SPACE GAME
STARTER KIT

By Ray Wenderlich

Space Game Starter Kit

Ray Wenderlich

By purchasing the Space Game Starter Kit, you have the following license:

- You are allowed to use and/or modify the source code in the Space Game Starter Kit in as many games as you want, with no attribution required.
- You are allowed to use and/or modify all art, music and sound effects that are included in the Space Game Starter Kit in as many games as you want, but must attribute Vicki Wenderlich of vickiwenderlich.com.
- The source code included in this Space Game Starter Kit is for your own personal use only. You are NOT allowed to distribute or sell the source code in the Space Game Starter Kit without prior authorization.
- Likewise, the chapters in this guide are for your own personal use only. You are NOT allowed to distribute or sell the chapters in this guide without prior authorization.

Table of Contents

Dedication

This starter kit is dedicated to the readers of raywenderlich.com.

Thank you for your continued readership and support!

Introduction

Welcome to the Space Game Starter Kit for iOS!

The Space Game Starter Kit includes full source code for a complete side-scrolling space game for iOS using Sprite Kit, Apple's 2D graphics framework introduced with iOS 7. The game is filled with asteroids, aliens, lasers and explosions—and of course, a bad-ass boss fight at the end!

With the Space Game Starter Kit, not only do you get full source code that you can use in your own games—you also get four epic-length chapters that show you how to build the entire game from scratch.

Whether you're a beginner or an advanced iOS developer, the Space Game Starter Kit gives you some great benefits:

- It serves as an example of a fully-functional game from which to study and learn.
- It has code, art, sound effects, music and particle systems that you can directly reuse in your own games.
- It's a starting point that you can extend to create your own game, saving you time and money.
- It is a great way to dive into iOS game development head first, or reinforce your existing knowledge.
- It teaches tips and tricks that you might not have come across before, such as how to: create a universal app that runs on both iPhone and iPad, define levels in a property list, use Bezier paths for enemy movement, create multiple enemy types and much more.
- By purchasing this starter kit, you've given back to raywenderlich.com, making future tutorials, forum support and starter kits possible.
- It is fun and relaxing to go through the step-by-step chapters and learn along the way!
- Plus, once you're finished, it's a lot of fun to show off what you made to your friends and family—they'll think you're an Xcode Jedi master.

Prerequisites

To use this starter kit, you need to be a member of the iOS developer program and have a Mac with Xcode installed. You'll also need an iPhone, iPod touch or iPad to use for testing, because the game uses the accelerometer, which is not available on the Simulator.

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

• http://www.raywenderlich.com/newsletter

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

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

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

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

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

How to Use the Space Game Starter Kit

There are several ways you can make use of the Space Game Starter Kit.

First, you can simply 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 find the related chapters and read up on any sections of code that confuse you. The table of contents can help with that, and the search tool is your friend!

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

Note that you don't necessarily have to go through each chapter—for instance, if you already know how to do everything in Chapter 1, you can skip straight to Chapter 2. The Space Game Starter Kit includes a version of the project for each chapter that includes all the code from the previous chapters, so you can pick up the game at any point you like.

Introducing the Third Edition

It's been a little over two years since I first wrote the Space Game Starter Kit, and a lot has changed since then!

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

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

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

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

I hope you all enjoy the third edition, and thank you again for purchasing the Space Game Starter Kit! ☺

Blast Off!

With this introduction complete, it's time to blast off into your space game.

Grab your favorite caffeinated beverage and some snacks, prop up your feet and get ready for some coding fun—and to blast some aliens!

> **Note:** If you have any questions as you go through the Space Game Starter Kit, please visit our forums at *http://www.raywenderlich.com/forums*.

About the Author

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 Editors

Greg Heo was the tech editor of this starter kit. He is an indie developer and tech partner at Ferocious Apps. He likes caffeine, codes with two-space tabs, and can be found at his standing desk at all hours of the day.

B.C. Phillips was the editor of this starter kit. He 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).

About the Artist

Vicki Wenderlich is a ceramic sculptor who was convinced two years ago to make art for her husband's iPhone apps. She discovered a love of digital art, and has been making app art and digital illustrations ever since. She is passionate about helping people pursue their dreams, and makes free app art for developers available on her website, http://www.vickiwenderlich.com.

Chapter 1: A Basic Space Shooter

In this first chapter, you're going to strap right in and make a simple version of a space game, where you pilot a cool spaceship and blast through a dangerous asteroid belt.

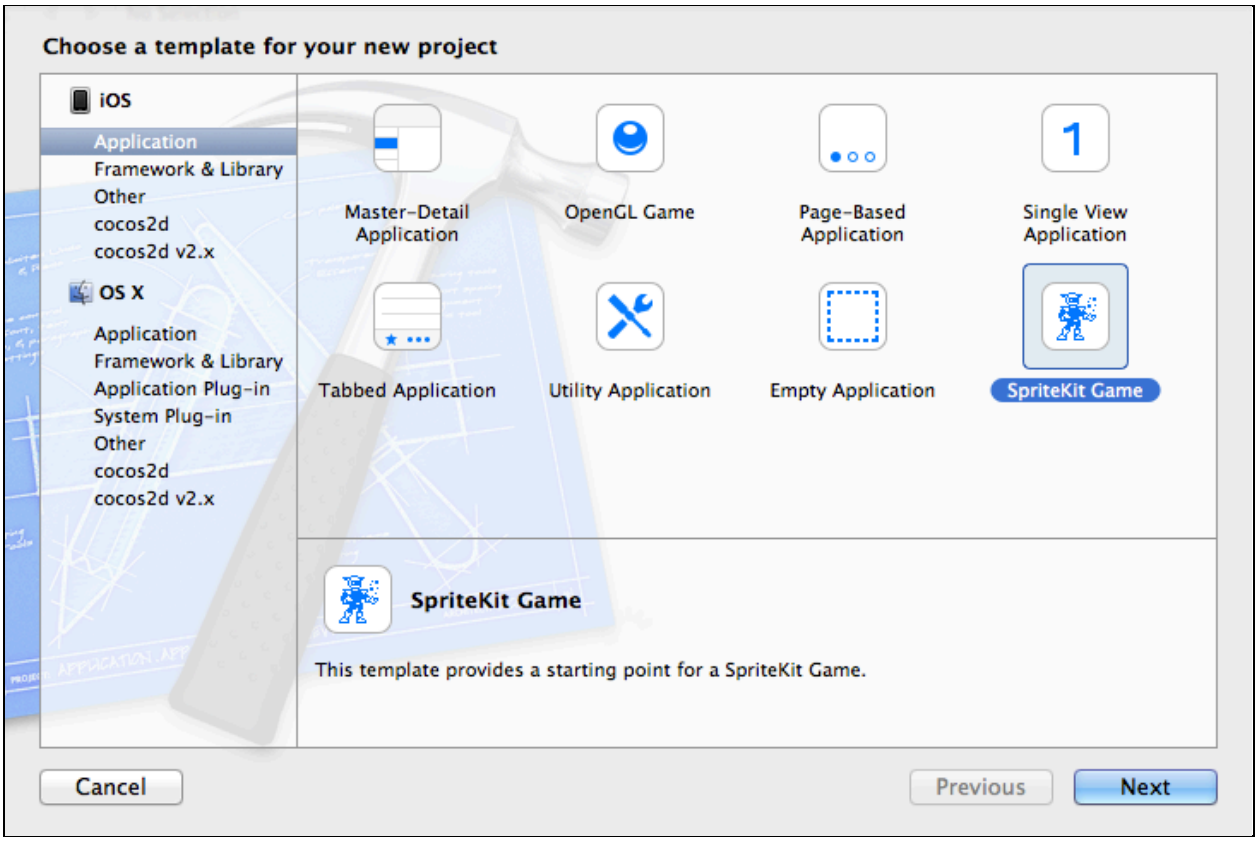
You may notice that this chapter is somewhat similar to the Space Shooter Sprite Kit Tutorial available on my site. However, there are some major differences in this version, such as iPad and Retina display support, a title/menu scene and better sprite preloading. So even if you've done that tutorial before, you should still go through this chapter from the beginning.

Without further ado, let's bravely go where no tutorial has gone before! ☺

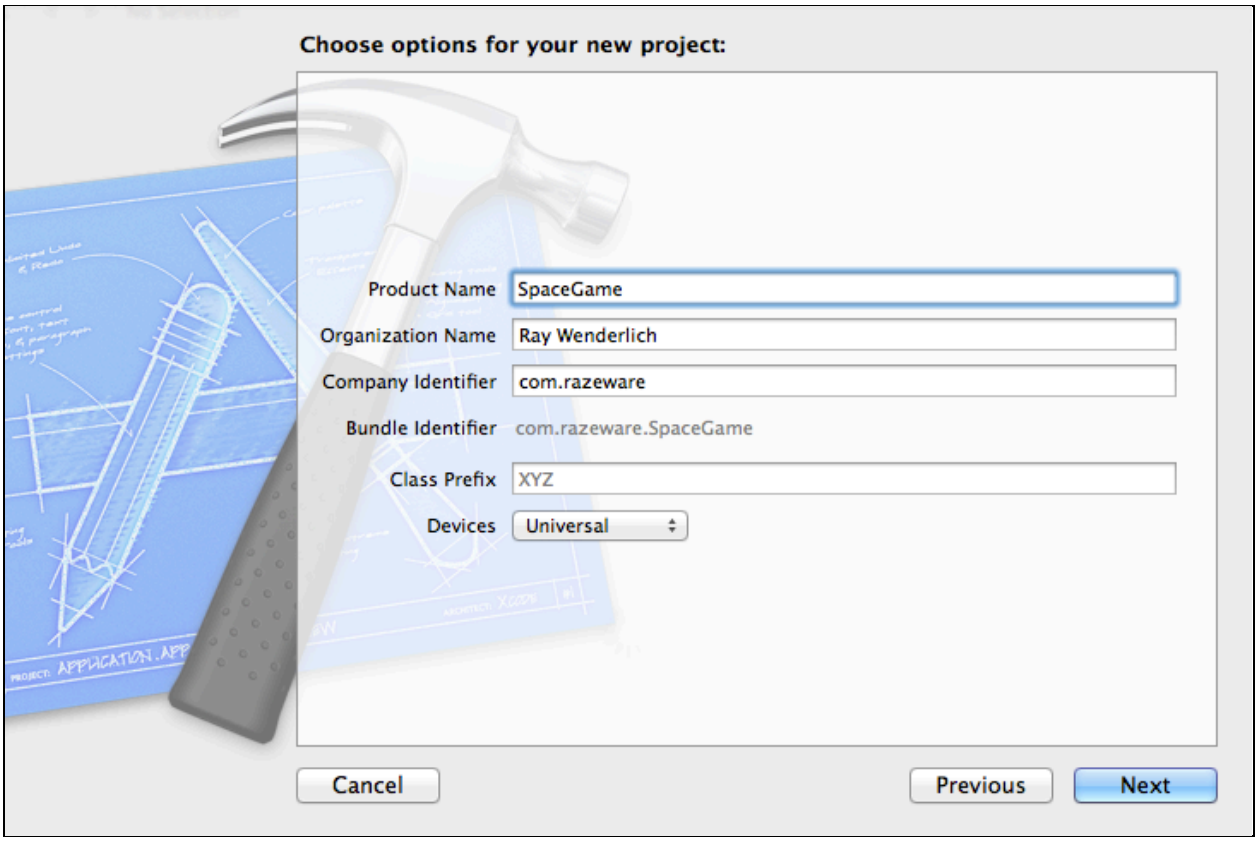
Hello, Sprite Kit!

Let's get started by creating a "Hello, World" Sprite Kit project and then modifying it to display the title of the Space Game Starter Kit.

Start up Xcode, go to **File\New\Project...** and choose the **iOS\Application\Sprite Kit Game** template, as you can see below:

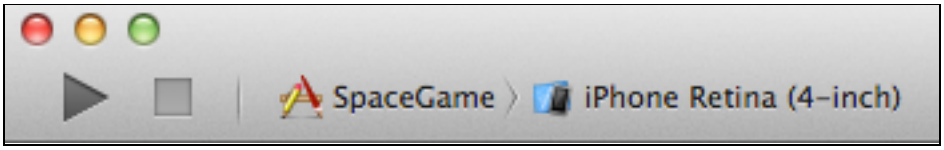


Click **Next**, enter **SpaceGame** for the Product Name, select **Universal** for Device Family and click **Next** again.



In the final step, choose a folder on your hard drive to save the project and click **Create**.

