# PLATFORMER GAME STARTER KIT

By Jake Gundersen

# Platformer Game Starter Kit

Jake Gundersen

# Table of Contents

# Dedication

To my boys, John and Eli.

# Introduction

One of the most familiar types of games is platformer games – especially if you grew up with a Nintendo, Super Nintendo, or Sega Genesis. Many of us have nostalgic memories of games like Mario, Sonic, Bionic Commando, and more!

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

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 game character. This feeling is strongest in the most well-designed games. The platformers that lacked this feeling of connection with the game avatar didn't enjoy the same level of success.
2. **Complex physics engines**. The second feature of the first generation of 8-bit games that made them so fun were the **complex physics engines** behind them. In earlier games, like *Pitfall*, for example, controlling the character was difficult. The character's movements were more static (every jump was the same height, no reversing direction in mid-jump, etc.) and as a result, the game was less enjoyable.
3. **Fluid connection between controller and movement**. The third essential tenet of game feel is a fluid connection between controller and 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. Games where it's too easy to press a wrong button, or hard to press the right one, can feel frustrating and disjointed. 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 Platform Game Starter Kit is to teach you the fundamental techniques involved in making platform games, with these three goals 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**. The Platform Game Starter Kit includes full source code for a complete side-scrolling platformer game for the iPhone, complete with tile maps, custom physics behavior, enemy AI, and more!
2. **Epic-length tutorials**. With the Platform Game Starter Kit, not only do you get full source code that you can use in your own games – but you also get 9 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 start creating your own platform game!

# Who this starter kit is for

The perfect audience for this Starter Kit is someone who knows the basics of making simple games for iOS using the Cocos2D library, but is a bit uncertain about how to make your own platformer game. By the time you have finished reading this Starter Kit, you will have the tools you need to get started!

If you are new to making games on iOS or the Cocos2D library, I recommend you go through our free How To Make A Simple iPhone Game with Cocos2D 2.X tutorial series available here:

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

Then, a good next step is to go through the Space Game Starter Kit or Beat 'Em Up Game Starter Kit if you have it. Between the introductory tutorial and one of those starter kits, you should have plenty of knowledge to follow along with this tutorial.

# Prerequisites

To use this starter kit, you need to have a Mac with Xcode installed. You also need an iPhone or iPod touch to test your code on, since the controls for the game work much better when you can actually touch the screen with two hands.

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

# How to use this starter kit

You have several ways you can make use of the Platformer Game Starter Kit.

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

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

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

Note you don't necessarily have to do each tutorial – if you already know how to do everything in Chapter 1, you can skip straight to Chapter 2 for example. The Platform Game Starter Kit includes a version of the project where it leaves off after each previous chapter that you can pick up from.

# 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! He lived deep underground with all manner of fantastic creatures but, he was kidnapped by evil robots and now he must battle his way back down to the underworld he came from.

Here's how you will be building this up, chapter by chapter:

## Chapter 1

You are here! In this chapter, you'll learn how to load game level assets, including a TMX tile map and parallax background assets. This will set the stage for the animated objects that you'll add later on.

## Chapter 2

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

## Chapter 3

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

## Chapter 4

Chapter 4 will introduce the rules that move Cyclops around. You'll enable the player to jump and to move left and right. You'll explore the physics rules that control the speed and momentum of the player.

## Chapter 5

In Chapter 5, you will add 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 Cyclops can jump, is injured or is dying.

The Cyclops and its enemies will all have a number of different possible states, including but not limited to walking, standing, jumping, double-jumping, wall-sliding, attacking and dying. Knowing the state of Cyclops will determine how fast he can move, what he should be doing and what he should look like.

## Chapter 6

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

Once your Cyclops is animated, your game will start to feel more like a real boy! Err… I mean game. ☺

## Chapter 7

In this chapter, you'll incorporate enemies into your game. The enemies will follow a lot of the same rules (and use the same code) for physics and collision detection as used for Cyclops.

You'll add patterns of behavior, from simple to complex, to create enemies that are both intelligent and challenging.

## Chapter 8

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

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

## Chapter 9

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

## And when you're done...

Once you've made your way through all of these chapters, not only will you be capable of creating a platformer game – 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, tiled maps and so forth that you will learn in this book 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'll definitely want to have them on hand as you go through the Starter Kit.

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

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

This is a great place to ask any questions you have about the book or making 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, Nature of Code.
- **sonicretro.org**: For your great resources on the implementation of Sonic the Hedgehog.
- **Rod Struogo and Ray Wenderlich:** For your work on Learning Cocos2D.
- **Ricardo Quesada and all the contributors to Cocos2D:** For developing such an amazing framework!
- 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

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

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

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

# About the artist

**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.

For more of his work please go to joe-mccormick.com.

# Chapter 1: Getting Started

This starter kit contains all the resources and instructions you need to a 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 files 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 ChpX End**

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 if you make a mistake, you can revert back to the correct state by using the project from the end of the previous chapter.

**3. Project Cyclops Final**

This is the final project 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 (either on 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 Starter Project instead. Build and run, and you'll see the main menu is funcitonal but when you start a level all you get is a blank screen with a heads-up display on it:

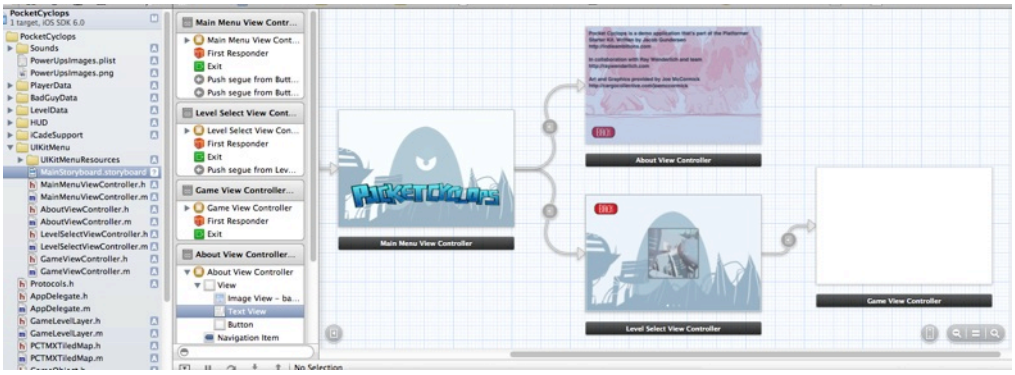
Implementing this screen is where you'll be spending your time on in this tutorial. The rest of the project is just a framework to save you time and get you started.

In the rest of this section, you'll go over each part of the starter project. Each section includes a small challenge where you will try to slightly tweak each part. I highly recommend you follow each challenge so you feel comfortable with each part of the code in this project.

## UIKit Menu Interface

The goal of this starter kit is to show you how to code the platform game gameplay, so to keep the focus on that I have included the code for the game's main menu for you.

Open **UIKitMenu\MainStoryboard.storyboard** to see the menu system – you'll see there are four view controllers inside. The menu system is created with UIKit instead of Cocos2D for a good reason. For designing and managing a menu system, it's much easier to use than Cocos2D!




This tutorial doesn't cover how to set up integration between Cocos2D and UIKit, but you should be able to grasp what's happening by peeking inside the classes. If that isn't enough, you can refer to the following post on how to integrate Cocos2D and UIKit (though this tutorial is a little older, the concepts remain the same):

http://www.raywenderlich.com/4817/how-to-integrate-cocos2d-and-uikit

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

## HUDLayer Class

The starter proejct also contains a class called `HUDLayer` that implents the Heads-Up Display (HUD). The HUD also has support iCade, a popular third party joystick for iOS devices. You can find these in the **HUD** and **iCadeSupport** groups.

I recommend taking a quick peek at HUDLayer.m so you can get an idea of what it's doing. Don't worry if you don't understand every line – just look to see how it's placing the buttons on the screen and watching for them to be tapped in `ccTouchesBegan:`.

If you're curious how to set this up from scratch, first see this tutorial on how to create a HUD:

http://www.raywenderlich.com/4666/how-to-create-a-hud-layer-with-cocos2d

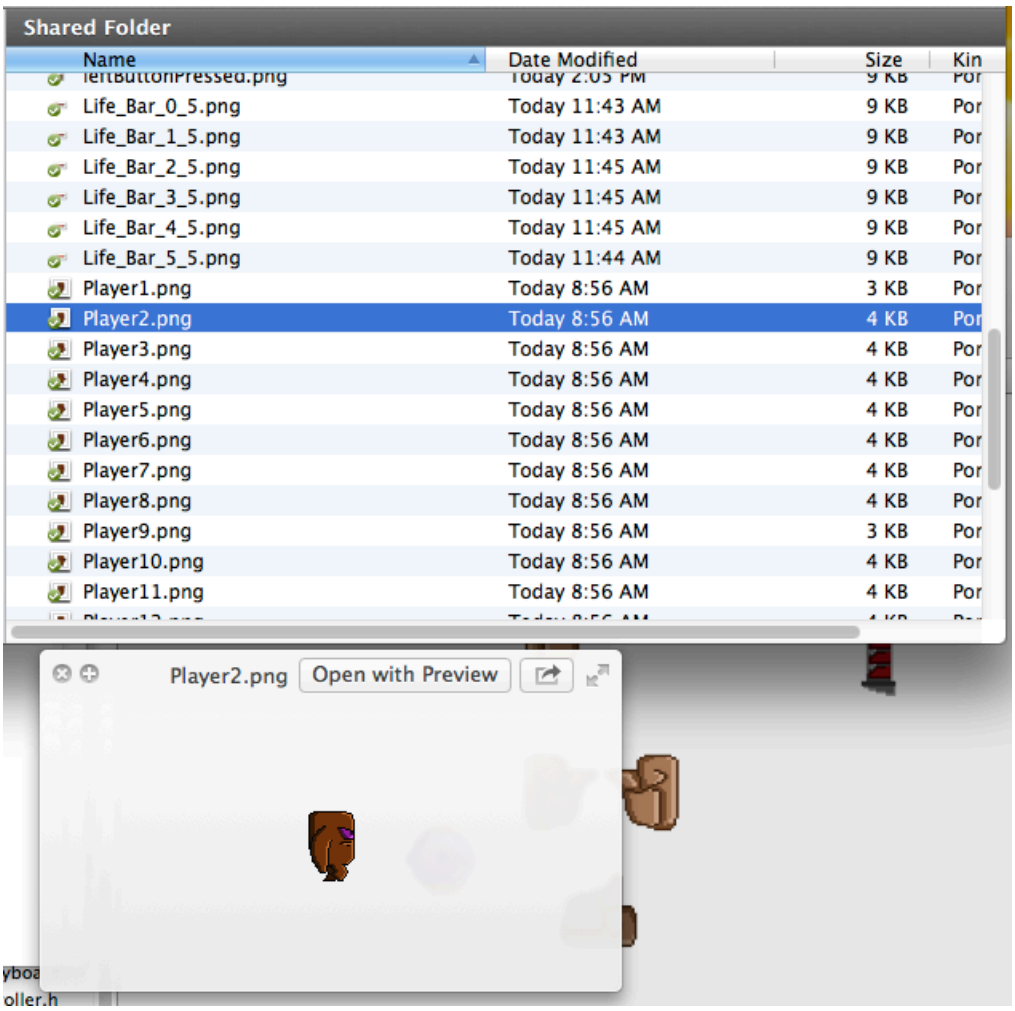
Then take a look at this tutorial on how to integrate iCade support:

http://www.raywenderlich.com/8618/adding-icade-support-to-your-game

**Challenge**: Try moving the health bar to the upper right of the screen instead of the upper left.

## Graphic Assets

You can find the raw sprites used for this game in the **Raw** folder. Take a look through at the cool artwork!




However, the sprites in this game are not used as individual files like this – instead, I have packed them into a number of sprite sheets using using Texture Packer. As you'll learn later in this Starter Kit, this makes the game much more efficient.

You can find the Texture Packer files inside the **Raw** folder, and you can find the premade sprite sheets inside the **PlayerData** and **BadGuyData** groups in the Starter Project. You can use the .tps files (Texture Packer files) to rebuild the sprite sheets, but you'll need to change the directories for the output files (plist/png) first. I'll cover how to import these into the project when the time comes.

**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

In this tutorial, you'll be importing frames from the sprite sheets into animation objects. Though I will be covering the code to do that, I'll be brief about it. If you want more detail, here's an excellent tutorial on how it works:

http://www.raywenderlich.com/1271/how-to-use-animations-and-sprite-sheets-in-cocos2d

**Challenge:** Make a slight tweak to **Raw\Player1.png** – maybe change the color or draw a cape. Then use TexturePacker to rebuild the sprite sheet (PlayerImages.png/plist). Open it in the Starter Project to make sure your update came through!

## Level Data

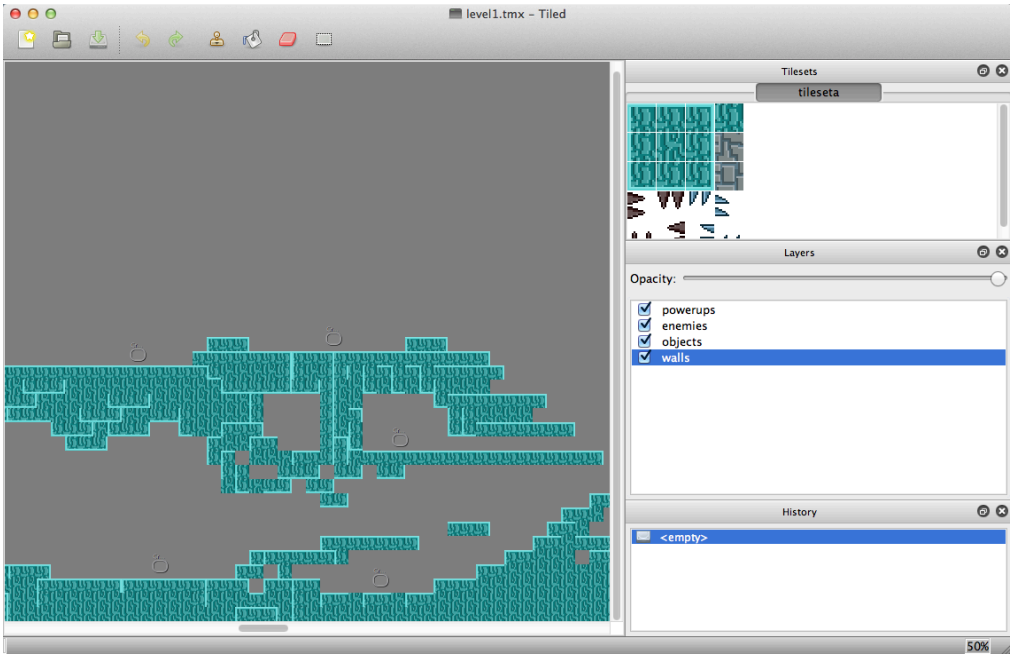
The game also includes several pre-built levels. These levels are `TMXTiledMap` files created using the popular Tiled map editor, and you can find them in the **LevelData** folder (level1.tmx, level2.tmx, and level3.tmx).

A `TMXTiledMap` file is a tile-based map system that relies on a grid of tiles and a tile set image to create a map. It also includes data about the positions of key placemarks (like the starting position of the player in the 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.

In order to follow this tutorial, you need to be familiar with the basics of working with `TMXTiledMap`**s** in Cocos2D. Fortunately, there's a tutorial for that! Check it out:

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



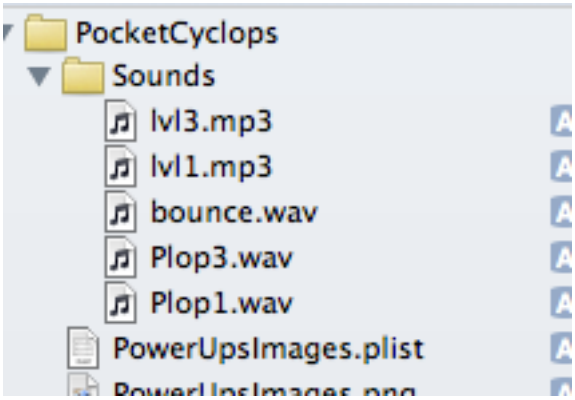

Feel free to play around with changing the level layout as you build the game!

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

## Music and Audio

I've provided a number of audio effects, sounds and music files for this game, which you can find in the **Sounds** group. Most of these were purchased at http://audiojungle.com. This is an affordable place to get high-quality music and sound assets for your game projects.

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




The provided starter project is a great template for games that have a `UIStoryboard` interface that segues into a Cocos2D game scene. You can reuse it in your own games rather than creating it from scratch every time. The `GameViewController` class has all the assets required to set up a `UIViewController` that contains a Cocos2D game view.

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

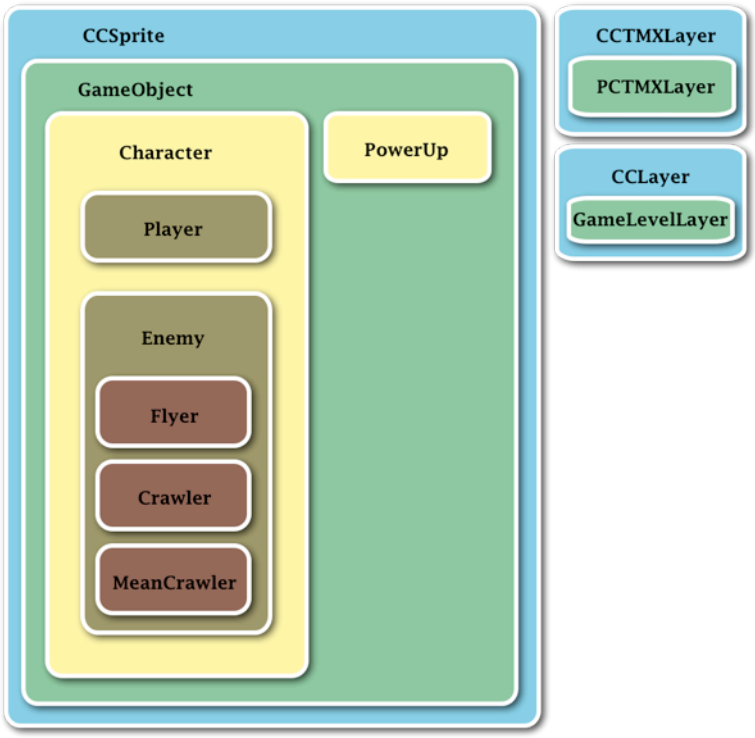
# Class hierarchy

Before you dive into coding, we should discuss the class hierarchy you will be using in this game.

As a programmer, an important part of game design is carefully considering 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 will take you to make your great platformer game.

Here's a diagram showing how all the classes of this platformer engine will fit together:




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

**1.** `CCLayer->GameLevelLayer`

a. Game Scene – This object contains the parent object that owns, either directly or indirectly, all the game objects. It also houses the primary methods for the physics engine, the game initialization/loading, and the run loop.

b. It has an `initWithLevel` method. This method takes in an integer that tells the layer what level object to load. For each level it loads a PLIST file containing a dictionary that has data about what other assets (sound, TMX Map, background images) to load.

c. It loads all these assets into memory and sets up the level.

d. The primary methods for the physics engine-based collision detection are found in this object.

e. It keeps the camera centered on Cyclops.

**2.** `CCTMXTiledMap->PCTMXTiledMap`

a. In addition to the standard `CCTMXTiledMap` class methods, this subclass adds methods that assist in identifying the tiles around a given location.

b. It also has convenience methods for testing whether objects can collide with a given tile (a wall layer tile).

**3.** `CCSprite->GameObject`

a. The `GameObject` is the parent class for all game sprites. It contains methods to load animations from sprite sheets (something potentially common to all game sprites).

**4.** `CCSprite->GameObject->PowerUp`

a. A `PowerUp` is a game object that enhances Cyclops' abilities.

**5.** `CCSprite->GameObject->Character`

a. A `Character` object will move around within the layer and collide with walls, floors, and other characters. It contains a number of attributes that are required for the physics engine, such as `desiredPosition` and `velocity`.

b. Each `Character` instance will have a bounding box method, called `collisionBoundingBox`, that returns a `CGRect` that will be used for collision detection. The `Character` superclass contains a default implementation (using the size of the sprite texture). Many subclasses of `Character` will override this method in order to allow for more precise control of the collision detection.

c. Each character has a state machine that controls the animations and logic for that entity. The `Character` 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 it receives when colliding with another character. There's an empty superclass implementation in the `Character` class.

**6.** `CCSprite->GameObject->Character->Player`

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

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

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

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

**7.** `CCSprite->GameObject->Character->Enemy`

a. In *Pocket Cyclops*, all enemies die after a single hit. This common `tookHit` implementation is found in the `Enemy` class.

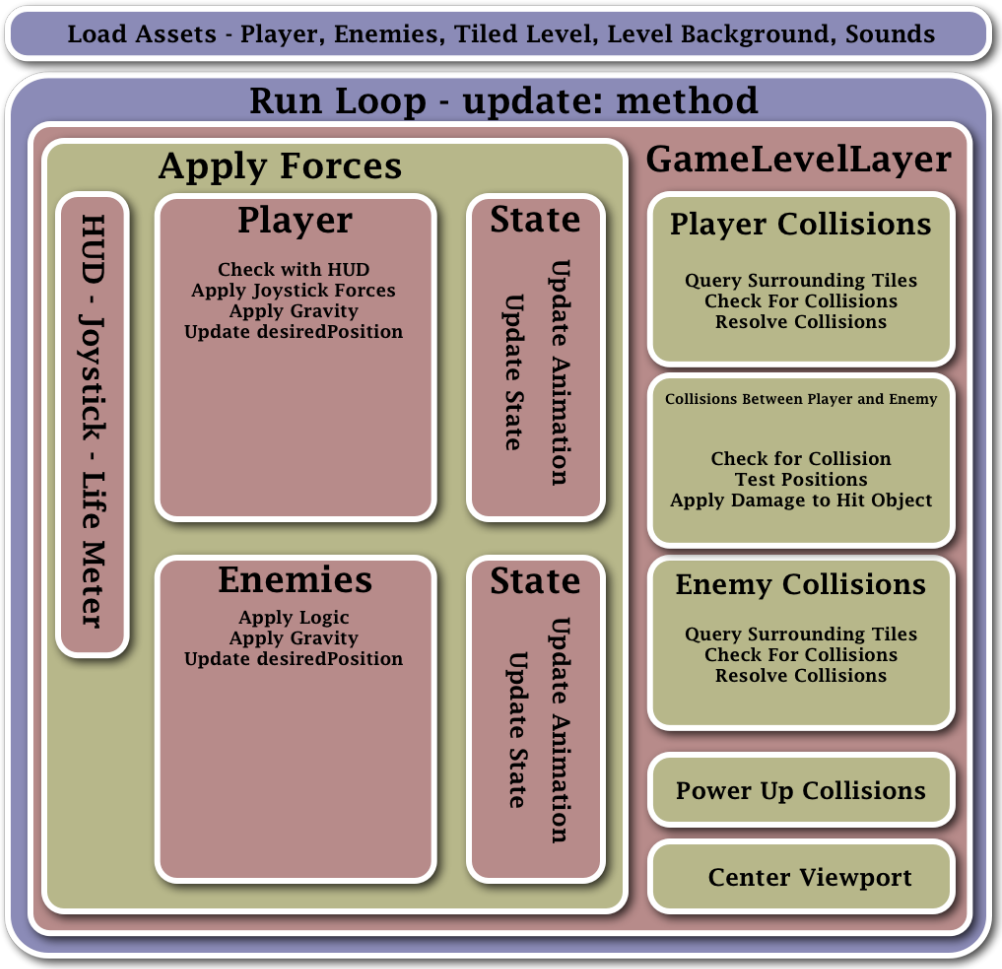
b. Some enemies need to have information about Cyclops' position or information about tiles in the tile map. The `Enemy` class contains these attributes.

**8.** `CCSprite->GameObject->Character->Enemy->`Individual Enemy Subclass

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

This has been a high-level overview of the individual subclasses of 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.

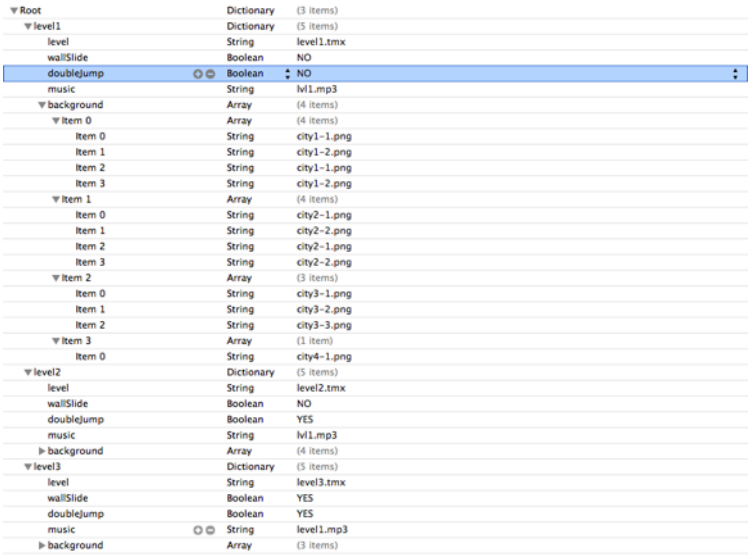



# Let's code!

Enough talk – it's finally time to code!

Don't worry, you'll start nice and easy to get your feet wet. You'll add some code to load the background and music.

I created a file called **LevelData\levels.plist** that contains information about each level – open it up to take a look:




This file contains information about the parallax background, the music for the level, the `TMXTilemap` and the state of the player's upgrades (**WallSlide** and **DoubleJump**) at the beginning of the level. You will know the player's upgrade state, because the levels have been designed so that the player has to find and use the power-up to complete the level.

Next, open **GameLevelLayer.m** and take a look at the initWithLevel method:

```
-(id) initWithLevel:(int)level
{
    if( (self=[super init])) {

    }
    return self;
}
```

You'll see it's pretty empty right now – this is what you'll be modifying! ☺

`initWithLevel` provides an integer that tells the initializing layer object which level is being loaded. You'll use that to load levels.plist into an `NSDictionary` object. Then you use the information provided in that `NSDictionary` to initialize the level properly.

Also, you need to keep a reference to the level currently being played, because you'll need that information later (like when it's time to move to the next level).

OK, so let's get to work! First add an instance variable to keep track of the loaded level by adding the following code in **GameLevelLayer.m** to the class extension below the imports:

```
@interface GameLevelLayer() {
    int currentLevel;
```

```
}

@end
```

Next load the music from the level data. For this you need to import the `SimpleAudioEngine` class.

Add the following line at the top of the file, next to the other `#import` statements (before/after, either way):

```
#import "SimpleAudioEngine.h"
```

Now modify `initWithLevel` as shown below:

```
-(id) initWithLevel:(int)level
{
    if( (self=[super init])) {
        //1
        currentLevel = level;
        //2
        NSString *path = [[NSBundle mainBundle]
          pathForResource:@"levels" ofType:@"plist"];
        NSDictionary *levelsDict = [NSDictionary
          dictionaryWithContentsOfFile:path];
        //3
        NSString *levelString =
          [NSString stringWithFormat:@"level%d", level];
        NSDictionary *lvlDict = levelsDict[levelString];
        //4
        NSString *bgMusic = lvlDict[@"music"];
        //5
        [[SimpleAudioEngine sharedEngine]
          playBackgroundMusic:bgMusic loop:YES];
        [SimpleAudioEngine sharedEngine].backgroundMusicVolume =
          0.25;
    }
    return self;
}
```

This does the following:

1. First store the level index for later use.
2. Next load **level.plist** into the `levelsDict` `NSDictionary` object.
3. Create 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, load `lvlDict` with the data for the current level.

4. Now that `lvlDict` contains the current level's information, you can use the "music" key to get the filename for the MP3 that is the background track for this level.
5. Instantiate the `SimpleAudioEngine` singleton and start playing the background music. It's a bit loud, so set the volume level to **.25** in order to keep it at a tolerable level.

That's it! If you build and run now and select a game level, you'll have the same view as before, but now you'll hear some groovy tunes playing!

Next on the agenda: loading a parallax level. Add this code to `initWithLevel` right after the line that sets the background music volume:

```
//1
CCParallaxNode *pNode = [CCParallaxNode node];
//2
NSArray *backGroundArray = [lvlDict objectForKey:@"background"];
//3
for (NSArray *nodeArrays in backGroundArray) {
    for (NSString *bgChunkFilename in nodeArrays) {
        //4
        CCSprite *bgNodeSprite = [CCSprite
spriteWithFile:bgChunkFilename];
        //5
        bgNodeSprite.anchorPoint = ccp(0.0, 0.0);
        //6
        int indx = [nodeArrays indexOfObject:bgChunkFilename];
        //7
        float indx2 = (float)[backGroundArray
indexOfObject:nodeArrays] + 1.0;
        float ratio = ((4.0 - (float)indx2) / 8.0);
        if (indx2 == 4.0) {
```

```
            ratio = 0.0;
        }
        //8
        [pNode addChild:bgNodeSprite z:(int)indx2 * -1
parallaxRatio:ccp(ratio, 0.6) positionOffset:ccp((indx * 2048),
30)];
    }
}
//9
[self addChild:pNode];
```

This code block creates a parallax node. 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. It's most often used as above, as a background to simulate depth by moving the layers in the back at speeds slower than the foreground layers.

1. First you create a new parallax node.
2. Next you retrieve the background information and load it into the `backGroundArray` object.
3. The `backGroundArray` object contains an array hierarchy that is two levels deep. Each layer has three objects and multiple images, and has to be broken into tiles.

   OpenGL (and therefore Cocos2D) has a maximum texture size that it can use. These levels are much larger horizontally than that maximum. In this code, the max is assumed to be 2048 x 2048. On newer devices that max is larger, and on the iPhone 3G and older devices it is lower. You don't need to worry about the lower maximum because those older devices don't support Cocos2D 2.0 at all, so they won't run any of the code you'll be writing here anyway.

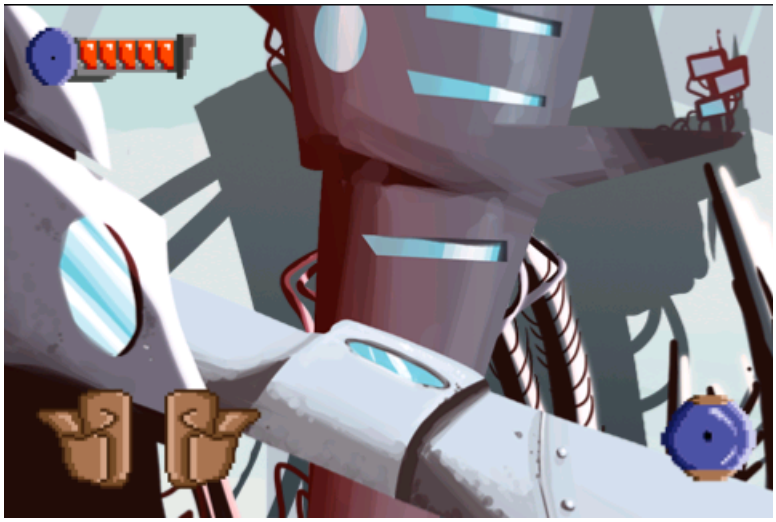
   You have to tile the background in 2048-wide chunks, which is why you're using a loop to load multiple images into each layer. However, each layer moves at a rate that is slower than the main tile map for that level. So each level is smaller than the layer in front of it.

   The entire level (the tile map) is 500 tiles wide. Each tile is 32 pixels. This results in a level that is 16,000 pixels wide. Each parallax layer moves slower than the one in front of it, so you need some fraction of 16,000 pixels to fill the entire level.
4. Within the two `for` loops each image is loaded as `bgNodeSprite`, a `CCSprite` object.
5. The `anchorpoint` is set at (0, 0) so that the bottom left corner of each background layer can be easily set equal to the bottom left of the coordinate system.
6. You find the index of the filename within the array. This is so that it can be positioned in the appropriate place. All the images within an individual layer are

set up end-to-end. The `indx` variable holds the relative position of the filename within the array and therefore, the position of the image within the entire layer.

7. You find a second index value, `indx2`. This is the position of the layer within the entire node. You find this value in order to assign a ratio, or a speed, to all the tiles within that layer. This value is called `ratio` and varies between 1/2 and 1/8 of the speed of the foreground tile map.
8. Once you've found these values, you use them to add each of the images as children of the parallax node. The call for that is `addChild:z:parallaxRatio:positionOffset:`. Using this method, you align each of the tiles and assign to them the appropriate speed and z-position.
9. Once the loop has finished with all the image tiles, you add it to the main layer.

Build and run now. You won't see any magic yet, because that requires movement. But you will see the beauty (…and the beast is coming!):



And that's it for now – now you are familiar with the starter project, and have written a bit of code to get you started.

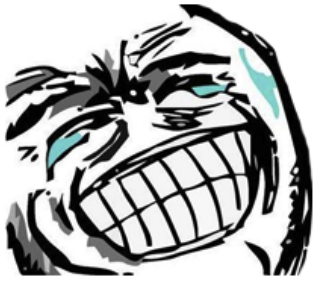
In the next chapter, the real fun begins! You'll add the tile map, and add Cyclops, the star of the game. And you'll start on the core of the platformer: the physics engine!

