingSound = [SKAction playSoundFileNamed:@"flyerdie.wav"
waitForCompletion:NO];
    self.playAttackSound = [SKAction
playSoundFileNamed:@"flyerattack.wav" waitForCompletion:NO];
    self.playCloseEyeSound = [SKAction
playSoundFileNamed:@"flyercloseeye.wav" waitForCompletion:NO];
  }
  return self;
```

```
}
```

Add this to `kStateDead`:

```
[self runAction:self.playDyingSound];
```

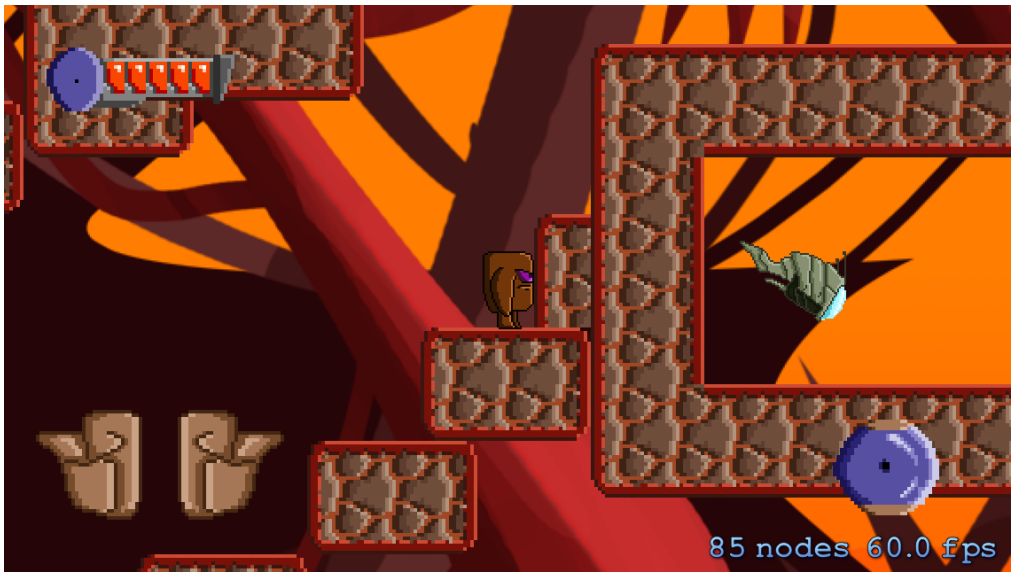
Now add this to `kStateAttacking`:

```
[self runAction:self.playAttackSound];
```

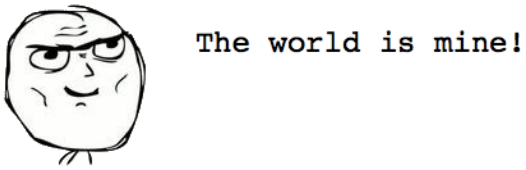
Finally, add this to `kStateHiding`:

```
[self runAction:self.playCloseEyeSound];
```

That's it! Build and run, and you should have a complete platformer game—with sound effects for all characters!



You have completed the Platformer Game Starter Kit. Congratulations! You can now do something awesome that you might not have been able to do before. ☺



Final Challenge: In this game, the cyclops has a single life. What would you need to do to give the player multiple lives? How would you let the player obtain more lives by collecting objects?

Parting thoughts

I hope you've enjoyed using this starter kit and learned a ton along the way! You now know how to make your own platformer physics engine, with tile maps, animations, enemy AI and more.

The next step is to create a platformer game of your own. The goal of this starter kit was to teach you the basic techniques for building a platformer game—now the rest is up to you.

You could add your own art, levels, enemies and power-ups. Maybe you want to make your main character shoot a weapon or maybe you want to add advanced AI. No matter what you might like to do, you have enough of a framework to figure out how to incorporate any of these elements yourself.

One piece of advice I'd offer, though, is to *start simple*. Just as you did in this starter kit, take the simplest idea you can and get it working. Then add features and polish step by step until you're happy with it.

Then, the most important part—submit your game to the App Store so others can enjoy your creation. And when you do, drop me a line—I'd love to see what you've come up with!

Thank You!

I hope you enjoyed the Platformer 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 does there.

I appreciate each and every one of you for taking the time to try out the Platformer Game Starter Kit. If you have an extra few minutes, I would really love to hear what you thought of this starter kit.