You now have a "Hello, World" Sprite Kit project that the template has set up for you. You can build and run this project if you like—just choose the **iPhone Retina (4-inch)** simulator from the scheme dropdown in the upper left and click the **Play** button to build and run:



At this point, you'll see the text "Hello World" appear on the screen. Also, if you tap anywhere on the screen, a spinning spaceship will appear:

Don't worry—you'll be making a much cooler space game than this! ☺

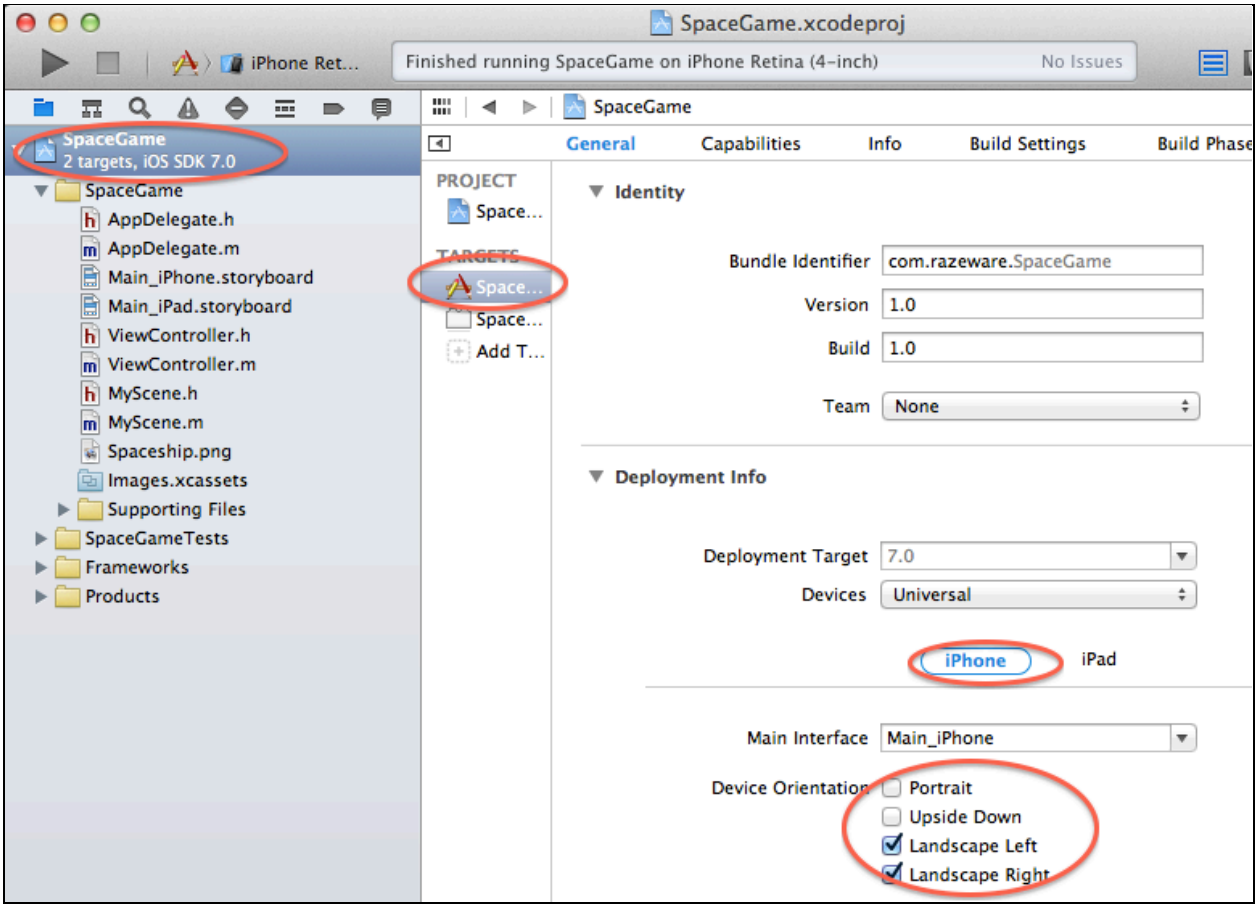
The first step is to remove all of the template code that Xcode sets up for you so you can have a clean starting point. To do this, open **MyScene.m** and replace the contents with the following:

```
#import "MyScene.h"

@implementation MyScene

#pragma mark - Init functions

-(id)initWithSize:(CGSize)size {
  if (self = [super initWithSize:size]) {
    self.backgroundColor = [SKColor blackColor];
  }
  return self;
}

@end
```

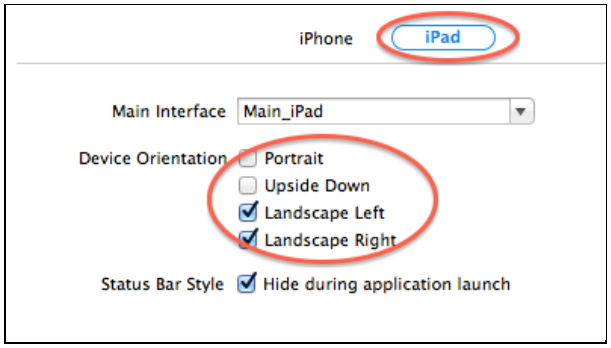
A Sprite Kit game is typically broken up into a set of screens, or **scenes**. The Sprite Kit template starts you out with a single scene called `MyScene`. Here you make the scene as simple as can be—you set the background color to black, and that's it!

Before you build and run again, there are a few final things you should do. You want the Space Game to run in landscape mode instead of the default portrait

mode. To fix this, first select **SpaceGame** in the Project Navigator and make sure the **SpaceGame** target is also selected. In the **General** properties in the **Deployment Info** section for the **iPhone**, uncheck the box for **Portrait** so that only **Landscape Left** and **Landscape Right** are selected:



Remember, you are making this space game a universal app that works on both the iPhone and iPad, so switch to the **iPad** settings and uncheck **Portrait** and **Upside Down** so that only **Landscape Left** and **Landscape Right** are selected there as well:



There are two more steps. First, open **ViewController.m** and replace `viewDidLoad` with the following:

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

  // Configure the view.
  SKView * skView = (SKView *)self.view;
  if (!skView.scene) {
    skView.showsFPS = NO;
    skView.showsNodeCount = NO;

    // Create and configure the scene.
    SKScene * scene = [MyScene sceneWithSize:skView.bounds.size];
    scene.scaleMode = SKSceneScaleModeAspectFill;

    // Present the scene.
    [skView presentScene:scene];
  }
}
```

This is required to make sure that Sprite Kit initializes `MyScene` with the correct scene size in landscape mode. For a more detailed explanation, check out Chapter 1 of *iOS Games by Tutorials*, "Sprites."

Finally, add this method to the bottom of **ViewController.m**, above the `@end` line:

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

This hides the status bar while the game is running, which makes for a nicer gameplay experience.

And that's it—build and run, and enjoy your clean blank slate!

Displaying the Title

Next let's pretty this up by displaying the game's title.

Open **MyScene.m** and modify the `@implementation` line to add a few instance variables you will need:

```
@implementation MyScene {
  SKNode *_gameLayer;
  SKNode *_hudLayer;
  SKLabelNode *_titleLabel1;
  SKLabelNode *_titleLabel2;
}
```

The first two variables represent the two layers the Space Game will have:

1. The `_gameLayer` will contain the gameplay objects such as the player ship, the asteroids, the alien ships and so on.
2. The `_hudLayer` will contain any text that shows up in the game. HUD stands for head-up display, which is a term borrowed from modern aircraft technology. In the context of a video game, it describes the tools and data that you want to overlay the `_gameLayer`.

Separating your game into logical layers like this is good practice. One reason in particular it will come in handy in the Space Game is that it will let you move all of the objects in the game layer at once without affecting the objects in the HUD layer.

The second couple of variables represent the two labels you will use to display the game's title.

> **Note:** I am commonly asked why I like to name my instance variables with underscores.
>
> It's just a personal preference. I like using underscores because when I'm looking through code and see an underscore, I instantly know that I'm working with an instance variable and not something else, like a local variable or property. This also happens to be the same convention Apple uses, so if you use it, you're in good company. ☺

Next, add this new method to create the two layers right after `initWithSize:`:

```
- (void)setupLayers {
  _gameLayer = [SKNode node];
  [self addChild:_gameLayer];
  _hudLayer = [SKNode node];
  [self addChild:_hudLayer];
}
```

The layers are just empty `SKNode` objects to start. You add them as children of `MyScene` and from now on, whenever you create a new node, you will add it as a child of either the game layer or the HUD layer, depending on where it belongs.

Next, you want to create the labels and add them to the HUD layer. However, to create the labels you need to choose a font size. Usually this is simple—you just pass the font size you want when you create the label.

Remember, though, that you want the Space Game to work just fine on both iPhone and iPad devices. On the iPhone, the screen size (in points) is either 480x320 (for 3.5" screens) or 568x320 (for 4" screens). On the iPad, the screen size (in points) is 1024x768. As you can see, the iPad is much bigger—roughly double so!

> **Note:** Sprite Kit uses points for positions and sizes rather than pixels. A point is defined as 1 pixel on non-retina display devices and 2 pixels on retina display devices.
>
> In other words, if you ask for the size of an iPad retina, you will get 1024x768 (points), instead of 2048x1536 pixels.
>
> This is a nice feature because it makes coding a bit easier. For example, you could place a space ship at (100,100) and it would be at the same position on both retina and non-retina displays.

Since the iPad screen size in points is roughly double the size of the iPhone's, you need to use a bigger font size on the iPad. To help with this, you'll write a little conversion routine. Add this method after `setupNodes`:

```
#pragma mark - Helper methods

- (CGFloat)fontSizeForDevice:(CGFloat)fontSize {
  if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    return fontSize * 2;
  } else {
    return fontSize;
  }
}
```

This helper method allows you to pass in a font size and, if the app is running on an iPad, the method doubles the size.

Now you have all the pieces you need to create the labels themselves. Add this method right after `setupLayers`:

```
- (void)setupTitle {
  NSString *fontName = @"Avenir-Light";

  // Title Label 1
  _titleLabel1 = [SKLabelNode labelNodeWithFontNamed:fontName];
  _titleLabel1.text = @"Space Game";
  _titleLabel1.fontSize = [self fontSizeForDevice:48.0];
  _titleLabel1.fontColor = [SKColor colorWithRed:0.7 green:0.7 blue:0.7 alpha:1.0];
  _titleLabel1.position = CGPointMake(self.size.width/2, self.size.height * 0.8);
  _titleLabel1.verticalAlignmentMode = SKLabelVerticalAlignmentModeCenter;
  [_hudLayer addChild:_titleLabel1];

  // Title Label 2
  _titleLabel2 = [SKLabelNode labelNodeWithFontNamed:fontName];
  _titleLabel2.text = @"Starter Kit";
  _titleLabel2.fontSize = [self fontSizeForDevice:96.0];
  _titleLabel2.fontColor = [SKColor colorWithRed:0.7 green:0.7 blue:0.7 alpha:1.0];
  _titleLabel2.position = CGPointMake(self.size.width/2, self.size.height * 0.6);
  _titleLabel2.verticalAlignmentMode = SKLabelVerticalAlignmentModeCenter;
  [_hudLayer addChild:_titleLabel2];
}
```

This method creates the labels and adds them to the HUD layer. Here are a few notes on the setup:

- The method uses one of the built-in fonts on the iPhone, Avenir-Light. At the time of writing this starter kit, Sprite Kit does not support bitmap fonts (for pre-created, fancy and efficient font effects) like Cocos2D does, so you either have to choose from the built-in fonts or use a third-party library like the one provided by

the creators of Glyph Designer (http://support.71squared.com/hc/en-us/articles/200037472).

- It colors the text to be gray. Note that this is one of the disadvantages of Sprite Kit's lack of built-in bitmap font support—whereas with a bitmap font system you could apply gradients or other neat effects, here the text must be a uniform color.
- It sets the size for each font using the helper routine you wrote earlier. Note that the second line should be bigger than the first.
- It sets the vertical alignment mode of each font to center so that when you position the fonts you're positioning the center of the text. To me, this makes thinking about placement easier in this case.
- It also positions each label on the screen. Note that it positions the labels using a multiple of the screen's width rather than hard-coded offsets. This puts the labels in the right spots regardless of how large the screen turns out to be.

There's just one last step—call your new setup functions. Add these lines to `initWithSize:`, right after setting the background color:

```
[self setupLayers];
[self setupTitle];
```

That's it! Build and run your project, and you'll see the title of the Space Game Starter Kit appear on the screen!

Making the Title Sexy

Your title has a cool feel to it, but in the Space Game Starter Kit, you're committed to making things gratuitously awesome.

You are going to modify the titles so that they zoom into view instead of just appearing on the screen. To do this, add the following lines to the bottom of `setupTitle`:

```
[_titleLabel1 setScale:0];
SKAction *waitAction1 = [SKAction waitForDuration:1.0];
SKAction *scaleAction1 = [SKAction scaleTo:1 duration:0.5];
[_titleLabel1 runAction:[SKAction sequence:@[waitAction1, scaleAction1]]];

[_titleLabel2 setScale:0];
SKAction *waitAction2 = [SKAction waitForDuration:2.0];
SKAction *scaleAction2 = [SKAction scaleTo:1 duration:1.0];
[_titleLabel2 runAction:[SKAction sequence:@[waitAction2, scaleAction2]]];
```

Now you start out each label with a scale of 0 and use Sprite Kit actions to make them zoom onto the screen.

If you're new to Sprite Kit actions, they're easy to use. You create an action based on what you want to do—for example, jump, rotate or scale—and pass in the appropriate parameters.

Then you simply run the action on the Sprite Kit node you want to perform the action, using the `runAction` method.

With two labels on the screen, you want a little delay so that one follows the other. The labels each wait a little with `waitForDuration:` and then zoom into view for a cool effect. You run both actions, one after the other, using a special action `sequence:`.

`sequence:` is easy to use—you simply give it an array with a list of the actions you want to run. It will then run them one after another, waiting for each to complete before moving to the next one.

So the labels run a `sequence:` where the first action is to wait for a bit (via `waitForDuration:`) and the second is to zoom into a given scale (via `scaleTo:`).

Build and run, and now you'll see the labels zoom in when the app starts up! The below screenshot captures the second line mid-zoom.

It's looking much cooler, but isn't quite at the awesome level. If you pay attention to the zoom, you might feel that it's too even in its rate of zooming, which doesn't feel natural.

Most things that move in nature take time to speed up, go at a nice clip for a while and then slow down before stopping. However, this crazy text just goes full-speed the entire time!

To get a more realistic feel, you're going to set up the text so that it moves quickly when the zoom begins and then slows down at the end. You can do this easily in Sprite Kit by setting the `timingMode` on the action to one of the following choices:

- `SKTimingModeLinear`: The default behavior, which is to perform the action at a constant speed.
- `SKTimingModeEaseIn`: Go slowly at the beginning and then speed up.
- `SKTimingModeEaseOut`: Go fast at the beginning and then slow down.
- `SKTimingModeEaseInEaseOut`: Go slowly at the beginning and end, but speed up in-between.

> **Note**: At the time of writing this starter kit, Sprite Kit only supports these four basic timing modes. However, there are a bunch of other timing modes you might want in your games, like elastic timing, bounce timing and shake timing. To learn how to add more advanced timing modes like these into your games, check out Chapter 18 of *iOS Games by Tutorials*, "Juice Up Your Games: Part 2."

Add these two lines to the bottom of `setupTitle` to set the timing mode for the move actions appropriately (changes highlighted):