> **Challenge:** To test out the parallax scrolling, try running an action on the layer to scroll the level forward.
>
> Hint: you will actually have to move the layer to the left in order to make the screen appear to move to the right! ☺

# Chapter 2: A Budding Physics Engine

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

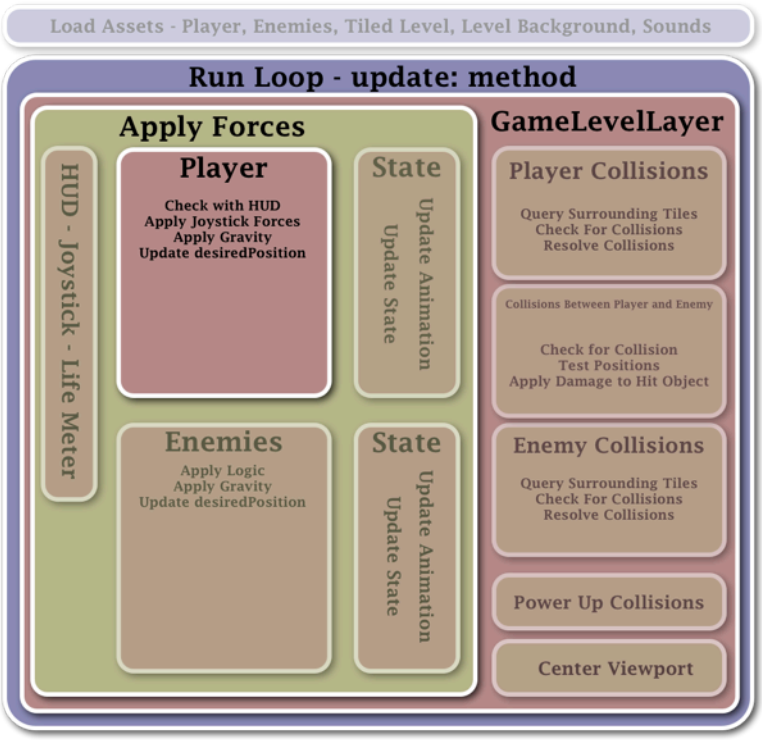
This chapter will focus specifically on applying forces to the `Player` object. You won't be checking the state of the buttons on the HUD quite yet, but you will learn how to effectively store forces in vectors, specifically in `CGPoint` structures, and how to work with and manipulate those forces.



**It is all unfolding according to my design!**

But first, to get to that point, you'll load some `CCTMXTiledMap` 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 was discussed in the first chapter.

Here's the portion of the overall plan that this chapter will cover (everything except the HUD):




# The Tao of physics engines

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

There are two main reasons why you'll be rolling your own instead of using pre-existing engines such as Box2D or Chipmunk:

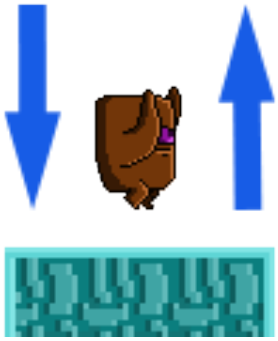
1. **Fine tuning**. To get the right feel for a platformer game, you need to be able to fine-tune the response of the engine. In general, platformers created using pre-existing engines don't feel like the classic games that you're used to.
2. **Simplicity**. Box2D and Chipmunk have a lot of extra capabilities 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**. The first job of a physics engine 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 Cyclops to make him jump. Over time, the force of gravity will act against that initial upward force, which will give you that nice classic parabolic jump pattern.

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



For example, when Cyclops hits the ground, collision detection will keep him from falling through. It will also be used when he lands on, or bumps into, an enemy.

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

# Physics engineering

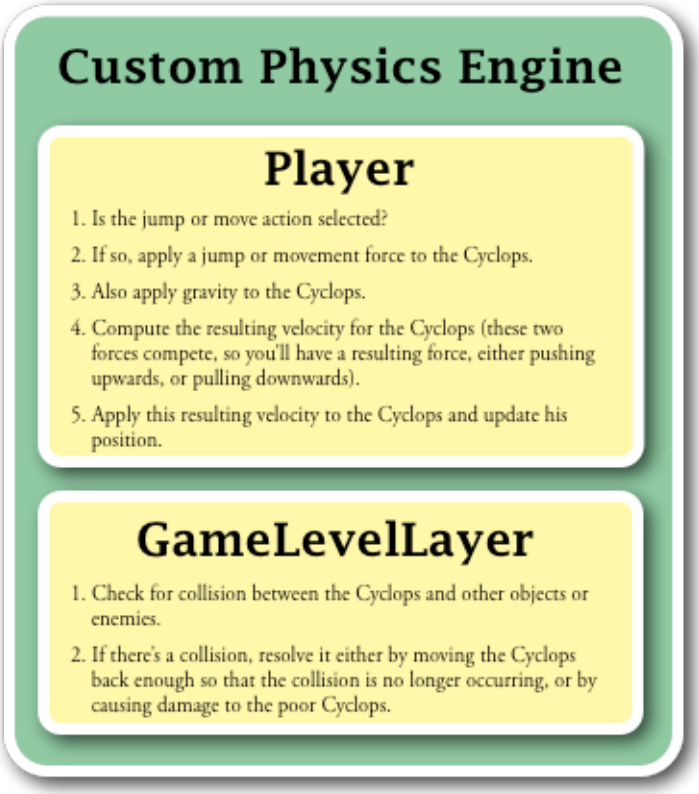
In the physics engine you'll create, Cyclops will have his own movement-describing variables: current velocity (speed and direction), acceleration, and position, among others. Using these variables, every movement you apply to him will follow this algorithm:

1. Is the jump or move action selected?
2. If so, apply a jump or movement force to Cyclops.
3. Also apply gravity to Cyclops.
4. Compute the resulting velocity for Cyclops. The jump or movement force and gravity compete with each other, so you'll have a final resulting force, either pushing upwards, or pulling downwards.
5. Apply this resulting velocity to Cyclops and update his position.
6. Check for collision between Cyclops and other objects or enemies.
7. If there's a collision, resolve it either by moving Cyclops back enough so that the collision is no longer occurring, or by causing damage to the poor Cyclops.

You'll run these steps for every frame.

In the game world, gravity is constantly pushing Cyclops down and through the ground, but the collision resolution step will put him back on top of the ground in each frame. You can also use this feature to determine if Cyclops is still touching the ground, and if not, disallow a jump action since Cyclops is probably already in mid-jump or has just walked off a ledge.


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




However, when you reach the sixth step — collision detection — you need to take all of the level features, such as walls, floors, enemies and other hazards, into consideration. So the collision detection step will be performed in each frame by the

`GameLevelLayer` – remember, that's the `CCLayer` subclass that will do a lot of the physics engine work.

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

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

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

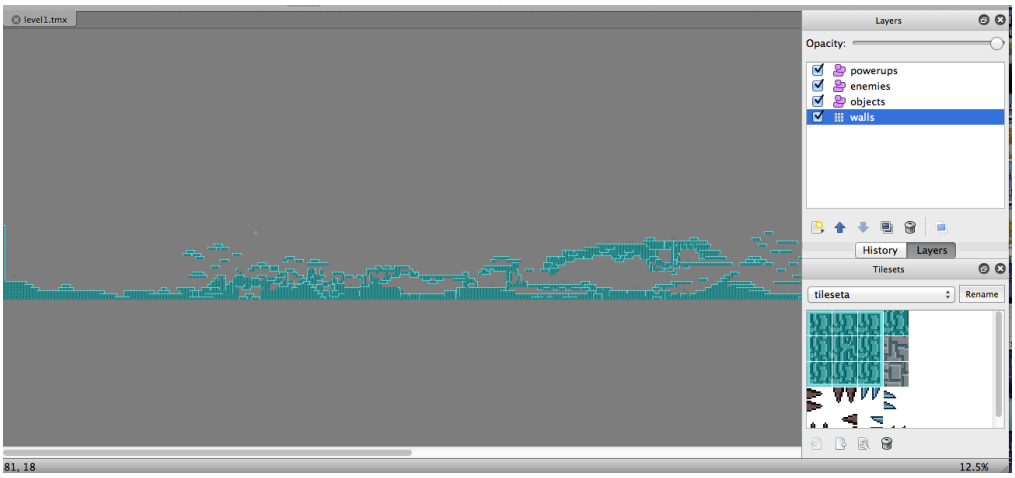
# In the beginning, there was a tile map

It's TMX time! Physics wouldn't be much use without geography. The next step in the process is bringing 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 up your Tiled map editor (download it if you don't have it already) and opening **level1.tmx** (found at PocketCyclops/level1.tmx in the project) from your project directory. You'll see the following:

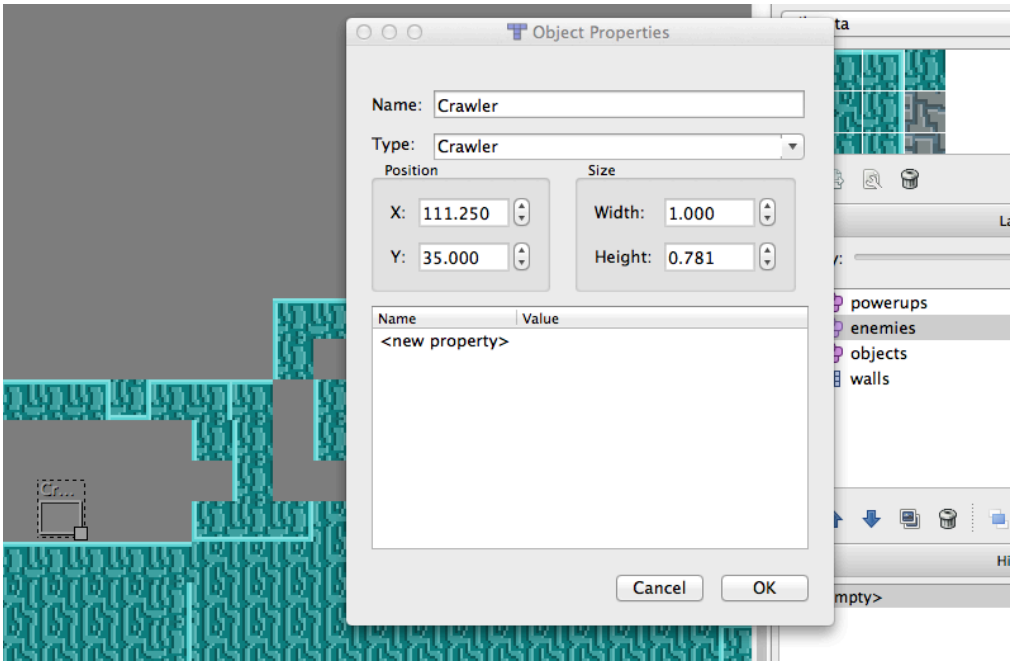



Take a look at the sidebar. There are four kinds of layers in this level file:

1. **Powerups** – This layer contains the objects that grant Cyclops additional powers.

2. **Objects** – This layer contains level features, like the starting point where Cyclops enters the layer and the exit point that triggers the end of the level.
3. **Enemies** – This layer, as the name implies, contains the various enemies.
4. **Walls** – This is the main layer, containing all the walls (or platforms and ground, if you prefer) for the layer.

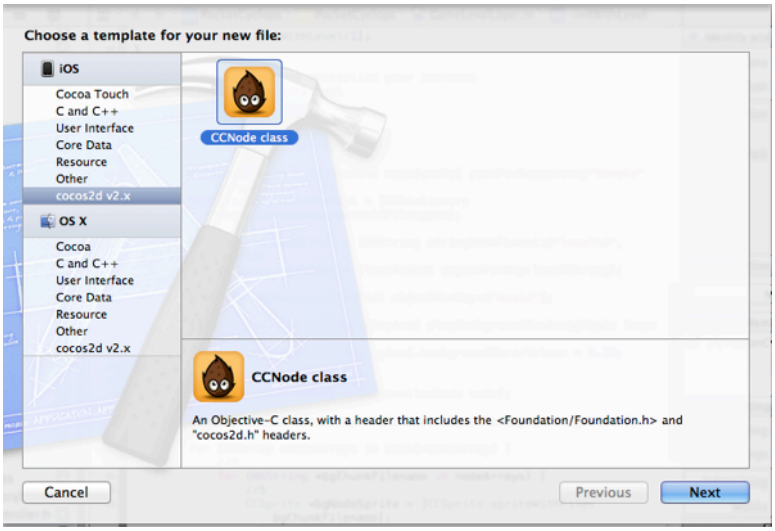
Notice that to add enemies, objects, or powerups, I added objects (i.e. the "gray rectangles") to the object layer, and gave them particular names, types, and properties. For example, if you right click on one of the enemy objects and choose Properties, you will see it has a name and type of Crawler:




You will be looking for objects with these names and types in code later on.

It's time to add the tile map to the game. But first, 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 `CCTMXTileMap` subclass.

Open the project in Xcode, if it isn't already open, and create a new class by going to **File->New->File**. Choose the **iOS\cocos2d v2.x\CCNode** class template. Make it a subclass of `CCTMXTiledMap` and give it the name **PCTMXTiledMap.m**. The default location for this new class should be in the same folder as the other files – that's fine.




You'll be adding to this class as you move forward. For now, use it as-is.

Next import the class into **GameLevelLayer.m**:

```
#import "PCTMXTiledMap.h"
```

Add an instance variable to the class extension (the `@interface` block) using this new class:

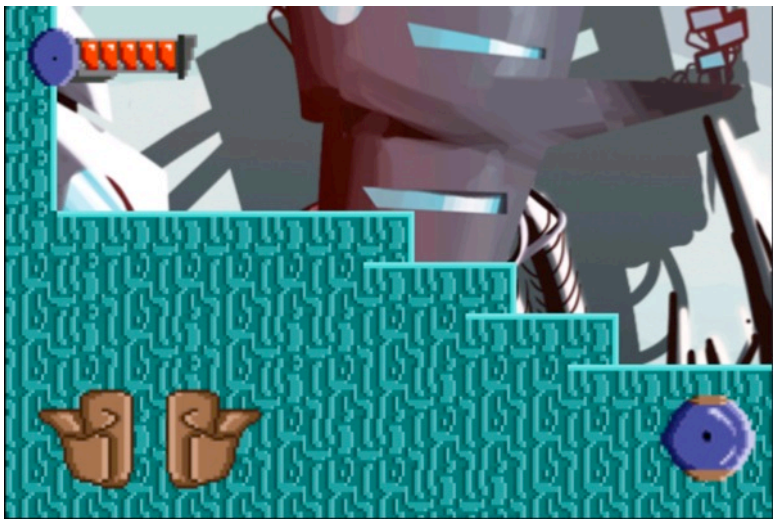
```
PCTMXTiledMap *map;
```

Now add the tile map to the layer! Add the following code to `initWithLevel:` right after the code that adds the parallax node to the layer:

```
NSString *lvlString = [lvlDict objectForKey:@"level"];
map = [PCTMXTiledMap tiledMapWithTMXFile:lvlString];
[self addChild:map];
```

In the above code you retrieve the name of the TMX file from the `lvlDict` object. Using that filename, you instantiate the map object with a standard `tiledMapWithTMXFile` initializer. Then you add that object to the layer.

If you build and run now, you should have tiles in your game levels:



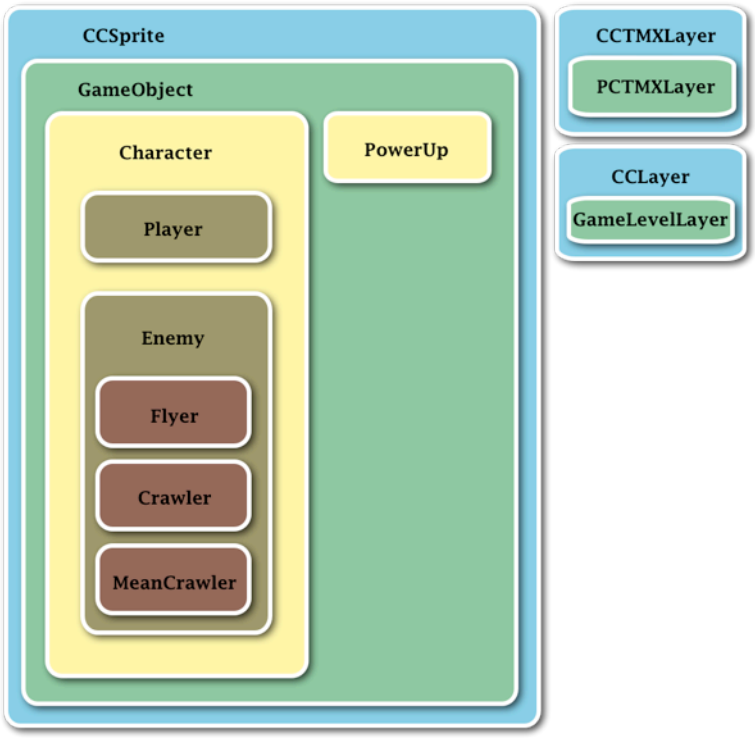
# Taxonomy of objects

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

Well, except that in order 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. But, you will come to see that in the end you'll write less code and have a much easier time maintaining that code.

I covered the class hierarchy briefly in the introductory chapter. You will now create those objects.

Once again, here's the diagram showing how all the classes of the engine will fit together:




As you can see, all of the objects will be children of the `CCSprite` class. The first one is the `GameObject` class.

Create the `GameObject` class now by going to **File->New->File**. Choose the **iOS\cocos2d v2.x\CCNode** class template, set the subclass to `CCSprite` and name it **GameObject.m**.

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 `GameObject` class is that every subclass will need to load animations, and the methods in the `GameObject` class will help do that.

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

The `Character` class will contain methods to:

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

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

Add the `Character` class by going to **File->New->File**. Select the **iOS\cocos2d v2.x\CCNode** class template, make it a subclass of `GameObject` and call it **Character.m.**

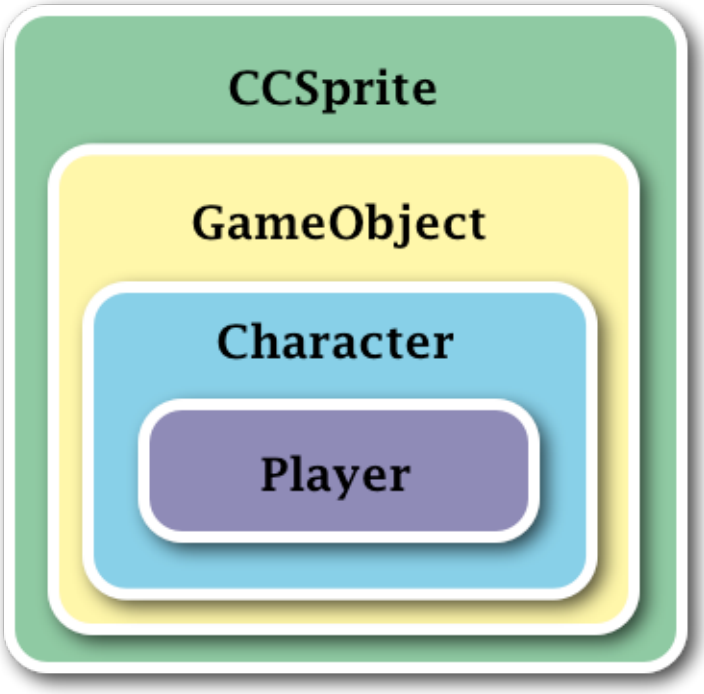
You need to change one thing in order to get this to work. Because `Character` is a subclass of `GameObject`, you have to import the **GameObject.h** header. You don't need to import cocos2d.h, however, because GameObject.h already does that. So change this line in **Character.h**:

```
#import "cocos2d.h"
```

To this:

```
#import "GameObject.h"
```

As this tutorial progresses 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 have the `Player` class in place yet. ☺ That's what you'll do next!

# Incarnating the hero

The `Player` class will be a subclass of `Character`, so create a new file by going to **File->New->File**, select the **iOS\cocos2d v2.x\CCNode** class template, set it as a subclass of `Character` and call it **Player**.

As you did for the `Character` class, change the #import line in **Player.h** from this:

```
#import "cocos2d.h"
```

To this:

```
#import "Character.h"
```

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

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

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

Next, add an instance variable to the `@interface` section:

```
Player *player;
```

It's time to add the player object. You might think it's as simple as calling `spriteWithFile`. You could certainly do that, but in this tutorial you'll be using sprite sheets instead.

A sprite sheet is a collection of individual sprites all organized into one image. Here's what Cyclops' sprite sheet looks like:

**Note:** Sprite sheets are often used with batching to increase drawing performance when you have many instances of the same sprite onscreen. In the case of the player object, there will only ever be one player instance on the screen at one time, so there's no need for batching.

So why use sprite sheets here? Another advantage of sprite sheets is you save texture memory by packing the sprites into a smaller space. Plus it's a good practice to follow in case you need to move to sprite batching later as your game becomes more graphically intensive.

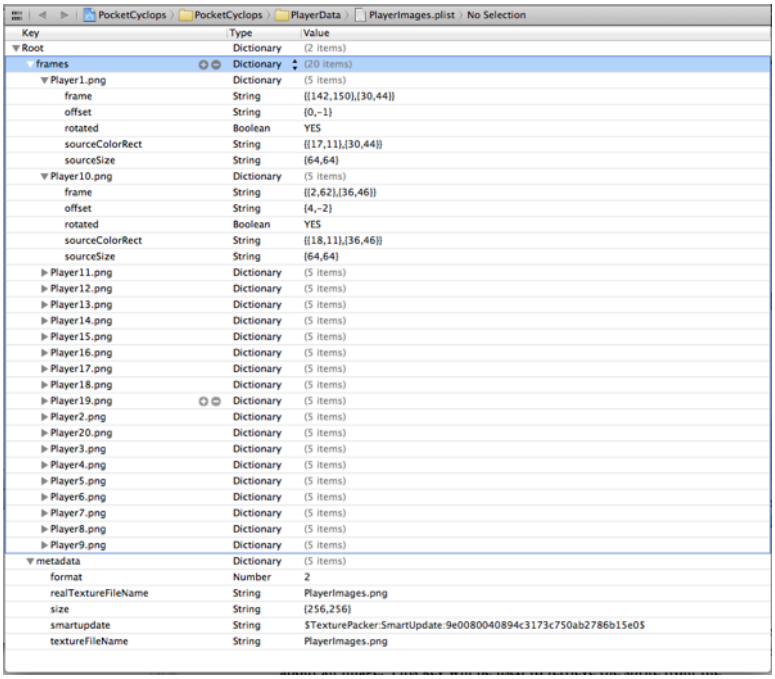
To learn more about the advantages of sprite sheets, check out this video: http://www.codeandweb.com/what-is-a-sprite-sheet

Add the following code to `initWithLevel:` in **GameLevelLayer.m** (right after the code that loads the tile map):

```
[[CCSpriteFrameCache sharedSpriteFrameCache]
  addSpriteFramesWithFile:@"PlayerImages.plist"];

player = [[Player alloc]
  initWithSpriteFrameName:@"Player1.png"];
[self addChild:player];
```

The first line loads the sprite sheet into the `CCSpriteFrameCache` singleton. This object is responsible for keeping track of all the loaded sprite sheets, the individual sprite images and their names.

Here's what the PLIST for the sprite sheet looks like:




What you see above is the PLIST file that describes the name and location of each individual image within the sprite sheet. The original file name is the key for the attributes of a specific image. This key will be used to retrieve the sprite from the `CCSpriteFrameCache`.

> **Note:** The sprite sheets used in this tutorial were created using Texture Packer. If you'd like to know more about sprite sheet creation and use, check out this tutorial:
>
> http://www.raywenderlich.com/1271/how-to-use-animations-and-sprite-sheets-in-cocos2d

The player object is created using the `initWithSpriteFrameName` initializer. This retrieves the correct sprite based on the key provided by `CCSpriteFrameCache`. Other than that, the `Player` sprite behaves just as if you used the `initWithName` initializer.

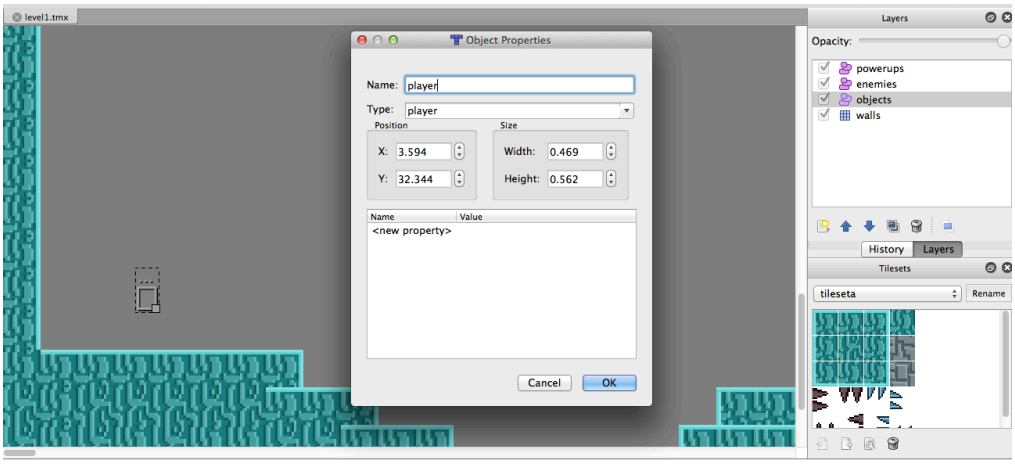
The player is then added to the layer. If you build and run now, you should see the player on the screen!

Can you spot him? Hah! He's hiding in the bottom-left corner.

You need to set his initial position. That's one of the pieces of information stored in the layer called **objects** 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 there, load up **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 of the data for the gray box within the objects layer (this is from Tiled).




The gray box has a name and a type, both "player". It also has a position and a size. You'll only be using the name and the position here. If you'd like to refresh your understanding of how to 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

Add this code to `initWithLevel:` to retrieve the player information (right after the code that creates the player object):

```
CCTMXObjectGroup *og = [map objectGroupNamed:@"objects"];
NSDictionary *playerObj = [og objectNamed:@"player"];
player.position = ccp([playerObj[@"x"] floatValue],
[playerObj[@"y"] floatValue]);
```

Here you first retrieve the `CCTMXObjectGroup` for the objects layer from the tile map.

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 value is stored in the x and y keys. You convert these values to floats (they are `NSNumbers`) and use the values to position the player.

> **Note:** You're using the new subscripting syntax (`playerOjb[@"x"]` instead of `[playerObj objectForKey:@"x"]`) – this is the new "literal syntax" introduced in Xcode 4.5.

Build and run now. Old Cyclops is right where he should be!

My, he doesn't look very happy to be there, does he?

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

# The gravity of your situation

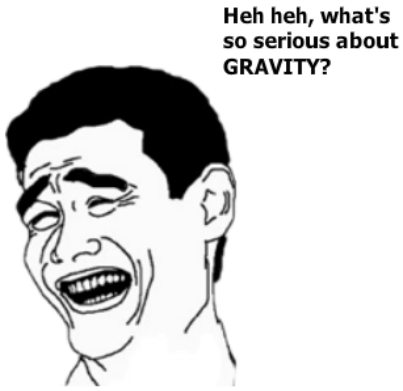
To build a platformer physics environment, you could write a complex set of branching logic that takes the player's state into account and decides which forces to apply based on that state.

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

Other forces don't just 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 gravity off. 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 is countered by friction, which gradually slows down the object until it stops.

This 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; gravity will always be applied.




# Playing God

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

This means that when Cyclops walks off a ledge, he'll continue to fall at an accelerating rate until he collides with something else. When you move him left or right, he won't stop moving as soon as you stop applying force; his momentum will carry him forward, while friction slows him down gradually until he stops.

As you continue building your platform game, you'll see that this logic will make 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 games.

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! ☺

# The law of the land: CGPoints and forces

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 moving 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.

You're going to be representing 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. "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 `CGPoint`s, you can rely on the various built-in functions provided by Cocos2D. You'll be making heavy use of functions such as `ccpAdd` (add two points), `ccpSub` (subtract one point from another) and `ccpMult` (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 will be acted upon in each frame by a number of forces, including gravity, walk/jump forces supplied by the user, and friction, which will slow (and eventually stop) Cyclops.

In each frame, you'll add all these forces together, and the resulting cumulative force will be added to the previous velocity of the player object to give you the current velocity. The current velocity will be scaled down to match the fractional time amount between each frame, and finally that scaled value will be used to move the player's position for that frame.

> **Note:** If any of this sounds confusing, Daniel Shiffman wrote an excellent tutorial on vectors that 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/

Start with gravity, since it's a constant force that will always be applied 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 `Character` class, because it is common to all moving game objects. In **Character.h** add this right before the `@end`:

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

Moving to **GameLevelLayer.m**, in `initWithLevel:`, add this line (before the end of the `if` statement):

```
[self scheduleUpdate];
```

This call is a built-in Cocos2D method that schedules a method named `update:` to be executed every frame.

Next add the `update:` method:

```
-(void)update:(ccTime)dt {
    [player update:dt];
}
```

This calls `update:` on the `player` class. You'll get an error telling you that `Player` doesn't have an `update:` method. You can fix that!

But instead of putting the method declaration into the `Player` class, add it to its superclass, `Character`. Every `Character` subclass will have an `update:` method (they'll be different, but they'll all have one).

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

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

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

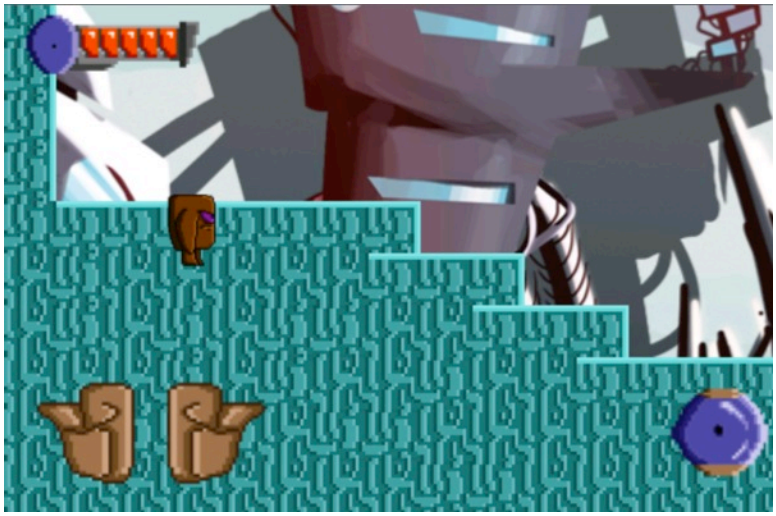
```
// 1
-(id)initWithSpriteFrameName:(NSString *)spriteFrameName
{
    if (self = [super initWithSpriteFrameName:spriteFrameName]) {
        self.velocity = ccp(0.0, 0.0);
    }
    return self;
}

-(void)update:(ccTime)dt
{

    // 2
    CGPoint gravity = ccp(0.0, -450.0);

    // 3
    CGPoint gravityStep = ccpMult(gravity, dt);

    // 4
    self.velocity = ccpAdd(self.velocity, gravityStep);
    CGPoint stepVelocity = ccpMult(self.velocity, dt);

    // 5
    self.position = ccpAdd(self.position, stepVelocity);
}
```

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

1. You've created a new `initWithSpriteFrameName:` method that uses `initWithSpriteFrameName:`. This initializer is the one used to load individual frames from a sprite sheet. Also, you initialize the velocity variable to (0,0).
2. Here you declare the value of the gravity vector (vector meaning the change in position). For each second of time, you accelerate the velocity of Cyclops 450 pixels towards the floor. If Cyclops starts from a standstill, at the one second mark he'll be moving at 450 pixels/second, at two seconds he'll be moving at 900 pixels/second, and so forth. Clear enough?
3. Next you use `ccpMult` to scale the acceleration down to the size of the current time step. Recall that `ccpMult` multiplies a `CGPoint`'s values by a float value and returns the `CGPoint` result. This way even when you're faced with a variable frame rate you'll still get consistent acceleration.
4. Once you've calculated the gravity for the current step, you add it to your current velocity. With the new velocity you've calculated, you have the velocity for a single time step. Again, you're doing these calculations in order to get consistent velocity no matter what the frame rate is.

5. Finally you use the velocity you calculated for this single step and `ccpAdd` to get the updated position for Cyclops.

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



See how Cyclops falls right through the floor? He won't get very far that way. It takes collision detection to make the ground solid beneath his feet, and that's what you'll do next.

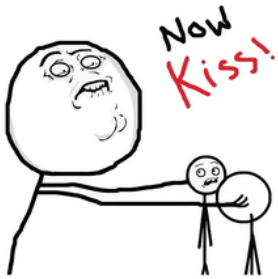
Congratulations on getting this far! This chapter and the next cover the most difficult concepts in this tutorial and you're already about half way through that!

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

You could use the accelerometer in the device, 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. In the last chapter, you focused on the first part, by simulating movement for the Cyclops through gravity. In this chapter, you will focus on the second part, by adding collision detection and resolution into the game!



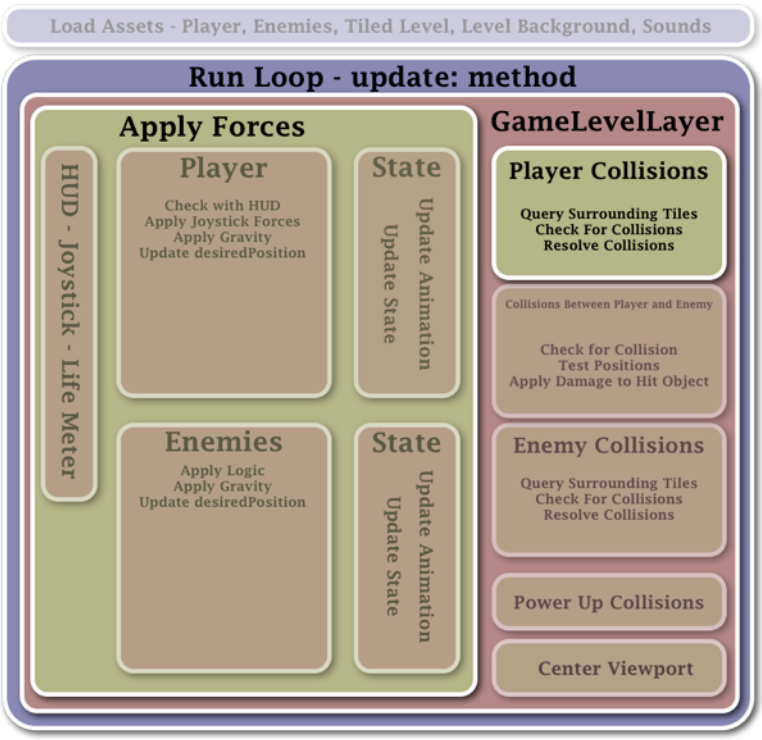

You will add the collision handling code to the `GameLevelLayer`. The `GameLevelLayer` has access to the tile map, the player, the enemies and all the other objects that you want to check for collisions.

You will need some methods to query the level for tiles that will generate collisions. To provide these, you'll add some new methods to `PCTMXTiledMap`, your subclass of `CCTMXTiledMap`.

You'll also look at different methods of resolving collisions and learn which one 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 from it.

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. Lucky for you, a platformer like *Pocket Cyclops* doesn't need a very complicated collision detection engine.