Please leave a comment on the official private forum for the Platformer Game Starter Kit at www.raywenderlich.com/forums. If you don't 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 send me an email. Although sometimes it takes me a while to respond, I do read each and every email, so please drop me a note!

Please stay in touch, and I look forward to checking out your platformer games!

Jacob Gundersen

fattjake@gmail.com

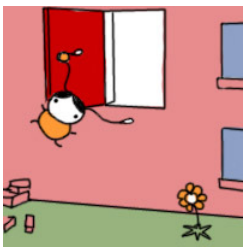
Appendix A: Interviews with Successful Platformer Game Devs

As a bonus to thank you for purchasing the Platformer Game Starter Kit, I've included an interview with two of the most successful iOS platformer game developers, Ben Hopkins and Tony McBride.

These guys have been down in the development trenches and emerged with some great games. Keep reading to learn more about them and how they did it. I'm sure you will benefit from their advice and perhaps pick up some additional tips and tricks!

About the interviewees

Ben Hopkins ("BH") is the developer behind *1-Bit Ninja*. He does coding for web, iOS and OS X. He is also a Papervision3D team. Other apps he has worked on include HoloToy, GLSL Studio, Simul80 and Oh Hi! Octopi! You can find Ben on Twitter as @kode80.



Tony McBride ("TM") is the programmer for Physmo's *Mos Speedrun*. He started programming more than 25 years ago on the BBC Micro before moving on to the Atari ST and PC/consoles. He met Nick (@physmo) at university and they started developing and selling games together. They've both had full-time jobs in the games industry and other programming positions and now build games for fun. You can find Tony on Twitter as @physmotone

The interview

1) What's the most important aspect of building a platformer for a mobile device? What's the greatest challenge? Did you have any epiphanies when solving a problem on the platform?

BH: The most fundamentally important aspects of a platform game on any system are the controls. On mobile devices, many of which have no physical buttons, this is especially true.

Platform games designed for mobile devices that employ virtual, onscreen controls tend to tailor their level design around the limited precision of the control scheme. Simple level layouts, forgiving jumps and gameplay mechanics are all trends I've noticed in these types of platform games.

On the other hand, the few platform games designed for mobile devices that feature increased complexity and/or difficulty often use a control scheme other than "traditional" onscreen virtual buttons.

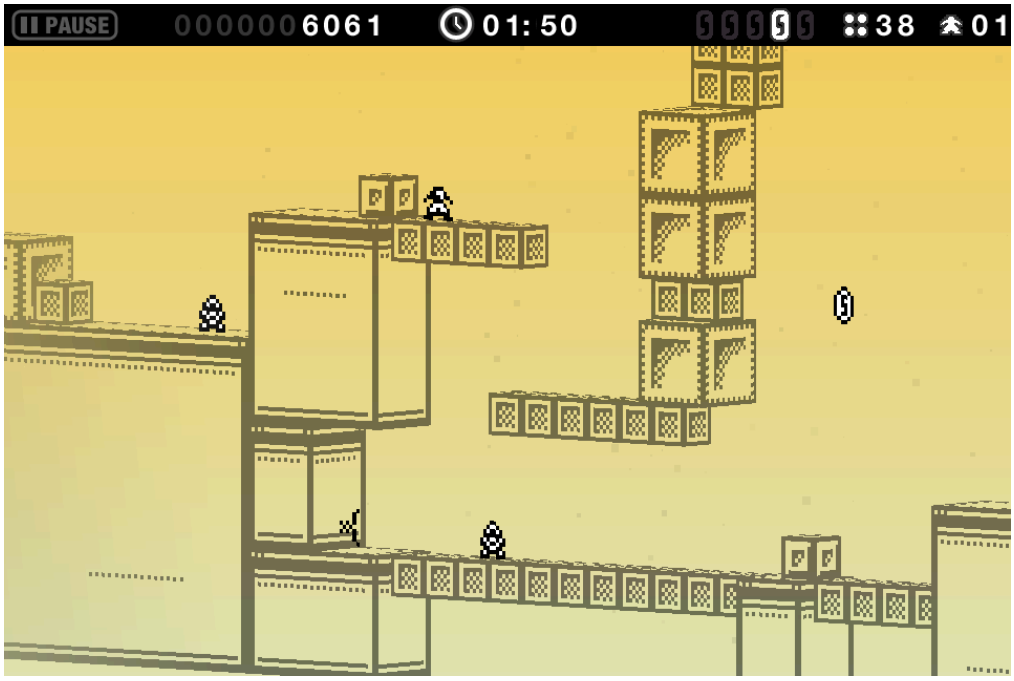
Wanting to create a tough "old school" platform game in *1-Bit Ninja*, I took the latter approach. Look at a device's strengths and weaknesses and design with those in mind; constraints offer a great opportunity to experiment with new ideas.