```
[_titleLabel1 setScale:0];
SKAction *waitAction1 = [SKAction waitForDuration:1.0];
SKAction *scaleAction1 = [SKAction scaleTo:1 duration:0.5];
scaleAction1.timingMode = SKActionTimingEaseOut;
[_titleLabel1 runAction:[SKAction sequence:@[waitAction1, scaleAction1]]];

[_titleLabel2 setScale:0];
SKAction *waitAction2 = [SKAction waitForDuration:2.0];
SKAction *scaleAction2 = [SKAction scaleTo:1 duration:3.0];
scaleAction2.timingMode = SKActionTimingEaseOut;
[_titleLabel2 runAction:[SKAction sequence:@[waitAction2, scaleAction2]]];
```

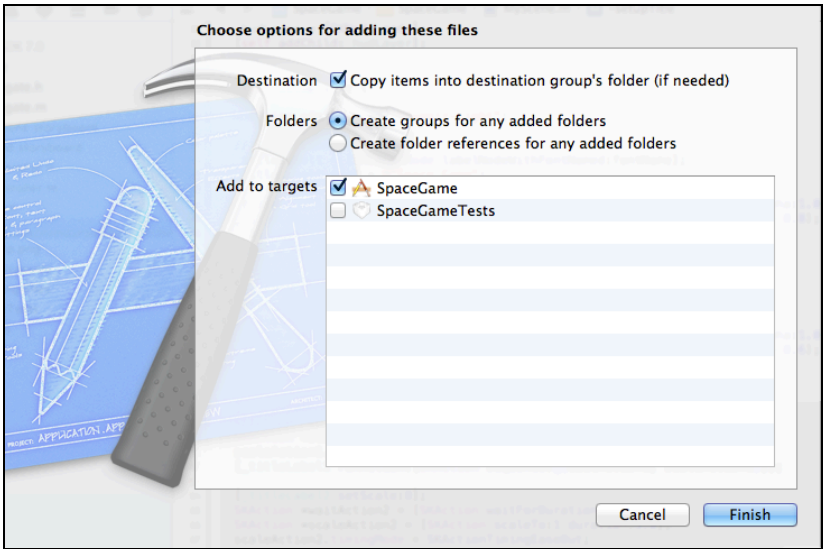
If you run the app again, you should see a subtle improvement in the zoom effect. Animations in UIKit typically use an easing curve to achieve a more realistic look, so this is a great practice to follow in Sprite Kit, too.

Adding the Art

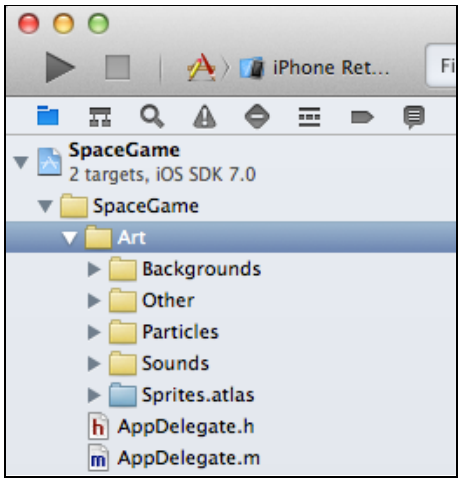
In the next section, you're going to start your game out with a bang by adding awesome music and sound effects, right from the beginning!

But first, you need to add the art and sounds for this game into your project. Find the resources that come along with this starter kit—you should have a folder named **Art**.

Once you find the **Art** folder, drag it into your Xcode project. A dialog will pop up—make sure that **Copy items into destination group's folder (if needed)** is checked, **Create groups for any added folders** is selected, make sure the **SpaceGame** target is checked and click **Finish**.



At this point, your Project Navigator should look like the following:



You'll notice there are five subfolders inside Art:

- **Backgrounds**: Contains the background images for the game—space dust, planets, distant galaxies and so on. Note that the background images are stored as separate images rather than in a texture atlas. That's because it's generally good practice to keep large images used only once in the background outside of a texture atlas.
- **Other**: Contains icons and default images you can use for the game.
- **Particles**: Contains particle systems for various special effects you will use in the game. These were created directly in Xcode with the built-in particle editor. We'll discuss these more later on.
- **Sounds**: Contains sound effects and music for the game. You will be using these shortly!
- **Sprites.atlas**: All of the non-background art for the game is inside this folder. Xcode automatically generates a texture atlas for these sprites since the folder ends with the **.atlas** extension. To learn more about texture atlases, check out Chapter 25 of *iOS Games by Tutorials*, "Performance: Texture Atlases".

As you look at the images inside these folders, you will notice that there are four versions of each image:

1. **imageName.png** (32x32px): The image that would be used on iPhone and iPod touch devices without retina displays. At the time of writing this starter kit, there are no iPhone or iPod touch devices without retina displays that run iOS 7—they all have retina displays! However, it's good practice to include this anyway, just in case and to be future-proof.
2. **imageName@2x.png** (64x64px): The image that will be used on iPhone and iPod touch devices with retina displays. This is saved as double the height and width of the normal iPhone image.

3. **imageName~iPad.png** (64x64px): The image that will be used on iPad devices without retina displays. For the Space Game Starter Kit, this image is exactly the same as the retina-sized iPhone image, since the dimensions of a retina iPhone and a non-retina iPad (in points) are fairly close.
4. **imageName@2x~iPad.png** (128x128px)**:** The image that will be used on iPad devices with retina displays. This is saved as double the height and width of the normal iPad image, or 4x that of the normal iPhone image.

When you load images in the Space Game Starter Kit, you will use the base name, like "imageName". Behind the scenes, since you are following the naming conventions above, Sprite Kit will choose the correct image for you automatically based on the device.

Adding SKTUtils

There's one more step before you can queue the music—adding some utiilty code into your game.

Inside the starter kit, you will find another directory called **SKTUtils**. Just as you did earlier for the Art folder, drag it into your Xcode project. A dialog will pop up—make sure that **Copy items into destination group's folder (if needed)** is checked, **Create groups for any added folders** is selected, make sure the **SpaceGame** target is checked and click **Finish**.

These are a bunch of helper methods and classes that come from our book *iOS Games by Tutorials*. The thing you'll use most frequently is a set of functions that make working with `CGPoints` easier—these are methods to add or subtract points, find lengths of vectors, and so on.

To make using the `SKTUtils` library easier, open **SpaceGame\Supporting Files\SpaceGame-prefix.pch** and add these lines to the bottom of the file:

```
#import "SKTUtils.h"
#import "SKTAudio.h"
#import "SKNode+DebugDraw.h"
#import "SKAction+SKTExtras.h"
#import "SKTEffects.h"
#import "SKEmitterNode+SKTExtras.h"
```

This precompiled header file is automatically included when a source file in the project is compiled. Adding these import lines means you'll have access to the helper methods and classes you'll need for your game from `SKTUtils`.

Feel free to peek around `SKTUtils` to see what's inside. I'll discuss these methods more as you use them throughout the starter kit.

Gratuitous Music and Sound Effects

Finally, it's time to add some groovy tunes!

You've already added the sound files to your project, so now you just have to add the code to play them.

> **Note:** If you're wondering how I made the sounds, I created the sound effects with a tool called CFXR, the voice sound effects with a microphone and Audacity, and the background music with Garage Band.

Start by opening **MyScene.m** and add these private instance variables:

```
SKAction *_soundExplosionLarge;
SKAction *_soundExplosionSmall;
SKAction *_soundLaserEnemy;
SKAction *_soundLaserShip;
SKAction *_soundShake;
SKAction *_soundPowerup;
SKAction *_soundBoss;
SKAction *_soundCannon;
SKAction *_soundTitle;
```

You're already familiar with `SKAction` from the game title animation, so the surprise here is that playing sounds are just actions, too!

You are going to preload each of the sounds by creating the actions to play each sound when you initialize the scene. That way, when you run the actions later, the sound effects will have been loaded and will play without a stutter.

Next add this method to set up the sounds, right after `setupTitle`:

```
- (void)setupSound {
  [[SKTAudio sharedInstance] playBackgroundMusic:@"SpaceGame.caf"];
  _soundExplosionLarge =
    [SKAction playSoundFileNamed:@"explosion_large.caf" waitForCompletion:NO];
  _soundExplosionSmall =
    [SKAction playSoundFileNamed:@"explosion_small.caf" waitForCompletion:NO];
  _soundLaserEnemy =
    [SKAction playSoundFileNamed:@"laser_enemy.caf" waitForCompletion:NO];
  _soundLaserShip =
    [SKAction playSoundFileNamed:@"laser_ship.caf" waitForCompletion:NO];
  _soundShake = [SKAction playSoundFileNamed:@"shake.caf" waitForCompletion:NO];
  _soundPowerup = [SKAction playSoundFileNamed:@"powerup.caf" waitForCompletion:NO];
  _soundBoss = [SKAction playSoundFileNamed:@"boss.caf" waitForCompletion:NO];
  _soundCannon = [SKAction playSoundFileNamed:@"cannon.caf" waitForCompletion:NO];
  _soundTitle = [SKAction playSoundFileNamed:@"title.caf" waitForCompletion:NO];
}
```

The first line uses a helper method from `SKTUtils` to play the background music. The rest of the lines create the sound playing actions in advance, as mentioned earlier.

Now that you've written this method, be sure to call it by adding the following line to `initWithSize:`, right *before* calling `setupLayers`:

```
[self setupSound];
```

And finally, you want to play a sound effect when the title appears. Add one more action to the sequence of actions on `_titleLabel1` in `setupTitle`:

```
[_titleLabel1 setScale:0];
SKAction *waitAction1 = [SKAction waitForDuration:1.0];
SKAction *scaleAction1 = [SKAction scaleTo:1 duration:0.5];
scaleAction1.timingMode = SKActionTimingEaseOut;
[_titleLabel1 runAction:
  [SKAction sequence:@[waitAction1, _soundTitle, scaleAction1]]];
```

Pretty easy, eh? Build and run your code, and you'll see your game is beginning to have some style!

Shooting Stars

However, right now your game has a big problem. How can you have a space game without stars?!

It would be awesome to have some stars shooting from the right-hand side of the screen to the left-side of the screen, to make the player feel like they're flying through space.

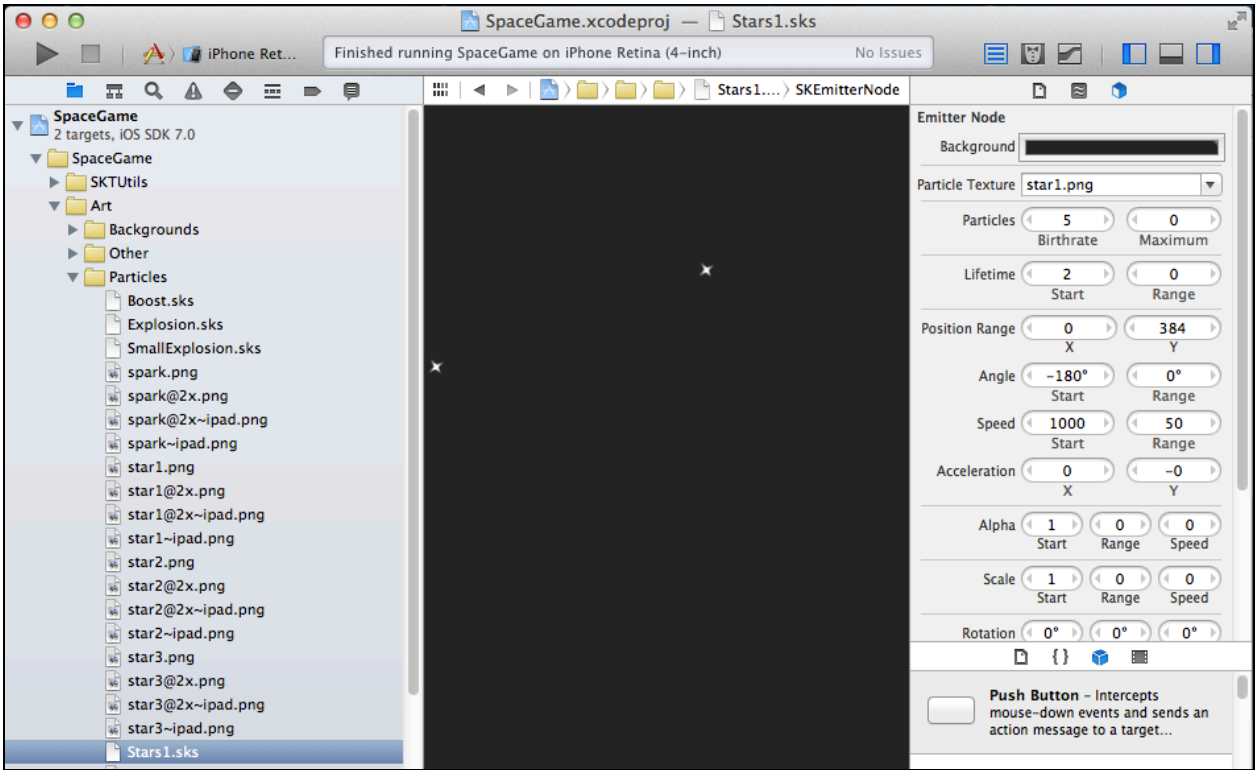
There are several ways you could implement this:

- You could have a background with a lot of stars drawn on it and move it from right to left. This would be easy, but has the disadvantage of requiring a large background, taking up texture memory.
- You could create a sprite for each star and move each of those from right to left. This would save a lot of texture memory compared with the solution above, since you'd only need one small texture per type of star. On the downside, your frame rate would suffer because having a lot of sprites on the screen at once is expensive.
- Or you could use particle systems! Particle systems are optimized to efficiently create large amounts of small colors or objects and move them across the screen. This is exactly what you need.

The easiest way to create a particle system is to use the built-in particle system editor that comes with Xcode. I've used this to create three particle systems for you

that shoot stars from right to left. You can find them in **Art\Particles\Stars1.sks**, **Stars2.sks**, and **Stars3.sks**.

Open **Stars1.sks**, make sure that the **Utilities** panel is open and that the **SKNode Inspector** is selected (the third tab). You will see the built-in particle system editor appear, allowing you to tweak any of the settings for the particle system:



Here are some of the important settings to note about the particle system:

- The particle texture is set to star1.png. Note that you have multiple versions of this image in you project for each device the app might run on, as discussed earlier.
- The birthrate is set to 5, with a maximum of 0 and a lifetime of 2. This means that the particle system creates five stars each second and that they last two seconds each before disappearing. A maximum of 0 means there is no upper limit to the number of stars the particle system can create.
- The angle is set to -180 degrees, which means the stars shoot to the left.
- The position range is set to vary 384 points along the y-axis. This means the stars spawn randomly along the height of the screen. Later, you will dynamically modify this property to be the exact height of the device running the game, since the height will be different on an iPad, for example.
- The speed is set to 1000 points per second.

Feel free to tweak any of these settings to get the effect you'd like to have. Once you're satisfied, let's add these to the game!

Start by adding the following new method in **MyScene.m**, right after `setupSound`:

```
- (void)setupStars {
  // 1
  NSArray *starsArray = @[@"Stars1.sks", @"Stars2.sks", @"Stars3.sks"];
  for (NSString *stars in starsArray) {
    // 2
    SKEmitterNode *emitter =
      [NSKeyedUnarchiver unarchiveObjectWithFile:
        [[NSBundle mainBundle] pathForResource:stars ofType:nil]];
    // 3
    emitter.position = CGPointMake(self.size.width*1.5, self.size.height/2);
    // 4
    emitter.particlePositionRange =
       CGVectorMake(emitter.particlePositionRange.dx, self.size.height * 1.5);
    // 5
    emitter.zPosition = -1;
    [_gameLayer addChild:emitter];
  }
}
```

You overlay the three particle systems to make a fuller-looking star field. Here's how the setup works:

1. First you create an array with the file names of the three particle systems you want to use.
2. For each array element, you create an `SKEmitterNode` with that file.
3. You set the position of the particle system to be offscreen to the right (1.5x the width) and in the middle of the screen.
4. You set the y-variance for the particle spawn range to be half the height of the screen, times 1.5. This makes it so that the star field is visible even if you zoom the layer out a bit, which you'll be doing later on in this tutorial to create a neat effect. If the particle system simply matched the exact screen dimensions, it would look really weird, like stars were only in one particular rectangle in space!
5. Finally, you set the z-positioning so the stars are in the background and you add the particle system to the game layer.

There's one last step. Simply add the line to call the above method to the bottom of `initWithSize:`, right after the call to `setupTitle`:

```
[self setupStars];
```

Build and run. Now it's starting to look like a space game!

Adding a Play Label

Your game has an awesome title, sound effects and music, but right now there's absolutely nothing to do.

In this section, you'll add a simple label to the game that says "Tap to Play". When the user taps the screen, the game can begin.

Start by opening **MyScene.m** and adding a private instance variable to keep track of the label. Add it right below the declarations for the title labels:

```
SKLabelNode *_playLabel;
```

Then add the following code to the bottom of `setupTitle`:

```
// Play Label
_playLabel = [SKLabelNode labelNodeWithFontNamed:fontName];
[_playLabel setScale:0];
_playLabel.text = @"Tap to Play";
_playLabel.fontSize = [self fontSizeForDevice:32.0];
_playLabel.fontColor = [SKColor colorWithRed:0.7 green:0.7 blue:0.7 alpha:1.0];
_playLabel.position = CGPointMake(self.size.width/2, self.size.height * 0.25);
_playLabel.verticalAlignmentMode = SKLabelVerticalAlignmentModeCenter;
[_hudLayer addChild:_playLabel];

SKAction *waitAction3 = [SKAction waitForDuration:3.0];
SKAction *scaleAction3 = [SKAction scaleTo:1 duration:0.5];
```

```
scaleAction3.timingMode = SKActionTimingEaseOut;
SKAction *scaleUpAction = [SKAction scaleTo:1.1 duration:0.5];
scaleUpAction.timingMode = SKActionTimingEaseInEaseOut;
SKAction *scaleDownAction = [SKAction scaleTo:0.9 duration:0.5];
scaleDownAction.timingMode = SKActionTimingEaseInEaseOut;
SKAction *throbAction = [SKAction repeatActionForever:[SKAction
sequence:@[scaleUpAction, scaleDownAction]]];
SKAction *displayAndThrob = [SKAction sequence:@[waitAction3, scaleAction3,
throbAction]];
[_playLabel runAction:displayAndThrob];
```

This creates an `SKLabelNode` that reads "Tap to Play" and adds it to the HUD layer below the other labels. It also construct a sequence of actions that makes the label zoom into view and then "throb " to draw the player's attention and create a neat effect.

Build and run, and you will see the play label appear on the screen.

Starting the Game

You need to add some code to detect when the user taps the Play button and start the game accordingly, but before you can do that, you need to add some code to keep track of the game's state.

This is because the game now has two states: before the game begins (the "main menu") and the gameplay itself. You only want to begin the game during the "main menu" state.

You are going to put the code to keep track of the game's state in a separate class, so that you have a firm foundation to build upon later as you continue to develop your game's logic. Yes, you will eventually have more than two game states!

To do this, create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **LevelManager** for Class, **NSObject** for Subclass of, click **Next** and then click **Create**.

Open **LevelManager.h** and replace the contents with the following:

```
typedef NS_ENUM(NSInteger, GameState)
{
  GameStateMainMenu = 0,
  GameStatePlay,
  GameStateDone,
  GameStateGameOver
};

@interface LevelManager : NSObject

@property (assign) GameState gameState;

@end
```

Here you create an enumeration for the four states you will have in your space game:

- `GameStateMainMenu`: The initial state, when the labels are on the screen.
- `GameStatePlay`: The gameplay state.
- `GameStateDone`: The game is finished and it's time to transition to the Game Over screen.
- `GameStateGameOver`: The Game Over screen has appeared and the game is waiting for the user to restart.

You also create a property to store the game's current state.

Next open **LevelManager.m** and replace the contents with the following:

```
#import "LevelManager.h"

@implementation LevelManager

- (id)init {
  if ((self = [super init])) {
    _gameState = GameStateMainMenu;

  }
  return self;
}
```

```
@end
```

This creates an initializer that simply initializes the game state to the main menu state.

Now let's set up the game to use this class. Open **MyScene.m** and import the new class:

```
#import "LevelManager.h"
```

Also add a private instance variable for the level manager:

```
LevelManager *_levelManager;
```

Add a new function to initialize the level manager, right after `setupStars`:

```
- (void)setupLevelManager {
  _levelManager = [[LevelManager alloc] init];
}
```

And call this method from `initWithSize:`, right after the call to `setupStars`:

```
[self setupLevelManager];
```

Add this method to the bottom of **MyScene.m**:

```
#pragma mark - Transitions

- (void)startSpawn {
  _levelManager.gameState = GameStatePlay;
  [self runAction:_soundPowerup];

  NSArray *nodes = @[_titleLabel1, _titleLabel2, _playLabel];
  for (SKNode *node in nodes) {
    SKAction *scaleAction = [SKAction scaleTo:0 duration:0.5];
    scaleAction.timingMode = SKActionTimingEaseOut;
    SKAction *removeAction = [SKAction removeFromParent];
    [node runAction:[SKAction sequence:@[scaleAction, removeAction]]];
  }
}
```

You will call this method when the user taps the screen while the game is in the main menu state. Eventually this method will initialize some of the game logic, too, but for now you just play the power-up sound effect and zoom out the labels.

To avoid duplicating code, the method puts each label and the menu item into an array, then loops through each item and runs an action to zoom it out to a scale of 0.

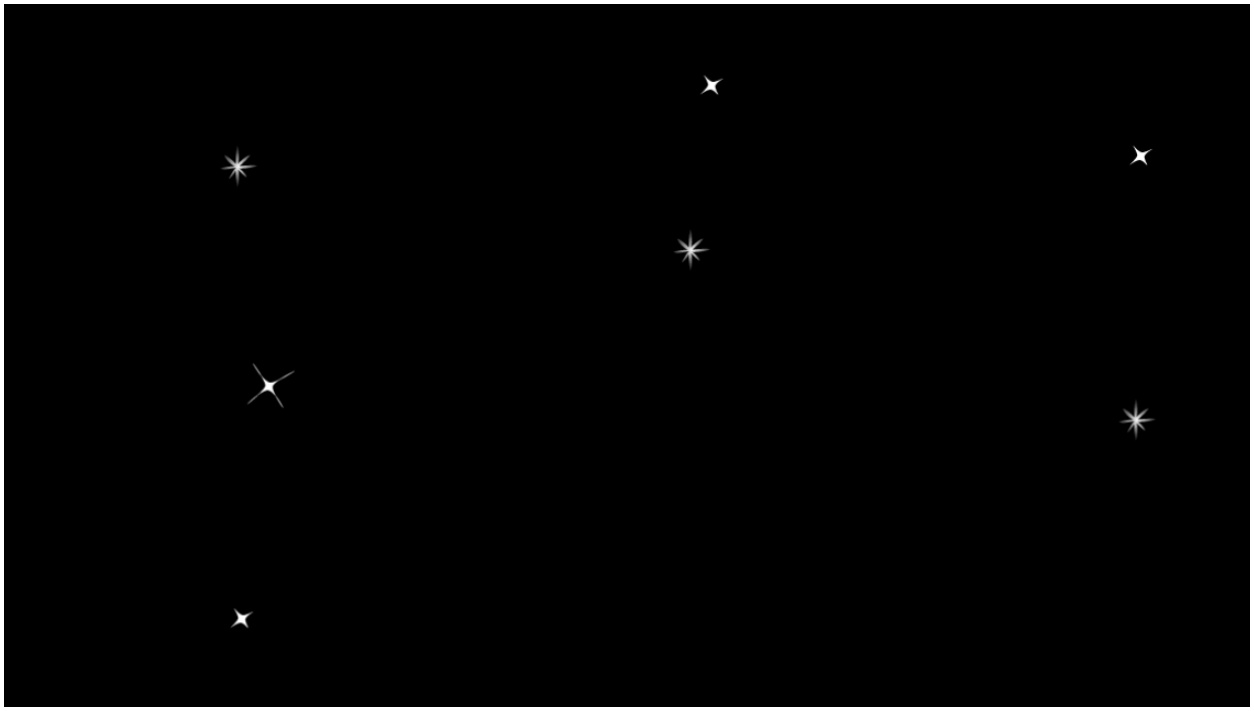
After each item is zoomed out, you need to remove it from the layer. Otherwise, you'd have these invisible Sprite Kit nodes sitting around consuming resources. To accomplish this, the last thing you put in the `sequence:` is a `removeFromParent` action to remove the items when the game switches state.

The last step is to add this new method to the bottom of **MyScene.m** to detect the tap, and call the `startSpawn` method if the game is in the main menu state:

```
#pragma mark - Touch detection

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {
  if (_levelManager.gameState == GameStateMainMenu) {
    [self startSpawn];
    return;
  }
}
```

`touchesBegan:withEvent:` is a standard `UIResponder` method to handle touch detection. Any touch onscreen will be enough to trigger this method and start the game.

Build and run, and now when you tap the screen everything should zoom out. You're ready to start work on the main action!



Adding Your Space Ship

The first thing you need in a space game, after stars, is a space ship to pilot around! In this section, you'll add your hero into the scene.

To keep your code clean, you are going to create a separate class for each type of object you add to the game. All of these classes will derive from a base class called `Entity` that will contain some common code, so let's start by creating it.

Create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **Entity** for Class, **SKSpriteNode** for Subclass of, click **Next** and then click **Create**.

Open **Entity.h** and replace the contents with the following:

```
@interface Entity : SKSpriteNode

@property (nonatomic, assign) NSInteger hp;
@property (nonatomic, assign) NSInteger maxHp;

- (instancetype)initWithImageNamed:(NSString *)name maxHp:(NSInteger)maxHp;

@end
```

All entities in the space game will have hit points, so here you create properties for the current and max HP.

Open **Entity.m** and replace the contents with the following:

```
#import "Entity.h"

@implementation Entity

- (instancetype)initWithImageNamed:(NSString *)name maxHp:(NSInteger)maxHp {
  if ((self = [super initWithImageNamed:name])) {
    _maxHp = maxHp;
    _hp = maxHp;
  }
  return self;
}

@end
```

Here you create a basic initializer that stores the hit points. Note that you call the superclass's initializer (`SKSpriteNode`) with the image to use for the sprite.

Now let's subclass this for the player ship. To do this, create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **Player** for Class, **Entity** for Subclass of, click **Next** and then click **Create**.

You don't need to make any changes to the header file but you do need to implement an initializer in the implementation. Open **Player.m** and replace the contents with the following:

```
#import "Player.h"

@implementation Player

- (instancetype)init {
  if ((self = [super initWithImageNamed:@"SpaceFlier_sm_1" maxHp:10])) {
  }
  return self;
}

@end
```

This passes "Spaceflier_sm_1" as the image name to use for the ship. You can find this image in the **Art\Sprites.atlas** folder that you added to your project earlier. Remember, since the folder ends with .atlas, Sprite Kit will automatically generate a texture atlas containing the sprites for you, which helps keep your game running efficiently.

This is one great thing about Sprite Kit. You don't have to do anything differently to use texture atlases than if you were using individual sprites—you simply create sprites as you normally would with `initWithImageNamed:`. If the sprites are in a texture atlas, Sprite Kit will detect that and load them from the automatically generated texture atlas automatically. Sprite Kit will also automatically choose the correct image variant based on the device, as discussed earlier.

Now let's try out your new class. Start by opening **MyScene.h** and replacing the contents with the following:

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

@class Player;

@interface MyScene : SKScene

@property (strong) Player * player;

@end
```

Here you create a property for the player class. Note that you are using a property rather than a private instance variable. That's because later in this starter kit, you will need to access the player sprite from another class.