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

In order to detect collisions for Cyclops, you'll need to query the tile map for the tiles that directly surround him. Then you'll use a few built-in iOS functions to test whether the character's bounding box is intersecting with the tile's bounding box.

> **Note**: Forget what a bounding box is? It's simply the smallest axis-aligned rectangle that a sprite can fit inside. Usually this is straightforward and 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 – Cocos2D 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`.

But first, 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 `boundingBox` 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, depending on the player sprite. You don't want to register a collision when an object overlaps this transparent space – only when it starts to overlap 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 first thing you'll do is add a method that will return the bounding box of Cyclops. Other objects will need this same method – for instance, enemies will also need to collide with the walls and floors of the level. So you'll want to add this method in a common place.

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

How about in the `Character` class? All classes derived from the `Character` class will 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 **Character.h**:

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

Add the method implementation (and a method stub to get rid of compiler warnings for missing implementations) to **Character.m**:

```
-(void)update:(ccTime)dt {
}

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

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

You'll be overriding this method in subclasses in order to return a `CGRect` that is smaller than the sprite frame `CGRect`. But it's still important to have a common ancestor method, so that even if you don't remember to implement it in a subclass, it 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` called `bounding` that starts from the created origin point (by moving half the defined width left and half the defined height down from the center). You're using constants for the height and width of the box, and there's a good reason for this.

Often when you have a number of different sprites that make up animations for a game character, they don't all have the same dimensions. Some may be taller or wider. If you're using the sprite 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. In order to rectify this, you need to use a constant height and width to create the bounding rectangle. Another way to accomplish this is to make sure that all the sprites are the same exact size, but the method you're going to use here is more reliable.

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 center.

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

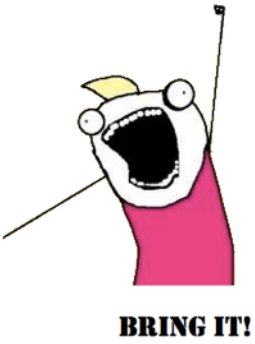
If you wanted to make your game more forgiving, you could shrink this box down a little more, so that the player would have to get even closer to collide with enemies.

Notice that your code is currently showing errors for `kPlayerWidth` and `kPlayerHeight` since these values aren't defined. To fix that, add the following `#define` statements in **Player.h** right after the `#import` line:

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

# The heavy lifting

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 methods in your `GameLevelLayer` to accomplish the collision detection:

- A method that returns the coordinates of the eight tiles that surround Cyclops' current location.
- A method to determine which, if any, of these eight tiles is a collision tile. Some of your tiles, like background tiles, won't have physical properties and therefore your Cyclops won't collide with them.
- A method to resolve those collisions in a prioritized way.

You'll create two helper methods that will make it easier to accomplish the above goals:

1. A method that calculates the tile position of the player.
2. A method that takes a tile's coordinates (tile map coordinates) and returns the `CGRect` in Cocos2D coordinates.

Tackle the helper methods first. You'll add these methods to the `PCTMXTiledMap` class you created earlier.

Add the following method declarations to **PCTMXTiledMap.h** (right before the `@end`):

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

Now add the methods themselves to **PCTMXTiledMap.m**:

```
- (CGPoint)tileCoordForPosition:(CGPoint)position
{
    float x = floor(position.x / self.tileSize.width);
    float levelHeightInPixels = self.mapSize.height *
      self.tileSize.height;
    float y = floor((levelHeightInPixels - position.y) /
      self.tileSize.height);
    return ccp(x, y);
}

-(CGRect)tileRectFromTileCoords:(CGPoint)tileCoords
{
    float levelHeightInPixels = self.mapSize.height *
      self.tileSize.height;
    CGPoint origin = ccp(tileCoords.x * self.tileSize.width,
      levelHeightInPixels - ((tileCoords.y + 1) *
      self.tileSize.height));
    return CGRectMake(origin.x, origin.y, self.tileSize.width,
      self.tileSize.height);
}
```

The first method gives you the coordinates of the tile based on the passed-in position. In order to get a tile position, you just divide the coordinate value by the tile size.

You need to invert the coordinate for the height, because the coordinate system of Cocos2D/OpenGL has an origin 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?

The second method reverses the process of calculating the coordinate. It takes in a tile coordinate and returns the `CGRect` in Cocos2D space of the given tile.

It multiplies the tile coordinate by tile size. Once again, you have to reverse 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.

Why do you add one to the tile 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 point it returned would be `19 * tileHeight`.

# Surrounded by tiles!

Now you'll create the method to retrieve the surrounding tiles. This method will return an array containing the global ID of the tile image (GID), the tile map coordinate for that tile, and information about the `CGRect` origin for that tile.

You'll be arranging this array in the order of priority 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 Cyclops, you'll need to set the flag that tells you whether Cyclops is currently touching the ground. You'll need this later so you know when it's OK to let the Cyclops jump!

First add the method declaration to **PCTMXTiledMap.h**:

```
- (NSArray *)getSurroundingTilesAtPosition:(CGPoint)position
forLayer:(CCTMXLayer *)layer;
```

Now add the method itself to **PCTMXTiledMap.m**:

```
-(NSArray *)getSurroundingTilesAtPosition:(CGPoint)position
forLayer:(CCTMXLayer *)layer {
    //1
    CGPoint plPos = [self tileCoordForPosition:position];
    //2
    NSMutableArray *gids = [NSMutableArray array];
    //3
    for (int i = 0; i < 9; i++) {
        int c = i % 3;
        int r = (int)(i / 3);
        CGPoint tilePos = ccp(plPos.x + (c - 1), plPos.y +
          (r - 1));
        //4
        int tgid = [layer tileGIDAt:tilePos];
        //5
        CGRect tileRect = [self tileRectFromTileCoords:tilePos];
        //6
        NSDictionary *tileDict = @{
        @"gid":@(tgid),
        @"x":@(tileRect.origin.x),
        @"y":@(tileRect.origin.y),
        @"tilePos":[NSValue valueWithCGPoint:tilePos]};

        [gids addObject:tileDict];
```

```
    }
    //7
    [gids removeObjectAtIndex:4];
    [gids insertObject:[gids objectAtIndex:2] atIndex:6];
    [gids removeObjectAtIndex:2];
    [gids exchangeObjectAtIndex:4 withObjectAtIndex:6];
    [gids exchangeObjectAtIndex:0 withObjectAtIndex:4];
    //8
    for (NSDictionary *d in gids) {
        NSLog(@"%@", d);
    }
    return (NSArray *)gids;
}
```

Phew – there's a lot of code here! Don't worry, we'll go over it in detail.

Before reviewing this code section-by-section, note that you're passing in a layer object and that your tile map has a number of different layers, as covered in the last chapter.

Having separate layers allows you to have different types of collision detection that are tailored for each layer. In this game, I set up the map so that all of the obstacles that need to be checked for collision detection are in a single layer – the Walls layer. However, in your game you might want to have other types of layers, such as a layer of things 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, now let's go through the code above section-by-section:

1. First retrieve the tile coordinates for the input position (which will be the position of Cyclops or an enemy).
2. Next create a new array for returning the surrounding tile information.
3. Then start a loop that will run nine times – because there are nine possible spaces around (and including) the character's space. The code calculates the positions of the nine tile positions and stores them in the `tilePos` variable.

> **Note**: You only need information for eight tiles, because you should never need to resolve a collision with the tile space in the center of the 3 x 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, Cyclops has moved at least half his width in a single frame. He shouldn't move this fast, ever – at least in this game!

> However, to make iterating through those eight tiles easy, just include the tile position of Cyclops and remove it at the end.

4. This section calls the `tileGIDAt:` method. This method returns the GID of the tile at a specific tile coordinate. If there's no tile at that coordinate, it returns zero.
5. Next use the helper method you created earlier to calculate the Cocos2D world space coordinates for each tile's `CGRect`.
6. Then store all that information in an `NSDictionary`. The collection of dictionaries is put into the return array.
7. Remove the Cyclops tile from the array and sort the tiles into a priority order. Resolve collisions with the tiles directly adjacent (below, above, left, right) to Cyclops first.
8. The final section is just for debugging and serves no other purpose. It simply logs the final array to the console so you can make sure that you are doing everything correctly.

Quite often, resolving the collision for the tile directly under the character (enemy or Cyclops) also resolves the collisions for the 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.


Fix middle block First!
This also fixes block 2.

Your collision detection routine makes certain assumptions about how to resolve collisions. Those assumptions are valid more often for adjacent tiles than for diagonal tiles, so you want to avoid collision resolution with diagonal tiles as much as possible.

Below is an image showing the order of the tiles in the array before and after the sorting in step #7. You can see that after sorting, the bottom, top, left and right tiles are resolved first. Knowing this order will also help you know when to set the flag indicating that Cyclops is touching the ground, so you know if he can jump or not – you'll set this up later.

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | | 5 |
| 6 | 7 | 8 |

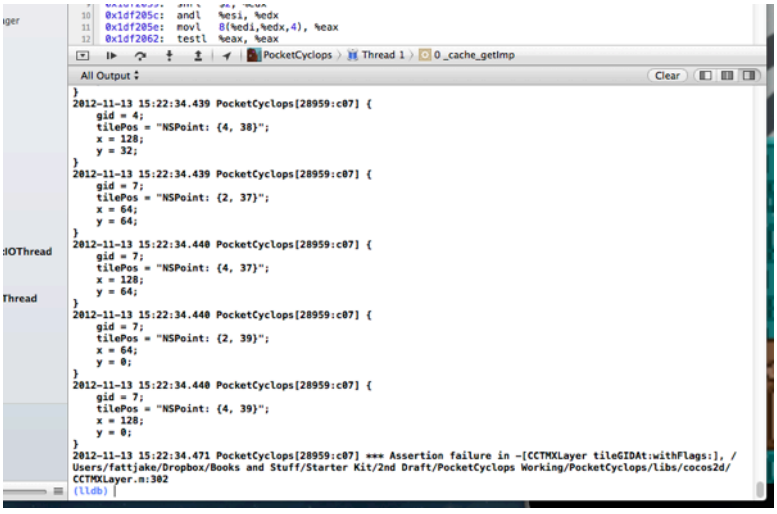
| | | |
|---|---|---|
| 5 | 2 | 6 |
| 3 | | 4 |
| 7 | 1 | 8 |

You're almost ready for the next build to verify that everything is correct! There are a few things to do first. Make the following changes in **GameLevelLayer.m**:

```
// Add to the @interface section
CCTMXLayer *walls;

// Add to the initWithLevel: method, after the map is added to //
the layer
walls = [map layerNamed:@"walls"];

// Add to the end of the update: method
[map getSurroundingTilesAtPosition:player.position
forLayer:walls];
```

The above code simply sets up an instance variable for the walls layer, which is later used to retrieve the surrounding tiles for the player every frame.

Build and run. You can see that you're getting tiles logged to the console, but unfortunately it crashes after a while:



At first you get a string of tile positions, but ultimately the game crashes with a `TMXLayer: invalid position` error message. This happens when the `tileGIDat:` method is given a tile position that is outside the boundaries of the tile map.

You're going to stop this from happening by implementing collision detection.

# Grounding Cyclops

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

If the player updates his position and then the `GameLevelLayer` finds a collision, you want your Cyclops to be moved back a little bit so he's no longer colliding.

So Cyclops needs a new variable that he can update, but one that will stay a secret between himself and the `GameLevelLayer` — call it `desiredPosition`.

You want the `Character` class (applies to both Cyclops and enemies) to calculate and store its desired position. But the `GameLevelLayer` 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 updating the actual sprite until all the tiles have been checked for collisions and resolved.

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

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

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

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

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

Finally, modify `collisionBoundingBox` in **Player.m** (it was overridden, 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);
```

```
}
```

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

Next, still in **Player.m**, make the following change to `update:` so that it's updating the `desiredPosition` property instead of the `position` property. Replace the line `self.position = ccpAdd(self.position, stepVelocity);` (at the very end of the method) with:

```
self.desiredPosition = ccpAdd(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). Add the following method to **GameLevelLayer.m**:

```
-(void)checkForAndResolveCollisions:(Character *)c {
    //1
    NSArray *tiles = [map getSurroundingTilesAtPosition:c.position
forLayer:walls];

    for (NSDictionary *dic in tiles) {
        //2
        CGRect pRect = [c collisionBoundingBox];
        //3
        int gid = [[dic objectForKey:@"gid"] intValue];

        if (gid) {
            //4
            CGRect tileRect = CGRectMake([[dic objectForKey:@"x"]
floatValue], [[dic objectForKey:@"y"] floatValue],
map.tileSize.width, map.tileSize.height);
            //5
            if (CGRectIntersectsRect(pRect, tileRect)) {
CGRect intersection = CGRectIntersection(pRect, tileRect);
                //6
                int tileIndx = [tiles indexOfObject:dic];

                if (tileIndx == 0) {
                    //tile is directly below the Character
```

```
                    c.desiredPosition = ccp(c.desiredPosition.x,
c.desiredPosition.y + intersection.size.height);

                } else if (tileIndx == 1) {
                    //tile is directly above the Character
                    c.desiredPosition = ccp(c.desiredPosition.x,
c.desiredPosition.y - intersection.size.height);

                } else if (tileIndx == 2) {
                    //tile is left of the Character
                    c.desiredPosition = ccp(c.desiredPosition.x +
intersection.size.width, c.desiredPosition.y);
                } else if (tileIndx == 3) {
                    //tile is right of the Character
                    c.desiredPosition = ccp(c.desiredPosition.x -
intersection.size.width, c.desiredPosition.y);
                } else {
                    if (intersection.size.width >
intersection.size.height) {
                        //7
                        //tile is diagonal, but resolving
collision vertically
                        float resolutionHeight;
                        if (tileIndx > 5) {
                            resolutionHeight =
intersection.size.height;
                        } else {
                            resolutionHeight = -
intersection.size.height;
                        }
                        c.desiredPosition =
ccp(c.desiredPosition.x, c.desiredPosition.y + resolutionHeight);
                    } else {
                        //tile is diagonal, but resolving
horizontally
                        float resolutionWidth;
                        if (tileIndx == 6 || tileIndx == 4) {
                            resolutionWidth =
intersection.size.width;
                        } else {
                            resolutionWidth = -
intersection.size.width;
                        }
                        c.desiredPosition =
ccp(c.desiredPosition.x + resolutionWidth, c.desiredPosition.y);
```

```
                        }
                    }
                }
            }
            //8
            c.position = c.desiredPosition;
        }
    }
```

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

Can you think of why? Because the `Character` 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 will be using the `desiredPosition` and the `collisionBoundingBox` methods to detect and resolve collisions with tiles in the level. The `Character` class is the class that 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 `Character` 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 `Enemy` class 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 those changes in one place.

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

1. First retrieve the set of tiles that surround the character. Next loop through each tile in that set.

   Each time you iterate through a tile, you check if there's a collision. If there is a collision, you resolve it by changing the `desiredPosition` attribute of the character.

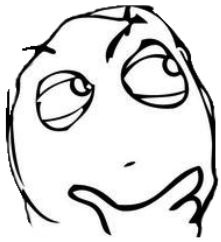
2. Inside the loop, first get the current bounding box for the character. As you saw earlier, this `desiredPosition` variable is the basis for the `collisionBoundingBox`. Each time a collision is found, the `desiredPosition` variable is changed so that it's no longer colliding with that tile. Often, that means that other tiles surrounding the character are no longer in collision either, and when the loop arrives at those tiles, you won't need to resolve those collisions again.

3. The next step is to retrieve the GID you stored for the tile in the dictionary. There may not be an actual tile in that position. If there isn't, you'll have stored a zero in the dictionary. In that case, the current loop iteration is over and the code moves on to the next tile.

4. If there is a tile at that position, you need to get the `CGRect` for that tile and store it in the `tileRect` variable. Now that you have a `CGRect` for the character and for the tile, you can use them to check for collisions.
5. To check for the collision, use `CGRectIntersectsRect`. If there is a collision, then use `CGRectIntersection` to get the `CGRect` 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.



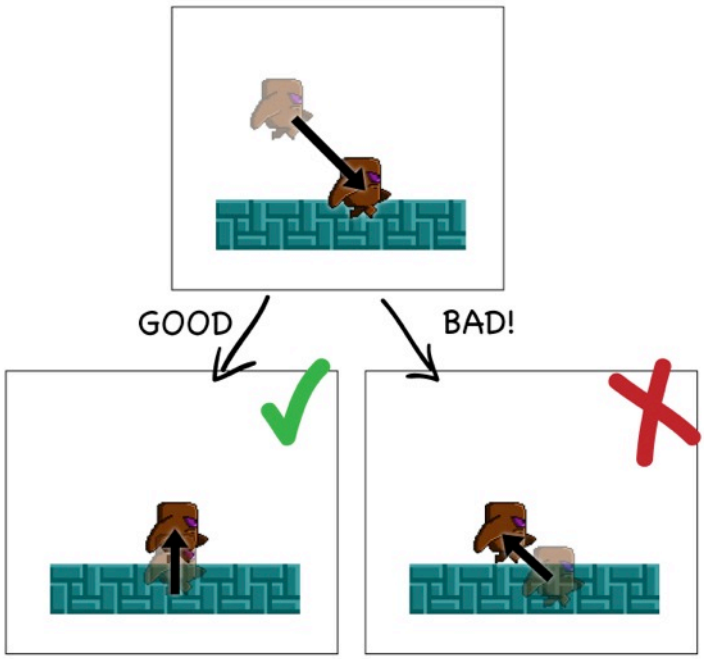
***Hmm... not as easy as it appears.***

You might think the best way to do so is to move the character backwards 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.

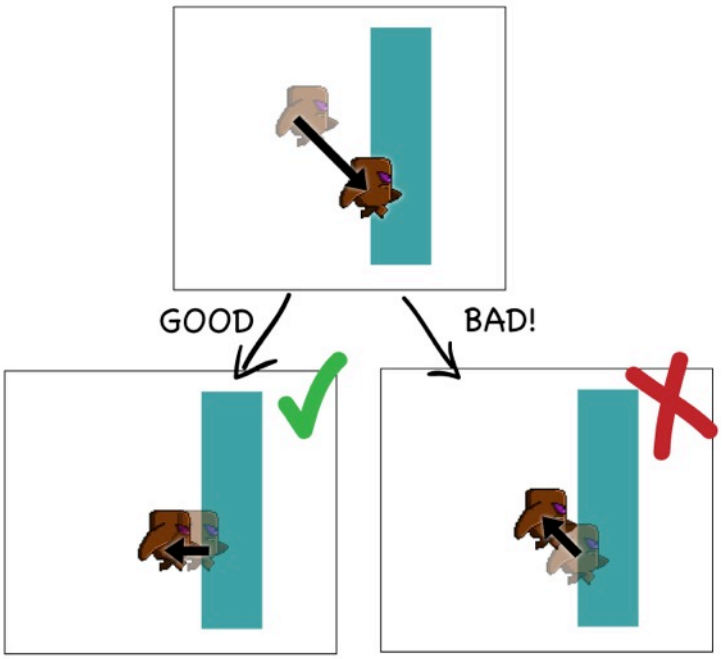
Consider this: gravity is constantly pulling the character down onto the tiles underneath him, and those collisions are constantly being resolved.

Imagine that the character is moving forward: he 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), he would need to move 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 Cyclops into the wall, for example, then his desired trajectory is diagonally downward and into the wall. Reversing this trajectory would move him upward and away from the wall — again, not the motion you want! You want him to stay on the outside of the wall without slowing or reversing his downward speed.

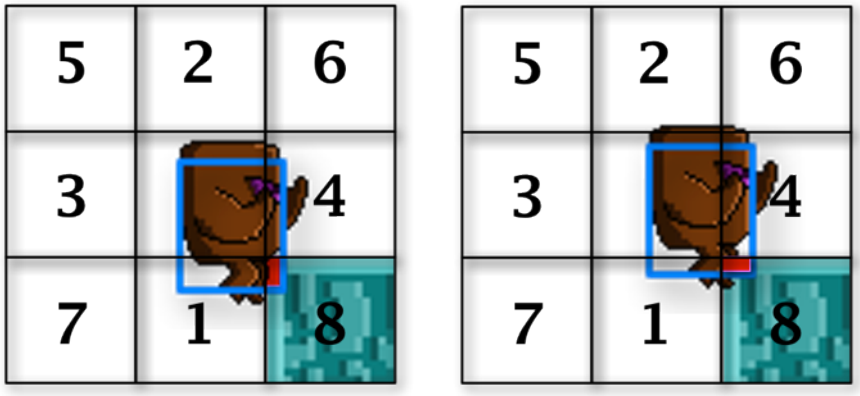



Therefore, you need to decide when to resolve collisions vertically, when to resolve them horizontally, and to handle both events as mutually exclusive cases.

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 Cyclops, you will always resolve that collision by moving him 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 him. 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.

Sounds 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 `character` 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` size to move the character back out of a collision state. Basically, look at the height or width, as appropriate, of the collision `CGRect` and use that value as the distance to move the character.

By now, you may have realized why you need to resolve tiles in a certain order. If not, consider the following example. Assume that the character is intersecting both the tile below him, and to the bottom right, as you can see in the following picture:


Fix middle block First!
This also fixes block 2.

The blue area (the tiny area to the right of the red bar) is tall and skinny, because that collision intersection only represents 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 tile directly beneath the character, then he'd no longer be in a collision state with the tile below and to 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:` method!

6. Here you get 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. Simple enough. However, once you get to the diagonal tiles, you'll implement the logic described in the section above.
7. 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 be moving the character either up or down, which you determine next by seeing if the tile index is greater than five (tiles six and seven are beneath the character). Based on that, you know whether you need to add or subtract the collision height. The horizontal collision resolution follows the same logic.
8. Finally, you set the position of the character to the final desired position calculated during collision detection.

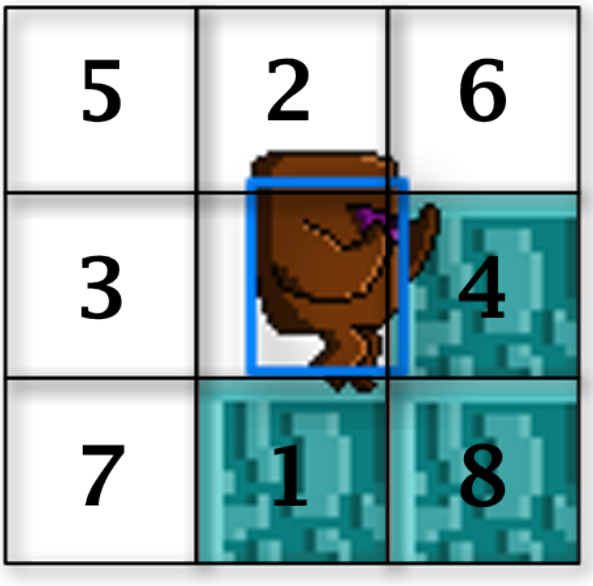
This method is the guts of your collision detection system. It's a basic system, and you may find that if your game moves very quickly or has other goals, you need to alter it to get consistent results. But for this particular game, it works well. ☺

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 position along the way:

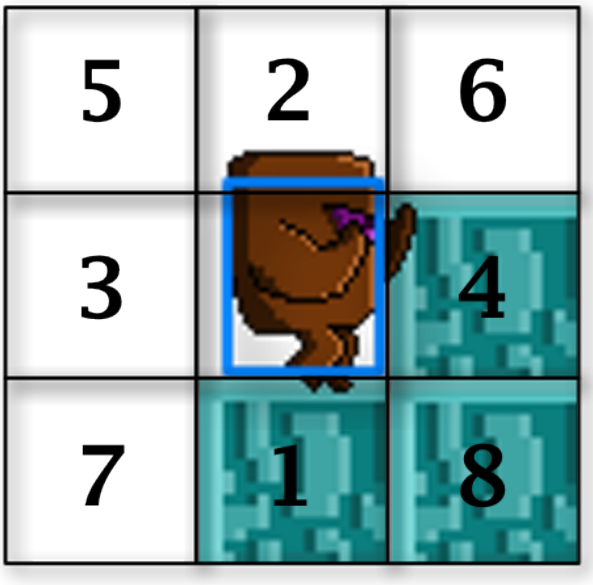
1. **Get tiles in order**: You've got collisions with three tiles 1, 4 and 8.


2. **Iterate through the tiles**.
3. **Check for GID**: If you have a tile at a given position, you will get a GID representing the tile image. If there's no GID, there's no tile there, and you can move on to the next tile position.
4. **Resolve collision**: You have a tile GID in slot one, so resolve that collision by moving the character's `desiredPosition` up by the height of the collision (retrieved by 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 (through tile 1) may not exist after you resolve the collision (by changing `desiredPosition`).



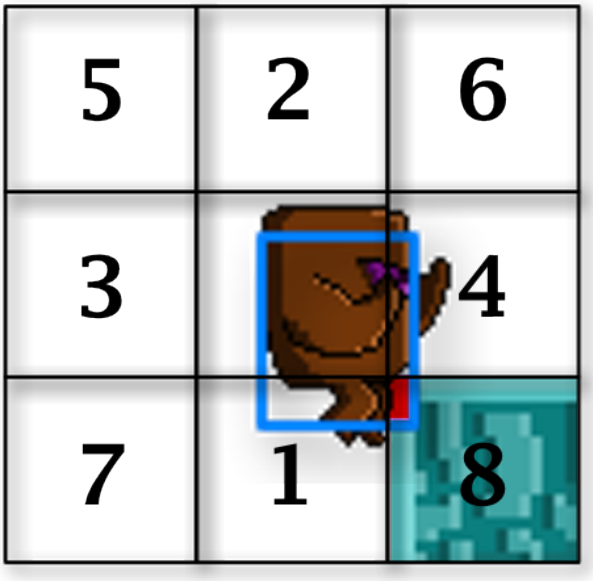

5. **Loop through tiles with GID 0**: Tiles 2 and 3 will return a GID of 0, so the loop will move to the next iteration in both cases, without testing for intersection or resolving collisions.
6. **Resolve collision with tile 4**: Because it's an adjacent tile and to the right, always resolve by subtracting 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.

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 Cyclops to the left and keeping the vertical position the same.

This case occurs often enough – think about when you are jumping up and over a tall obstacle. The player will be pushing into the side of the object (think a Mario Brothers pipe) all the way up (probably). Until that collision rectangle is wider than tall, the player will move up smoothly.

However, at the last frame before he would clear the pipe, it's possible that the collision will be wider than tall, in which case the resolution would put him 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 great 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! The collision detection method is complete and is used to check for collisions with the player (and later, with each enemy) every frame.

Let's put it to use! Replace this line in `update:` in **GameLevelLayer.m**:

```
[map getSurroundingTilesAtPosition:player.position
    forLayer:walls];
```

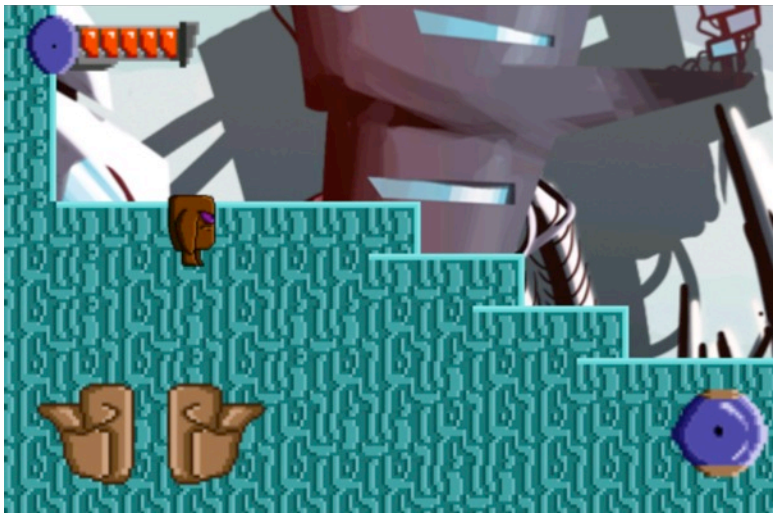
With this:

```
[self checkForAndResolveCollisions:player];
```

You can also remove the log statements in `getSurroundingTilesAtPosition:forLayer:` in **PCTMXTiledMap.m** (section #8) since they are no longer needed:

```
for (NSDictionary *d in gids) {
        NSLog(@"%@", d);
}
```

Build and run now:



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

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

Every frame, the force of gravity is added to `self.velocity`. This means that the character is constantly accelerating downward, pushing harder and harder into the floor.

You are constantly adding speed to the character's downward trajectory until it is greater than the size of the tile — you're moving through an entire tile in a single frame, which is a problem that was mentioned earlier.

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

If you don't do this, you'll get weird behaviors, such as moving through tiles as you saw above, or where your character jumps against a low ceiling and floats 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 that s/he can't jump off of thin air. You'll set up that flag now.

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

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

Now add the indicated lines (marked by comments saying `///// Here`) to your monster `checkForAndResolveCollisions:` method in **GameLevelLayer.m**:

```
-(void)checkForAndResolveCollisions:(Character *)c {
    //1
    NSArray *tiles = [map getSurroundingTilesAtPosition:c.position
forLayer:walls];
    c.onGround = NO; //////Here

    for (NSDictionary *dic in tiles) {
        //2
        CGRect pRect = [c collisionBoundingBox];
        //3
        int gid = [[dic objectForKey:@"gid"] intValue];

        if (gid) {
            //4
            CGRect tileRect = CGRectMake([[dic objectForKey:@"x"]
floatValue], [[dic objectForKey:@"y"] floatValue],
map.tileSize.width, map.tileSize.height);
            if (CGRectIntersectsRect(pRect, tileRect)) {
                //5
```

```
                CGRect intersection = CGRectIntersection(pRect,
tileRect);
                //6
                int tileIndx = [tiles indexOfObject:dic];

                if (tileIndx == 0) {
                    //tile is directly below the Character
                    c.desiredPosition = ccp(c.desiredPosition.x,
c.desiredPosition.y + intersection.size.height);
                    c.velocity = ccp(c.velocity.x, 0.0);
//////Here
                    c.onGround = YES; ///////Here
                } else if (tileIndx == 1) {
                    //tile is directly above the Character
                    c.desiredPosition = ccp(c.desiredPosition.x,
c.desiredPosition.y - intersection.size.height);
                    c.velocity = ccp(c.velocity.x, 0.0);
//////Here

                } else if (tileIndx == 2) {
                    //tile is left of the Character
                    c.desiredPosition = ccp(c.desiredPosition.x +
intersection.size.width, c.desiredPosition.y);
                } else if (tileIndx == 3) {
                    //tile is right of the Character
                    c.desiredPosition = ccp(c.desiredPosition.x -
intersection.size.width, c.desiredPosition.y);
                } else {
                    if (intersection.size.width >
intersection.size.height) {
                        //7
                        //tile is diagonal, but resolving
collision vertically
                        float resolutionHeight;
                        if (tileIndx > 5) {
                            resolutionHeight =
intersection.size.height;

                            if (c.velocity.y < 0) { ///////Here
                                c.onGround = YES; ///////Here
                                c.velocity = ccp(c.velocity.x,
0.0); ///////Here
                            }  ///////Here
                        } else {
```

```
                            resolutionHeight = -
intersection.size.height;