TM: I think the most important aspects of building a platformer for a mobile device are the touch controls. We worked hard on Mos Speedrun to get the buttons in the best place and to respond quickly. We also allowed the player to move the buttons around on the iPad to get the most comfortable placement.

Another important aspect was the game camera—we tried hard to make it look ahead of the player so they could see as much of the level as possible.

2) What is the most important thing to keep in mind to make a platformer game fun?

TM: When we designed Mos Speedrun, we wanted the player to be able to make the "perfect" run through the level. To do this, we placed the enemies and jumps in specific places so the player wouldn't have to stop at any point in the level. We also added replay value by having various tasks in each level—collect all the coins, find the hidden skull, etc.



BH: Variety and progression in level design are very important—trying to make interactive elements such as enemies unique from one another so that each one presents a different advantage or obstacle to the player, and then combining those elements in interesting ways through level design. Constantly going back to previous levels to ensure that repeating layouts are kept to a minimum is important and can help with the overall feel from one level to the next.

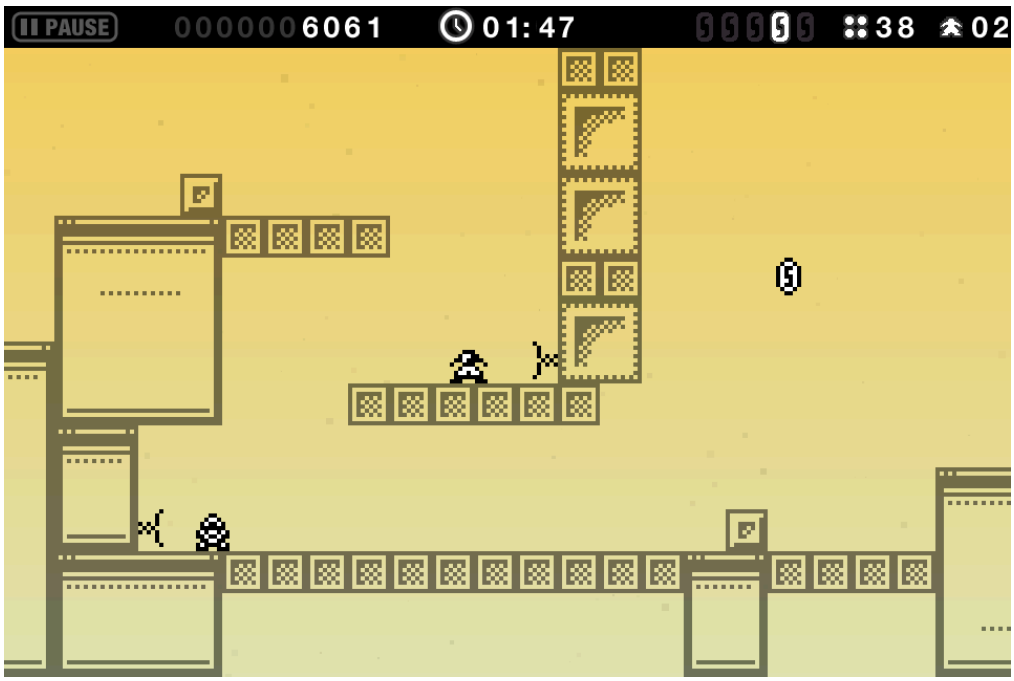
Above all, experiment. I always have a number of "test" levels that are simply one or two rooms I use for trying out ideas that can then be fleshed out and included in full levels.

3) How do you handle physics in your game? An open source physics engine? Custom-built? Any tricks that you came up with to solve specific problems in your game?

TM: The physics in *1-Bit Ninja* are custom but very simple—just velocity and gravity. Collision detection with the level is done through a simple 2D lookup into the level's tilemap data. All movements are hardcoded in each element's update function, the majority just using velocity but some using sine curves, and only check for level/entity collisions if needed by that specific entity.

BH: We used pixel collision in our game rather than real physics. We wanted the collision detection to be very accurate so the player wouldn't feel cheated if they were killed by an enemy that they didn't touch.