You also forward declare the `Player` class with the `@class` keyword, rather than import the header file directly, for more efficient compilation.

Next, switch to **MyScene.m** and import Player.h at the top of the file:

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

Then add a method to initialize the player right after `setupLevelManager`:

```
- (void)setupPlayer {
  _player = [[Player alloc] init];
  _player.position = CGPointMake(-_player.size.width/2, self.size.height * 0.5);
  _player.zPosition = 1;
  _player.name = @"player";
  [_gameLayer addChild:_player];
}
```

This sets the player's position to be offscreen to the left along the x-axis. When you begin the game, you will make the player fly in. It also sets the z-position of the player to 1 so it appears in front of everything else.

> **Note:** If you're confused how `-player.size.width/2` means "offscreen to the left along the x-axis", here's why.
>
> When you set the position of a sprite, by default you're setting the position of the center of the sprite. If you set the position of the sprite to 0 along the x-axis, half of the sprite would be onscreen and half would be offscreen.
>
> So to get the sprite fully offscreen, you move it to the left by half the width of the sprite. You get the width of the ship by `player.size`. Voilà!

Call this method in `initWithSize:`, right after calling `setupLevelManager`:

```
[self setupPlayer];
```

Next add this new method to spawn the player, right after `startSpawn`:

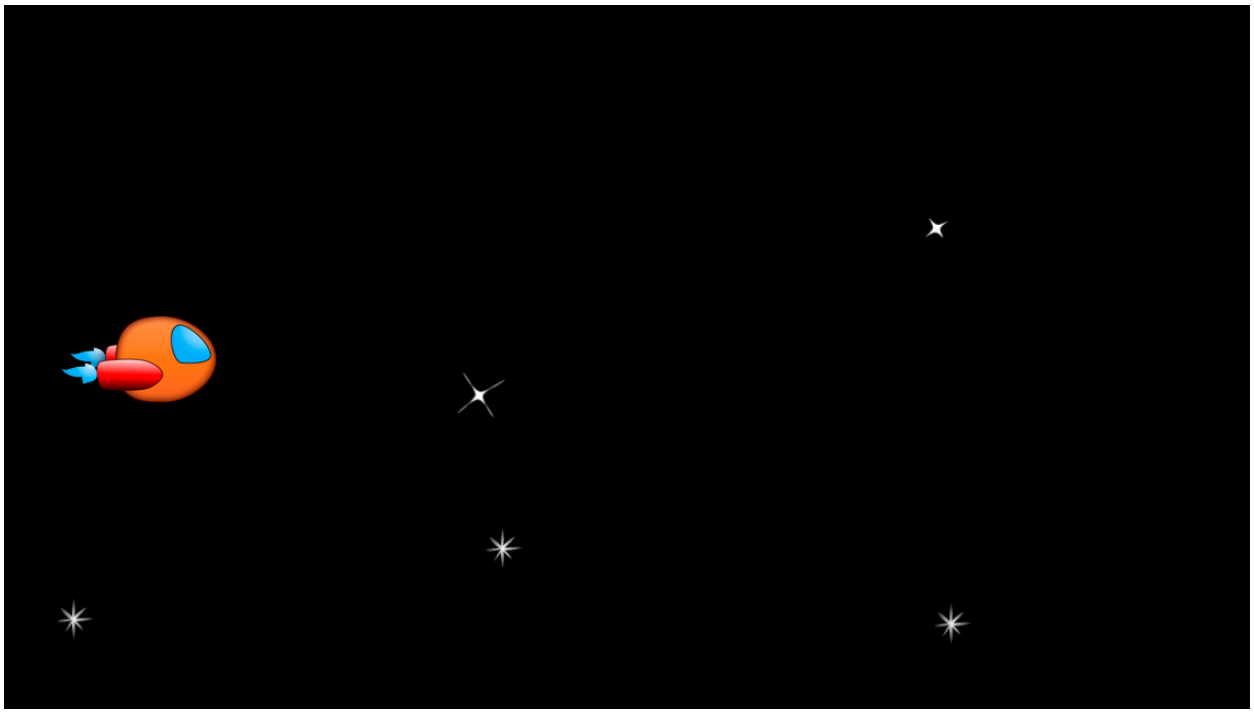
```
- (void)spawnPlayer {
  SKAction *moveAction1 = [SKAction moveBy:
    CGVectorMake(_player.size.width/2 + self.size.width * 0.3, 0) duration:0.5];
  moveAction1.timingMode = SKActionTimingEaseOut;
  SKAction *moveAction2 = [SKAction moveBy:
    CGVectorMake(-self.size.width * 0.2, 0) duration:0.5];
  moveAction2.timingMode = SKActionTimingEaseInEaseOut;
  [_player runAction:[SKAction sequence:@[moveAction1, moveAction2]]];
}
```

This makes the player ship fly in from the left and then backtrack a bit for a cool effect.

Finally, call this new method at the bottom of `startSpawn`:

```
[self spawnPlayer];
```

Build and run, and—awesome, you've got a space ship!



Animating the Ship

If you look at the art for the game under **Art\Sprites.atlas**, you'll see that there are two animation frames for the ship while it's moving—**SpaceFlier_sm_1.png** and **SpaceFlier_sm_2.png**.

So rather than displaying just the first image as you're doing right now, why not animate that ship?

Animating a sprite is ridiculously easy in Sprite Kit. There are just two steps:

- Create an array of `SKTextures`, specifying the images that make up the animation.
- Create an `animateWithTextures:` action and run it on the sprite, specifying the array of textures you created earlier.

Note that the `animateWithTextures:` action runs the animation only once. A common technique you'll use is to wrap the animation in a `repeatActionForever:` action so that it keeps going until you tell it to stop.

Let's try this out! Add the following code to **Player.m**, right after `init`:

```
- (void)setupAnimation {
  NSArray * textures = @[
      [SKTexture textureWithImageNamed:@"SpaceFlier_sm_1"],
```

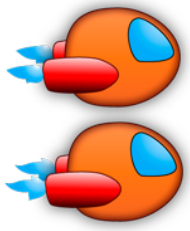
```
      [SKTexture textureWithImageNamed:@"SpaceFlier_sm_2"],
    ];
  SKAction *animation = [SKAction animateWithTextures:textures timePerFrame:0.2];
  [self runAction:[SKAction repeatActionForever:animation]];
}
```

This creates an array of textures using both images, according to step 1, above. Then it runs a new `animateWithTextures:` action on the ship, wrapped in a `repeatActionForever:` action, according to step 2, above.

Now call this method from the `if` block inside `init`:

```
[self setupAnimation];
```

That's it—Build and run, and your ship now has full thrusters!

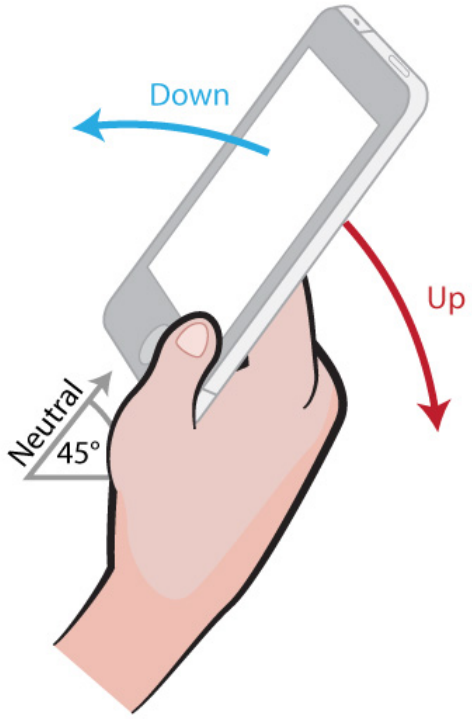


Moving with the Accelerometer

Your game sure looks cool, but it isn't very interesting yet because the ship just sits in the same spot the entire time!

In this game, you're going to keep the ship in the same X position, moving the background from right to left to simulate the ship moving forward. However, the player will be able to modify the ship's Y position by tilting the device to make the ship move up or down.

Let's make the ship move along the y-axis based on how the player tilts the device. The diagram below shows the "neutral" position, where the ship remains still:



If you tilt the device away from you, so that it's more flat, the ship should move up, and if you tilt the device towards you, so that it's move vertical, the ship should move down.

To receive data from the accelerometer, you need to use the Core Motion framework. Add this line to **MyScene.m** to import the framework:

```
@import CoreMotion;
```

> **Note**: The `@import` keyword is new to iOS 7. It's a more efficient way to both import the header file and link in the framework library. To learn more, check out this tutorial:
>
> http://www.raywenderlich.com/49850/whats-new-in-objective-c-and-foundation-in-ios-7

Within Core Motion, you'll use the `CMMotionManager` class to access the accelerometer data. Create a private instance variable for this next:

```
CMMotionManager *_motionManager;
```

Then add this method to configure the motion manager to receive accelerometer updates, right after `setupPlayer`:

```
- (void)setupMotionManager {
  _motionManager = [[CMMotionManager alloc] init];
  _motionManager.accelerometerUpdateInterval = 0.05;
  [_motionManager startAccelerometerUpdates];
}
```

This configures the motion manager to receive updates every 1/60th of a second and tells it to start listening for accelerometer updates.

Call this method in `initWithSize:`, right after the call to `setupPlayer`:

```
[self setupMotionManager];
```

Then add this method to the bottom of the file:

```
#pragma mark - Update methods

- (void)updatePlayer {
  CGFloat kFilteringFactor = 0.75;
  static UIAccelerationValue rollingX = 0, rollingY = 0, rollingZ = 0;

  rollingX = (_motionManager.accelerometerData.acceleration.x * kFilteringFactor) +
    (rollingX * (1.0 - kFilteringFactor));
  rollingY = (_motionManager.accelerometerData.acceleration.y * kFilteringFactor) +
    (rollingY * (1.0 - kFilteringFactor));
  rollingZ = (_motionManager.accelerometerData.acceleration.z * kFilteringFactor) +
    (rollingZ * (1.0 - kFilteringFactor));

  CGFloat accelX = rollingX;
  CGFloat accelY = rollingY;
  CGFloat accelZ = rollingZ;

  NSLog(@"accelX: %f, accelY: %f, accelZ: %f", accelX, accelY, accelZ);
}
```

This code comes directly from Apple sample code and filters the accelerometer data so that it's not so "jiggly". Don't worry if you don't understand this; all you need to know is that it smooths out the data so that the player ship's movement looks more natural. If you're insatiably curious, this is called a high-pass filter, and you can read about it on Wikipedia's High Pass Filter entry.

For now, you just print out the filtered values for x, y and z so that you can see what you're getting.

All you need to do now is call this method every frame. You can do this by overriding the scene's `update:` method and calling `updatePlayer` instead, but while you're at it you should add some code to keep track of the time elapsed since the

last frame—you'll need this frequently in the game and Sprite Kit doesn't keep track of it for you.

Add these two new private instance variables:

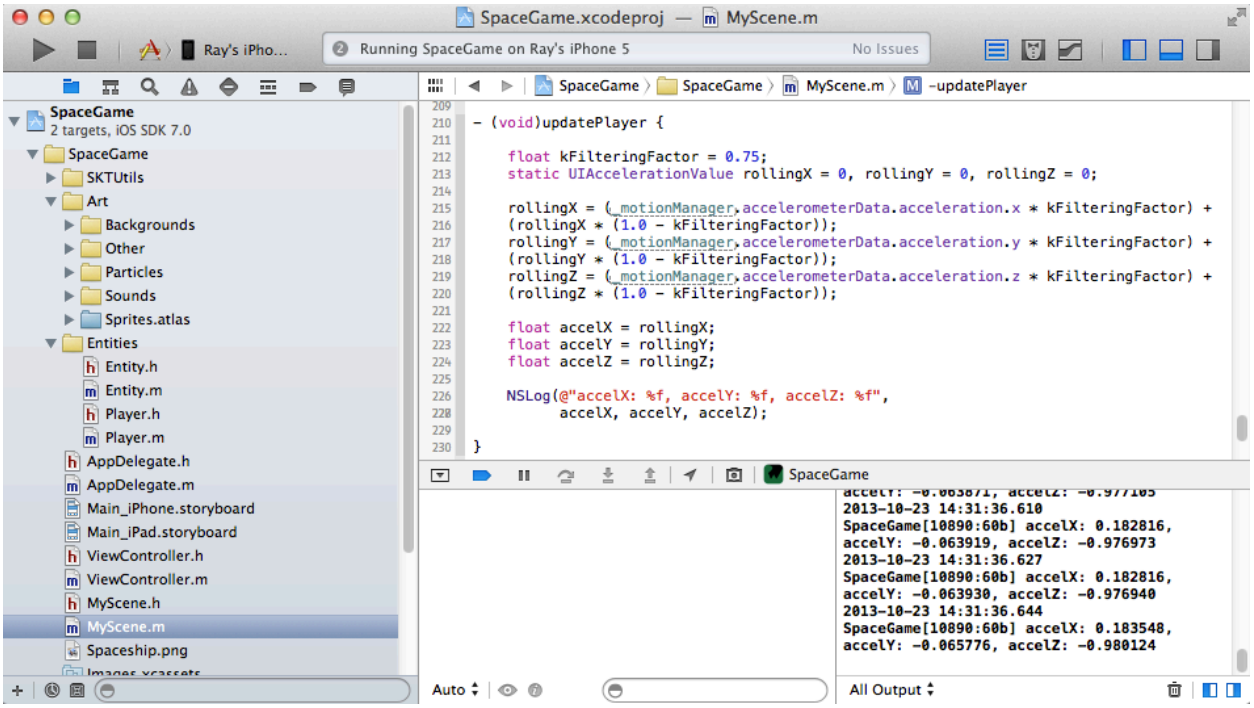
```
NSTimeInterval _lastUpdateTime;
NSTimeInterval _deltaTime;
```

Then add this method right after `updatePlayer`:

```
- (void)update:(CFTimeInterval)currentTime {

  if (_lastUpdateTime) {
    _deltaTime = currentTime - _lastUpdateTime;
  } else {
    _deltaTime = 0;
  }
  _lastUpdateTime = currentTime;

  if (_levelManager.gameState != GameStatePlay) return;

  [self updatePlayer];
}
```

This keeps track of the time elapsed since the last frame and stores the value in `_deltaTime`. It then calls `updatePlayer` every frame, as long as the game state is `GameStatePlay`.