                            if (c.velocity.y > 0) { ///////Here
                                c.velocity = ccp(c.velocity.x,
0.0); ///////Here
                            }  ///////Here
                        }
                        c.desiredPosition =
ccp(c.desiredPosition.x, c.desiredPosition.y + resolutionHeight);
                    } else {
                        //tile is diagonal, but resolving
horizontally
                        float resolutionWidth;
                        if (tileIndx == 6 || tileIndx == 4) {
                            resolutionWidth =
intersection.size.width;
                        } else {
                            resolutionWidth = -
intersection.size.width;
                        }
                        c.desiredPosition =
ccp(c.desiredPosition.x + resolutionWidth, c.desiredPosition.y);
                    }
                }
            }
        }
        //8
        c.position = c.desiredPosition;
    }
}
```

You start off by setting the character's `onGround` property to `NO`. Then each time the character has a tile under them (either adjacent or diagonally) you set `onGround` to `YES` and set the velocity to zero.

Also, if the character has an adjacent tile above them, you set their velocity to zero. This will make 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.

On more minor point, 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 his 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 up, you still resolve the collision in the same way, but you don't change his velocity or set the onGround flag.

If you build and run now, you should have a Cyclops on solid ground!

The collision detection engine for the game is now in place. This is the hardest part of making a platformer game. If you've made it this far, congratulations! That a big accomplishment, and a lot more fun awaits you.

**Challenge:** The current state of the physics engine gets a single tile coordinate for the center point of the player (it will be the same for enemies) and then gets the surrounding eight tiles.

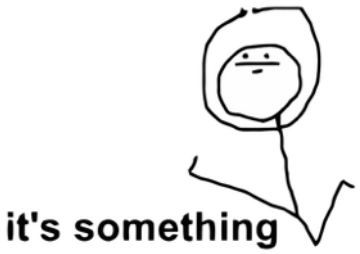
This structure will work as long as the collision bounding box is less than 2 x 2 tiles (64 x 64 pixels). If it were larger than that, you'd potentially have a collision in a tile that was farther than one tile width away.

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

Here's a hint, currently the collidable tiles surround the center tile, in the case of a tile larger than 2x2, there would be multiple tiles in the center, surrounded by a right (however large). Figure out which tiles are the outermost ring, alter the getSurroundingTilesAtPosition method to return that set.

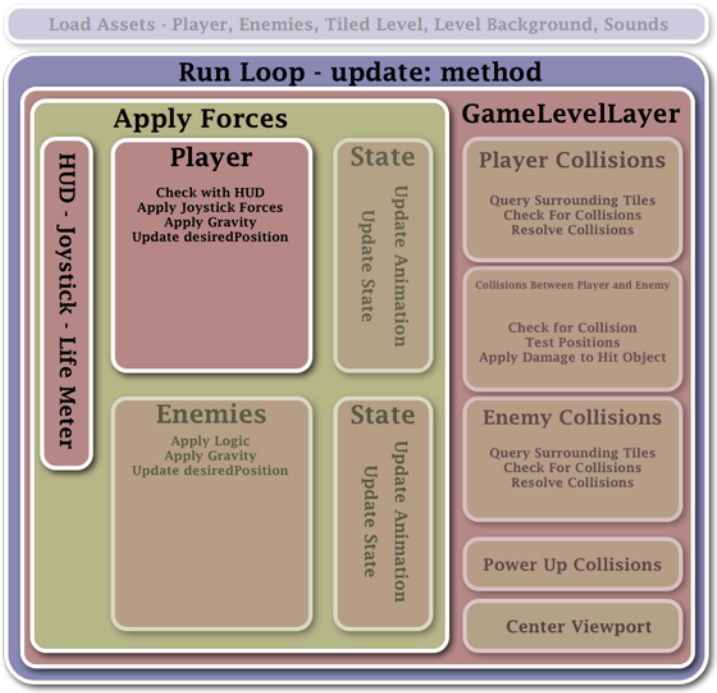
# Chapter 4: Moving the Player

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

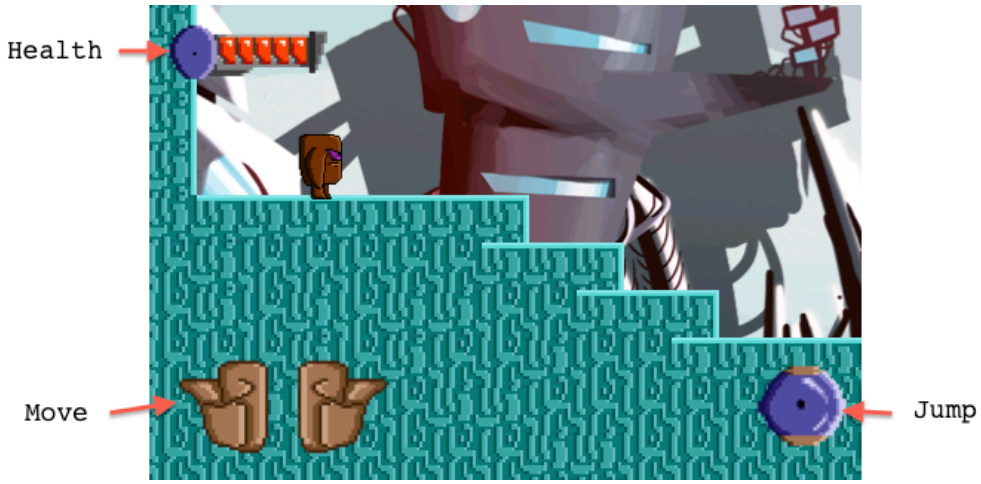



This will be a short and fun chapter, so limber up and get ready to do some coding!

Here's how this chapter fits into your overall game map (The HUD portion, plus the section of the `Player` class that queries the HUD state and applies physics forces):




# Head's Up!




I have already added the user interface for the HUD into the project for you. You can see the health bar in the upper left, the movement buttons in the lower left, and a jump button in the lower right.

The HUDLayer is a separate layer that is added to the scene on top of the `GameLevelLayer`. You can find the code for this inside **HUD\HUDLayer.m**. Take a few moments to look through the file. At this point, when you touch the buttons nothing happens except for updating the buttons to make them look like they are pressed.

Of course, you want to change this so when you press the buttons it moves the player around instead. But how do you make the player object do something, considering it's in a completely separate layer? Similarly, you'll sometimes need the player object tell the HUD to do something, such as updating the player's health display when the player is harmed.

There are lots of ways to get the HUD and the player talking to each other. One way would be to create delegates and set up the HUD to communicate button presses by sending messages, either to the `Player` class or to the `GameLevelLayer` class. You could also set the HUD as a delegate of those classes, and they could communicate back to the HUD when the player's life score changes.

Another approach would be to use the observer pattern by setting up `NSNotification`**s** 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

Later, you will indeed use `NSNotification`**s** to update the player's life indicator. For now, though, you'll focus just on implementing movement. To do this, you will query the state of the buttons from the player's `update:` method. If the buttons are pressed, you'll know it's time to move the player!

Here's why you'll have the player poll the joystick state (rather than the joystick notifying the player). In the next chapter, you'll explore the player's **state machine**, which will control whether the player is jumping or running. The animations that Cyclops displays at any point will depend on changes to the state machine.

> **Note**: A state machine is a system where the character can have one – and only one – state at a time. The state dictates the current visual representation (single frame or animation) of the player and provides information about which states are valid when transitioning from one state to another. 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 use input from the HUD in the state machine as well. What that means is that each 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 the HUD code is already written, you'll do most of this chapter's work within the `Player` class.

First you need an import for the `HUDLayer` class in **Player.m**:

```
#import "HUDLayer.h"
```

Next move to `update:` and add the following code before the existing code:

```
//1
HUDLayer *h = (HUDLayer *)[[[CCDirector sharedDirector]
runningScene] getChildByTag:25];
//2
joystickDirection jd = [h getJoystickDirection];
//3
CGPoint joyForce = ccp(0,0);

//4
if (jd == kJoyDirectionLeft) {
    self.flipX = YES;
    joyForce = ccp(-kWalkingSpeed, 0);
} else if (jd == kJoyDirectionRight) {
    self.flipX = NO;
    joyForce = ccp(kWalkingSpeed, 0);
}
//5
CGPoint joyForceStep = ccpMult(joyForce, dt);
//6
self.velocity = ccpAdd(self.velocity, joyForceStep);
```

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

1. The starter project adds the `HUDLayer` class to the scene with a tag of 25 to make it easy to get a pointer to it wherever you are in the code. The first step is to get a pointer to the HUD layer.
2. `HUDLayer` has a number of `enums` that describe the state of the buttons. `joystickDirection` is an `enum` that can be either `kJoyDirectionRight` or `kJoyDirectionLeft`. This step uses `getJoystickDirection` to retrieve that value. The HUD is set so that only one direction button can be engaged at a time, because it usually would make no sense to have both buttons pressed in the context of a platformer.
3. A new force variable is introduced and set to zero. This value will store a force value in the X dimension. You could use a `float` here, but in order to easily add the force later using `ccpAdd`, you use a `CGPoint` instead.
4. Next, the `if` statement sets `joyForce` to positive or negative `kWalkingSpeed` if either of the buttons is pressed. If neither is pressed, it simply leaves the value at zero. `kWalkingSpeed` is a constant that you'll add in just a minute.

   This section also sets the `flipX` variable. Every `CCSprite` has a `flipX` variable that can be used to horizontally flip the image. This makes coding certain parts easier because you don't need both a right- and left-facing image. Here `flipX` is set depending on the direction, so that the player is facing the right way.

5. The `joyForce` value is scaled 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.

6. Finally, the scaled value is added to the current velocity.

The only thing left to is to define the `kWalkingSpeed` value. Add this line to **Player.h**:

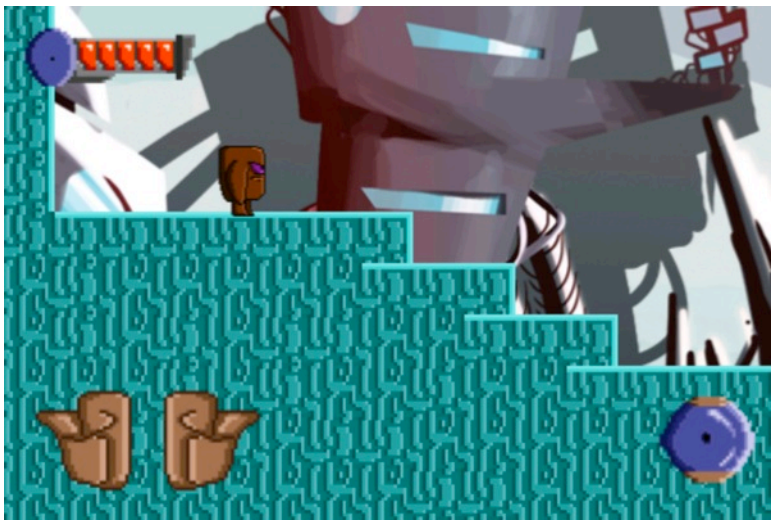
```
#define kWalkingSpeed 1600
```

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

If the player was at a stand still, after 1/60 of a second, his speed is 27 pixels per second. After the second time step, 54 pixels per second, etc. That value (27 / 54 / 71 etc.) is then scaled again by 1/60 to find how much he actually moved that time step (the first time step only moves him half a pixel!) Increasing this value decreases the time it takes him to get to his top speed, and you want him to reach top speed in less than a second. So it needs to be a pretty big value.

That's long enough to make it feel like his a physical object, because real objects don't go from zero to full speed instantly, but do so over time. That feels like momentum. 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. That just feels better.

Build and run now. Then tap the right button a couple of times and see what happens:



Once you tap the button, you'll see the Cyclops move forward like he has ice skates.

This looks a bit weird because he keeps moving even if you're no longer holding down the button. This is because nothing is slowing him down. As you'll recall, your physics simulation requires the application of force to change momentum. Once there's force involved, the momentum causes continued movement until a counter force is applied.

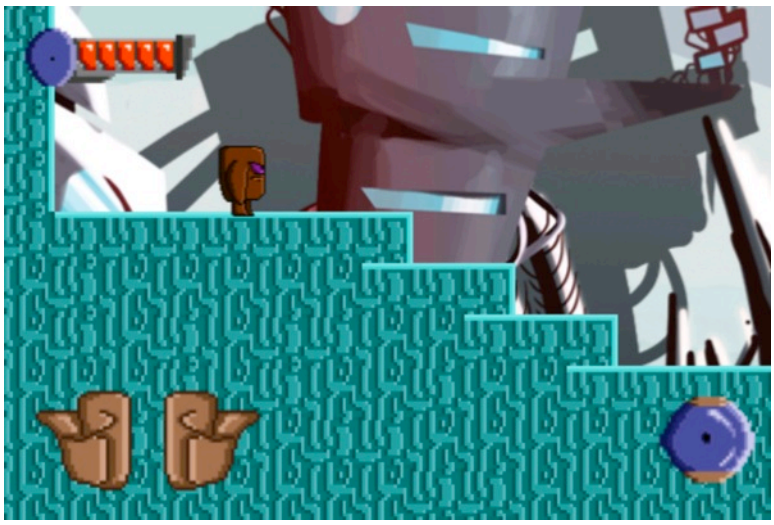
To make things more realistic, you need to introduce friction into the equation. This is a value that you'll call damping, something less than 1.0, that will be multiplied by the resulting velocity every frame. Add the following `#define` to **Player.h**:

```
#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 sets `self.desiredPosition`:

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

Build and run. Now when you tap the joystick Cyclops will move and then have a short slide to a stop. Hey, this is beginning to feel like a platformer!



If you hold down the button he'll accelerate, 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 will mean that it will take longer for Cyclops to come to a stop on his own. This might be appropriate for slippery surfaces, but remember that the acceleration will accumulate much more quickly with a higher `kWalkingSpeed` value.

You also want to set a maximum speed. These three values – `kWalkingSpeed`, `kDamping` and `kMaxSpeed` – will control how quickly Cyclops reaches his top speed, what the top speed is and how quickly he stops on his own (assuming that you don't press the opposite button).

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

```
#define kMaxSpeed 250
```

Then add this line before the final line updating `desiredPosition` in `update:` in **Player.m**:

```
self.velocity = ccpClamp(self.velocity,
    ccp(-kMaxSpeed, -kMaxSpeed), ccp(kMaxSpeed, kMaxSpeed));
```

The `ccpClamp` 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 min point of -250, -250 and a max of 250, 250.

So, why is the `kWalkingSpeed` 1600 if the `kMaxSpeed` is 250? Remember that the `kWalkingSpeed` is scaled to the time step, so for each step it adds 1/60$^{th}$ (about 26) to the velocity value. This means it will reach the max (not taking into account the damping, which slows it down a bit) in less than a second, which is what you want.

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

Damping creates a practical upper speed limit (add 26 to 250 and multiply by .85 to end up with 234). So when you build and run, the set max values won't actually be reached.

However, as you tweak the feel of your game, it's good to include this upper bound in order to have the maximum amount of control. Also, remember that your update method isn't always 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 occurs, your delta time (dt) value could be very large, which would greatly increase the value added to the previous velocity, and without the 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. ☺

# Pocket paparazzi

This game is starting to turn out nicely, but it is a bit disappointing because it doesn't take long for Cyclops to disappear off the right side of the screen.

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

```
-(void)setViewpointCenter:(CGPoint) position {

    CGSize winSize = [[CCDirector sharedDirector] winSize];

    int x = MAX(position.x, winSize.width / 2);
    int y = MAX(position.y, winSize.height / 2);
    x = MIN(x, (map.mapSize.width * map.tileSize.width)
            - winSize.width / 2);
    y = MIN(y, (map.mapSize.height * map.tileSize.height)
            - winSize.height/2);
    CGPoint actualPosition = ccp(x, y);
```

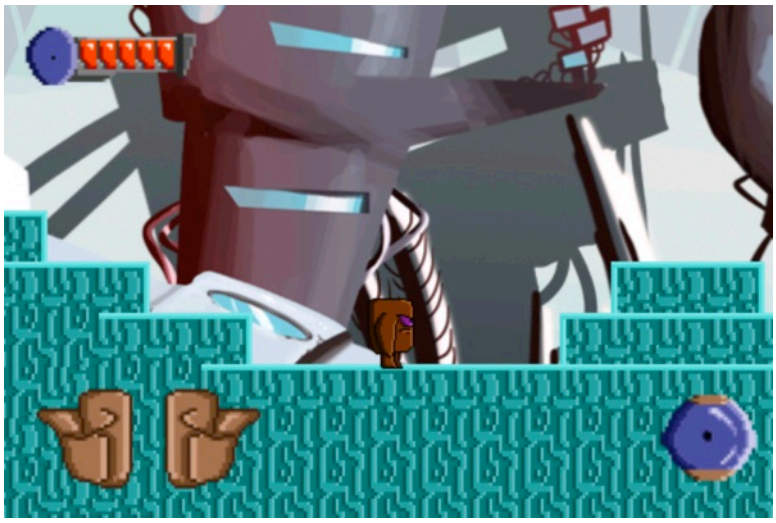
```
    CGPoint centerOfView = ccp(winSize.width/2,
      winSize.height/2);
    CGPoint viewPoint = ccpSub(centerOfView, actualPosition);
    self.position = viewPoint;

}
```

This code is 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

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

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

This will be a big payoff for a tiny amount of work, so build and run to check it out!



Now the camera will follow the player, and you'll also get to enjoy the parallax scrolling action you set up in Chapter 1.

# The joy of jumping

The Cyclops can now move around quite well. But once he hits a valley, he's not able to go anywhere since he can't get up those peaks. ☹ Time to make your Cyclops jump!

The jump is the distinguishing feature of the platformer and the element that leads to most of the fun. You want to make sure that the jumping movement is fluid and

feels right. In this tutorial, you'll implement the jump algorithm used in *Sonic the Hedgehog*, as described here: http://info.sonicretro.org/Sonic_Physics_Guide

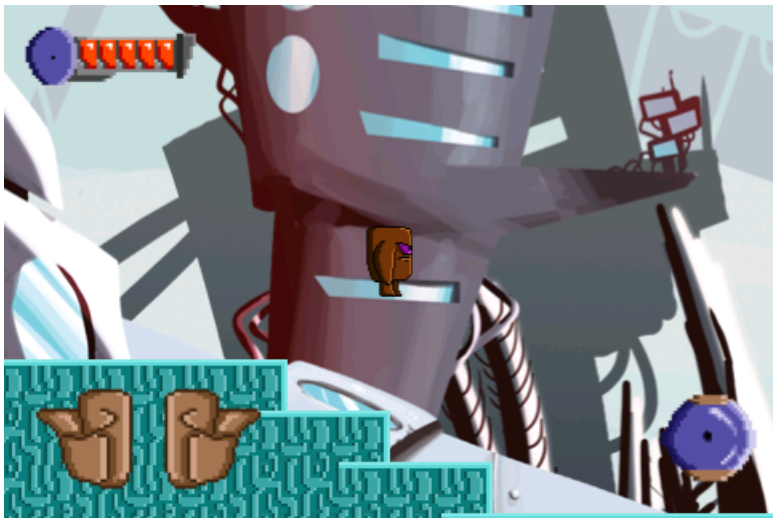
First add the following constant to **Player.h**:

```
#define kJumpForce 400
```

Next, in **Player.m**, add this code to `update:` immediately after the line `self.velocity = ccpAdd(self.velocity, joyForceStep);`:

```
jumpButtonState js = [h getJumpButtonState];

if (js == kJumpButtonOn) {
    if (self.onGround) {
        self.velocity = ccp(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 and wait until gravity pulls him back down again.

In modern platform games, users expect much finer control over the jump action. You want controllable, completely unrealistic (but fun as hell) *Mario Brothers* or *Sonic* style jumping where you 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'll 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:

```
jumpButtonState js = [h getJumpButtonState];

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

Since you're relying on a new constant, `kJumpCutoff`, you need to add it to **Player.h** 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 – but on your device this time instead of the simulator. It will be helpful to run on a device from now on so it's easy to use both the movemenet and jump buttons at the same time.

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.

Speaking of testing devices – it's worth pointing out that this project comes with iCade support. I won't be covering iCade integration in this tutorial, but you can read all about it in the following tutorial:

http://www.raywenderlich.com/8618/adding-icade-support-to-your-game

That tutorial was written based on the code I wrote when I added iCade support for this tutorial, so the two ought to be easy to understand together.

If you have an iCade, fire it up and play through the level!

If you are old enough to remember the glory days of the arcades, doesn't this bring back great nostalgia? ☺

# Kangaroo or Cyclops?

Notice that the only precondition for the jump action is that `self.onGround` is set to `YES` (and, of course, that the jump button is pressed). This means that if you hold down the jump button, Cyclops will jump as soon as he 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, and some very successful games do display this behavior. But, being the perfectionist you are, you're going to fix it. ☺

The expectation is that in order to complete a new jump, you need to release the button before pressing it again. This will let you press down on the button to jump, and keep it pressed until you want to jump again. A new jump requires that you release the button and press it again.

You need a new Boolean variable that will record the state of the jump button. Call it `jumpReset`. This variable will keep track of whether the jump button was released in between jumps and will prevent a new jump until the button is released.

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

```
@interface Player () {
    BOOL jumpReset;
}
@end
```

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

Now set the `jumpReset` flag in `initWithSpriteFameName:` (this can go after the line that sets `self.velocity`):

```
jumpReset = YES;
```

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

```
jumpButtonState js = [h getJumpButtonState];

if (js == kJumpButtonOn) {
    //1
    if (self.onGround && jumpReset) {
        self.velocity = ccp(self.velocity.x, kJumpForce);
        //2
        jumpReset = NO;
    }
} else {
    if (self.velocity.y > kJumpCutoff) {
        self.velocity = ccp(self.velocity.x, kJumpCutoff);
    }
    //3
    jumpReset = YES;
}
```

There are three changes to the original code:

1. You now check that both `self.onGround` and `jumpReset` are set to `YES`.
2. After the jump force is applied to `self.velocity`, `jumpReset` is set to `NO`. This prevents the jump force from being applied again until `jumpReset` is set back to `YES`. You only set this to `NO` when the jump force is actually applied. If you continuously press the jump button, `jumpReset` will be set to `NO` only in the very first frame, when the player first jumps off the ground.

3. In the `else` block, which is triggered if the jump button isn't pressed, the `jumpButton` is set back to `YES`.

Build and run again. You should see that the kangaroo jump behavior is gone, giving users better control of 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. Pat yourself on the back – you deserve it. ☺

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

But some games allow the player to "float" at the top of a jump for just a little bit. Games with that sort of jump feel more forgiving and allow the player to get up and around platforms more easily.

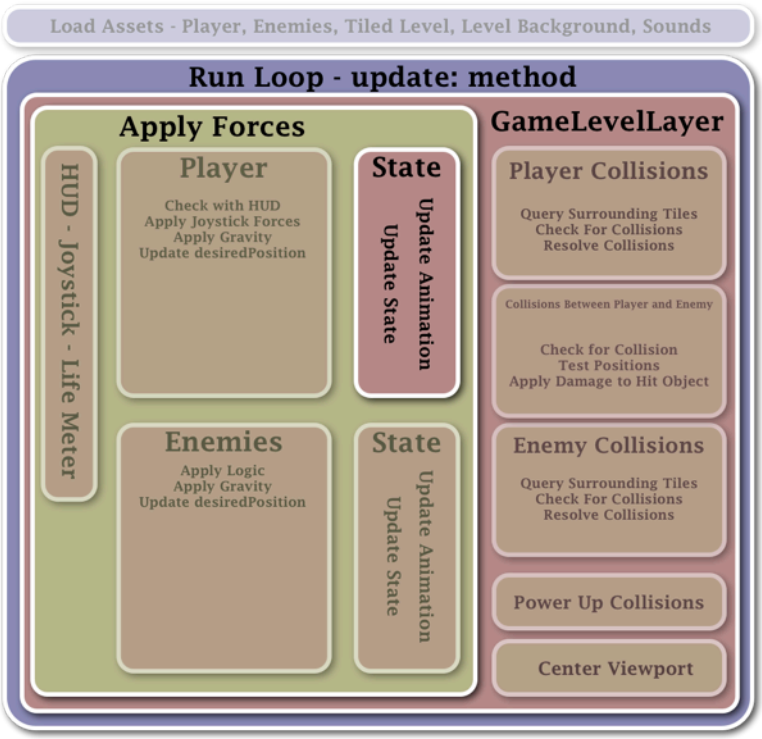
How would you go about implementing that behavior in this game?

# Chapter 5: State Machine

The focus of this chapter is the all-important state machine. Many games, including this one, use a state machine to control the appearance and behavior of its characters.

Your state machine will control which character animation is playing at any given time (e.g., jumping or walking) and will help the game logic determine which character actions are valid, when. For example, Cyclops can't jump from a falling state, can't wall slide if touching the ground, etc.

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



# What is a state machine?

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

- 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 in order to transition into certain states. The current state will help determine which tests should be run to change states.

For example, one of the upgrades that Cyclops will obtain during the course of the game is the wall slide ability, which lets you slowly slide down along vertical walls instead of dropping rapidly like you usually would.

In order to transition into the wall slide state, there are three conditions:

1. Cyclops needs to be falling;
2. The user needs to be pressing the direction button towards the wall;
3. Cyclops needs to be colliding with the wall.

Thus, Cyclops needs to be in a falling, jumping, or double-jumping state (but falling down, rather than shooting up), and he needs to be pushing into the walls.

Second, the state machine will determine the current running animation for a character. You'll have different animations for jumping, running, wall sliding, etc. When a state change is triggered, the game will load and run a new animation or 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 more complex than what a state machine can easily accommodate. However, the state machine is an easy way to organize and track character state, and it'll serve *Pocket Cyclops* just fine.

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

# Calling all states!

Because enemies as well as Cyclops will rely on a state machine, you'll add the code that handles state change and definition to the `Character` 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.

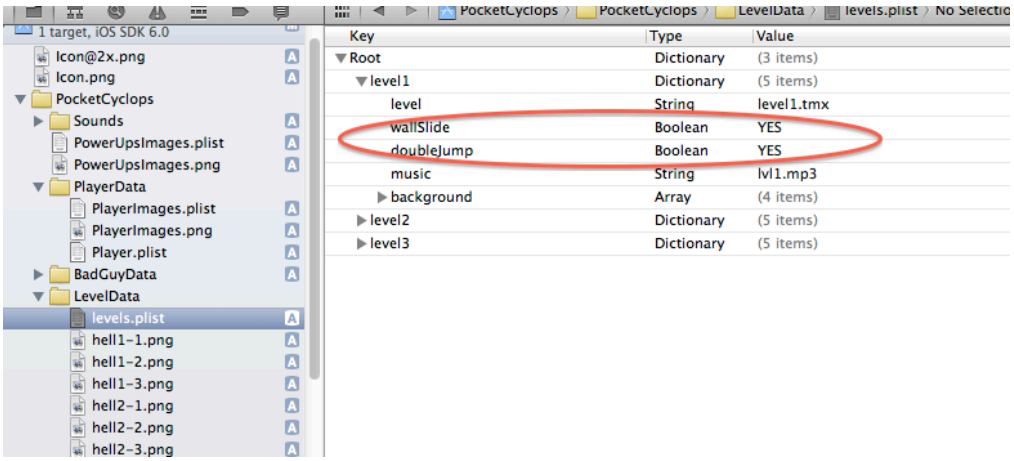
You need at least eight states for the `player` class (the enemies will add a few more). Add them as an `enum` by inserting the following code after the `#import` statement in **Character.h**:

```
typedef enum {
    kStateJumping,
    kStateDoubleJumping,
    kStateWalking,
    kStateStanding,
    kStateDying,
    kStateFalling,
    kStateDead,
    kStateWallSliding
} CharacterStates;
```

At any given time Cyclops will be in one and only one of these states. You can use the present state to determine if you can and should transition to another state. For example, in order to enter the double-jumping state, Cyclops will have to already be in the jumping state.

In order to easily test and write the state machine, you are going to turn on all of the upgrades for the first level. There's a **levels.plist** file that sets up the initial state of the player and the level. Inside this file, there are two Booleans for the power-ups.

Select the **levels.plist** file (it's inside the LevelData folder) in the Project Navigator. 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 Cyclops a free pass! In a later chapter, you'll make Cyclops earn these abilities by obtaining the relevant power-up objects.

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

```
@property (nonatomic, assign) CharacterStates characterState;
-(void)changeState:(CharacterStates)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 stub (which will be overridden in subclasses) to **Character.m**:

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

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

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

The method first checks to see if the `newState` is equal to the current state of the player, since there's nothing to do if the state has not changed. Then you log the new state and set the player state to `newState`.

That's it for now. Later you'll change the character animation or the displayed sprite, depending on the 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 state for walking, standing, and jumping.

First add the following line to the beginning of `update:` (still in **Player.m**):

```
CharacterStates newState = self.characterState;
```

Now, in the block that sets the jump force, find the line that sets `jumpReset` to `NO` and add these lines immediately after it:

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

Then add this code before the line that declares the `gravity` `CGPoint`:

```
if (self.onGround && jd == kJoyDirectionNone) {
    newState = kStateStanding;
} else if (self.onGround && jd != kJoyDirectionNone) {
```

```
    newState = kStateWalking;
}
[self changeState:newState];
```

The first step creates the `newState` variable. There will be occasions when you will set the `newState` variable multiple times in the course of a single update loop.

In order to avoid calling `changeState:` multiple times, you create a variable and run through all your tests, changing the variable as you go. Once you're done with the tests, you call `changeState:` to set the final state value.

In the case of these three state tests, they are all mutually exclusive and the value 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 of a test, and more a matter of simply setting the state when the jump force is first applied. The jumping state starts with the application of a jump force and continues until Cyclops hits the ground again. The state won't be overridden by later tests as long as `onGround` is `NO`.

Why do you set `self.onGround` to `NO` after you've applied a jump force? What do you think?

Remember how the physics engine works. 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 jumping force to Cyclops, his `velocity` property is set, ensuring that he'll move upwards. But this won't actually happen until the next loop of the `GameLevelLayer`'s `update:` method. So, immediately after the jump is set within the `Player`'s `update:` method, `onGround` is still set to `YES`.

This means that when you get to the last tests, which look for `onGround` and set the walking or standing states, `onGround` will still be set to `YES` even though the player might actually be jumping. That's why you set `onGround` to `NO` – to ensure that when you get to the standing or walking tests, they don't trigger because `onGround` is still set to `YES`.

The later tests check for `onGround` and whether the character is being directed left or right, and choose either the walking or standing states. The way it is set up, because of the momentum created in the physics engine, you'll move from walking to standing when the user stops pressing the button.

However, this may or may not be the actual moment Cyclops stops moving in that direction. Depending on the physics values you have set, Cyclops may still be sliding left, but the walking animation will stop to be replaced with the standing animation.

This is what you want in most cases. The alternative is that the user stops pressing a button and, because of momentum, the walking animation continues to play, which would look a bit strange because the user is no longer pressing the button.

Build and run your game. If you walk or jump around, you'll the console reporting the changing state value. These values are `enums`, so they are being represented as integer values that simply indicate their location in the `enum` list (the first state, `kStateJumping`, is zero).

```
2012-11-14 10:34:42.461 PocketCyclops[32487:c07] cocos2d: **** WARNING **** CC_ENABLE_GL_STATE_CACHE is disabled. To
improve performance, enable it by editing ccConfig.h

AudioStreamBasicDescription:  2 ch,  44100 Hz, 'lpcm' (0x00000029) 32-bit little-endian float, deinterleaved
2012-11-14 10:34:43.805 PocketCyclops[32487:c07] Change State 3
2012-11-14 10:34:45.539 PocketCyclops[32487:c07] Change State 0
2012-11-14 10:34:46.422 PocketCyclops[32487:c07] Change State 3
2012-11-14 10:34:46.739 PocketCyclops[32487:c07] Change State 0
2012-11-14 10:34:47.872 PocketCyclops[32487:c07] Change State 3
2012-11-14 10:34:48.306 PocketCyclops[32487:c07] Change State 2
2012-11-14 10:34:50.022 PocketCyclops[32487:c07] Change State 3
```

# The wall slide

At long last it's time to add the states for wall sliding and the double jump! These states make the state machine tests significantly trickier. First you'll tackle the test for wall sliding.

Here are the conditions under which wall sliding can occur:

1. **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 Cyclops into a wall. Falling downward next to a wall shouldn't be enough.

Let's give this a shot!

## Up against the wall?

The first thing you need is a new property in **Character.h**:

```
@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'll need to set it up in `checkForAndResolveCollisions:` in `GameLevelLayer`.

Do you remember this diagram from the chapter on collision detection?

Re-ordered

| | | |
|---|---|---|
| 5(4) | 2(1) | 6(5) |
| 3(2) | | 4(3) |
| 7(6) | 1(0) | 8(7) |

The diagram shows the order of the tiles in the collision detection array. The index (shown in parentheses) of the tile is one less than its place, because 0 is the first index in the array.

Therefore, you want to find those places where indices 2 (tile 3, left-middle), 3 (tile 4, right-middle), 6 (tile 7, left-diagonal-down), and 7 (tile 8, right-diagonal-down) are resolved. Those are the places where there's a possible collision with the wall.

You are going to set `onWall` to `YES` when there's a collision resolved in tiles 3 and 7 to the left of the player and 4 and 8 to the right. You won't count collisions resolved with tiles 5 and 6.

Why not? You don't want the wall slide to occur if Cyclops is only hanging on by the corner of a diagonal tile above him. He needs more of a surface for a successful slide. This choice is based on my design preference. You can include those tiles in a wall slide if you like. ☺

So the task is now to find the places in `checkForAndResolveCollisions:` 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.

In **GameLevelLayer.m**, find the line at the beginning of `checkForAndResolveCollisions:` where `c.onGround = NO;` and add this line right after it:

```
c.onWall = NO;
```

Now you'll find all the places in the code where any of the tiles with indices 2, 3, 6 or 7 are resolved, and add the line that sets `c.onWall` to `YES` and set `c.velocity.x` to `0.0`.

To do this, add the following code to the `if` blocks where `tileIndx` is 2 or 3 (add it after the line that sets `c.desiredPosition`):

```
c.onWall = YES;
```

```
c.velocity = ccp(0.0, c.velocity.y);
```

In the case of tile indices 6 or 7, there are two scenarios: a horizontal resolution and a vertical one. You only want to set `onWall` if the resolution is horizontal, so you need to find the place in the code where this happens.

Right after the last instance of this line:

```
c.desiredPosition = ccp(c.desiredPosition.x + resolutionWidth,
c.desiredPosition.y);
```

Add this:

```
if (tileIndx == 6 || tileIndx == 7) {
    c.onWall = YES;
}
c.velocity = ccp(0.0, c.velocity.y);
```

In case the above directions were hard to follow, here's the whole method with the modifications marked by comments:

```
-(void)checkForAndResolveCollisions:(Character *)c {

    NSArray *tiles = [map getSurroundingTilesAtPosition:c.position
forLayer:walls];
    c.onGround = NO;
    c.onWall = NO; //////Here

    for (NSDictionary *dic in tiles) {

        CGRect pRect = [c collisionBoundingBox];
        int gid = [[dic objectForKey:@"gid"] intValue];

        if (gid) {
            CGRect tileRect = CGRectMake([[dic objectForKey:@"x"]
floatValue], [[dic objectForKey:@"y"] floatValue],
map.tileSize.width, map.tileSize.height);
            if (CGRectIntersectsRect(pRect, tileRect)) {

                CGRect intersection = CGRectIntersection(pRect,
tileRect);

                int tileIndx = [tiles indexOfObject:dic];

                if (tileIndx == 0) {
                    //tile is directly below the Character
```

```
                    c.desiredPosition = ccp(c.desiredPosition.x,
c.desiredPosition.y + intersection.size.height);
                    c.velocity = ccp(c.velocity.x, 0.0);
                    c.onGround = YES;
                } else if (tileIndx == 1) {
                    //tile is directly above the Character
                    c.desiredPosition = ccp(c.desiredPosition.x,
c.desiredPosition.y - intersection.size.height);
                    c.velocity = ccp(c.velocity.x, 0.0);

                } else if (tileIndx == 2) {
                    //tile is left of the Character
                    c.desiredPosition = ccp(c.desiredPosition.x +
intersection.size.width, c.desiredPosition.y);
                    c.onWall = YES; //////Here
                    c.velocity = ccp(0.0, c.velocity.y);
//////Here
                } else if (tileIndx == 3) {
                    //tile is right of the Character
                    c.desiredPosition = ccp(c.desiredPosition.x -
intersection.size.width, c.desiredPosition.y);
                    c.onWall = YES; //////Here
                    c.velocity = ccp(0.0, c.velocity.y);
//////Here

                } else {
                    if (intersection.size.width >
intersection.size.height) {

                        //tile is diagonal, but resolving
collision vertically
                        c.velocity = ccp(c.velocity.x, 0.0);

                        float resolutionHeight;
                        if (tileIndx > 5) {
                            resolutionHeight =
intersection.size.height;
                            c.onGround = YES;
                        } else {
                            resolutionHeight = -
intersection.size.height;
                        }
                        c.desiredPosition =
ccp(c.desiredPosition.x, c.desiredPosition.y + resolutionHeight);
                    } else {
```

```
                        //tile is diagonal, but resolving
horizontally
                        float resolutionWidth;
                        if (tileIndx == 6 || tileIndx == 4) {
                            resolutionWidth =
intersection.size.width;
                        } else {
                            resolutionWidth = -
intersection.size.width;
                        }
                        c.desiredPosition =
ccp(c.desiredPosition.x + resolutionWidth, c.desiredPosition.y);

                        //////Here
                        if (tileIndx == 6 || tileIndx == 7) {
                            c.onWall = YES;
                        }
                        c.velocity = ccp(0.0, c.velocity.y);

                    }
                }

            }
        }

        c.position = c.desiredPosition;
    }
}
```

Notice the last line in the last block of code added – it sets the player's x velocity to zero. The character has a small amount of momentum as the x velocity decays. He won't move perceptibly, but he'll still be colliding with the block next to him. This can last almost a full second, depending on the physics values you set.