4) Do you have any tips on good level design? Did you encounter any constraints designing levels on a mobile device?



BH: The design constraints in *1-Bit Ninja* naturally came from the control scheme. As the player has no "back" button, extra care had to be taken in avoiding areas they could get stuck in. There are several elements that can reverse the player's direction, such as springs and moving platforms, which enabled some interesting design puzzles. But their use also required extra planning and play testing.

If a spring reversed the player's direction, for example, then there needs to be a corresponding spring somewhere to put them back on track. Added complications came from enemies. Since the player can gain extra height by bouncing off of an enemy and enemies can move, there is the chance that an enemy may move to an unexpected area, allowing the player to springboard off of it and reach an unintended point while reversed.

During the initial planning stage, I created a spreadsheet that listed all the levels in the game. I then allocated the various interactive elements and enemy types to each level, giving an even distribution of unique encounters across the whole game. When it came time to design each level, I would check the spreadsheet and only use the elements I'd previously allocated.

5) What's your approach to enemy AI behaviors?

TM: We wanted the AI behaviors to be predictable so the players didn't get annoyed. Because it's a speedrun game, the movement couldn't be random and the enemies had to be in the same positions each time you played the game.

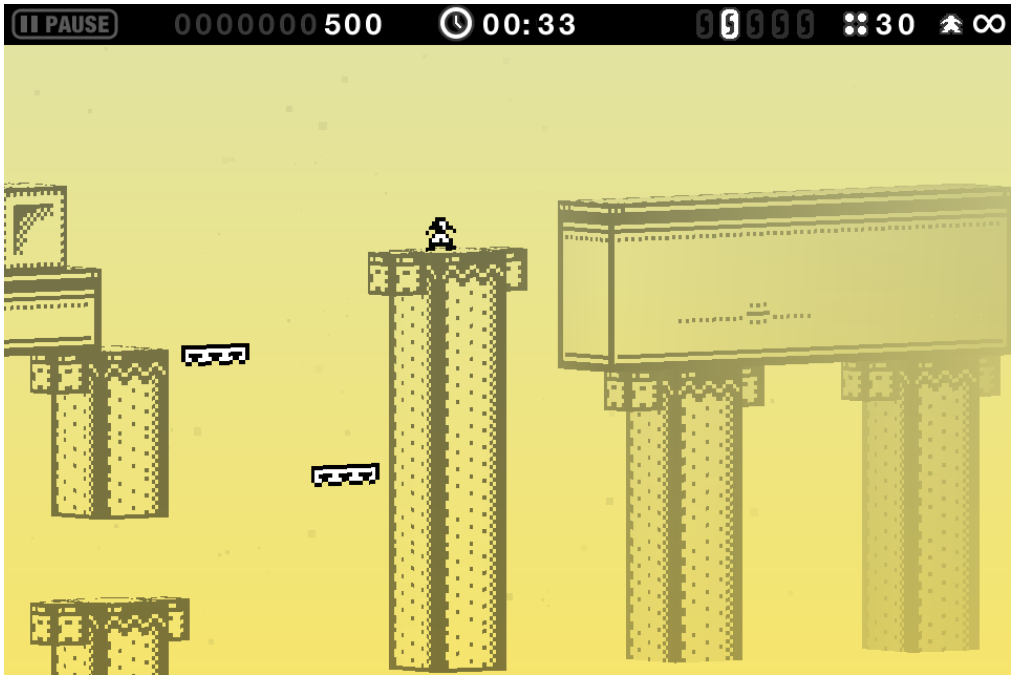
BH: I try and think of enemies as moving, interactive extensions of the level itself. At the most basic level, I group enemies into horizontal and vertical obstructions. Arrows always shoot from right to left, vases shoot shurikens up which then fall

back down. These two types alone cause the player to respond in a relatively predictable manner and thus can be used to nudge the player in a certain direction.

An important aspect that I try to maintain with all enemies is a warning phase; this is a type of visual cue of not only an impending action but also the timing of the impending action. Before an arrow is shot it moves in and out once as if aiming. Before a vase shoots a shuriken it pauses, does an up-down dance and then fires. The timings on these warning phases are always the same and I believe after a while the player picks up on these timings instinctually, knowing when it is and isn't safe.