Build and run this code—on your device, not the Simulator, because the Simulator does not have accelerometer support. As you run the code, look at your console output and you'll see the accelerometer values displaying in the console log:



Notice that as you tilt your iPhone up and down, `accelX` goes up and down. You're going to base the ship's movement on this value.

You're going to use +/- 0.6 as the starting value for `accelX`. The further away from +/- 0.6 the accelerometer input goes, the faster the ship will move up or down.

Your strategy is simply to update a variable for the intended speed of the ship's movement as you get accelerometer input, and then use this variable to move the ship the appropriate amount every frame.

This is better than trying to immediately move the ship based on accelerometer input, because accelerometer input doesn't necessarily come in at the rate of exactly once per frame.

Enough talk—let's see what this looks like in code! Add this code at the bottom of `updatePlayer`:

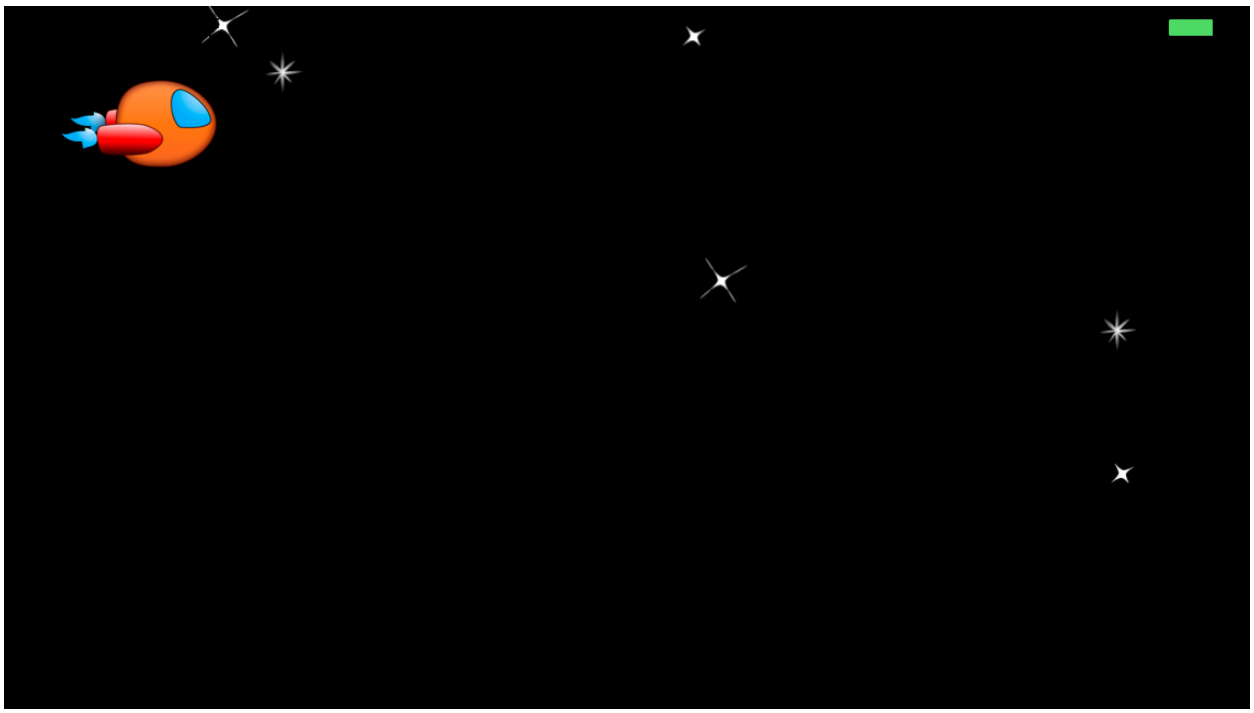
```
CGFloat kRestAccelX = 0.6;
CGFloat kPlayerMaxPointsPerSec = self.size.height*0.5;
CGFloat kMaxDiffX = 0.2;

CGFloat accelDiffX = kRestAccelX - ABS(accelX);
CGFloat accelFractionX = accelDiffX / kMaxDiffX;
CGFloat pointsPerSecX = kPlayerMaxPointsPerSec * accelFractionX;

CGFloat playerPointsPerSecY = pointsPerSecX;
CGFloat maxY = self.size.height - _player.size.height/2;
CGFloat minY = _player.size.height/2;
```

```
CGFloat newY = _player.position.y + (playerPointsPerSecY * _deltaTime);
newY = MIN(MAX(newY, minY), maxY);
_player.position = CGPointMake(_player.position.x, newY);
```

This first figures out the difference between `accelX` and 0.6 (`accelDiffX`). It then divides the difference by 0.2 to compute an acceleration fraction (`accelFractionX`). This fraction will be be near 0 when `accelDiffX` is close to 0.6, and near 1.0 when `accelDiffX` is close to 0.8.

Finally, you set the points per second the ship should move to be equal to the maximum speed to move (half the height of the screen per second) times the acceleration fraction, and then move the ship that amount. There is also some code to prevent the ship from moving past the top or bottom edges of the screen. Voilà!

You're done! Build and run the code, and now you should be able to move your spaceship by tilting your device.



Now that you're done implementing movement with the accelerometer, you might want to comment out that `NSLog` statement displaying the accelerometer values; it's just slowing down your game at this point.

Creating an Asteroid Belt

You have tunes and you have a moving spaceship, but you are missing something major—things to smash!

It's time to add the first enemy type to your game: asteroids.

You're going to use a very simple strategy:

- Every so often, you'll spawn an asteroid.
- You'll start the asteroid offscreen to the right at a random y-value between the top and bottom of the screen.
- You'll move the asteroid from the right to the left at a random speed.
- For kicks, you'll also make the asteroids different sizes!

To start, you need a class for the asteroids. Asteroids will inherit from the same superclass as the player ship, `Entity`. Create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **Asteroid** for Class, **Entity** for Subclass of, click **Next** and then click **Create**.

Open **Asteroid.h** and replace the contents with the following:

```
#import "Entity.h"

typedef NS_ENUM(NSInteger, AsteroidType) {
  AsteroidTypeSmall = 0,
  AsteroidTypeMedium,
  AsteroidTypeLarge,
  NumAsteroidTypes
};

@interface Asteroid : Entity

@property (assign) AsteroidType asteroidType;

- (instancetype)initWithAsteroidType:(AsteroidType)asteroidType;

@end
```

Here you create an enumeration defining the three different kinds of asteroids in the game—small, medium and large asteroids.

Switch to **Asteroid.m** and replace the contents with the following:

```
#import "Asteroid.h"

@implementation Asteroid

- (instancetype)initWithAsteroidType:(AsteroidType)asteroidType {
  int maxHp;
  float scale;
  switch (asteroidType) {
  case AsteroidTypeSmall:
    maxHp = 1;
    scale = 0.25;
```

```
    break;
  case AsteroidTypeMedium:
    maxHp = 2;
    scale = 0.5;
    break;
  case AsteroidTypeLarge:
    maxHp = 4;
    scale = 1.0;
    break;
  default:
    return nil;
  }
  if ((self = [super initWithImageNamed:@"asteroid" maxHp:maxHp])) {
    self.asteroidType = asteroidType;
    [self setScale:scale];
  }
  return self;

}

@end
```

In the initializer, you check to see what kind of asteroid you are creating and set the hit points and scale appropriately.

Now that you have created an asteroid class, let's put it to use. Switch to **MyScene.m** and import the new class:

```
#import "Asteroid.h"
```

Then add this new method to the bottom of the file:

```
#pragma mark - Asteroids

- (void)spawnAsteroid
{
  CGFloat const moveDurationLow = 2.0;
  CGFloat const moveDurationHigh = 10.0;

  Asteroid *asteroid = [[Asteroid alloc]
    initWithAsteroidType:arc4random_uniform(NumAsteroidTypes)];
  asteroid.name = @"asteroid";
  asteroid.position = CGPointMake(
    self.size.width + asteroid.size.width/2,
    RandomFloatRange(0, self.size.height));
  [_gameLayer addChild:asteroid];

  [asteroid runAction:
    [SKAction sequence:@[
```

```
      [SKAction moveBy:CGVectorMake(-self.size.width*1.5, 0)
        duration:RandomFloatRange(moveDurationLow, moveDurationHigh)],
      [SKAction removeFromParent]
    ]
  ]];
}
```

You will call this method whenever you want to spawn an asteroid. It creates an asteroid offscreen to the right at a random y-location. It then makes the asteroid move across the screen to the left at a random speed between `moveDurationLow` and `moveDurationHigh`, and removes it from the scene when it is done moving (which will happen when the asteroid is offscreen to the left).

Add these two new instance variables:

```
NSTimeInterval _timeSinceLastAsteroidSpawn;
NSTimeInterval _timeForNextAsteroidSpawn;
```

You will use these to keep track of how long it's been since the spawning of the last asteroid, and when the next asteroid should spawn.

Next add this new method to spawn asteroids periodically:

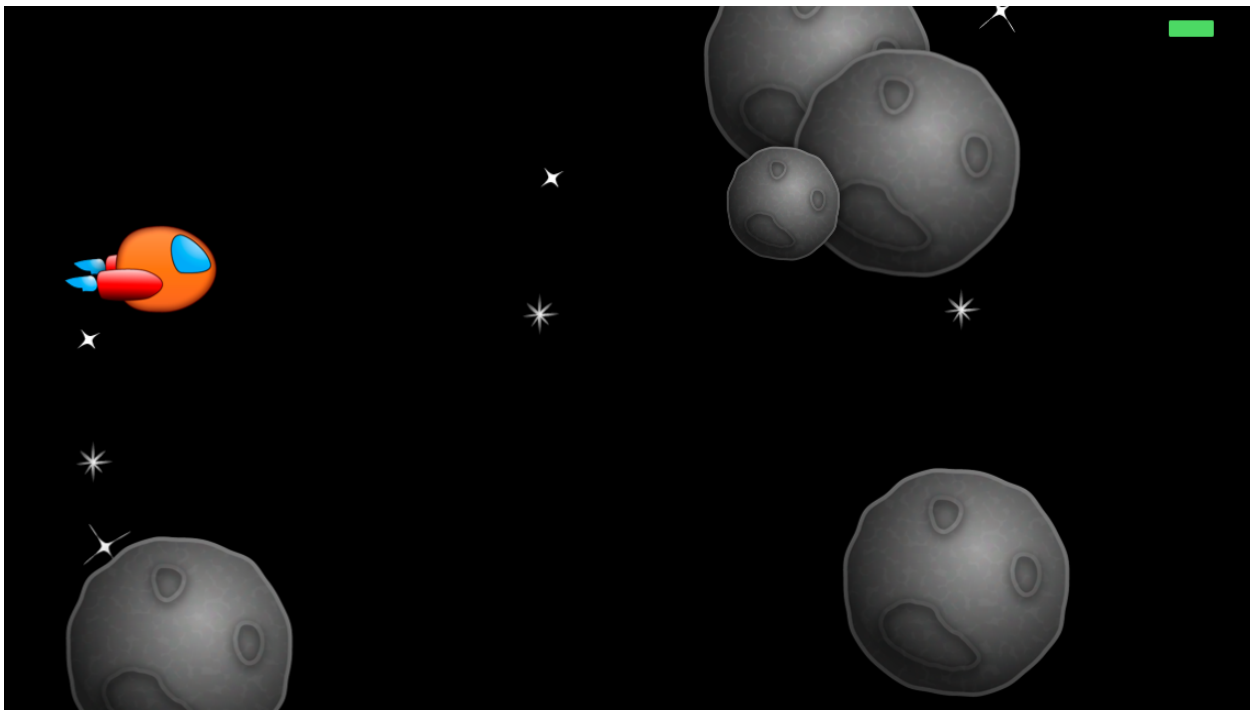
```
- (void)updateAsteroids {
  NSTimeInterval const spawnSecsLow = 0.2;
  NSTimeInterval const spawnSecsHigh = 1.0;

  _timeSinceLastAsteroidSpawn += _deltaTime;
  if (_timeSinceLastAsteroidSpawn > _timeForNextAsteroidSpawn) {
    _timeSinceLastAsteroidSpawn = 0;
    _timeForNextAsteroidSpawn = RandomFloatRange(spawnSecsLow, spawnSecsHigh);
    [self spawnAsteroid];
  }
}
```

This checks to see if it's time to spawn an asteroid and if so, calls the method you wrote previously. Currently this is set up to spawn an asteroid every 0.2 to 1.0 seconds.