This means that for a short time after the user stops pressing towards the wall, Cyclops will still be colliding and triggering a wall slide. This is unacceptable, as the user will expect wall sliding to stop immediately after he stops pressing towards the wall. Zeroing the x velocity fixes this and is more accurate, as Cyclops has stopped moving.

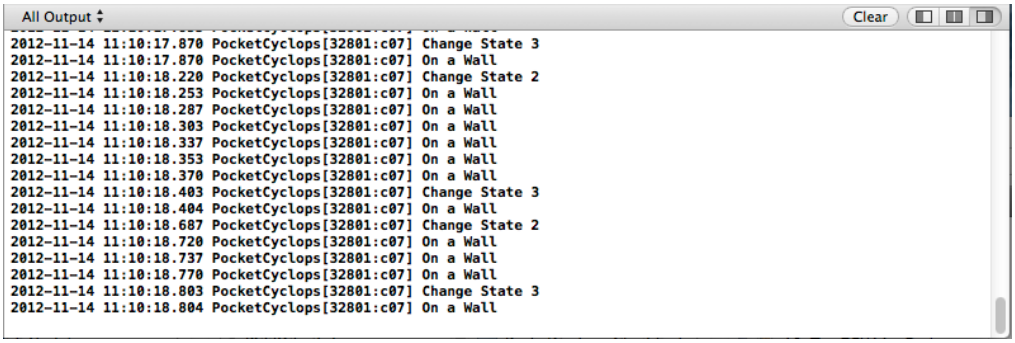
Now you can return to the `Player` class `update:` method and take advantage of this new information that tells you whether a wall collision is occurring. First just dip your toe in the waters by testing for the value.

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 Cyclops into a wall. The Xcode console should show that you are colliding with a wall, but only for as long as you press him into one.




If, when you press Cyclops into a wall, you have state change and "On a Wall" notifications in your console (make sure that the wall notifications stop immediately when you release the button), you have done everything right. You are ready to move on to the state change test.

Now return to `update:` in **Player.m**, and modify the code that tests for the walking and standing states to the following:

```
if (self.onGround && jd == kJoyDirectionNone) {
    newState = kStateStanding;
} else if (self.onGround && jd != kJoyDirectionNone) {
    newState = kStateWalking;
} else if (self.onWall && self.velocity.y < 0) { ////Here
    newState = kStateWallSliding;
}
```

The third test that you just added (the one marked by ////Here above) will transition Cyclops to a wall sliding state if `onWall` is set to `YES` and if his velocity is negative (Cyclops is falling).

You don't need to test for `self.onGround`. If `self.onGround` had been `YES`, the execution would never reach that third test – it would have triggered a true condition and executed one of the previous blocks. The fact that the third `if else` test 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. You 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 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 `update:` in **Player.m**, find the `if` statement that begins the jumping block, the one that comes after the `if (js == kJumpButtonOn)` line. Alter it thus:

```
if ((self.onGround || self.characterState == kStateWallSliding) &&
jumpReset) {
```

This just updates the jump code so 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 Cyclops is falling.

Build and run. You should now be able to make Cyclops jump off the 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, after '`jumpReset = NO;`' add this code:

```
if (self.characterState == kStateWallSliding) {
     int direction = -1;
     if (self.flipX) {
          direction = 1;
     }
     self.velocity = ccp(direction * kJumpOut, self.velocity.y);
}
```

You check to see if you're jumping off the wall and if you are, you add some force to push the player away from the wall.

You also initialize a new `direction` variable to -1. You then check `flipX` for the player. As mentioned before, `CCSprite` has as `flipX` property that allows you to flip

it so that you don't need a different sprite for a left-facing versus a right-facing image.

Here you simply check the state of `flipX` in order to determine whether you're against a right or left wall. Since you must be pressing against a wall to be wall sliding, the direction that the player is facing will determine the state of the `flipX` variable, which then tells you in which direction is the wall.

Now add the `#define` statement for the new constant to **Player.h**:

```
#define kJumpOut 360
```

Build and run again. Cyclops should be pushed away from the wall when wall sliding and jumping like any decent wall sliding jumper.

The other thing that wall sliding should do is slow the descent of the player. In order to do that, you want to check the velocity of the player after all the other velocity calculations are complete.

Add a new `#define` to **Player.h**:

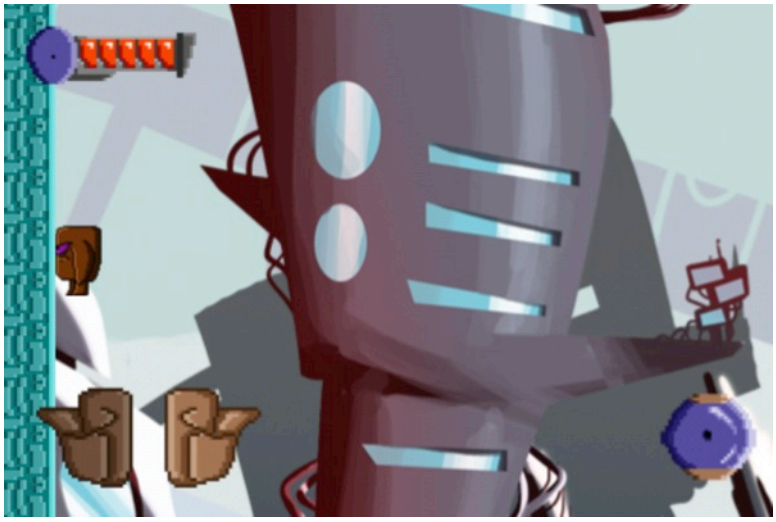
```
#define kWallSlideSpeed -30
```

This value is the maximum speed per second that Cyclops will fall during a wall slide. Now add this code to `update:` in **Player.m**, right before the last line, the line that sets `desiredPosition` (if you still have the `onWall` logging as the last bit of code in `update:`, remove it):

```
if (self.characterState == kStateWallSliding) {
    float fallingSpeed = clampf(self.velocity.y, kWallSlideSpeed,
0);
    self.velocity = ccp(self.velocity.x, fallingSpeed);
}
```

This code checks if the player is wall sliding and applies a clamp to the y component of the velocity variable, setting the upper bound to zero. If the player 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 function requires a max, and that's why zero was selected!

If you build and run now, you should have a Cyclops who slowly slides down walls:



Notice that you still have one little problem. If Cyclops starts wall sliding and then moves in the other direction (letting go of the wall), he still falls slowly. What's going on here?

It's because there's nothing in your code to take Cyclops out of his wall sliding state. Once a wall slider, always a wall slider, as they say… (What, they don't say that?)

You need to beef up the test at the end of `update:` (in **Player.m**), so that if the user stops pressing Cyclops into a wall, he moves back to the falling.

In `update:` add an `else` to the end of the `if` section that sets the `newState` variable, so that it looks like this (this block is before the line `[self changeState:newState];`):

```
if (self.onGround && jd == kJoyDirectionNone) {
    newState = kStateStanding;
} else if (self.onGround && jd != kJoyDirectionNone) {
    newState = kStateWalking;
} else if (self.onWall && self.velocity.y < 0 &&
self.canWallSlide) {
    newState = kStateWallSliding;
    ////Here
} else if ( self.characterState == kStateDoubleJumping || newState
== kStateDoubleJumping) {
    newState = kStateDoubleJumping;
```

```
} else if ( self.characterState == kStateJumping || newState ==
kStateJumping){
    newState = kStateJumping;
} else {
    newState = kStateFalling;
}
```

Now, no matter what else has happened, this section will set `newState` to one of the five states for the player.

I've added handling the kStateDoubleJumping in this block. I'll show you how to handle entering that state in the very next section.

Double-check your logic. What conditions are left in that last, catch-all `else` block?

Cyclops cannot be on the ground. Cyclops could be `onWall` and moving upward, in which case jumping is the correct state. At this point, you check if the state set in the last update loop, or if the new state set in the block handling user input (pressing the jump button), is either the jumping or double jumping states.

The only thing that takes the player out of the jumping or double jumping states is landing on the ground or pressing against a wall (or pressing jump to move from jumping to double jumping). If the player was in one of those state previously, or if the jump button has been pressed, 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's velocity changes (when he stops moving upwards). You could easily add that test here. However, 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 will occur when either, 1) the Cyclops walks of a ledge, or 2) when he exits the wall sliding state. That second option is the bug we're trying to fix!

This stuff gets complex quickly!

Build and run now, and 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 Cyclops the ability to jump a second time from mid-air.

The double jump requires two conditions:

1. The current player state 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 will check if Cyclops is already in the jumping state. If he is, it will put him into a double-jump state.

This new test will come first and the existing code will become an `else if` part of the same `if` statement. It should look like the following (this goes right after the `if (js == kJumpButtonOn) {` line in `update:` in **Player.m**):

```
if ((self.characterState == kStateJumping || self.characterState
== kStateFalling) && jumpReset) {
    self.velocity = ccp(self.velocity.x, kJumpForce);
    jumpReset = NO;
    newState = kStateDoubleJumping;
} else if ((self.onGround || self.characterState ==
kStateWallSliding) && jumpReset) { //Old if statement
```

This code (inside the `if` block) is identical to the other jumping code. The only difference is that you are setting the state to double jump. You can see that you are able to enter the double jump state from either a jumping state or a falling sate. This means that if the Cyclops walks off a ledge, he can still do a mid-air jump.

It's going to be a little tricky to determine whether 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 on.

Refer back to the section that tests for state and sets the `newState` variable. It should look like this:

```
if (self.onGround && jd == kJoyDirectionNone) {
    newState = kStateStanding;
} else if (self.onGround && jd != kJoyDirectionNone) {
    newState = kStateWalking;
} else if (self.onWall && self.velocity.y < 0 &&
self.canWallSlide) {
    newState = kStateWallSliding;
    ////Here
} 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 the `newState` variable is set the first time you enter 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 Cyclops has jumped once or twice. That's why this requires a little more maintenance. You could achieve the same result by creating a previous state variable that gets set when the state changes.

One side effect of this test, and it's a desired effect, is that if Cyclops transitions to a wall sliding state, and then back to a jumping state, he can then double jump again. So by touching a wall, the player gets back their second jump.

This is what you want for this game, but it's worth pointing out that, if you didn't want the game to behave this way, you would need to create a Boolean to keep track of whether another double jump was available or not. You could, for example, give the double jump back to the player only after they touch 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:(ccTime)dt
{
    CharacterStates newState = self.characterState;

    HUDLayer *h = (HUDLayer *)[[[CCDirector sharedDirector]
runningScene] getChildByTag:25];

    joystickDirection jd = [h getJoystickDirection];

    CGPoint joyForce = ccp(0,0);

    if (jd == kJoyDirectionLeft) {
        self.flipX = YES;
        joyForce = ccp(-kWalkingSpeed, 0);
    } else if (jd == kJoyDirectionRight) {
        self.flipX = NO;
        joyForce = ccp(kWalkingSpeed, 0);
    }


    CGPoint joyForceStep = ccpMult(joyForce, dt);

    self.velocity = ccpAdd(self.velocity, joyForceStep);

    jumpButtonState js = [h getJumpButtonState];

    if (js == kJumpButtonOn) {

        if (self.characterState == kStateJumping && jumpReset) {
            self.velocity = ccp(self.velocity.x, kJumpForce);
            jumpReset = NO;
            newState = kStateDoubleJumping;
        } else if ((self.onGround || self.characterState ==
kStateWallSliding) && jumpReset) {
            self.velocity = ccp(self.velocity.x, kJumpForce);

            jumpReset = NO;
            if (self.characterState == kStateWallSliding) {
                int direction = -1;
                if (self.flipX) {
                    direction = 1;
                }
                self.velocity = ccp(direction * kJumpOut,
self.velocity.y);
            }
```

```
            self.onGround = NO;
        }
    } else {
        if (self.velocity.y > kJumpCutoff) {
            self.velocity = ccp(self.velocity.x, kJumpCutoff);
        }

        jumpReset = YES;
    }

    if (self.onGround && jd == kJoyDirectionNone) {
        newState = kStateStanding;
    } else if (self.onGround && jd != kJoyDirectionNone) {
        newState = kStateWalking;
    } else if (self.onWall && self.velocity.y < 0 &&
self.canWallSlide) {
        newState = kStateWallSliding;
        ////Here
    } 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 = ccp(0.0, -450.0);

    CGPoint gravityStep = ccpMult(gravity, dt);

    self.velocity = ccpAdd(self.velocity, gravityStep);
    CGPoint stepVelocity = ccpMult(self.velocity, dt);
    self.velocity = ccp(self.velocity.x * kDamping,
self.velocity.y);

    self.velocity = ccpClamp(self.velocity, ccp(-kMaxSpeed, -
kMaxSpeed), ccp(kMaxSpeed, kMaxSpeed));

    if (self.characterState == kStateWallSliding) {
```

```
        float fallingSpeed = clampf(self.velocity.y,
kWallSlideSpeed, 0);
        self.velocity = ccp(self.velocity.x, fallingSpeed);
    }

    self.desiredPosition = ccpAdd(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 player is fully powered-up. Cyclops can wall slide and double jump to his 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. ☺

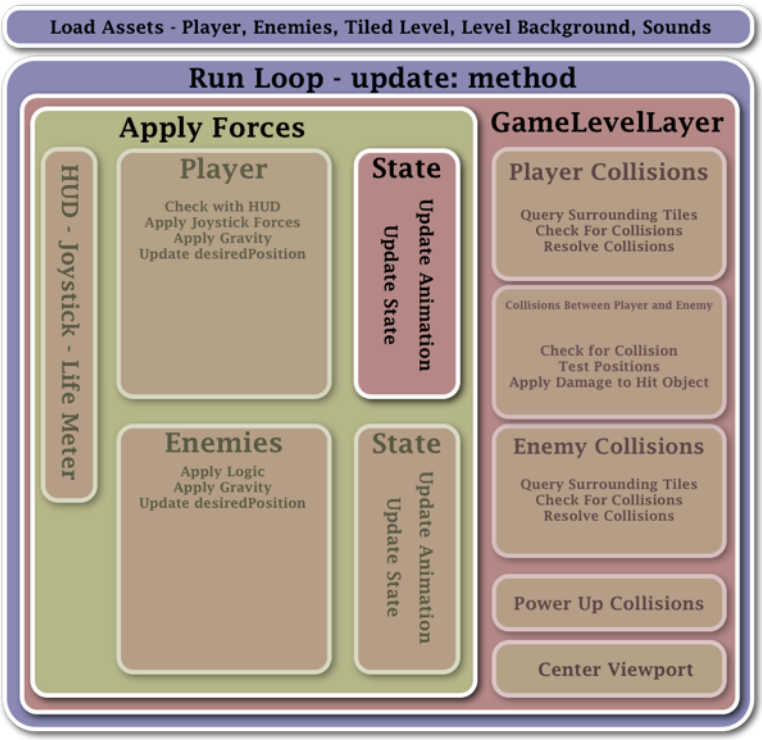
But don't think you're done with the state machine! It will make plenty of appearances in later chapters, as you add character animations, enemies and power-ups to the game.

**Challenge:** In the current implementation you can enter the double jump state from mid air after walking off a ledge or after 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 need to be added?

# Chapter 6: Animations

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 Cyclops, and lay the groundwork for enemy and power-up animations. Afterwards, your game will look a lot more like something you would enjoy playing.

Here's where this chapter fits in the scheme of things:



# State machine diplomacy

Now that the state machine is working, you can use state transitions to switch 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. For this reason, you return from `changeState:` right at the beginning if the `newState` is equal to `self.characterState` (bail out if the state hasn't changed).

Open **Player.m** and take a look at `changeState:`. Notice that 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, animate the character. From this chapter onward, your game will start 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 comes from *Learning Cocos2D* by Rod Strougo and Ray Wenderlich. It involves setting up a PLIST object (which is provided as part of the starter project) where you list out all of the sprite frames that make up an animation. The advantage of putting this in a plist is it makes it easy to tweak.

You will write a method to load these animations for Cyclops and the monsters. 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 `GameObject` because both the player and the monsters derive from that. Open **GameObject.h** and add the following method declarations:

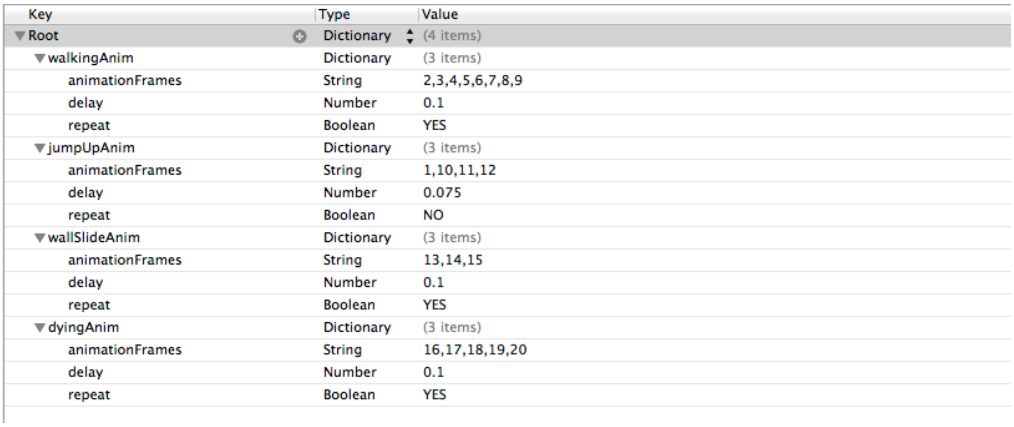
```
-(CCAnimation*)loadAnimationFromPlist:(NSString *)animationName
  forClass:(NSString *)className;
-(void)loadAnimations;
```

`loadAnimations` will be overridden in subclasses of `GameObject`, so you just need an empty implementation here. Add the following code to **GameObject.m**:

```
-(void)loadAnimations {
    //override this method
}
```

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 (`CCAnimation`) from a PLIST file .

Let's take a look at the format of the PLIST file you will be using. Open **PlayerData\Player.plist** and you will see the following:



| Key | Type | Value |
|---|---|---|
| Root | Dictionary | (4 items) |
| walkingAnim | Dictionary | (3 items) |
| animationFrames | String | 2,3,4,5,6,7,8,9 |
| delay | Number | 0.1 |
| repeat | Boolean | YES |
| jumpUpAnim | Dictionary | (3 items) |
| animationFrames | String | 1,10,11,12 |
| delay | Number | 0.075 |
| repeat | Boolean | NO |
| wallSlideAnim | Dictionary | (3 items) |
| animationFrames | String | 13,14,15 |
| delay | Number | 0.1 |
| repeat | Boolean | YES |
| dyingAnim | Dictionary | (3 items) |
| animationFrames | String | 16,17,18,19,20 |
| delay | Number | 0.1 |
| repeat | Boolean | YES |

This file is an `NSDictionary` that contains a set of `NSDictionary` objects, one for each animation. The key for the 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. This will be parsed and used to load the frames from the sprite sheet. More about this in a moment.

The second piece of information is the delay, which is a float that gives the amount of time in seconds between each frame change. For example, if there are 5 frames in an animation sequence and the delay value is .1, then during one second the animation will play completely twice – since 5 x 0.1 is 0.5, and that means the animation takes half a second to complete.

The final value is a Boolean that tells the `CCAnimation` object if the animation should loop. If it's set to `NO`, the animation will play only once.

Add this method to **GameObject.m**:

```
-(CCAnimation*)loadAnimationFromPlist:(NSString *)animationName
forClass:(NSString *)className {
    //1
    NSString *path = [[NSBundle mainBundle]
pathForResource:className ofType:@"plist"];
    NSDictionary *plistDictionary = [NSDictionary
dictionaryWithContentsOfFile:path];

    //2
    NSDictionary *animationSettings = [plistDictionary
objectForKey:animationName];

    //3
```

```
    CCAnimation *animation = [CCAnimation animation];

    //4
    animation.delayPerUnit = [[animationSettings
objectForKey:@"delay"] floatValue];

    //5
    NSString *animationFrames = [animationSettings
objectForKey:@"animationFrames"];
    NSArray *animationFrameNumbers = [animationFrames
componentsSeparatedByString:@","];

    //6
    for (NSString *frameNumber in animationFrameNumbers) {
        NSString *frameName = [NSString
stringWithFormat:@"%@%@.png",className,frameNumber];
        [animation addSpriteFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:frameName]];
    }

    //7
    return animation;
}
```

Here's a step-by-step breakdown of the above method:

1. The code in the first section loads the PLIST 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, the specific animation dictionary (based on the animation name) is retrieved from `plistDictionary`. Since the PLIST file contains all the animations for the given class, this method is called once for each animation, passing the name of the class and the name of the animation into the method. Each time it's called, the result will be assigned to a `CCAnimation` object.
3. This line creates the animation object that will be returned.
4. Next you retrieve the value for the delay and use it to set the animation's `delayPerUnit` property.
5. Then you retrieve the list of animation frames, a comma-separated list of frame numbers. So it needs to be parsed into an `NSArray` of strings, one for each frame number. This is done with a call to `componentsSeparatedByString` – it takes a string and separates it using the specified character. What's returned is an array of the individual string components. It's handy-dandy and I like it. ☺
6. Next is a `for` loop that creates a frame identifier for each of the frames contained in the array from the previous step. Once the frame name is created, you use it to add the actual sprite frame to the animation via `addSpriteFrame:`.

You use the name of the sprite frame to retrieve it from `CCSpriteFrameCache`. Of course, you need to add the sprite frames to the cache before you run this method. We'll cover that next.

If you use TexturePacker (which is what I did for this tutorial), the names of the frames will be their individual file names. So for this method of loading frames into animations to work, make sure that the file names follow the correct naming convention.

7. Finally, you return the `CCAnimation`.

All the game classes that have animations to load will use the above method. Again, be sure that all file names and Texture Packer assets follow the proper naming conventions, where each sprite frame's name is based on the class name and the frame number.

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 instance variables to keep track of the various animations. So add the following to the `@interface` section at the top of **Player.m** (where you put the `jumpReset BOOL` earlier):

```
CCAnimation *walkingAnim;
CCAnimation *jumpUpAnim;
CCAnimation *wallSlideAnim;
```

You can probably guess what each of these animations are, based on their names. You may also have noticed that the provided PLIST files have another animation – the dying animation. You'll add that one later on.

Implement the `loadAnimations` method in **Player.m** as follows:

```
-(void)loadAnimations {
    wallSlideAnim = [self loadAnimationFromPlist:@"wallSlideAnim"
forClass:@"Player"];
    walkingAnim = [self loadAnimationFromPlist:@"walkingAnim"
forClass:@"Player"];
    jumpUpAnim = [self loadAnimationFromPlist:@"jumpUpAnim"
forClass:@"Player"];
}
```

You can see that each animation name in the calls to `loadAnimationFromPlist:forClass:` corresponds to the animation name in the PLIST.

You want to call `loadAnimations` during the initialization of the class. Because this method is common to all the `GameObject` subclasses, add the call to **GameObject.m** as follows:

```
-(id)initWithSpriteFrameName:(NSString *)spriteFrameName {
    if (self = [super initWithSpriteFrameName:spriteFrameName]) {
        [self loadAnimations];
    }
    return self;
}
```

You're going to use `initWithSpriteFrameName:` when initializing all the objects in the game that come from sprite sheets, and in this game, everything comes from a sprite sheet. ☺ So you can now be sure that `loadAnimations` will be called first, before any of the subclass initialization code runs.

You can't actually tell until you complete the next step, but your animations are now loaded and ready to use. It's time to put the icing on the cake!

# Animations deploy!

Now that you have your animations loaded into instance variables, it's time to run the animations when you switch states. To do this, replace `changeState:` in **Player.m** with the following:

```
-(void)changeState:(CharacterStates)newState {
    if (newState == self.characterState) {
        return;
    }

    self.characterState = newState;

    //1
    [self stopAllActions];

    //2
    id action = nil;

    //3
    switch (newState) {
        case kStateStanding:
            //4
            [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"Player1.png"]];
            break;
        case kStateFalling:
            [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"Player10.png"]];
            break;
```

```
        case kStateWalking:
           //5
            action = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:walkingAnim]];
            break;
        case kStateWallSliding:
            action = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:wallSlideAnim]];
            break;
        case kStateJumping:
            action = [CCAnimate actionWithAnimation:jumpUpAnim];
            break;
        case kStateDoubleJumping:
            [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"Player10.png"]];
            break;
        default:
            //6
            [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"Player1.png"]];
            break;
    }

    //7
    if (action) {
        [self runAction:action];
    }
}
```

Everything before section #1 is code previously in the method, but you're removing the `NSLog` statement that was there previously, since you no longer need it. Let's go through the rest section-by-section:

1. You call `stopAllActions` 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 for one frame before the animation takes over.

   If there isn't an animation running, calling `stopAllActions` won't do anything. No harm, no foul. ☺

2. You create a new animation object and initialize it to `nil`. It can contain an animation after all the state checks or it can still be `nil`, depending on the state.
3. This starts the state checks. 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 player to a specific sprite frame. If Cyclops is standing, for example, he doesn't have an animation.

4. This sets the standing state. You call `setDisplayFrame:` and retrieve a `CCSpriteFrame` from the cache. The sprite frames are retrieved using the name of the original image file.

   The falling state is also a static frame and so uses the `setDisplayFrame:` call as well.

5. This sets the walking animation to the action variable you set up earlier. The animation is simply information about a series of frames and the timing for those frames. In order to run an animation, it needs to be encapsulated in a `CCAction` object. This line creates a `CCAction` with the `CCAnimate` call. It encapsulates that in a `CCRepeatForever` action type so that the animation will run indefinitely (until it is stopped via the call to `stopAllActions` in section #1, that is).

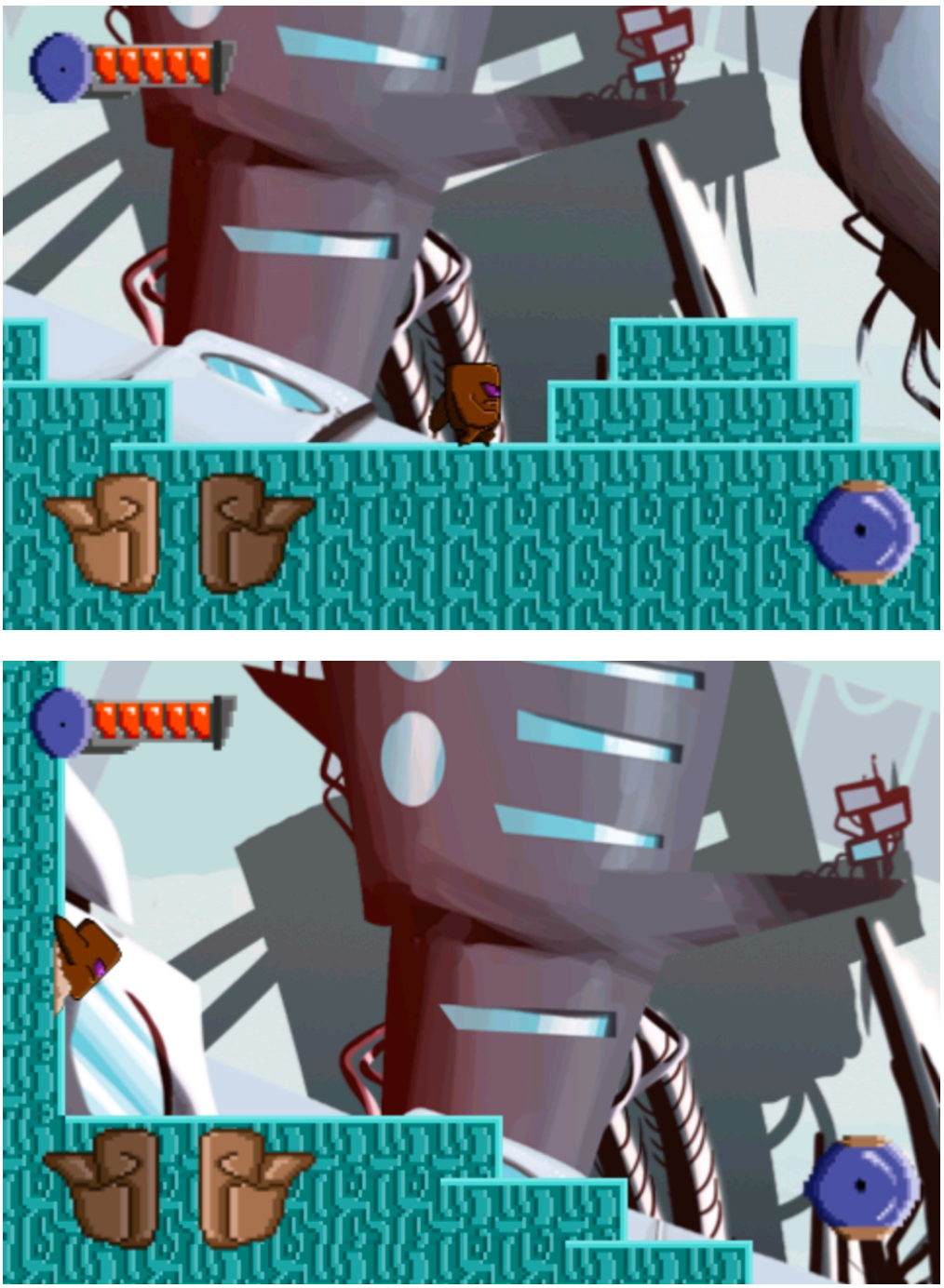
   Walking and wall sliding animations will repeat for as long as you continue that action. The jump animation will only run once at the beginning of the jump behavior. So notice that in the case of the jump state, you aren't wrapping the animations in a `CCRepeatForever`.

   Every player state does one of these two things: either set the frame or start an animation.

6. This section sets the default character animation. If the state variable is not equal to any of your enumerated states, it will choose this option, which is the same as the standing state. This should never get executed because you should always have a state that is one of the enumerated states. You could also have skipped `kStateStanding` and just used this default for that. You're doing it this way just to be thorough.

7. Finally, you run a test to see if `action` was set (otherwise it 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. The call to `runAction:` actually starts the animation.

That's it! Build and run the game and you should have a Cyclops that is running, wall sliding and jumping.



Now that you have a fully working player, 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 over half of the entire starter kit!

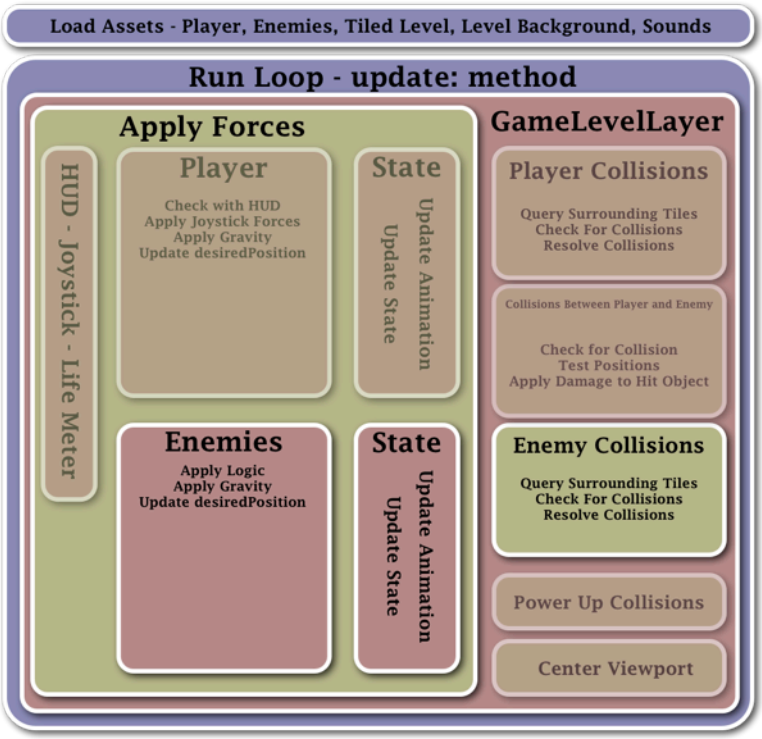
**Challenge:** In this game, animations are driven by the `characterState` variable. There's exactly one animation for every state. But what if you had a game where the player had different suits/costumes? Think of Mario with his white suit or raccoon suit. How would you incorporate that into this game structure?

# Chapter 7: Enemies

It's time to raise the stakes. The game is now playable, but Cyclops has it way too easy without anyone to oppose him!


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 atack 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 will be the simplest: call him the **simple crawler**. He will move in a specific direction and the only stimulus that will change his action is a bump into a wall, which will cause him to switch direction.

The second enemy will be much like the first, but more intelligent. Call him the **mean crawler.** This one will pursue the player. When necessary, he will be able to jump over things in his pursuit of Cyclops.

The third enemy will be the meanest of them all, fearsome for his ability to fly. However, in order for him to be beatable, he will only be able to purse Cyclops when Cyclops has his back to the enemy, like a ghost in *Super Mario World*. This enemy will be forced to freeze when the player is looking at him – unless Cyclops gets too close, so watch out! Call him the **two-faced flyer**.

These enemies will use the same logic as the player. The collision detection method will be the same. They will use a state machine to determine the current animation and expected behavior.

While these enemies are fairly simple, they will give you the tools to write much more complex AI.

# The enemy onslaught

Time to get started building the enemy army!

Create a new file with the **iOS\cocos2d v2.x\CCNode class** template. Name the class **Enemy** and make it a subclass of **Character**.

Now change the `#import "cocos2d.h"` statement in **Enemy.h** to:

```
#import "Character.h"
```

Repeat the same process as above to create a new class named `Crawler`, except make it a subclass of `Enemy`. Once you've created the class, change the `#import "cocos2d.h"` line in **Crawler.h** to:

```
#import "Enemy.h"
```

Now that you've got 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 `GameLevelLayer` to keep track of all the enemies (not just crawlers).

Switch to **GameLevelLayer.m** and add this to the `@interface` block:

```
NSMutableArray *enemiesArray;
```

Now add the following to the `#import` section:

```
#import "Crawler.h"
```

Next you're going to add the block of code that creates the enemies. Since it's a bit long and complex, you'll create a separate method that you'll call from `initWithLevel:`.

Before you write the actual method, add the following code to `initWithLevel:` before the `[self scheduleUpdate]` line:

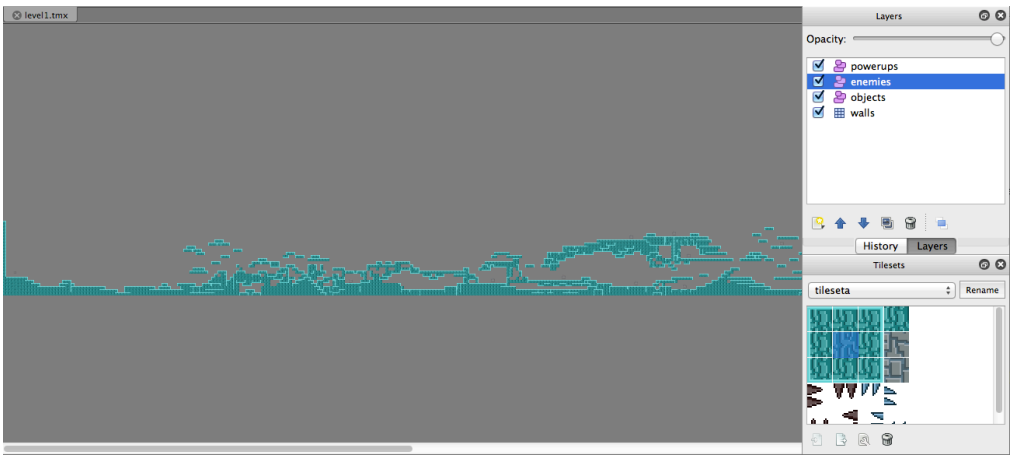
```
[self loadEnemies];
```

Now add the method:

```
-(void)loadEnemies {
    //1
    CCTMXObjectGroup *enemies = [map objectGroupNamed:@"enemies"];
    //2
    enemiesArray = [NSMutableArray array];
    //3

    NSString *spriteFramesFile = @"CrawlerImages.plist";
    //4
    [[CCSpriteFrameCache sharedSpriteFrameCache]
addSpriteFramesWithFile: spriteFramesFile];
    //5
    CCSpriteBatchNode *objectSpriteSheet = [CCSpriteBatchNode
batchNodeWithFile:@"CrawlerImages.png"];
    //6
    [self addChild:objectSpriteSheet];
    //7
    for (NSDictionary *object in enemies.objects) {
        //8
        NSString *firstFrameName = @"Crawler1.png";
        //9
        Enemy *objectInstance = [[Crawler alloc]
initWithSpriteFrameName:firstFrameName];
        //10
        objectInstance.position = ccp([[object objectForKey:@"x"]
floatValue], [[object objectForKey:@"y"] floatValue]);
        //11
        [objectSpriteSheet addChild:objectInstance];
        //12
        [enemiesArray addObject:objectInstance];
    }
}
```

It's a pretty big chunk of code, so here's a line-by-line explanation:

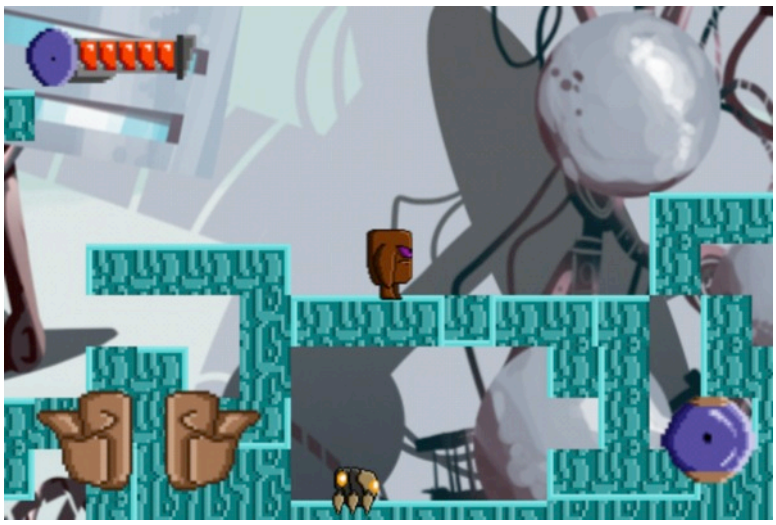
1. The first step is to retrieve data about the enemies from the TMX Tile Map layer. This comes in the form of a `CCTMXObjectGroup`. If you take a look again at the provided tile map files in the Tiled editor, you'll see an object layer (or group) called "enemies", which contains a bunch of objects specifying where you should spawn enemies. This is what you're loading here.




2. Initialize the `enemiesArray`. This mutable array keeps track of all the level's enemies.
3. Set up the crawler's sprite frames file.
4. Use the sprite frame file name to add the sprite frames to `CCSpriteFrameCache`. This is the same as when you added the player's sprite frames.
5. Create a `CCSpriteBatchNode` object using the crawler's sprite sheet. More detail on this in a minute.
6. Add the `CCSpriteBatchNode` to the layer as a child.
7. Loop through the `CCTMXObjectGroup` you created earlier. The `objects` property is an array containing an `NSDictionary` with all the enemy info, including position, type, name and any custom data elements.
8. Create a variable that identifies the frame that you'll use to initialize the crawler object.
9. Now use the value from #8 to initialize the crawler object. You set `objectInstance` as an instance of `Enemy` instead of `Crawler`. This is so that you can use this same loop later to initialize other types of enemies without having to create a different loop for each type of enemy.

   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.
10. Set the position of the enemy using the x and y keys in the `NSDictionary`.
11. Add the enemy to the `CCSpriteBatchNode`. When you use a batch node, you must add the individual sprites to the batch node instead of to the layer. This way the batch node can optimize the drawing for the layer by combining the drawing code for all the sprites it contains. This is explained in greater detail at the end of this section.
12. Finally, add the enemy to the `enemiesArray`.

Build and run now, and walk through the level until you find the first enemy (it might take a little bit to find him).



He's just sitting there, not unlike Cyclops at the beginning. Now you can apply physics to the enemy, just as you did for Cyclops!

**Note:** `CCSpriteBatchNode` is an object that's designed to help draw sprites in a more efficient way. If each object on screen has its own `CCSprite` object, when it comes time to draw the scene, each object is drawn in a single drawing call. This means that a scene with fifty enemies would call the draw method fifty times.

When you use a `CCSpriteBatchNode`, you put all the enemy images onto a single texture called a sprite sheet or sprite atlas. This allows you to call draw one time and draw all the enemies at once. It takes longer than drawing a single enemy, but it's much quicker than drawing fifty enemies, one at a time.

Most mobile games that have lots of graphical assets will make use of `CCSpriteBatchNode`s, and so will you for this tutorial. I suspect that in this game it's not absolutely necessary, but it's good practice for when you make your own game, which will undoubtedly have more levels, more enemies, and would probably require the added efficiencies of a sprite batch node.

I've provided individual sprite sheets for the player, each enemy, the HUD, and the power-up objects – all of them made with TexturePacker. Had I wanted to, I could have created a single sprite sheet containing all the game graphics (except for the tile map and background art). This would have been the most efficient method, but it's easier conceptually to keep them all separate, and it works just fine.

If you want to learn more about sprite sheets, check out these tutorials:

http://www.raywenderlich.com/1271/how-to-use-animations-and-sprite-sheets-in-cocos2d

http://www.raywenderlich.com/2361/how-to-create-and-optimize-sprite-sheets-in-cocos2d-with-texture-packer-and-pixel-formats

A sprite sheet consists of a PNG file containing all the individual images and a PLIST with information about the name and location of each frame within the collection.

If you are confused by the terminology (I was when first exposed to it), you can think of the sprite sheet as the actual PNG file. The `CCSpriteBatchNode` is the object that does the work of drawing the various sprites, sourced from the sprite sheet, into the scene. The PLIST file just tells the `CCSpriteBatchNode` where everything is by giving the layout of the frames within the PNG file.

# Let's get physical (again)

Just as with Cyclops, the physics rules for enemies are applied in two places. The enemy's `update:` method will apply gravity and movement, and the `GameLevelLayer` will resolve collisions.

The `update:` method will also contain the enemy AI. This first crawler will be very simple, so you can start there. Add the following method to **Crawler.m**:

```
-(void)update:(ccTime)dt {

    CGPoint gravity = ccp(0.0, -450.0);
    CGPoint gravityStep = ccpMult(gravity, dt);

    self.velocity = ccpAdd(self.velocity, gravityStep);
    self.desiredPosition = ccpAdd(self.position,
ccpMult(self.velocity, dt));
}
```

This is just gravity for now. 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!

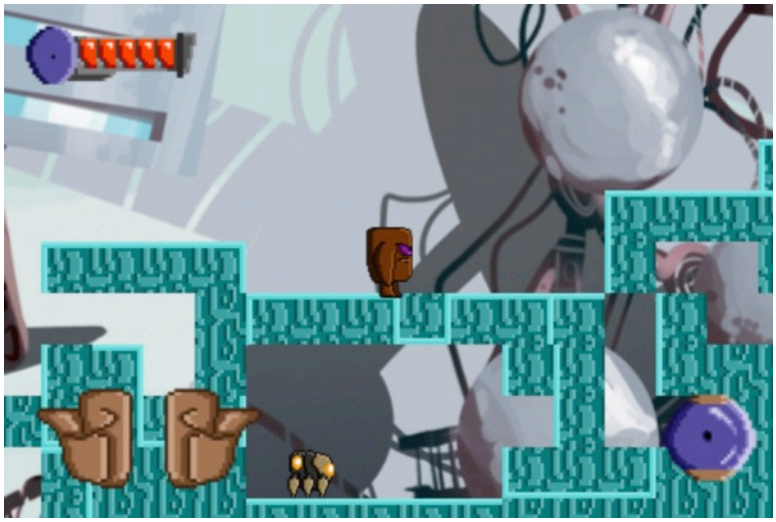

Switch to **GameLevelLayer.m** and add the following to `update:` (before the line that calls `setViewpointCenter`):

```
for (Enemy *e in enemiesArray) {
    [e update:dt];
    [self checkForAndResolveCollisions:e];
}
```

This `for` loop iterates over every enemy in the `enemiesArray`. 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:` on each enemy. This exactly mirrors the process applied to the player.

Build and run, and you'll see that your enemies are now being affected by gravity and that collisions are also active.



# Your first taste of AI

Now that the enemies are subject to the laws of physics, you can start giving 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, 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 that will 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):

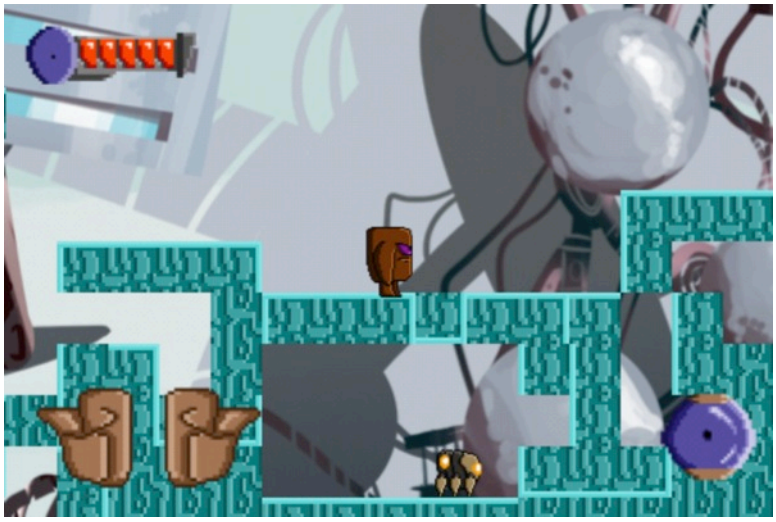
```
//1
if (self.onGround) {
    //2
    if (self.flipX) {
        self.velocity = ccp(-kMovementSpeed, 0);
    } else {
        self.velocity = ccp(kMovementSpeed, 0);
    }
    //3
} else {
    self.velocity = ccp(self.velocity.x * 0.98, self.velocity.y);
}

//4
if (self.onWall) {

    self.velocity = ccp(-self.velocity.x, self.velocity.y);
    //5
    if (self.velocity.x > 0) {
        self.flipX = NO;
    } else {
        self.flipX = YES;
    }
}
```

The above is pretty straightforward, but here's a section-by-section breakdown:

1. Detect whether the crawler is on the ground or not. If it's on the ground, then it's crawling, but if it has crawled off a ledge, then it would be falling.

2. If the crawler's on the ground, you can use the `flipX` property to determine which direction the crawler should be going. Based on this value, you set the velocity to the `kMovementSpeed` variable. This will be negative if the crawler is facing (and crawling towards) the left side of the screen.
3. If the crawler has gone off a ledge, then you want to slow its velocity in the x direction so that it mostly falls straight down, instead of floating diagonally down, which would happen without the damping `self.velocity.x * 0.98`.
4. If the crawler has hit a wall, then set the `onWall` property to `YES`. You can use this property to determine when it's time to switch direction. Switching direction is as easy as setting the x property of `self.velocity` to its corresponding negative value. (If it's already negative, then this will switch 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 Cyclops – at least for now! ☺

# The many faces of a monster

Now it's time to liven up these crawlers with some animations, using the same technique as for the player. 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 you switch state.

Do you remember the first step? Load those animations! Create an instance variable to keep track of the `CCAnimation` object by adding the following to **Crawler.m** (right before the `@implementation` line):

```
@interface Crawler () {
    CCAnimation *walkingAnim;
}
@end
```

For now, you have just one animation. Later you'll add others for the simple crawler, and the more complex enemies will also have more animations.

Now add the `loadAnimations` method:

```
-(void)loadAnimations {
    walkingAnim = [self loadAnimationFromPlist:@"walkingAnim"
forClass:@"Crawler"];
}
```

This is nothing to break a sweat over. The crawler will have two main states: walking and falling. The falling state doesn't have an animation – it will just be a single frame.

The next step is to put together the state machine that will control the transitions to go along with the `update:` method.

Add the following method:

```
-(void)changeState:(CharacterStates)newState {
    if (newState == self.characterState) {
        return;
    }

    [self stopAllActions];
    self.characterState = newState;

    id action = nil;

    switch (newState) {
        case kStateWalking:
            action = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:walkingAnim]];
            break;
        case kStateFalling:
            [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"Crawler1.png"]];
            break;
        default:
            break;
    }
```

```
    if (action != nil) {
        [self runAction:action];
    }
}
```

This should look familiar! The only difference here from the player's `changeState:` method is that you only have two states for the crawler.

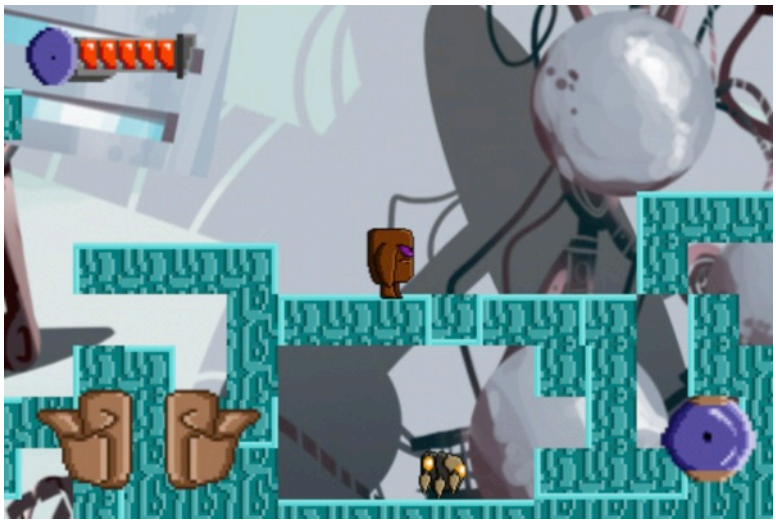
The final step is to trigger `changeState:` at the appropriate time via `update:`. So in `update:`, after the test for `self.onGround`, change the state to `kStateWalking`:

```
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. He'll be 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 Cyclops, the next chapter will take up that subject.

Continuing on with the enemies, I think this crawler is a pretty good start. But he's easy to avoid. You need a crawler that's a little more obnoxious. Hence, the mean crawler. ☺

Create a new file with the **iOS\cocos2d v2.x\CCNode class** template. Name the class **MeanCrawler** and make it a subclass of `Enemy`. Change the `#import "cocos2d.h"` line to:

```
#import "Enemy.h"
```

This mean crawler will do a few things that the simple crawler can't. For one, he'll pursue the player instead of just blindly walking back and forth. Second, the mean crawler will be able to jump – both to climb stairs in order to get to Cyclops, and to jump onto Cyclops if Cyclops tries to jump over him.

This new crawler will be much harder to avoid!

> **Note**: Are you wondering why you created a new `Enemy` subclass instead of making mean crawler a subclass of `Crawler`? The reason is that even though they look very similar, the two types of crawlers don't actually share enough code to make it worthwhile to create `MeanCrawler` by subclassing `Crawler`.

## Distinguishing among the ranks

The first thing to do is add new properties that will let the enemy know where Cyclops is. Add the following imports to **Enemy.h**:

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

Then add the new properties:

```
@property (nonatomic, weak) Player *player;
@property (nonatomic, weak) PCTMXTiledMap *map;
```

Now you need to set these properties when you initialize an `Enemy` instance and add it to the layer. In fact, you need to make some changes to the code that loads the enemies so that you can load different types.

Update `loadEnemies` in **GameLevelLayer.m** so it looks like this:

```
-(void)loadEnemies {
    CCTMXObjectGroup *enemies = [map objectGroupNamed:@"enemies"];

    enemiesArray = [NSMutableArray array];
    //1
    NSMutableArray *enemyTypes = [NSMutableArray array];
```

```
    NSMutableArray *enemyBatchNodes = [NSMutableArray array];

    for (NSDictionary *enemy in enemies.objects) {
        //2
        NSString *enemyType = [enemy objectForKey:@"type"];
        //3
        if (![enemyTypes containsObject:enemyType]) {
            //4
            NSString *spriteFramesFile = [NSString
stringWithFormat:@"%@Images.plist", enemyType];
            //5
            [[CCSpriteFrameCache sharedSpriteFrameCache]
addSpriteFramesWithFile: spriteFramesFile];
            //6
            CCSpriteBatchNode *enemyBatchNode = [CCSpriteBatchNode
batchNodeWithFile:[NSString stringWithFormat:@"%@Images.png",
enemyType]];
            //7
            [self addChild:enemyBatchNode];
            //8
            [enemyTypes addObject:enemyType];
            //9
            [enemyBatchNodes addObject:enemyBatchNode];
        }
        //10
        NSString *firstFrameName = [NSString
stringWithFormat:@"%@1.png", enemyType];
        //11
        Enemy *enemyInstance = [[NSClassFromString(enemyType)
alloc] initWithSpriteFrameName:firstFrameName];
        //12
        enemyInstance.position = ccp([[enemy objectForKey:@"x"]
floatValue], [[enemy objectForKey:@"y"] floatValue]);
        //13
        enemyInstance.player = player;
        enemyInstance.map = map;
        //14
        int enemyIndx = [enemyTypes indexOfObject:enemyType];
        CCSpriteBatchNode *node = [enemyBatchNodes
objectAtIndex:enemyIndx];
        //15
        [node addChild:enemyInstance];
        [enemiesArray addObject:enemyInstance];
    }
}
```

There are a few lines in this new method that are the same as before, but there are enough changes to merit a detailed explanation:

1. Create two temporary `NSMutableArray`**s**. For each new enemy type, you need to set up a separate `CCSpriteBatchNode` for that class. These two arrays will help keep track of that.

   The first array holds the names of the classes. You'll just use this array to check if you've already set up the `CCSpriteBatchNode` for that class. The second array holds references to the `CCSpriteBatchNode`**s** themselves.

2. Save the class name of the `Enemy` subclass. This is specified in the TMX tile map using the "type" field.

3. Check the `enemyType` to see if you've already added it to the `enemyTypes` array. If you have, you can infer that you've already set up the `CCSpriteBatchNode` for that type and skip the next several lines.

4. If you haven't already set up the `CCSpriteBatchNode`, then you need to do that. Create a string that holds a reference to the PLIST that describes the sprite sheet for this particular enemy type.

   The primary thing you accomplish with this new method is the ability to initialize any `Enemy` subclass that exists in your tile map. You can use the value in the "type" field to dynamically create a reference to any class type/sprite sheet.

5. Add the sprite sheet from step #4 to the `CCSpriteFrameCache`.

6. Set up the `CCSpriteBatchNode` object.

7. Add the `CCSpriteBatchNode` to the layer.

8. Add the new class to the `enemyTypes` array so that you won't set up a `CCSpriteBatchNode` for this class again.

9. Add the `CCSpriteBatchNode` to the `enemyBatchNodes` array so that you can retrieve the batch node for this particular enemy type when you next need it.

10. Save the name of the first frame for this particular enemy sprite sheet.

11. This is the magic of the whole block. You can use `NSClassFromString()` to get a reference to a class from a string. By setting the "type" field to the name of the class, you can create an instance of any class quite easily.

    You use it here to create an instance of the `Enemy` class and you initialize the class with the `firstFrameName` value that you created in the previous line.

12. Set the position of the new `Enemy` instance using the information loaded from the `TMXTileMap` dictionary for that object. It's the same logic as before.

13. Next, you set the player and map attributes. This will allow you to get the player's information, including position, from within an `Enemy` subclass. You can also use the `PCTMXTiledMap` object's methods to get the tile coordinate for a certain position. You'll need this to program the logic governing how the enemy follows the player, for example.

14. Get the right `CCSpriteBatchNode` from the `enemyBatchNodes` array.

15. Finally, add the `Enemy` instance to the `CCSpriteBatchNode`.

It's time to test the code by building and running, but since the enemies are now being loaded 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. So you need to create one more class.

Do that by creating a new file with the **iOS\cocos2d v2.x\CCNode class** template. Make it a subclass of `Enemy` and name it `Flyer`. Change the `#import "cocos2d.h"` line to:

```
#import "Enemy.h"
```

Now add an import statement for the new class to **GameLevelLayer.m**:

```
#import "Flyer.h"
```

One final thing you must do is to include a minimal `update:` method. If you recall your logic for collision resolution, it relies on the `desiredPosition` property. If this value isn't set, it will (probably) be zero. A `desiredPosition` of 0,0 is invalid when you ask for the surrounding tiles and this will result in a `TMXLayer` error.

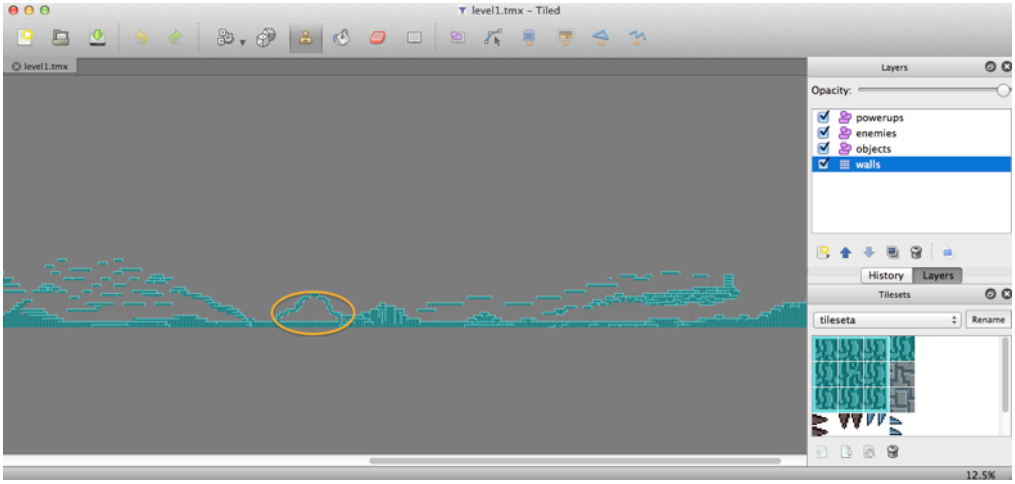
So add the following code to both **MeanCrawler.m** and **Flyer.m**:

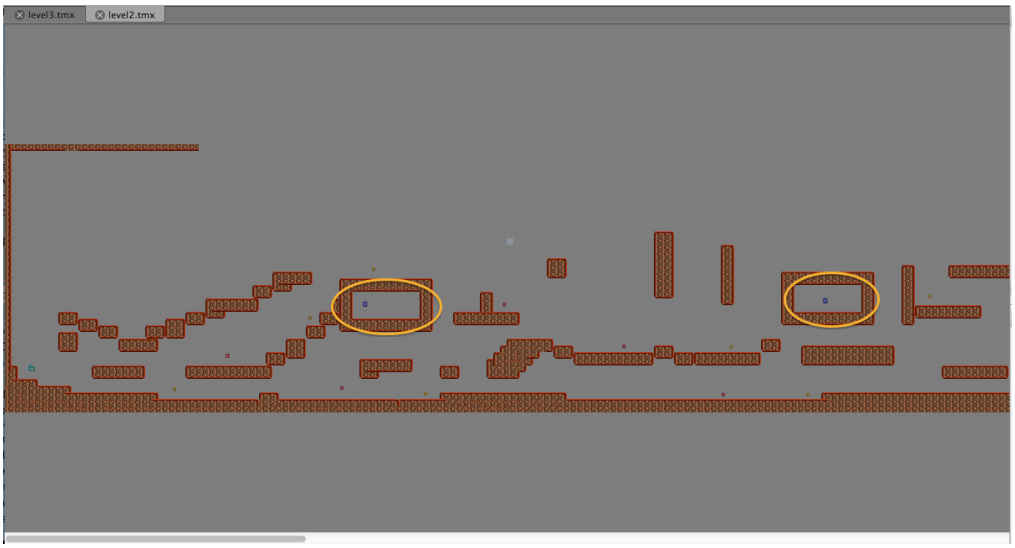
```
-(void)update:(ccTime)dt {
    self.desiredPosition = self.position;
}
```

Now you can build and run. You should have both mean crawlers and flyers appearing in your levels, though they'll just be floating idly in the air until you modify the `update:` method to give them some purpose.

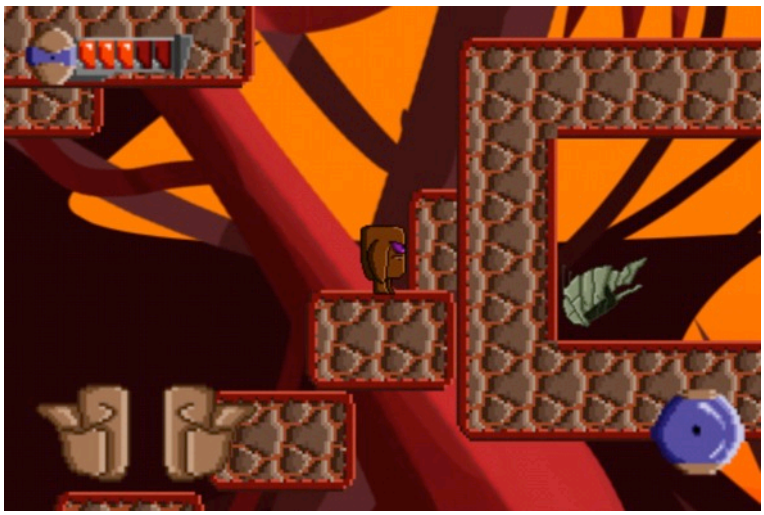
The first level has mean crawlers towards the end. To see a flyer though, you need to go to Level 2.

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 Cyclops and his enemies won't last for long. ☺

## Marching orders

Now that your game can successfully load 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 () {
    CCAnimation *walkingAnim;
    CCAnimation *jumpUpAnim;
}
@end
```

And then add the following new method inside the @implementation:

```
-(void)loadAnimations {
    walkingAnim = [self loadAnimationFromPlist:@"walkingAnim"
      forClass:@"MeanCrawler"];
    jumpUpAnim = [self loadAnimationFromPlist:@"jumpUpAnim"
      forClass:@"MeanCrawler"];
}
```

This code should make sense to you at this point. Eventually all characters will have an animation for their death, but you'll add those after you add the logic that damages characters.

Next add the `changeState:` method. This is also standard by now:

```
-(void)changeState:(CharacterStates)newState {
```

```
    if (newState == self.characterState) {
        return;
    }

    [self stopAllActions];
    id action = nil;
    self.characterState = newState;

    switch (newState) {
        case kStateWalking:
            action = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:walkingAnim]];
            break;
        case kStateJumping:
            action = [CCAnimate actionWithAnimation:jumpUpAnim];
            break;
        default:
            break;
    }

    if (action != nil) {
        [self runAction:action];
    }

}
```

Now for the interesting part – adding the logic. ☺ But first, you need to add the `kMovementSpeed` constant to **MeanCrawler.m** after the `#import` line:

```
#define kMovementSpeed 60
```

With that done, replace the placeholder `update:` method you added earlier with the following:

```
-(void)update:(ccTime)dt {
    //1
    float distance = ccpDistance(self.position,
self.player.position);

    if (distance > 1000) {
        self.desiredPosition = self.position;
        return;
    }

    //2
CGPoint gravity = ccp(0.0, -450.0);
```

```
CGPoint gravityStep = ccpMult(gravity, dt);
    //3
    if (self.player.position.x > self.position.x) {
        self.velocity = ccp(kMovementSpeed, self.velocity.y);
    } else {
        self.velocity = ccp(-kMovementSpeed,  self.velocity.y);
    }

    //4
    self.velocity = ccpAdd(self.velocity, gravityStep);

    //5
    if (self.velocity.x > 0) {
        self.flipX = NO;
    } else {
        self.flipX = YES;
    }

    //6
    CGPoint velocityStep = ccpMult(self.velocity, dt);
    self.desiredPosition = ccpAdd(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 player hasn't reached a section of the level where this enemy resides (at least 1000 pixels away), you don't want the enemy to pursue the player. You want the enemy to wait until Cyclops gets closer. This code returns (bails out of `update:` after setting the `desiredPosition`) until Cyclops gets within 1000 pixels 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. Test whether the player is left or right of the mean crawler and set the crawler's velocity (based on `kMovementSpeed`) accordingly.
4. Add the current speed to the current velocity.
5. Set the `flipX` according to the direction in which the mean crawler is moving.
6. Finally, you set the `desiredPosition` by creating a scaled `velocityStep` and adding it to the current position.

Build and run now. You should have mean crawlers that pursue 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 pixels.

Add the following code to **Crawler.m** at the beginning of the crawler**'s** `update:` method, before all of the existing code:

```
float distance = ccpDistance(self.position, self.player.position);
if (distance > 1000) {
    self.desiredPosition = self.position;
    return;
}
```

## These crawlers also 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 the logic that changes the mean crawler's state.

Replace `update:` in **MeanCrawler.m** with the following improved version:

```
-(void)update:(ccTime)dt {
    //1
    float distance = ccpDistance(self.position,
self.player.position);
    if (distance > 1000) {
        self.desiredPosition = self.position;
        return;
    }

    //2
    if (self.onGround) {
        [self changeState:kStateWalking];
    }

    //3
    CGPoint myTileCoord = [self.map
tileCoordForPosition:self.position];
    CGPoint twoTilesAhead;

    //4
    if (self.player.position.x > self.position.x) {
        twoTilesAhead = ccp(myTileCoord.x + 2, myTileCoord.y);
        self.velocity = ccp(kMovementSpeed, self.velocity.y);
    } else {
        twoTilesAhead = ccp(myTileCoord.x - 2, myTileCoord.y);
        self.velocity = ccp(-kMovementSpeed,  self.velocity.y);
    }

    //5
    twoTilesAhead = ccpClamp(twoTilesAhead, ccp(0,0),
ccp(self.map.mapSize.width, self.map.mapSize.height));

    //6
    if ([self.map isWallAtTileCoord:twoTilesAhead]) {
        //7
        if (self.onGround) {
            self.velocity = ccp(self.velocity.x, kJumpForce);
            [self changeState:kStateJumping];
        }
    }

    //8
    if (distance < 100 && (self.player.position.y -
self.position.y) > 50) {
```

```
        if (self.onGround) {
            self.velocity = ccp(self.velocity.x, kJumpForce);
            [self changeState:kStateJumping];
        }
    }

    //9
    CGPoint gravity = ccp(0.0, -450.0);
    CGPoint gravityStep = ccpMult(gravity, dt);

    //10
    self.velocity = ccpAdd(self.velocity, gravityStep);

    //11
    if (self.velocity.x > 0) {
        self.flipX = NO;
    } else {
        self.flipX = YES;
    }

    //12
    CGPoint velocityStep = ccpMult(self.velocity, dt);
    self.desiredPosition = ccpAdd(self.position, velocityStep);
}
```

Here's a full explanation of this improved method:

1. Check how close the player is to the crawler and bail out if he's not close enough.
2. Check to see if the mean crawler is on the ground and if it is, set its state to walking.
3. Calculate the current tile position of the mean crawler and declare a new tile coordinate variable that will hold the position of the tile that is two tiles in front of the mean crawler, based on the current direction of the crawler.
4. Set `velocity` – this was existing code. Also, calculate the tile position two tiles in front of the mean crawler's current position.
5. If you ask for a tile position outside the bounds of the current map, the app will crash. This line makes sure that the `twoTilesAhead` tile position stays within the bounds of the map.
6. Next check whether the `twoTilesAhead` location contains a wall. If it does, you want your mean crawler to jump.
7. This is the jump logic. You don't want the mean crawler to jump if it's not touching the ground, so first check to see if it's on the ground. Then change the y component of the velocity vector to `kJumpForce` (which you'll add next). Once you

apply the jump force to the mean crawler, you need to change its state to jumping.

8. There is one other scenario in which you'd like the mean crawler to jump. If the player gets close and tries to jump over the mean crawler, the crawler should jump up to try and hit the player. This block performs that test if the player is within 100 pixels and at least 100 pixels above the mean crawler's position.
9. – 12. These sections were covered previously.