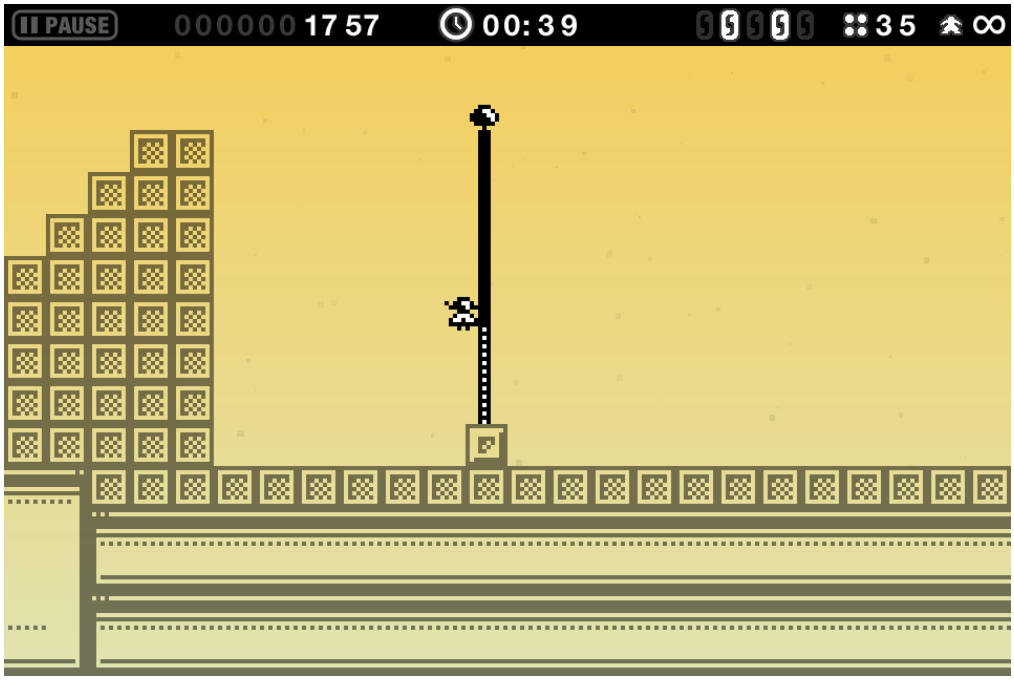
6) Any favorite features that ended up being cut from your game for whatever reason—in order to ship, performance problems, etc.?

TM: Not really. We would have liked to have more levels but it's quite time consuming to create new graphics and enemies for each level type.



BH: In my initial prototype for *1-Bit Ninja*, I had branching paths within the levels. Whole parts of the level would rotate dramatically, allowing the player to unlock and explore different areas within that one level. As I moved on from the initial prototype into more fleshed out, playable levels, it became clear that the degree of complexity both in terms of design and technology would require a large amount of time to sustain. So I decided to focus on the simpler linear level format you see in the game today. This is without a doubt the feature I am most excited to revisit in the sequel.

7) What's your favorite platformer of all time, on any hardware?



BH: It has to be *Super Mario World* on the SNES—that for me is platform game perfection. There are quite a few games that immediately come to mind with greater complexity, better graphics, etc. but when it comes down to it, *Super Mario World* is the game that holds the strongest memories for me. The variety of enemies, the secrets, the switch palaces, ghost houses, Yoshi, Star World, all tied together cohesively by the unified world map… that game was amazing! Incidentally, I've been playing a lot of *New Super Mario Bros. U* on the Wii U and I think this is the closest Nintendo has come to recreating the magic of *Super Mario World*.