Finally, call this method at the bottom of `update:`:

```
[self updateAsteroids];
```

Build and run your code, and practice navigating your ship through a randomized asteroid belt!



Note: At this point in previous versions of this starter kit, I introduced a technique called sprite caching where you pre-allocate and re-use sprites rather than create them on demand as you're doing here.

However, Sprite Kit contains many optimizations under the hood that operate automatically, including object pooling, so this technique does not give the same performance benefits as it did with Cocos2D. Therefore, I recommend you add sprite caching only if you've done performance testing and found that sprite allocation is a bottleneck for your game. In cases where you don't need it, like in this game, omitting it keeps your code simpler and easier to understand.

If you'd like to learn more about performance and object pooling in Sprite Kit, check out chapters 25 and 26 of *iOS Games by Tutorials*, "Performance and Testing" and "Performance: Tips and Tricks."

Lasers Go Pew, Pew!

Asteroids, check! Lasers… not so much. So let's get your pew-pew on and add some killer lasers to blast those asteroids to bits.

As before, you are going to create a new class to represent the lasers. Create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **PlayerLaser** for Class, **Entity** for Subclass of, click **Next** and then click **Create**.

You don't need to make any changes to the header file, but switch to **PlayerLaser.m** and replace the contents with the following:

```
#import "PlayerLaser.h"

@implementation PlayerLaser

- (instancetype)init {
  if ((self = [super initWithImageNamed:@"laserbeam_blue" maxHp:1])) {
  }
  return self;
}

@end
```

Switch to **MyScene.m** and import the new class:

```
#import "PlayerLaser.h"
```

Then add this method to the bottom of the file to spawn a laser:

```
#pragma mark - Lasers and Cannons

- (void)spawnPlayerLaser
{
  PlayerLaser *laser = [[PlayerLaser alloc] init];
  laser.position = CGPointMake(_player.position.x + 6, _player.position.y - 4);
  laser.name = @"laser";
  [_gameLayer addChild:laser];

  laser.alpha = 0;
  [laser runAction:[SKAction fadeAlphaTo:1.0 duration:0.1]];
  SKAction *actionMove =
    [SKAction moveToX:self.size.width + laser.size.width/2 duration:0.75];
  SKAction *actionRemove = [SKAction removeFromParent];
  [laser runAction:[SKAction sequence:@[actionMove, actionRemove]]];

  [self runAction:_soundLaserShip];
}
```

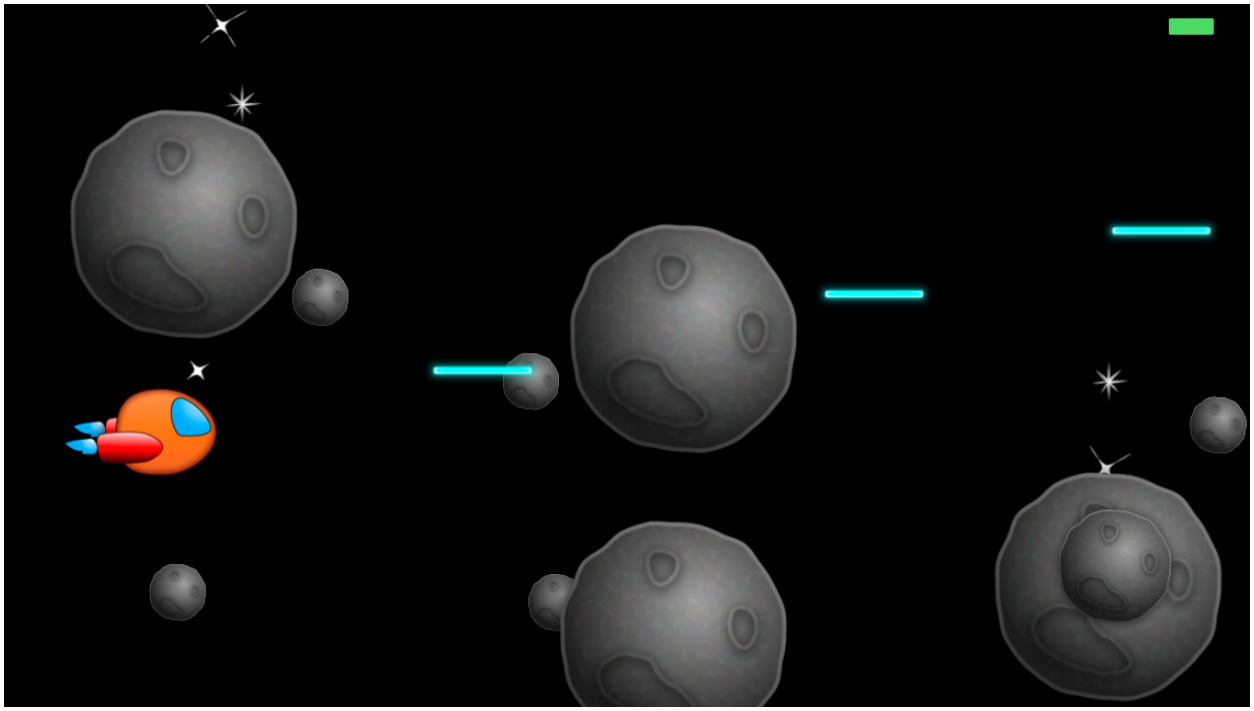
This creates a laser to the right of the player's position and runs an action to move it the entire width of the screen to the right. This means it will overshoot the edge of the screen, but that's OK—the important thing is that it goes past the edge of the screen.

Once the laser finishes moving, the method removes it from the scene.

You want to spawn a laser whenever the player taps in the gameplay state, so add this to the bottom of `touchesBegan:withEvent:`:

```
if (_levelManager.gameState == GameStatePlay) {
  [self spawnPlayerLaser];
  return;
}
```

That's it. Build and run your code, and go pew-pew!



Collision Detection

You've got lasers and asteroids, but there's a huge lack of satisfaction because nothing explodes!

Let's add some collision detection. You'll want to know if a laser hits any asteroids and if so, destroy both the laser and the asteroid.

In the previous version of this starter kit, I recommended that you start with simple bounding box collision detection before moving to a more precise method: physics-based collision detection. That's because it took a little work to correctly integrate a physics engine with Cocos2D.

However, Sprite Kit has a built-in physics engine that is tightly integrated with the rest of the framework and is super-easy to use. In fact, it's so easy that when you need collision detection I recommend you use the physics engine straight away rather than look for an alternative method. So that's what you'll be doing here!

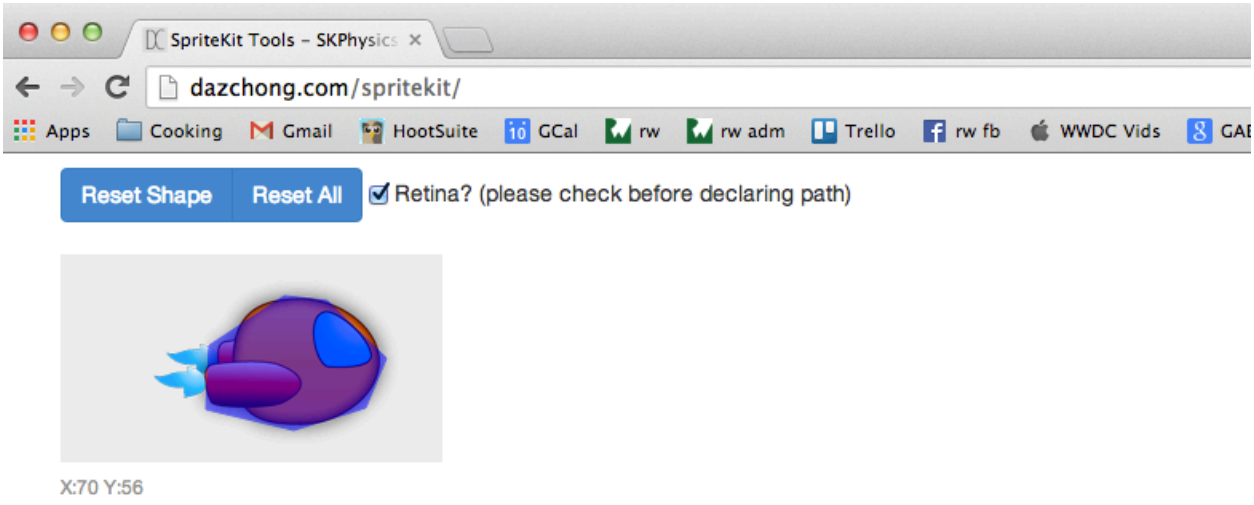
Every `SKNode` in Sprite Kit, like the player ship, an asteroid or a laser, can have a physics body attached. This gives you a way to represent the shape of the object

that you can use for collision detection purposes. Note that the shape doesn't need to match up exactly to the sprite—usually it's a simpler approximation.

Sprite Kit has methods for creating physics bodies that are either circles, rectangles or arbitrary polygons. For this game, you'll be defining a polygon shape for each of the sprites that maps to the shape of the sprite. All you need to define a polygon shape are the coordinates within the sprite that make up the shape.

How do you get these coordinates? Well, you can measure with a tool like Photoshop if you'd like, but I came across this handy web tool that makes the process even easier: http://dazchong.com/spritekit/

This free web tool allows you to drop a sprite onto a canvas and click around the sprite to define your polygon shape. For example, here's an image of my dragging **spaceflier_sm_1@2x.png** onto the editor and tracing a rough polygon around the sprite:



Output

```
SKSpriteNode *sprite = [SKSpriteNode spriteNodeWithImageNamed:@"SpaceFlier_sm_1@2x.png"];

CGFloat offsetX = sprite.frame.size.width * sprite.anchorPoint.x;
CGFloat offsetY = sprite.frame.size.height * sprite.anchorPoint.y;

CGMutablePathRef path = CGPathCreateMutable();

CGPathMoveToPoint(path, NULL, 70 - offsetX, 51 - offsetY);
CGPathAddLineToPoint(path, NULL, 83 - offsetX, 50 - offsetY);
CGPathAddLineToPoint(path, NULL, 102 - offsetX, 32 - offsetY);
CGPathAddLineToPoint(path, NULL, 95 - offsetX, 16 - offsetY);
CGPathAddLineToPoint(path, NULL, 73 - offsetX, 9 - offsetY);
CGPathAddLineToPoint(path, NULL, 45 - offsetX, 16 - offsetY);
CGPathAddLineToPoint(path, NULL, 46 - offsetX, 36 - offsetY);

CGPathCloseSubpath(path);
```

Note that when you are defining a shape, you want to define as few vertices as possible. The more vertices you add to a shape, the more resources it takes from the physics engine.

The tool then gives you some rough code you can use to define the physics bodies in Sprite Kit based on what you clicked. You don't necessarily have to use the sample code as-is. In this game, for example, I pulled out the coordinates but used slightly different code.

I have already used this tool to figure out the coordinates for each sprite in the space game. I will list out the coordinates I got below, but if you'd like the practice, you can replace these with your own coordinates by using this tool as you continue through the tutorial.

When you're tracing sprites for this game, be sure to use the @2x version of each sprite and that "Retina" is checked. That way the coordinates will be in points and the tool will know you are using a retina image that has two pixels per point. As for the iPad and iPad retina, you will write a few conversion routines that adapt the coordinates appropriately.

Let's start with that. Open **Entity.m** and add the following new methods:

```
- (void)moveToPoint:(CGPoint)point path:(CGMutablePathRef)path offset:(CGPoint)offset
{
  if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    CGPathMoveToPoint(path, NULL, point.x*2 - offset.x, point.y*2 - offset.y);
  } else {
    CGPathMoveToPoint(path, NULL, point.x - offset.x, point.y - offset.y);
  }
}

- (void)addLineToPoint:(CGPoint)point path:(CGMutablePathRef)path
offset:(CGPoint)offset {
  if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
    CGPathAddLineToPoint(path, NULL, point.x*2 - offset.x, point.y*2 - offset.y);
  } else {
    CGPathAddLineToPoint(path, NULL, point.x - offset.x, point.y - offset.y);
  }
}
```

These two methods are wrappers for the `CGPathMoveToPoint` and `CGPathAddLineToPoint` functions. They allow you to specify the positions in the @2x retina coordinates that you traced on the web page, and then they convert the positions appropriately if you are running on the iPad. Remember, on the iPad the artwork is double the size in points.

While you're adding helper methods to this file, add one more that you'll need soon:

```
- (void)collidedWith:(SKPhysicsBody *)body contact:(SKPhysicsContact*)contact {
```

```
}
```

Whenever the scene detects an entity colliding with another entity, it will call this method. Right now it's just a stub, but later you'll override these for each subclass to have different behavior.

You're almost ready to start creating the physics bodies using these helper methods, but first you should define an enumeration for the various types of physics bodies you'll be adding to the game. To do this, open **Entity.h** and add this enumeration to the top of the file, above the `@interface` line:

```
typedef NS_OPTIONS(NSInteger, EntityCategory)
{
  EntityCategoryPlayer      = 1 << 0,
  EntityCategoryAsteroid    = 1 << 1,
  EntityCategoryAlien       = 1 << 2,
  EntityCategoryPlayerLaser = 1 << 3,
  EntityCategoryAlienLaser  = 1 << 4,
  EntityCategoryPowerup     = 1 << 5
};
```

Here you define a bitmask for each category of objects you'll add to the game. So far you have only the player ship, asteroid and player laser, but you'll be adding aliens, alien lasers and power-ups later on in the tutorial.