You need to set the value of `kJumpForce` in order for this block to work, so add the following line at the top of **MeanCrawler.m**:

```
#define kJumpForce 250
```

> **Note**: When you compile your game after adding the above constant, you might notice a compiler warning about the `kJumpForce` macro being redefined. This is because the `kJumpForce` constant was also defined in **Player.h**.
>
> If you're like me and would like to eliminate all compiler warnings, you can simply move the `#define` line in **Player.h** to **Player.m**, since that definition is not required outside the `Player` class.

The other piece required to make this code work is the method that checks whether a certain tile coordinate is a wall coordinate. To implement it, first add the following method declaration to **PCTMXTiledMap.h**:

```
-(BOOL)isWallAtTileCoord:(CGPoint)tileCoord;
```

Next add the method to **PCTMXTiledMap.m**:

```
-(BOOL)isWallAtTileCoord:(CGPoint)tileCoord {
    CCTMXLayer *layer = [self layerNamed:@"walls"];
    int tgid = [layer tileGIDAt:tileCoord];
    if (tgid) {
        return YES;
    } else {
        return NO;
    }
}
```

Build and run now. Your mean crawler should be acting much meaner!

# The meanest of them all

The game's beginning to come alive now. But there's one last type of enemy with missing logic: the flyer. He lives up to his name.

The flyer's behavior will follow this pattern:

1. If the player is not close (farther than 1,000 pixels), stay dormant.
2. If the player is facing the flyer, go to sleep.
3. If the player is facing away, pursue him.
4. If the player gets close, attack!

You'll tackle `changeState:` first, since it's pretty routine by now. There are some new states that the flyer requires, so add them to the `CharacterState` enum in **Character.h**:

```
typedef enum {
    kStateJumping,
    kStateDoubleJumping,
    kStateWalking,
    kStateStanding,
    kStateDying,
    kStateFalling,
    kStateDead,
    kStateWallSliding,
    kStateAttacking,
    kStateSeeking,
    kStateHiding
} CharacterStates;
```

Next, the flyer has two animations. So add the instance variables for those via a class extension at the top of **Flyer.m**:

```
@interface Flyer () {
    CCAnimation *seekingAnim;
    CCAnimation *attackingAnim;
}
@end
```

Now implement `loadAnimations:`:

```
-(void)loadAnimations {
    seekingAnim = [self loadAnimationFromPlist:@"seekingAnim"
     forClass:@"Flyer"];
    attackingAnim = [self
     loadAnimationFromPlist:@"attackingAnim" forClass:@"Flyer"];
}
```

And then `changeState:`:

```
-(void)changeState:(CharacterStates)newState {
    if (newState == self.characterState) {
        return;
    }
    [self stopAllActions];
    self.characterState = newState;
    id action = nil;

    switch (newState) {
        case kStateSeeking:
            action = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:seekingAnim]];

            break;
        case kStateHiding:

            [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"Flyer4.png"]];

            break;
        case kStateAttacking:
            action = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:attackingAnim]];
            break;
        default:
            break;
```

```
    }
    if (action != nil) {
        [self runAction:action];

    }
}
```

Third time's the charm, right? All of this logic mirrors what you've done before. The three states are:

- Seeking (when the player's back is turned to the flyer);
- Hiding (when the player is facing the flyer);
- Attacking (when the player gets too close).

Build and run now. You won't see any changes in the behavior of the flyer (because the `Flyer` class `update:` method has not been modified 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:(ccTime)dt {
    //1
    float distance = ccpDistance(self.position,
self.player.position);
    if (distance > 1000) {
        self.desiredPosition = self.position;
        return;
    }

    //2
    float 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 = ccpNormalize(ccpSub(self.player.position,
self.position));
    self.velocity = ccpMult(v, speed);

    //7
    if (self.position.x < self.player.position.x) {
        self.flipX = NO;
    } else {
        self.flipX = YES;
    }

    //8
    CGPoint stepVelocity = ccpMult(self.velocity, dt);
    self.desiredPosition = ccpAdd(self.position, stepVelocity);
}
```

Here's your last method breakdown of this chapter:

1. This is similar to the other enemy `update:` methods. It looks to see if Cyclops is in the vicinity, and returns if he is not.
2. Create a variable that will be populated by the `if` statement in the next section. There are three possible speeds for the flyer, one for each of its states.
3. If Cyclops is really close, less than 100 pixels, then the flyer attacks, regardless of whether the player is facing him or not. This first block sets that up by setting the flyer's speed to 100 and changing its state to attacking.
4. Check whether Cyclops is looking in the direction of the flyer (you know that the player is farther than 100 pixels, or the first block would have executed and this

test wouldn't be running). If the player is looking at the flyer, then the flyer's state is changed to hiding and its speed is set to zero.

5. If neither of the above tests were true, then you can infer that Cyclops is farther than 100 pixels and has his back to the flyer. In this case, the flyer's speed is set to a moderate pace of 60 and its state is set to seeking.

6. Retrieve a normalized vector, first by calculating a `CGPoint` that is the distance between the flyer and the player. You then normalize the vector by scaling its length to one. So, for example, if you had a distance that was 100 pixels high and 50 pixels wide, normalizing it would change it to .9, .45, a vector with a length of one. This is called a **unit vector**.

   The vector can then be scaled up to give each time step a constant speed in whichever direction the flyer needs to move to seek the player. The second line does just that.

7. Look at the direction in which the flyer should be moving (towards the player) and set the `FlipX` value accordingly.

8. Finally, 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.

That's it! Build and run again. Your flyers should seek, attack and hide. (But remember that you'll have to move to Level 2 in order to encounter flyers.)

You now have three working enemy types, and a framework for creating enemy AI. You should see how you can create more enemies, or enemies with more complex behaviors, by increasing the number of states and logic in the `update:` method.

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 you will start to make the game a lot more dangerous, by making those enemies cause 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, one thing to consider is adding an enemy that has different attack states that you cycle through.

# Chapter 8: Damage & Death-Dealing

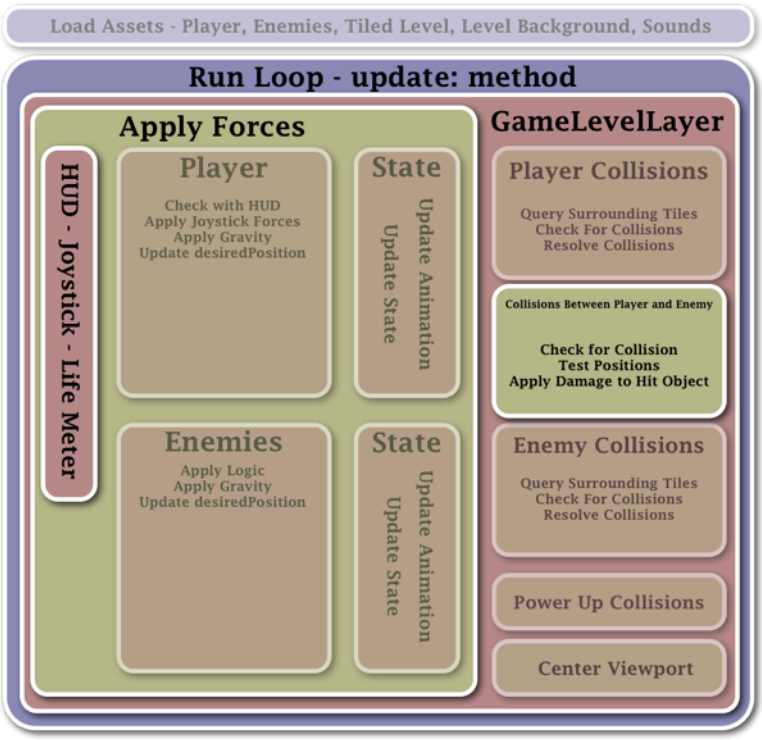
Now that you've got fully-functioning army of enemies, it is time to end the armistice and bring them into battle with Cyclops.


Here are the terms of engagement:

- If Cyclops lands on top of an enemy, he will damage the enemy. You'll add logic that will allow multiple hits for a kill, but all your enemies will die after one hit.
- Any other collision with an enemy will damage Cyclops.
- Cyclops has enough life to take five hits before he dies.

You also need to create conditions under which all the characters can be killed. This will require death animations and state change logic for Cyclops and each of the enemies.

Here are the sections of the overall game plan you'll be tackling in this chapter:




# When monsters collide

The first step is to create a method that will determine when Cyclops has collided with an enemy.

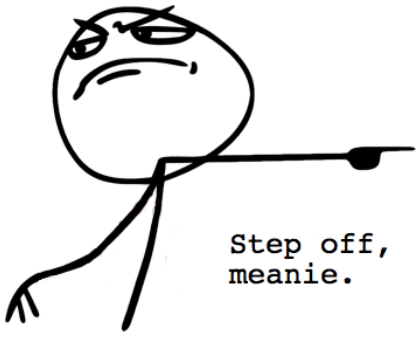
In order to make the method more efficient, you'll add a new property to enemies to determine whether they are active. If an enemy is farther away than 1,000 pixels, you'll consider it to be in a dormant state and won't include it in collision testing.

Add the following property to **Character.h**:

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

This property will do two things. For `Enemy` subclasses, it will be set to `NO` if the distance between the player 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 Cyclops, you'll use this property to give him some temporary protection after taking damage from an enemy. There will be a cool-down period after every hit during which Cyclops is invulnerable. If not for this, he could be hit too many times in quick succession during a single collision.




Modify `update:` in each of the `Enemy` subclasses (**Crawler.m, MeanCrawler.m, Flyer.m**). Each of them has the following code block at the very beginning of `update:`:

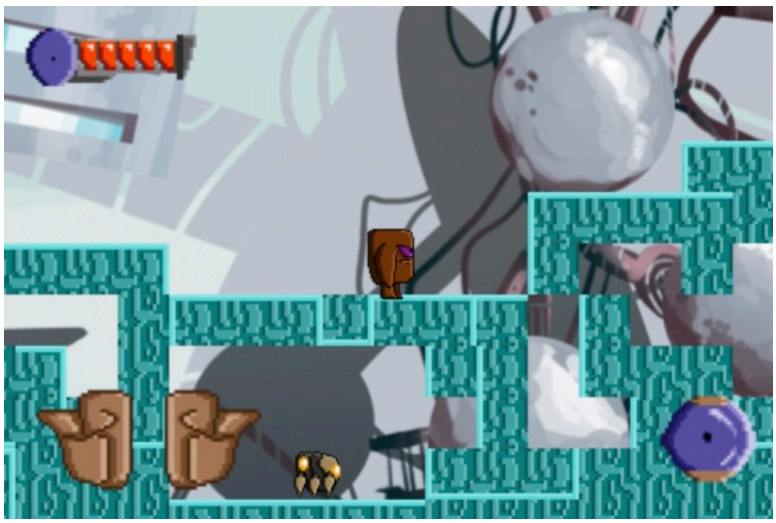
```
float distance = ccpDistance(self.position,
  self.player.position);
if (distance > 1000) {
    self.desiredPosition = self.position;
    return;
}
```

Replace or modify the above so that it matches the following:

```
float distance = ccpDistance(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 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 (i.e. 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 **GameLevelLayer.m**:

```
-(void)checkForEnemyCollisions:(Enemy *)e {
    //1
    if (e.isActive && player.isActive) {
        //2
        if (CGRectIntersectsRect(player.collisionBoundingBox,
e.collisionBoundingBox)) {
            //3
            CGPoint playerFootPoint = ccp(player.position.x,
player.position.y - player.collisionBoundingBox.size.height / 2);
            //4
            if (player.velocity.y < 0 && playerFootPoint.y >
e.position.y) {
                //5
              [e tookHit:player];
            } else {
                [player tookHit:e];
            }
        }
    }
}
```

This method will be called from `update:` each loop. You'll add that next, but first, here's what each line does.

1. Check if the current enemy is active (inactive enemies couldn't possibly collide with Cyclops).
2. Check for a collision between the bounding boxes of the enemy and Cyclops.

3. Find a point that represents the bottom center of Cyclops' bounding box. This is the foot point. You'll use it to determine whether Cyclops is hitting the top of the enemy.
4. This tests whether the foot point of the player is above the enemy and whether the player is falling. That's is enough to determine whether the Cyclops is coming down on top of the enemy.
5. Call `tookHit:` for the player or the enemy, depending on who should be damaged by the collision.

Now implement `tookHit` in the `Character` class. Each subclass can have its own implementation of `tookHit`, but since you want it to be common to both `Player` and `Enemy` subclasses, it makes sense to add it to the `Character` class.

First add the declaration to **Character.h**:

```
-(void)tookHit:(Character *)character;
```

Then add the method to **Character.m**:

```
-(void)tookHit:(Character *)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.

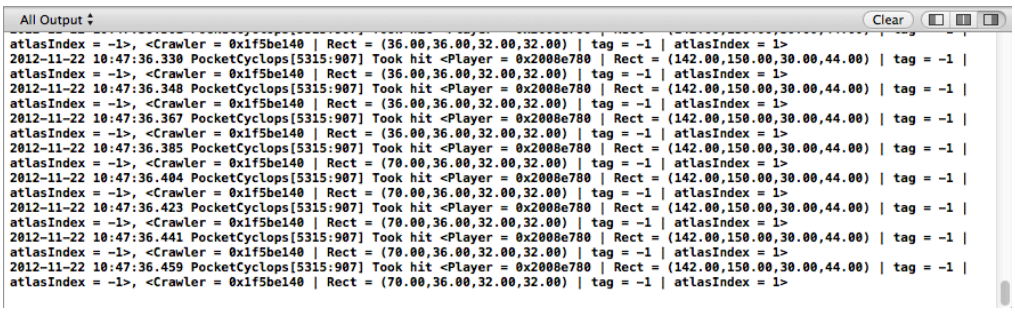
Now add the following line to **GameLevelLayer.m**'s `update:` method, inside the loop that iterates through the `enemiesArray`, after the call to `checkForAndResolveCollisions:`:

```
 [self checkForEnemyCollisions:e];
```

One last thing to do before you check if this is working. The player needs to have their `isActive` flag set to `YES` during initialization. Add the following line to `initWithSpriteFrameName:` in **Player.m**, after the line that initializes `self.velocity`:

```
self.isActive = YES;
```

Build and run. You should see collisions logged in the Xcode console. The first object is the one dealing the hit and the second is the one 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, no damage is given or taken yet, and Cyclops and his enemies still seem to be on friendly terms. You're about to change that!

# Squashing the meanies

All the enemies in the game will die after one hit, so their logic is easy to implement, and a good place to start.

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 **Enemy.m**:

```
-(void)tookHit:(Character *)character {
```

```
    self.life = self.life - 100;
    if (self.life <= 0) {
        [self changeState:kStateDead];
    }
}
```

This code references a new property, `life`. Enemies will all have a `life` value of 100 and so a single hit will finish them.

> **Note**: Although the code's not using it, `tookHit:` takes a `Character` instance as a parameter. If you ever wanted to have different characters deal different amounts of damage, you could do that by inspecting the class of the character instance using `NSClassFromString()`.

You need to add the `life` property to the `Character` class, since the player will also have life (just more of it). Do so by adding this line to **Character.h**:

```
@property (nonatomic, assign) int life;
```

Now initialize the property for all enemies by adding the following initializer to **Enemy.m**:

```
-(id)initWithSpriteFrameName:(NSString *)spriteFrameName {
    if (self = [super initWithSpriteFrameName:spriteFrameName]) {
        self.life = 100;
    }
    return self;
}
```

Each `Enemy` 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. Add the following `CCAnimation` instance variable to the class extensions for **Crawler.m**, **MeanCrawler.m**, **Flyer.m**:

```
CCAnimation *dyingAnim;
```

Now change `loadAnimations` for all three subclasses by adding this line to **Crawler.m**:

```
dyingAnim = [self loadAnimationFromPlist:@"dyingAnim"
forClass:@"Crawler"];
```

And this line to **MeanCrawler.m**:

```
dyingAnim = [self loadAnimationFromPlist:@"dyingAnim"
forClass:@"MeanCrawler"];
```

And this line to **Flyer.m**:

```
dyingAnim = [self loadAnimationFromPlist:@"dyingAnim"
forClass:@"Flyer"];
```

Now add the `kStateDead` branch to each of the `changeState:` methods (at the end, before the `default` case) of each of the `Enemy` subclasses:

```
    case kStateDead:
        action = [CCSequence actions:[CCAnimate
actionWithAnimation:dyingAnim],
                  [CCCallFunc actionWithTarget:self
selector:@selector(removeSelf)], nil ];
        break;
```

This code creates a `CCSequence`, an action that allows you to chain a number of animations or other actions together. In this case, you are first running the animation and then calling `CCCallFunc`, an action that calls a user-defined function. Next you need to set up that function, `removeSelf`.

Add the method declaration `removeSelf` to **Enemy.h**:

```
-(void)removeSelf;
```

And add the implementation to **Enemy.m**:

```
-(void)removeSelf {
    self.isActive = NO;
}
```

When you want to remove a `CCSprite` from its parent `CCLayer`, there's a method called `removeFromParentAndCleanup:` that you can use. However, in the case of enemies, you also need to remove them from the `enemiesArray` in `GameLevelLayer`. So the `GameLevelLayer` needs to know that it's time to remove the object. But how will it know when to remove them?

Well, you're going to be creating a check that looks at the `enemiesArray` for enemies that have their state set to `kStateDead` and `isActive` set to `NO`. `kStateDead` itself isn't sufficient, because you need to give the `dyingAnimation` several loops in order 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 `GameLevelLayer` to accomplish this. Replace the current `update:` implementation with this modified version:

```
-(void)update:(ccTime)dt {
    [player update:dt];
    [self checkForAndResolveCollisions:player];

    //1
    NSMutableArray *enemiesThatNeedDeleting = [NSMutableArray
array];

    for (Enemy *e in enemiesArray) {
        [e update:dt];
        [self checkForAndResolveCollisions:e];
        [self checkForEnemyCollisions:e];

        //2
        if (!e.isActive && e.characterState == kStateDead) {
            [enemiesThatNeedDeleting addObject:e];
        }

    }

    //3
    for (Enemy *e in enemiesThatNeedDeleting) {
        [enemiesArray removeObject:e];
        [e removeFromParentAndCleanup:YES];
    }

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

Here's a blow-by-blow of the above changes:

1. You add a new array to hold those enemies that need to be deleted. This is necessary because you can't remove objects from an array while iterating through its contents.
2. Within the loop that iterates through the enemies, you check for your two conditions. If the test is satisfied, then you add the enemy to the array that will contain all the deleted enemies.
3. Finally, you iterate through the delete array. You first remove the enemies from the `enemiesArray`. Then you call `removeFromParentAndCleanup:` to remove them from the `GameLevelLayer` 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 player, jumping etc. This isn't a zombie game! ☺

So add this test to the beginning of `update`: in each `Enemy` subclass (in **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 now. You should be able to jump on enemies to trigger their dying animation, removing them from the level. STOMP!

# You can dish it, but can you take it?

Handling damage to the player is a bit more complicated. First, Cyclops will have enough life to withstand five hits, and you'll need to account for that. Second, you'll need to update the life meter on the `HUDLayer` to reflect Cyclops' 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 Cyclops' transparency to indicate to the user when they are invulnerable, and restore 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 Cyclops has taken damage.

Finally, wouldn't it be nice if Cyclops got a little bounce after he lands on top of an enemy, as in other platformer games such as Mario Bros.? You'll add that, too.

## Life before death

First initialize the value of the `life` property at the end of the `if` condition in `initWithSpriteFrameName:` in **Player.m**:

```
self.life = 500;
```

Now you can add the `tookHit:` implementation to **Player.m**:

```
-(void)tookHit:(Character *)character {
    //1
    self.life = self.life - 100;
    if (self.life < 0) {
        self.life = 0;
    }
    //2
    [[NSNotificationCenter defaultCenter]
postNotificationName:@"LifeUpdate" object:self userInfo:@{@"life"
: @((float)self.life / 500.0)}];
    //3
    if (self.life <= 0) {
        //4
        [self changeState:kStateDead];
    } else {
        //5
        self.opacity = 122;
        self.isActive = NO;
        //6
        if (self.position.x < character.position.x) {
            self.velocity = ccp(-kKnockback / 2, kKnockback);
        } else {
            self.velocity = ccp(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. Cyclops starts with 500 life points, and you reduce his life by 100 per hit, meaning five hits before Cyclops is dead. This coincides with the number of segments in the life bar on the `HUDLayer`.

2. This is how you communicate with the `HUDLayer`. There are a couple of different ways you could have done this, but here you use `NSNotificationCenter` to post a notification that will include a dictionary containing the current value of the player's `life` property.

   The `HUDLayer` is set as an observer of this notification. When it fires, the `HUDLayer` will receive the notification, take the `life` property and use that value to update the life bar. You'll get to that code soon.

3. Next, check to see if the value of `life` property is zero.
4. If it is, the player is dead and you change their state to `kStateDead`.
5. If the player is still alive after the hit, then you need to do a few things. You change the player's opacity to half to indicate that the player is in the cool-down state. You set the `isActive` property to `NO`. This will temporarily stop collisions from being detected.
6. Test whether the enemy is to the right or left of the player. Depending on which side the collision occurred, you set the player to be knocked back in the opposite direction.
7. Finally, reverse those modifications to the player back to resume collisions after the cool-down period ends. This line calls a function 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 function to end the cool-down period:

```
-(void)coolDownFinished {
    self.opacity = 255;
    self.isActive = YES;
}
```

This just resets the values of the two variables set above so that Cyclops is again fully visible and collision-enabled.

Now add the handling of `kStateDead` to the player's `changeState:` method by inserting the following right before the `default` handler in `changeState:`:

```
case kStateDead:
    action = [CCSequence actions:
              [CCAnimate actionWithAnimation:dyingAnim],
              [CCDelayTime actionWithDuration:0.5],
```

```
                [CCCallFunc actionWithTarget:self
selector:@selector(endGame)],
                 nil ];
    break;
```

This new state calls the `dyingAnim` animation and then the `endGame` function. Add `dyingAnim` to the instance variable list in the class extension at the top of **Player.m**:

```
CCAnimation *dyingAnim;
```

Then initialize it in `loadAnimations` as follows:

```
dyingAnim = [self loadAnimationFromPlist:@"dyingAnim"
  forClass:@"Player"];
```

Next you want to add a check to `update:` so that you can no longer move the player after their 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;
}
```

Now add an empty implementation for `endGame`. For now, this function doesn't do anything much. 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 Cyclops' transparency increase after taking a hit. Time out!

If you take enough hits, you should see the dying animation for Cyclops:

Great! Cyclops has joined us mortals and is able to take damage and die. The game is almost complete!

## Allez-oop!

Remember how I mentioned giving Cyclops a little bounce after he lands on top of an enemy? You're going to implement that now. In the process, you'll also fix another issue.

Testing the game just now, did you notice that when Cyclops lands on top of an enemy, he takes damage as well? That isn't supposed to happen. If the player jumps on an enemy, only the enemy should take a hit. You'll see how to ensure this when you implement the bounce.

First add the following method declaration to **Player.h**:

```
-(void)bounce;
```

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

```
-(void)bounce {
    //1
    self.velocity = ccp(self.velocity.x, kJumpForce / 3);
    //2
    self.isActive = NO;
    //3
    [self performSelector:@selector(coolDownFinished)
      withObject:nil afterDelay:.5];
}
```

Here's what's happening:

1. Add an upward force, 1/3 the strength of a jump, to the player.
2. Sometimes when Cyclops jumps on top of an enemy, he'll be in collision for multiple loops. In order to avoid the player getting hit when they deliver a blow, you need to set the player's `isActive` property to `NO`. This will prevent the next loop from another collision.
3. Call the method that turns `isActive` back to `YES`.

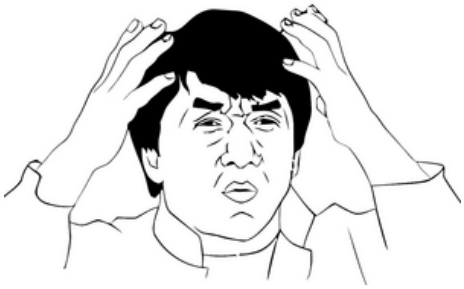
Now you just need to call the above method. Add this line to `checkForEnemyCollisions:` in **GameLevelLayer.m**, right before the `[e tookhit:player];` line:

```
[player bounce];
```

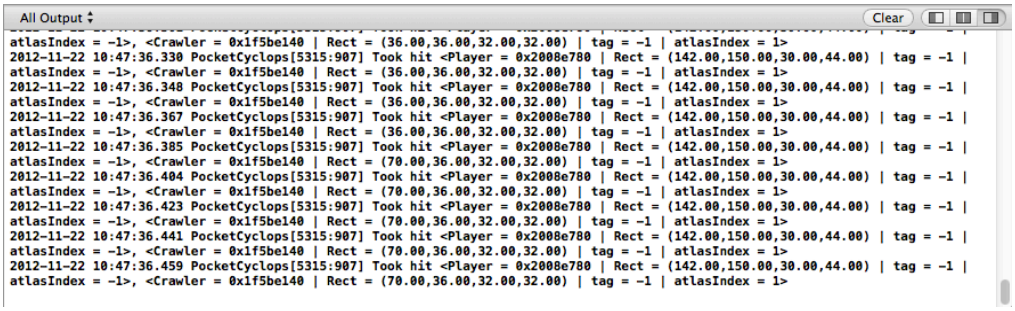
Build and run again. The player should no longer take a hit when damaging an enemy. And of course, Cyclops gets a little bounce after inflicting his blow:

## Updating the HUD

You may have noticed the life bar on the `HUDLayer` changing with each hit. What is that, magic?



Also, there's a function logging the value passed to the `HUDLayer`.




As you may have guessed, the `HUDLayer` already contains the code that reacts to the `NSNotification` and updates the life bar accordingly. While this means there's no need for you to add it yourself, I've provided the relevant code below so you can consider how it works.

It's pretty simple! Open **HUDLayer.m**, and you'll see this code in the `init` method:

```
[[NSNotificationCenter defaultCenter] addObserver:self
selector:@selector(setLifeMeter:) name:@"LifeUpdate" object:nil];
```

This registers the `HUDLayer` as being interested in when the LifeUpdate notification occurs. When it does, the `setLifeMeter` method will be called. And here's the implementation of that:

```
-(void)setLifeMeter:(NSNotification *)note {
    float pct = [[[note userInfo] objectForKey:@"life"]
      floatValue];
    int num = (int)(pct * 5);
    NSString *lifeFrame = [NSString
      stringWithFormat:@"Life_Bar_%d_5.png", num];
    [lifeMeter setDisplayFrame:[[CCSpriteFrameCache
      sharedSpriteFrameCache] spriteFrameByName:lifeFrame]];
}
```

Using the `life` value in the `userInfo` dictionary, you generate a number between one and five. Then you use that to get the appropriate sprite frame to load onto the `lifeMeter` object.

Notifications are pretty handy, eh? That's all it takes to communicate with the HUD.

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!

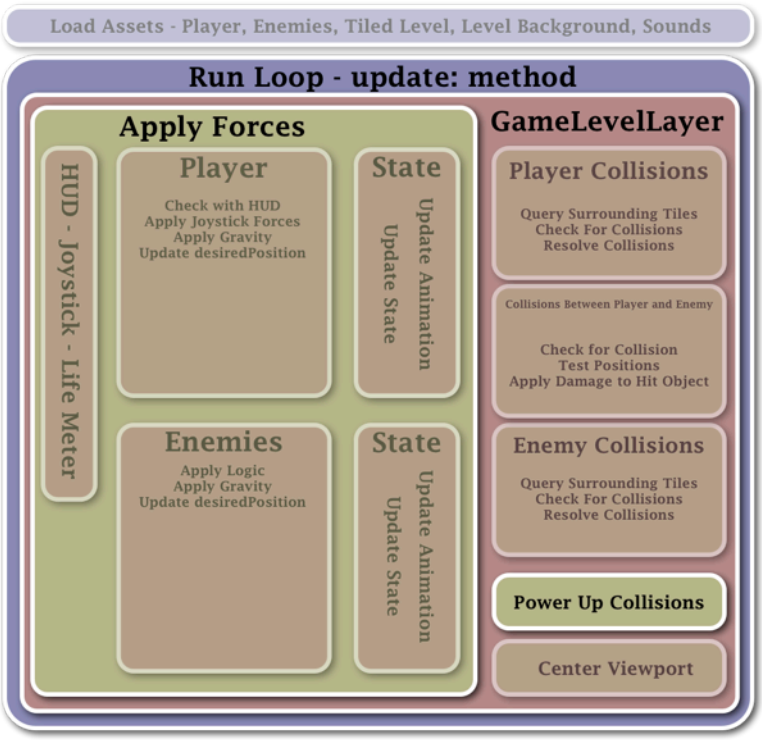
# Chapter 9: Finishing Touches

Your platform game has all the fundamentals in place, but some final stitches remain:

- You need to set up the power-up objects so that 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 some 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:




# Collisions with objects of power

Recall that a few chapters back, you turned on both the 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 scott-free. Now it's time to make him earn those powers!

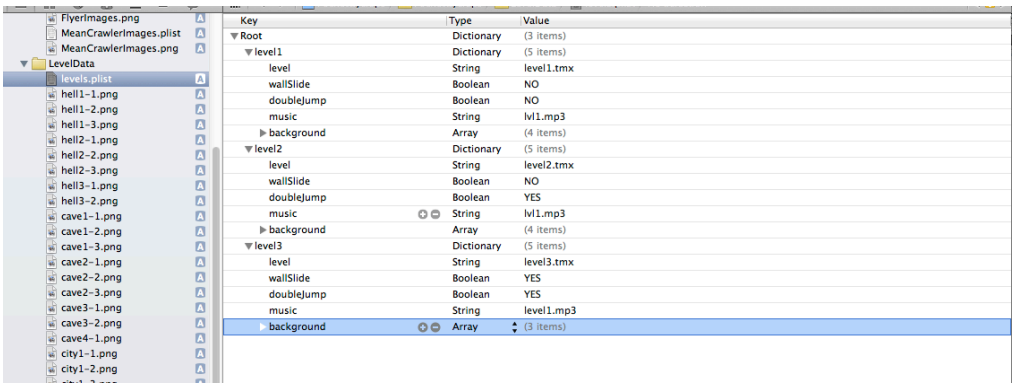
You'll add new objects to the levels and when a collision happens between Cyclops and one of these objects, it will activate either the wall slide or the double jump. But you've got to take these abilities away from Cyclops before you can give them back!

## Demoting Cyclops

Go into **level.plist** (in the **LevelData** folder) and expand the dictionaries within it. Change both the `doubleJump` and `wallSlide` attributes to `NO` in the level 1 dictionary. In the level 2 dictionary, change `wallSlide` to `NO`.

Yep, you guessed it: Cyclops will obtain the `doubleJump` in the first level and the `wallSlide` in the second. But at the beginning of those levels, the player won't yet possess those powers.

The PLIST should look like this:




The player now needs two new variables 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 `initWithLevel:` in **GameLevelLayer.m** and find the line that sets the player's position. Immediately after that line, add the following code:

```
player.canDoubleJump = [[lvlDict objectForKey:@"doubleJump"]
boolValue];
player.canWallSlide = [[lvlDict objectForKey:@"wallSlide"]
boolValue];
```

This sets the player'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 the double jump. Find this line in `update:` in **Player.m**:

```
if (self.characterState == kStateJumping && jumpReset) {
```

And change it to this:

```
if (self.characterState == kStateJumping && jumpReset &&
self.canDoubleJump) {
```

This is the point in your logic where you determine if the player should enter the double-jump state. Now, if the `canDoubleJump` property is `NO`, the double-jump state will never be triggered. All the other logic is unaffected by this change.

The wall slide is next. Find this line:

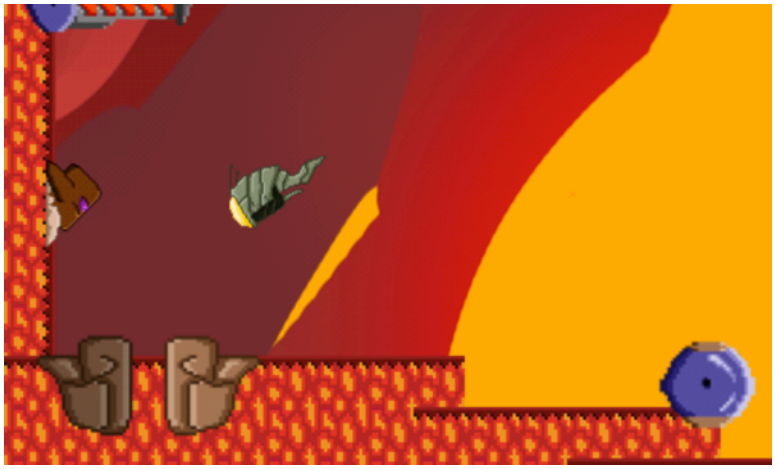
```
} else if (self.onWall && self.velocity.y < 0) {
```

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 state is never entered.

Build and run now. Test different levels to verify that double jumping and wall sliding are no longer available to you in level 1, but still work in level 3.



## Power-up objects

Good job! Now for some fun – it's time to add the power-up objects that will give the player 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 Cyclops eats for breakfast.

Code-wise, you're going to create a new `PowerUp` object that will be a subclass of `GameObject`. This way, `PowerUp` objects will be initialized by `initWithSpriteFrameName:` and will 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, in your own hit game. ☺

Create a new file with the **iOS\cocos2d v2.x\CCNode class** template. Make it a subclass of `GameObject` and name the object **PowerUp**.

Open the new **PowerUp.h** file. Change the import statement from `#import "cocos2d.h"` to:

```
#import "GameObject.h"
```

Now, still in **PowerUp.h**, add a new property:

```
@property (nonatomic, strong) NSString *powerUpType;
```

This will 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 **GameLevelLayer.m** and import the new class:

```
#import "PowerUp.h"
```

Next add a new instance variable to the class extension:

```
NSMutableArray *powerUpsArray;
```

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 Cyclops gets a power-up he'll always have it. However, when you create your own game it's likely that you'll want more than one in a single level. Hence, the array.

Adding power-ups to the level and checking for collisions will be similar to what you did with the enemies, but a bit less complicated.