TM: I think it would have to be *Super Mario World* on the SNES. :)

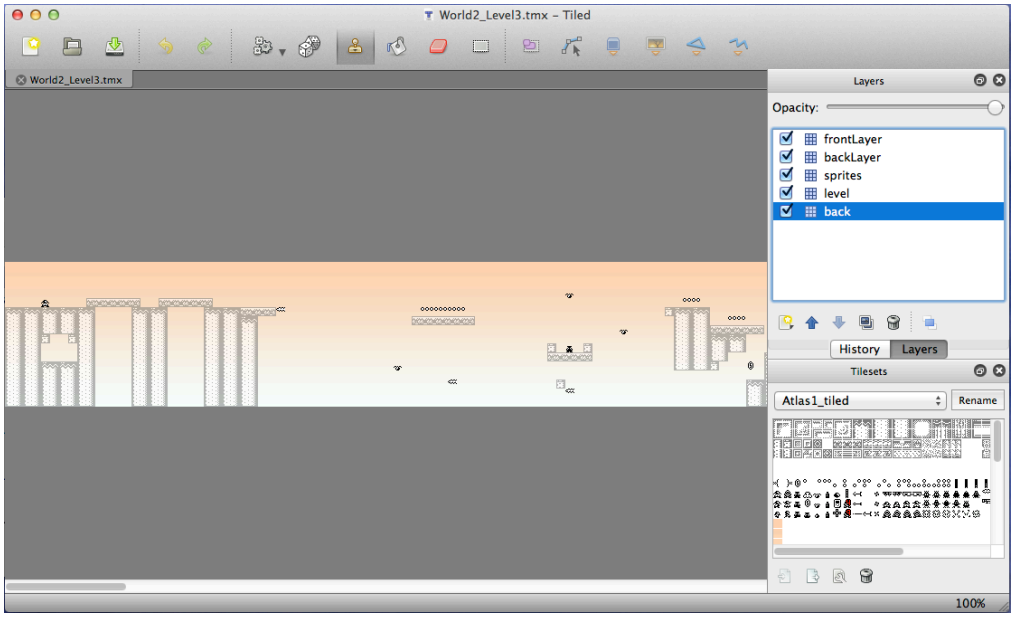
8) Are there any other tips you'd like to share with aspiring platformer game designers?

BH: Play test, play test, play test! It can be tempting to simply add things based on known constraints (the player can move this fast, jump this high, jump this far) and call it a day; my favorite levels, however, are the ones I spent the most time playing from start to finish over and over again, tweaking elements as I go. My least favorite levels are the ones I took both literal and figurative shortcuts on.

I think of the flow through a level much like a skater trying to hit that perfect line. Moving interactive elements in a level, such as enemies, platforms and power-ups, all have a huge impact on how the player progresses and, being time sensitive, these things directly affect the "flow". Starting a level at some point other than the beginning can result in an offset to the time that a player hits certain obstacles, which increases the chance of a notable break in the "flow" when played from the beginning. The "flow" is, of course, invisible to the player—that is, until they hit

their stride and things appear to fall into place naturally, giving a real sense of excitement and achievement.

9) How did you create levels for your game? Did you write your own level editor? If so, what was programmed in and what was it like?



BH: I designed the levels in *1-Bit Ninja* using the free open source editor Tiled (http://www.mapeditor.org). I broke the levels up into three layers: foreground, solid and background. The solid layer is the actual level that the player interacts with, the foreground layer appears in front of the solid layer on the 3D plane and the background layer appears behind.

I wrote a conversion tool that reads in the Tiled XML files and outputs a custom binary format for use in the game. The game loads the level's map data into memory at startup and as the player progresses through the level, the game generates the 3D mesh dynamically using a basic cube algorithm that creates appropriately-sized and positioned cubes based on the tile's ID and layer in the map file. Unneeded faces such as shared sides are also removed at the mesh generation stage to optimize the geometry. Tiles can define different side textures for use by the 3D mesh generator and these I simply hardcoded in a header file.

TM: We wrote our own level editor to create the game levels. We created it alongside the game so you could start playing the level at any point while you were editing without saving and starting the game—this helped us to tweak the levels to get the perfect speedrun. We did all of our development on the PC in C++ and ported to iOS when we needed to do a build.

10) What's the basic algorithm you used for your jumping behavior?

TM: It's a pretty simple bit of code—we just have a gravity vector and an (x,y) velocity vector, which we add to the player's position each time. We have some friction on the ground, acceleration and a maximum speed for the player, too. We also added two different types of jumps—a small one when the player presses the button quickly and a large one when the players holds the button down for longer.

BH: The jumping behavior in *1-Bit Ninja* is as simple as you can get. While the player is pressing the jump button, the game sets the player's vertical velocity to a certain value. When the player releases the jump button, the game no longer sets the player's vertical velocity and gravity takes care of the rest.

As a result of this setup, the height of the jump is dependent on how long the player holds the jump button. Bouncing off of enemies also works in the same way. When the game detects a collision between the player's feet and the enemy's head, it adds a certain value to the player's vertical velocity. If the player has the jump button pressed when this collision occurs, the game adds a greater value, allowing the player to springboard off of enemies.

That's it!

I'd like to give a huge thanks to Ben and Tony for taking the time to share their insights and experiences with us!

I hope these interviews have been interesting and helpful, and that your future platformer game is a great hit as well!