To load the power-up objects, add a new method in **GameLevelLayer.m**:

```
-(void)loadPowerUps {
    //1
    CCTMXObjectGroup *powerUps = [map
objectGroupNamed:@"powerups"];
    //2
    powerUpsArray = [NSMutableArray array];
    //3
```

```
    [[CCSpriteFrameCache sharedSpriteFrameCache]
addSpriteFramesWithFile:@"PowerUpsImages.plist"];
    //4
    for (NSDictionary *powerUp in powerUps.objects) {
        //5
        NSString *powerUpType = [powerUp objectForKey:@"type"];
        //6
        NSString *spriteFrameName = [NSString
stringWithFormat:@"%@.png", powerUpType];
        //7
        PowerUp *pu = [[PowerUp alloc]
initWithSpriteFrameName:spriteFrameName];
        //8
        pu.position = ccp([[powerUp objectForKey:@"x"]
floatValue], [[powerUp objectForKey:@"y"] floatValue]);
        //9
        pu.powerUpType = powerUpType;
        //10
        [powerUpsArray addObject:pu];
        [self addChild:pu];    }
}
```

Here's a step-by-step walkthrough of what this method is doing:

1. Load the `CCTMXObjectGroup` from the tile map file. There's a layer named **powerups** that contains all the power-up objects for each level.
2. Create the array that will be used to manage the power-ups.
3. Load the sprite frames for the power-up objects. This is hardly more efficient than using individual PNG files in this particular case, because there are only two images in this sprite sheet. It's more for the sake of convenience, since you've done it that way for all the other objects.

   You won't be using a `CCSpriteBatchNode` because there will only be one power-up drawn per level, so there's no gain from doing so.
4. Start a loop that iterates through all the power-up objects in the layer. There's only one in this game, but again, this structure is more convenient and could be expanded to accommodate multiple power-up objects.
5. Load the `type` value for the power-up object. You'll use this value later to determine whether it's a double jump or wall slide power-up.
6. Create the name of the sprite frame.
7. Initialize a new `PowerUp` object using the sprite frame name just created.
8. Set the position the same way you've done before, from the information contained in the object dictionary.
9. Set the `powerUpType` attribute using the value from step #5.

10. Add the object to both the layer and the array.

This code should all look fairly familiar. Now add the following line to `initWithLevel:` 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:

## The return of Super Cyclops

As you may have predicted, touching the power-up berry doesn't do anything yet. Time to fix that!

Add the following method to **GameLevelLayer.m**:

```
-(void)checkForPowerUpsCollisions {
    //1
    NSMutableArray *removeArray = [NSMutableArray array];
    //2
    for (PowerUp *p in powerUpsArray) {
        //3
        if (CGRectIntersectsRect([player collisionBoundingBox],
p.boundingBox)) {
            //4
            if ([p.powerUpType isEqualToString:@"DoubleJump"]) {
                player.canDoubleJump = YES;
            } else {
                player.canWallSlide = YES;
            }
            //5
```

```
                [removeArray addObject:p];
            }
        }
        //6
        for (PowerUp *p in removeArray) {
            //7
            [powerUpsArray removeObject:p];
            [p removeFromParentAndCleanup:YES];
        }
    }
```

Here's what's happening:

1. Create a deletion array that will contain a power-up that you've retrieved. As mentioned before, you can't remove an object from an array while you are iterating through that array, so you add it to a new array. Then, you use the new remove array to delete those objects from the powerUpsArray.
2. Start iterating through all the power-up objects in the `powerUpsArray`.
3. Test whether the bounding box of the power-up object and the collision bounding box of the player intersect. If they do, then you need to trigger the logic that will give the player the power-up ability and remove the power-up from the level.
4. Test whether the `powerUpType` string is "DoubleJump". If it is, then set `canDoubleJump` to `YES`. If it's not, then it must be "WallSlide", so enable the ability.
5. Add the power-up to the deletion array.
6. Iterate through `removeArray`.
7. Removal entails both removing the `PowerUp` object from the layer by calling `removeFromParentAndCleanup:` and removing it from `powerUpsArray`.

The only thing left to do is call this new method each frame via `update:`. Add the following line before the call to `setViewpointCenter`:

```
    [self checkForPowerUpsCollisions];
```

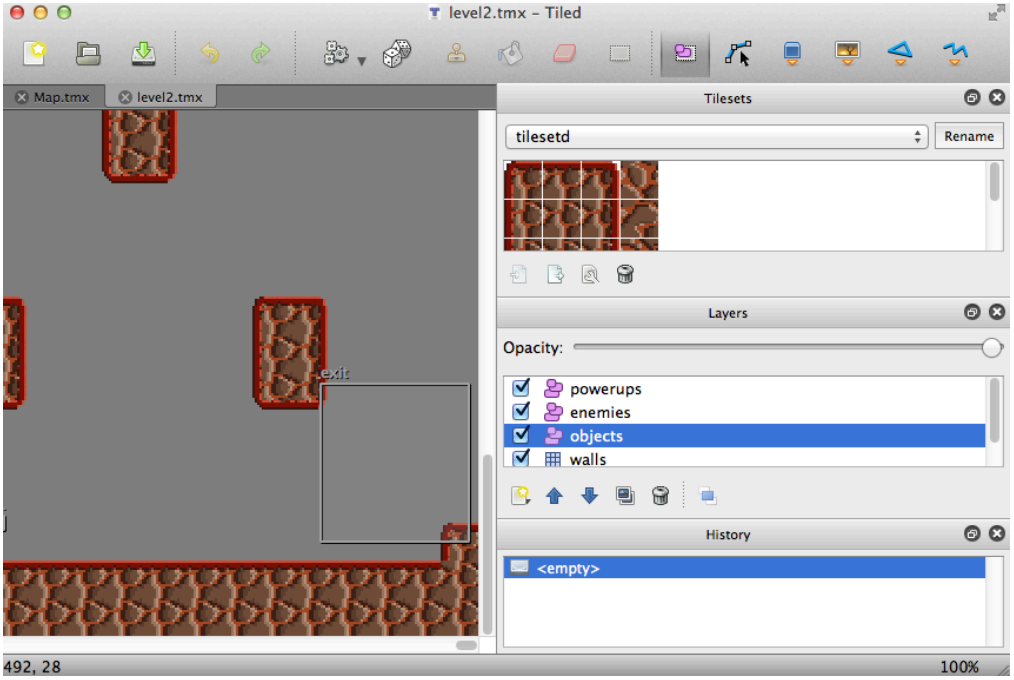
Build and run. You should now be able to find and obtain the double-jump power-up in the first level, and the wall slide power-up in the second level.

At this time there's no visual cue when the player picks up the power-up, but that might be a good thing to practice adding on your own! You could make the character blink, fire a particle system, pause the game ala *Metroid*, etc.

> **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 start 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 player is within 100 pixels 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 instance variable for the exit point. Add this to the class extension in **GameLevelLayer.m**:

```
CGPoint exitPoint;
```

Then in `initWithLevel:`, after the line that loads the `PowerUp` objects, add the code to set up the exit point:

```
NSDictionary *exit = [og objectNamed:@"exit"];
exitPoint = ccp([exit[@"x"] floatValue] + ([exit[@"width"]
floatValue] / 2), [exit[@"y"] floatValue] + ([exit[@"height"]
floatValue] / 2));
```

This loads the information about the exit (a rect) into a dictionary. Then it calculates the center point for the exit and sets the `exitPoint` variable to that center point.

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 **GameLevelLayer.m**:

```
-(void)checkForExit {
    //1
    float distanceToExit = ccpDistance(player.position,
exitPoint);
    if (distanceToExit < 100) {
```

```
        //2
        if (currentLevel < 3) {
            //3
            int nextLevel = currentLevel + 1;
            [[CCDirector sharedDirector]
replaceScene:[GameLevelLayer sceneWithLevel:nextLevel]];
        } else {
            //4
            CCLabelTTF *win = [CCLabelTTF labelWithString:@"You
Win" fontName:@"Marker Felt" fontSize:60];
            CGSize winSize = [[CCDirector sharedDirector]
winSize];
            win.position = ccp(winSize.width / 2.0 -
self.position.x, 160 - self.position.y);
            [self addChild:win];
            //5
            [self performSelector:@selector(restart)
withObject:nil afterDelay:3.0];
        }
    }
}
```

This method is simple enough:

1. Determine if the player is close enough to the exit to end the level. `ccpDistance` returns a `float` value that's the distance in pixels between two points, in this case the player and the exit.
2. Check the current level. If you're on a level before 3, then you want to progress to the next level.
3. If you want to progress to the next level, you need to call `replaceScene` and pass a new scene. Here you use the `GameLevelLayer` initializer that takes a level number.
4. If the player has reached the end of level 3, they've won the game, so let them know! You need to position the label so that it's centered on the screen. However, the current layer is moved as Cyclops progresses through the level, so if you just put it at position 240, 160, it will be off the screen (unless Cyclops dies before moving beyond the starting position). For this reason, you need to take the current layer position into account when calculating the label position.
5. Finally, dismiss the view and go back to the level selection screen. The `restart` method will do that part, but here you call the method after a three-second delay so that the user can see the win label first.

Now you need to call the above method as the last call in `update:`. Add that code:

```
[self checkForExit];
```

Next implement the `restart` method (still in **GameLevelLayer.m**):

```
-(void)restart {
    [[NSNotificationCenter defaultCenter]
postNotificationName:@"restart" object:nil];
}
```

Because you don't have a direct way to talk to the `ViewController` (the controller of the Cocos2D view), you need a way to pass a message. You could create a delegate relationship, or even just pass a reference, but you'll use an `NSNotification` instead since it's nice and simple.

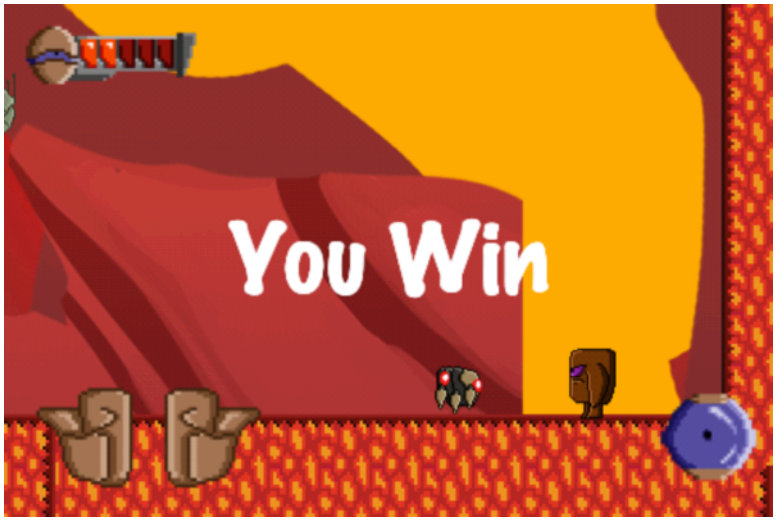
The `GameLevelLayer` will post the notification, and the `GameViewController` (included in the UIKit menu resources provided with the starter project) will observe it.

This code is already included in the provided resources, and so you don't need to add it. You can see the observer set up in `viewDidLoad` in **GameViewController.m**:

```
[[NSNotificationCenter defaultCenter]
addObserverForName:@"restart" object:nil queue:[NSOperationQueue
mainQueue] usingBlock:^(NSNotification *note) {
    [director replaceScene:[CCScene node]];
    [director pause];
    [[SimpleAudioEngine sharedEngine] stopBackgroundMusic];
    [self.navigationController popViewControllerAnimated:YES];
}];
```

This code block first pauses the director, stops the background music, and then pops the current view so that execution will return to the level select menu.

Build and run. You should now be able to move from level to level, and win the game!



# In the event of defeat

Winning's no fun if you can't lose, right? So 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 `GameLevelLayer` 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 Cyclops' 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 `GameLevelLayer` about the end of the game.

But how does the `Player` class identify the object to which it needs to send the message? Each `CCNode` has a parent property that points to the immediate parent object of that node. This is perfectly adequate in your case since the `GameLevelLayer` happens to be the parent for the `Player` instance.

> **Note**: There is one disadvantage to this technique. If your code became more complicated, or if you were building these classes as part of a team, then you might not know how many nodes are between the `Player` and the `GameLevelLayer`. In that case, just using the parent would not be a reliable way to access that object, and you might want to pass the layer as a parameter to the init method of the `Player`.

Replace `endGame` in **Player.m** with the following:

```
-(void)endGame {
    GameLevelLayer *gll = (GameLevelLayer *)self.parent;
    [gll gameEnded];
}
```

However, the `Player` class doesn't know anything about `GameLevelLayer`. So you need to import that class. Add this import at the top of **Player.m**:

```
#import "GameLevelLayer.h"
```

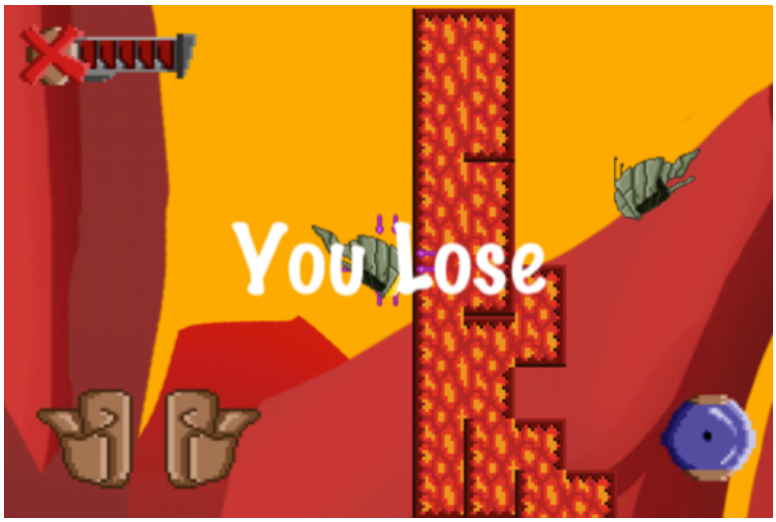
`GameLevelLayer` currently doesn't have a `gameEnded` method. First add the method declaration to **GameLevelLayer.h**:

```
-(void)gameEnded;
```

Then implement the method in **GameLevelLayer.m**. It looks a lot like the code for the win scenario:

```
-(void)gameEnded {
    CCLabelTTF *gameOver = [CCLabelTTF labelWithString:@"You Lose"
fontName:@"Marker Felt" fontSize:60];
    CGSize winSize = [[CCDirector sharedDirector] winSize];
    gameOver.position = ccp(winSize.width / 2.0 - self.position.x,
160 - self.position.y);

    [self addChild:gameOver];
    [self performSelector:@selector(restart) withObject:nil
afterDelay:5.0];
}
```

Build and run. Find an enemy and let it take your 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.

First you need to import the `SimpleAudioEngine` in the right place. Since both enemies and the player will trigger sound effects, you can import it in the `Character` class so that all of its subclasses will have access to `SimpleAudioEngine`.

Add the following line to **Character.h**:

```
#import "SimpleAudioEngine.h"
```

Start with the player's sounds. You want one sound for jumping, another for double jumping, one for dying and one for when Cyclops bounces off the top of an enemy. The first three can be added to `changeState:` easily, so start there.

Alter the switch statement in `changeState:` in **Player.m** as follows:

```
switch (newState) {
    case kStateStanding:
        [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"Player1.png"]];
        break;
    case kStateWalking:
        action = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:walkingAnim]];
        break;
    case kStateWallSliding:
        action = [CCRepeatForever actionWithAction:[CCAnimate
actionWithAnimation:wallSlideAnim]];
        break;
    case kStateJumping:
        [[SimpleAudioEngine sharedEngine]
playEffect:@"jump1.wav"];
        action = [CCAnimate actionWithAnimation:jumpUpAnim];
        break;
    case kStateDoubleJumping:
        [[SimpleAudioEngine sharedEngine]
playEffect:@"jump2.wav"];
        [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"Player10.png"]];
        break;
    case kStateDead:
        [[SimpleAudioEngine sharedEngine]
playEffect:@"player_die.wav"];
        action = [CCSequence actions:
                  [CCAnimate actionWithAnimation:dyingAnim],
                  [CCDelayTime actionWithDuration:0.5],
                  [CCCallFunc actionWithTarget:self
selector:@selector(endGame)],
                  nil ];
        break;
    default:
        [self setDisplayFrame:[[CCSpriteFrameCache
sharedSpriteFrameCache] spriteFrameByName:@"Player1.png"]];
        break;
}
```

You can see that in the cases of the jump, double jump and dying states, you've added a line that calls `playEffect:` on the `SimpleAudioEngine` singleton. The `playEffect` method takes a string that's the name of an audio file (it's best if you use a WAV).

For the fourth player sound effect, add the following call to the beginning of the `bounce` method (before all the existing code):

```
[[SimpleAudioEngine sharedEngine] playEffect:@"bounce.wav"];
```

Just like above, it adds a call to `playEffect:`.

Now you'll do the same thing to the `Enemy` subclasses. All the enemies will play a sound when they die, for a satisfyingly good riddance.

Start with the `Crawler` class. Inside `changeState:`, add this line to the `kStateDead` case:

```
[[SimpleAudioEngine sharedEngine] playEffect:@"crawler_die.wav"];
```

Now in the `MeanCrawler` class, add that very same line to `changeState:` (again for the `kStateDead` case). In addition to that, add this line for `kStateJumping`:

```
[[SimpleAudioEngine sharedEngine] playEffect:@"crawler_jump.wav"];
```

The `Flyer` class also needs some sounds. You'll add three to the `Flyer`.

Add this to `kStateDead`:

```
[[SimpleAudioEngine sharedEngine] playEffect:@"flyerdie.wav"];
```

Now this to `kStateAttacking`:

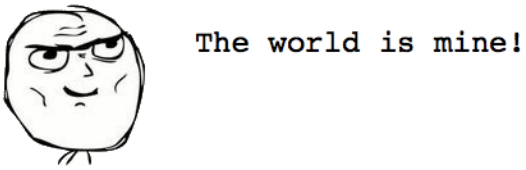
```
[[SimpleAudioEngine sharedEngine] playEffect:@"flyerattack.wav"];
```

Finally, add this to `kStateHiding`:

```
[[SimpleAudioEngine sharedEngine]
playEffect:@"flyercloseeye.wav"];
```

That's it! Build and run, and you should have a complete platformer game – with sound effects and all!

Congratulations! You have completed the Platformer Game Starter Kit! You can now do something awesome that you might not have been able to do before. ☺




**Final Challenge:** In this game, Cyclops has a single life. What would you need to do give the player multiple lives? How would you let the player obtain more lives by collecting objects?

# Parting thoughts

I hope you've enjoyed going through 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 to making a platformer game – now the rest is up to you!

You could add your own art, levels, enemies, powerups. Maybe you want to make your main character shoot a weapon, or maybe you want to add some more advanced AI. No matter what you might like to do, you now 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 start simple. Just like you did in this starter kit, take the simplest idea you can and get it working. Then you can keep adding features and polish step by step until you're happy with it.

Then the most important part – submit it 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 works on there.

I appreciate each and every one of you for taking time to try out the Platformer Game Starter Kit. If you have an extra second, I would really love to hear what you thought of this Starter Kit!

Please leave a comment on the official private forums for the Platformer Game Starter Kit at www.raywenderlich.com/forums. If you do not have access to the forums, you can sign up here:

http://www.raywenderlich.com/forum-signup

Or if you'd rather reach me privately, please don't hesitate to shoot me an email. Although sometimes it takes me a while to respond, I do read each and every email, so please drop me a note!

Please stay in touch, and I look forward to checking out your platformer games!

Jacob Gundersen

fattjake@gmail.com

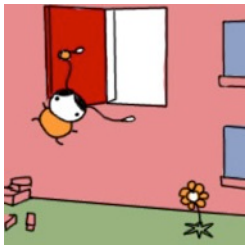
# Appendix A: Interviews with Successful Platformer Game Devs

As a bonus to thank you for purchasing the Platformer Game Starter Kit, I've included some interviews with some of the most successful iOS platformer game developers, so you can get some tips, tricks, and advice from them.

Keep reading to learn some more about guys who have been down in the trenches and have released some great games. Huge thanks to Ben and Tony for taking the time to share their insights and experiences!

# About the Interviewees

**Ben Hopkins** (BH) is the developer behind **1-Bit Ninja**. He does coding for web, iOS and OSX. Papervision3D team member. Subsonic scientist. Other apps include HoloToy, GLSL Studio, Simul80 and Oh Hi! Octopi! You can find him 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 onto the Atari ST and PC/Consoles. He met Nick (@physmo) at University and they started developing and selling our own games together. They've both had full-time jobs in the games industry and other programming positions so now we just do games for fun. You can find him 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 aspect of a platform game on any system is the controls. On mobile devices, many of which have no physical buttons, this especially true.

Platform games designed for mobile devices that employ virtual, on screen 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 utilize a control scheme other than traditional on screen 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 weakness's and design with those in mind, constraints offer a great opportunity to experiment with new ideas.

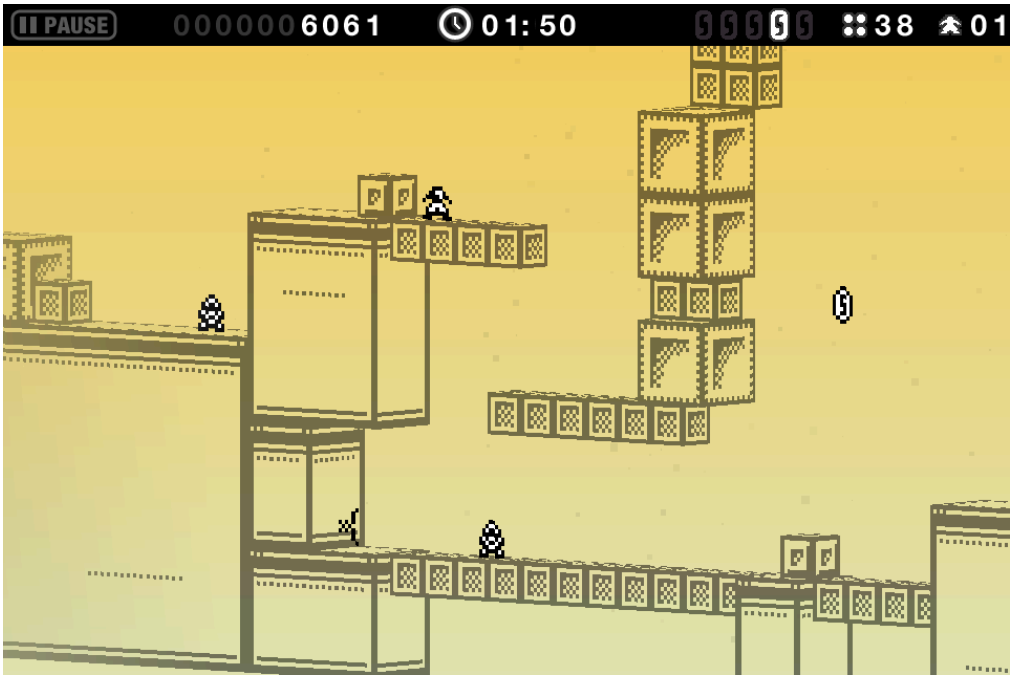

**TM:** I think the most important aspect of building a platformer for a mobile device is the touch controls. We worked hard on Mos Speedrun to get the buttons in the

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 in order 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 different tasks in each level (collect all the coins, find the hidden skull etc...)




BH: Variety and progression in level design is very important. Trying to make interactive elements such as enemies etc. unique from one another so that each one presents a different advantage or obstacle to the player, 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.

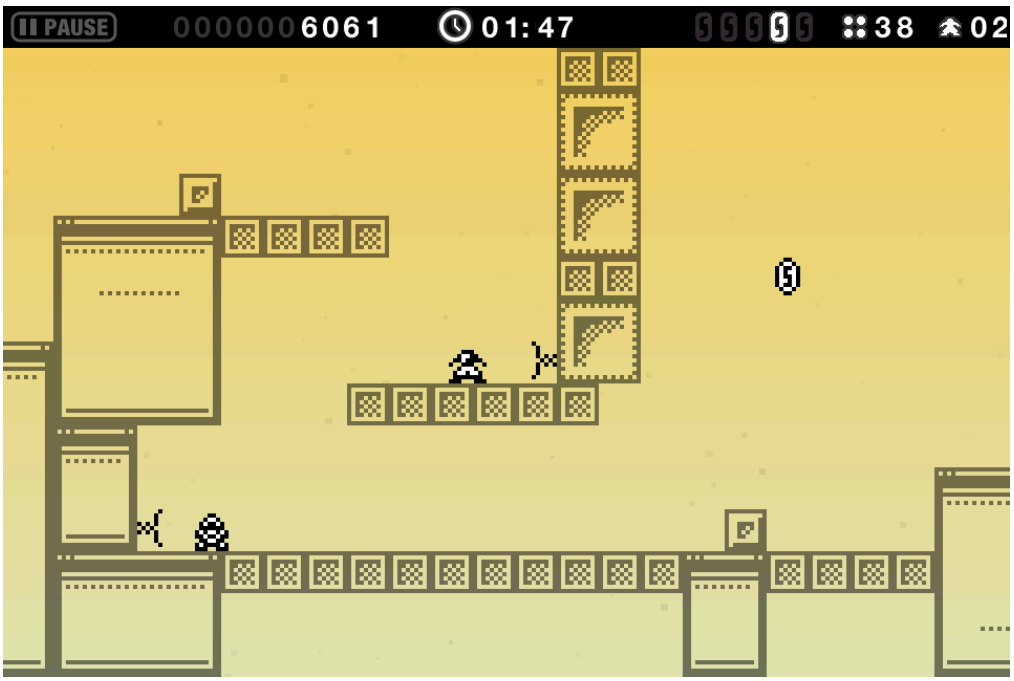
Above all, experiment, I always have a number of 'test' levels that are simply one or two rooms used for trying out ideas that can then be fleshed out and included in full levels.

*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 elements update function, the majority just using velocity, some using sine curves and only checking 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 that enabled some interesting design puzzles however their use also required extra planning and play testing.

If the player's direction is reversed by a spring, for example, then there needs to be a corresponding spring somewhere to put them back on track. Added complications come 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 it and reach an unintended point while reversed.

During the initial planning stage I created a spreadsheet that listed all 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?*

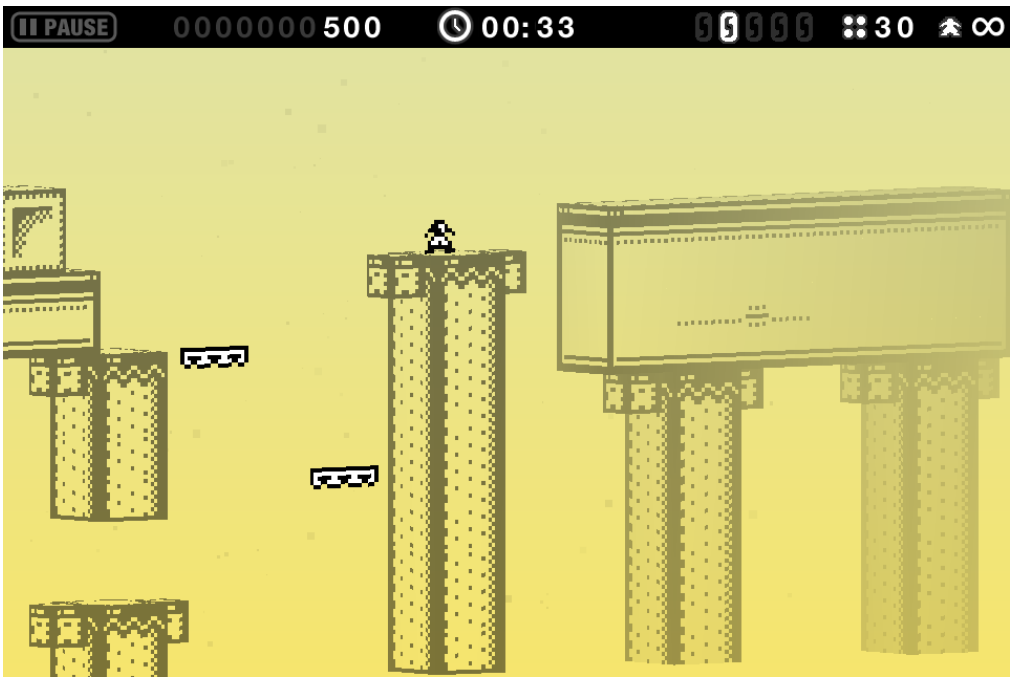

**TM:** We wanted the AI behaviours to be predictable so the players didn't get annoyed. Because it's a speedrun game the movement couldn't be random and the enemies had to be in the same positions each time you played the game.

**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.

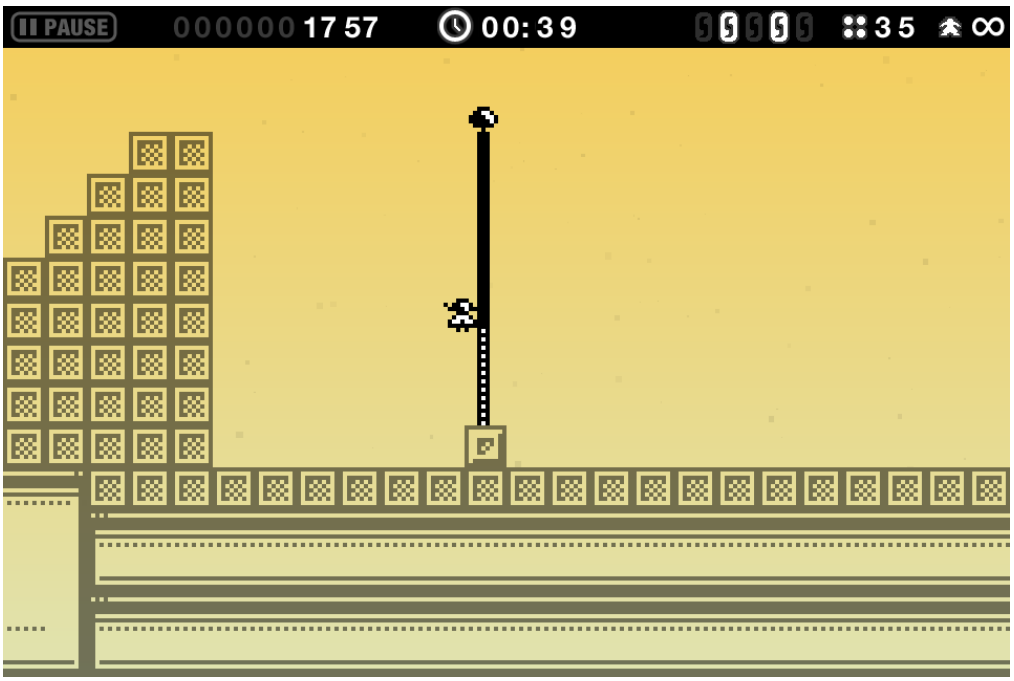
*6) Any favorite features that ended up being cut from your game (for whatever reason, in order to ship, performance problems, etc)?*

**TM:** Not really, we would have liked to have more levels but it's quite time consuming creating 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 level of complexity both in terms of design and technology would require a large amount of time 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 have come to recreating the magic of Super Mario World.

**TM:** Think it would have to be Super Mario on the SNES : )

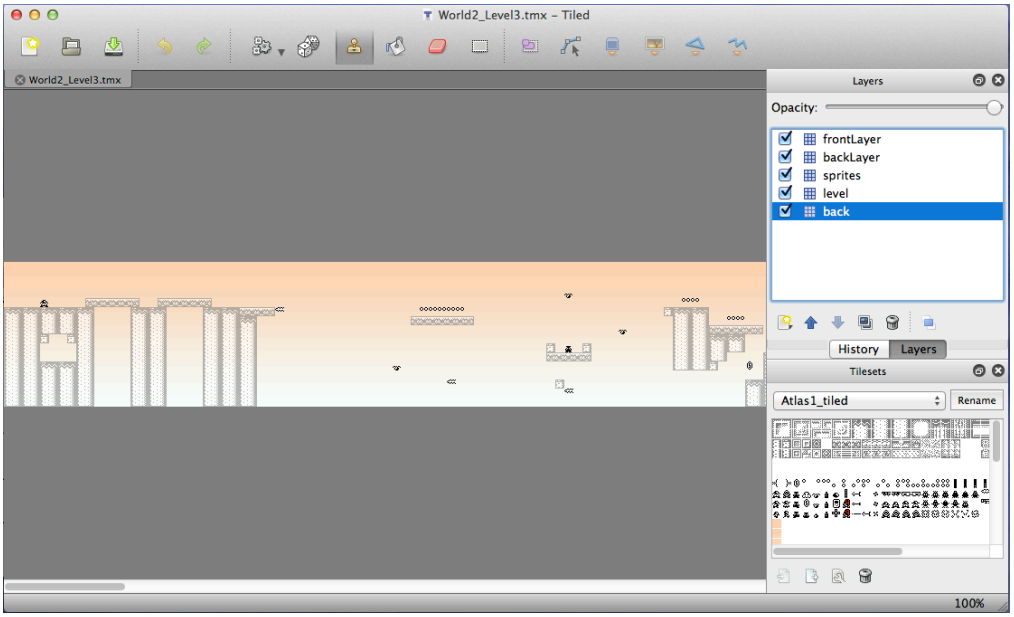
*8) Any other tips that you'd like to share with aspiring platformer game designers?*

**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 powerups all have a huge impact on how the player progresses and being time sensitive these

things directly effect 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:** The levels in 1-bit Ninja were all designed using the free open source editor Tiled (http://www.mapeditor.org/). Levels are broken up into 3 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 level's map data is loaded into memory at startup and as the player progresses through the level the 3D mesh is generated dynamically using a basic cube algorithm that creates appropriately sized and positioned cubes based on the tiles 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 are 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. All our development is done 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 XY velocity vector which we add to the players position each time. We have some friction on the ground, acceleration and a maximum speed for the player too. We also added 2 different types of jumps, a small one when the button is pressed quickly and a large one when the button is held down for longer.


**BH:** The jumping behavior in 1-bit Ninja is as simple as you can get. While the jump button is pressed the player's vertical velocity is set to a certain value, when the jump button is released the player's vertical velocity is no longer set 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 for. Bouncing off of enemies also works in the same way, when a collision is detected between the player's feet and the enemy's head, a certain value is added to the player's vertical velocity, if the player has the jump button pressed when this occurs the value added is greater allowing the player to springboard off of enemies.

# That's It!

Again, huge thanks to Ben and Tony for taking the time to share their experience and advice with us.

I hope that these interviews have been interesting and helpful, and that your future platformer game is a great hit as well!