Also, add declarations for the three helper methods you just wrote somewhere below the `@interface` line:

```
- (void)addLineToPoint:(CGPoint)point path:(CGMutablePathRef)path
offset:(CGPoint)offset;
- (void)moveToPoint:(CGPoint)point path:(CGMutablePathRef)path offset:(CGPoint)offset;
- (void)collidedWith:(SKPhysicsBody *)body contact:(SKPhysicsContact*)contact;
```

Now let's put these to use. Open **Player.m** and add this new method:

```
- (void)setupCollisionBody {
  // 1
  CGPoint offset = CGPointMake(self.size.width * self.anchorPoint.x,
    self.size.height * self.anchorPoint.y);
  CGMutablePathRef path = CGPathCreateMutable();
  [self moveToPoint:CGPointMake(70, 51) path:path offset:offset];
  [self addLineToPoint:CGPointMake(83, 50) path:path offset:offset];
  [self addLineToPoint:CGPointMake(102, 32) path:path offset:offset];
  [self addLineToPoint:CGPointMake(95, 16) path:path offset:offset];
  [self addLineToPoint:CGPointMake(73, 9) path:path offset:offset];
  [self addLineToPoint:CGPointMake(45, 16) path:path offset:offset];
  [self addLineToPoint:CGPointMake(46, 36) path:path offset:offset];
  CGPathCloseSubpath(path);
  self.physicsBody = [SKPhysicsBody bodyWithPolygonFromPath:path];
```

```
  // 2
  [self attachDebugFrameFromPath:path color:[SKColor redColor]];

  // 3
  self.physicsBody.categoryBitMask = EntityCategoryPlayer;
  self.physicsBody.collisionBitMask = 0;
  self.physicsBody.contactTestBitMask = EntityCategoryAsteroid |
    EntityCategoryAlienLaser | EntityCategoryPowerup | EntityCategoryAlien;
}
```

Let's go over this method section by section:

1. This first part creates the polygon path for the ship. The code uses the same coordinates from the web tool mentioned earlier, except here the helper methods convert the positions appropriately for the iPad. Feel free to replace these coordinates with your own shapes that you got using the web tool or Photoshop, for practice.
2. This uses a helper method from `SKTUtils` to attach a debug frame to the shape. A debug frame will help you visualize the physics shape to make sure it's working OK and matches up well to your sprite.
3. This sets the category of the sprite to player, and sets it to detect collisions with asteroids, alien lasers, power-ups and aliens. Note that the code sets `collisionBitMask` to 0, because you don't want the player ship to actually bounce off of anything—you just want to receive notifications when the objects collide so that you can run some custom code.

Next, call this method from `init`, right after the call to `setupAnimation`:

```
[self setupCollisionBody];
```

You're almost ready to test this out—you just have to create the physics world in your layer and configure it to receive contacts. To do this, open **MyScene.m** and add the following right before the `@implementation` to mark the class as implementing the `SKPhysicsContactDelegate`:

```
@interface MyScene() <SKPhysicsContactDelegate>
@end
```

Then add this new method after `setupPlayer`:

```
- (void)setupPhysics {
  self.physicsWorld.contactDelegate = self;
  self.physicsWorld.gravity = CGVectorMake(0, 0);
}
```

This sets the scene as the delegate of the physics world. It also sets gravity to the zero vector since the game takes place in outer space, where there is no gravity.

Call this method in `initWithSize:`, right after the call to `setupMotionManager`:

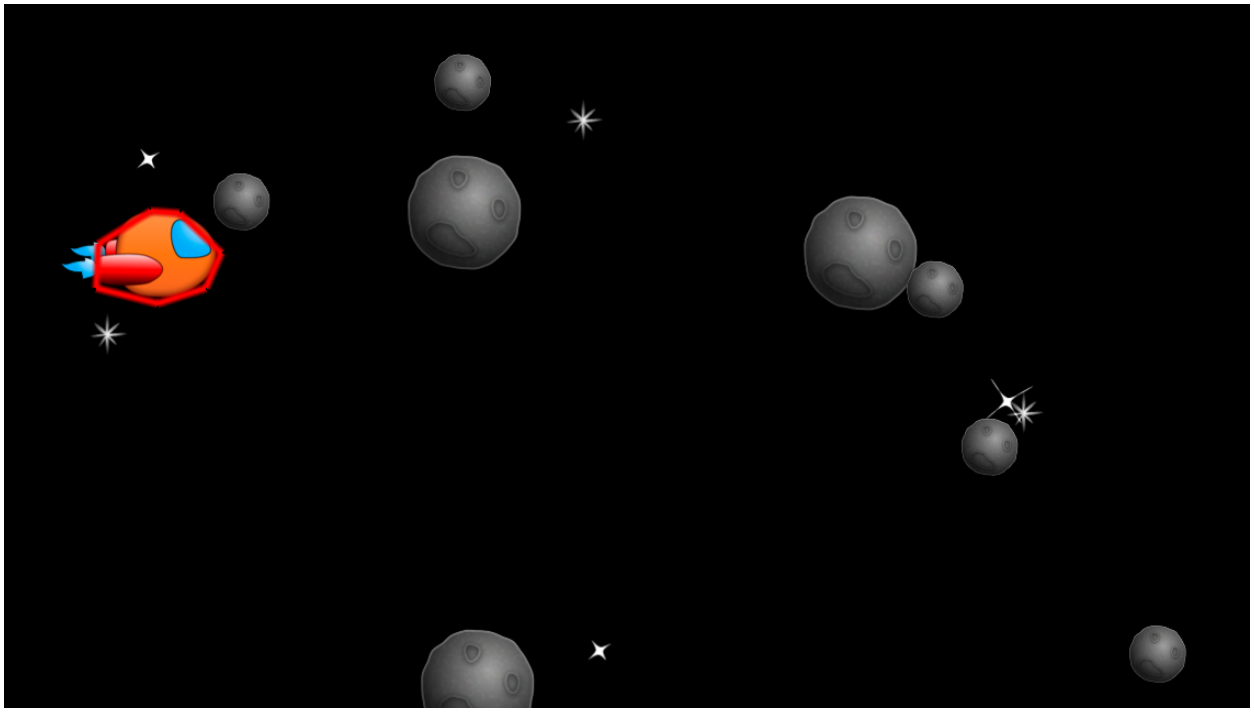
```
 [self setupPhysics];
```

Finally, add the `SKPhysicsContactDelegate` method that is called when two physics bodies collide. Add it to the bottom of the file:

```
#pragma mark - Physics functions

- (void)didBeginContact:(SKPhysicsContact *)contact
{
  SKNode *node = contact.bodyA.node;
  if ([node isKindOfClass:[Entity class]]) {
    [(Entity*)node collidedWith:contact.bodyB contact:contact];
  }

  node = contact.bodyB.node;
  if ([node isKindOfClass:[Entity class]]) {
    [(Entity*)node collidedWith:contact.bodyA contact:contact];
  }
}
```

This checks to see if the `SKNodes` for the physics bodies that collide are `Entity` classes. They should be, but it's good to check anyway. If they are, the code calls the stub method you wrote in `Entity` earlier, which you will later override in each subclass.

That's it—time to test! Build and run on your device, and you should see a debug shape drawn around your ship that matches what you traced:



Great! Now that you have the idea down, let's repeat this for the other shapes. I'm going to give you the coordinates I traced using the web tool, but you can replace these coordinates with your own if you'd like the practice. I also will not include a discussion of the code, since it's the same idea as defining the collision bodies for the player ship.

Add this to **Asteroid.m**:

```
- (void)setupCollisionBody {
  CGPoint offset = CGPointMake(self.size.width * self.anchorPoint.x,
    self.size.height * self.anchorPoint.y);
  CGMutablePathRef path = CGPathCreateMutable();
  [self moveToPoint:CGPointMake(30, 105) path:path offset:offset];
  [self addLineToPoint:CGPointMake(47, 119) path:path offset:offset];
  [self addLineToPoint:CGPointMake(78, 123) path:path offset:offset];
  [self addLineToPoint:CGPointMake(105, 112) path:path offset:offset];
  [self addLineToPoint:CGPointMake(123, 83) path:path offset:offset];
  [self addLineToPoint:CGPointMake(119, 47) path:path offset:offset];
  [self addLineToPoint:CGPointMake(99, 24) path:path offset:offset];
  [self addLineToPoint:CGPointMake(46, 22) path:path offset:offset];
  [self addLineToPoint:CGPointMake(25, 45) path:path offset:offset];
  [self addLineToPoint:CGPointMake(16, 79) path:path offset:offset];
  CGPathCloseSubpath(path);

  self.physicsBody = [SKPhysicsBody bodyWithPolygonFromPath:path];
  [self attachDebugFrameFromPath:path color:[SKColor redColor]];

  self.physicsBody.categoryBitMask = EntityCategoryAsteroid;
```

```
  self.physicsBody.collisionBitMask = 0;
  self.physicsBody.contactTestBitMask = EntityCategoryPlayerLaser |
EntityCategoryPlayer;
}
```

Add a call to this method from `initWithAsteroidType:`, *before* the call to `setScale::`

```
 [self setupCollisionBody];
```

> **Note**: It's important to create the physics body before setting the scale so that the physics body is located in the proper place when scaling the shape. Otherwise, you will see the physics body at a strange offset to the shape.

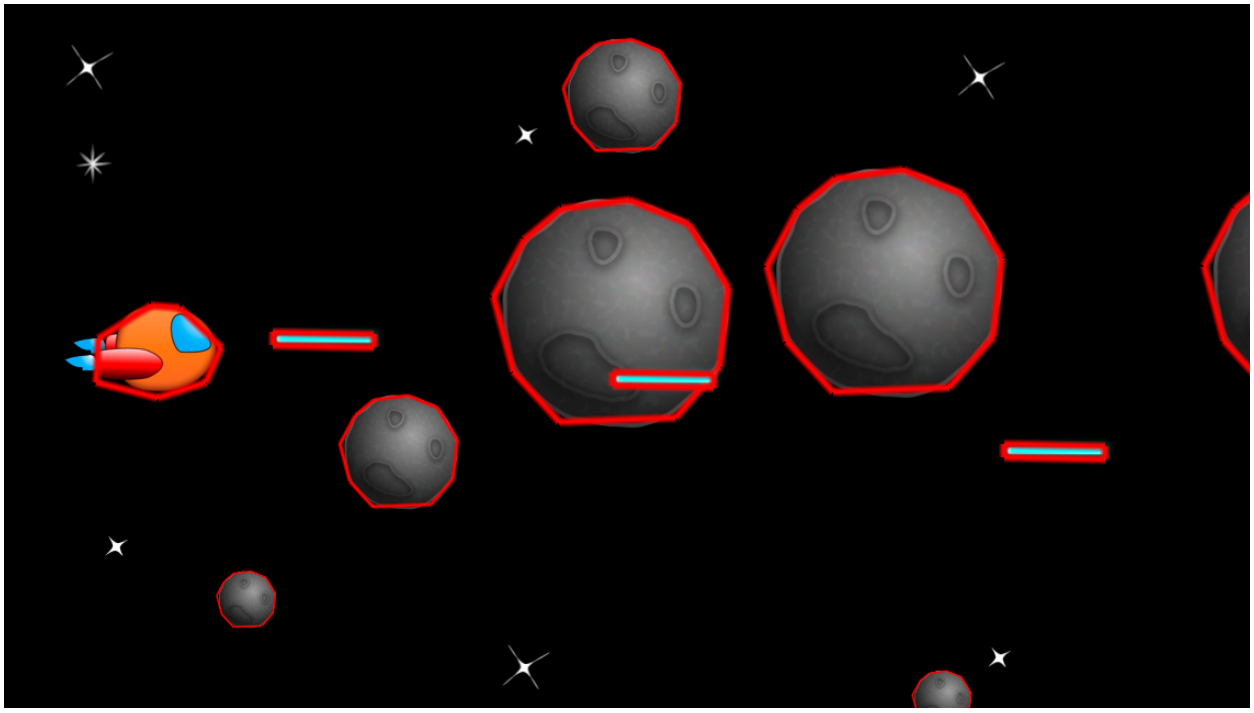
Add this to **PlayerLaser.m**:

```
- (void)setupCollisionBody {
  CGPoint offset = CGPointMake(self.size.width * self.anchorPoint.x, self.size.height
* self.anchorPoint.y);
  CGMutablePathRef path = CGPathCreateMutable();
  [self moveToPoint:CGPointMake(7, 12) path:path offset:offset];
  [self addLineToPoint:CGPointMake(53, 11) path:path offset:offset];
  [self addLineToPoint:CGPointMake(53, 5) path:path offset:offset];
  [self addLineToPoint:CGPointMake(7, 6) path:path offset:offset];
  CGPathCloseSubpath(path);

  self.physicsBody = [SKPhysicsBody bodyWithPolygonFromPath:path];
  [self attachDebugFrameFromPath:path color:[SKColor redColor]];

  self.physicsBody.categoryBitMask = EntityCategoryPlayerLaser;
  self.physicsBody.collisionBitMask = 0;
  self.physicsBody.contactTestBitMask = EntityCategoryAsteroid | EntityCategoryAlien;
}
```

Add a call to this method from `init`, inside the `if` block:

```
[self setupCollisionBody];
```

Build and run again, and now everything has collision shapes:



You've made a lot of progress and you're almost done. Now that you have the physics shapes defined and are calling the collision callback method, you need to implement the callback method appropriately on each subclass.

For now, you will just destroy the asteroids when they're hit by lasers—there will be no effect if an asteroid hits the player ship.

You want to play a sound effect when an asteroid is hit by a laser beam, and `MyScene` is responsible for playing sounds, so you need to add a helper method for that first. Add the following method to **MyScene.m**:

```
- (void)playExplosionLargeSound {
  [self runAction:_soundExplosionLarge];
}
```

Then open **MyScene.h** and declare the method:

```
- (void)playExplosionLargeSound;
```

Finally, open **Asteroid.m** and import the scene header at the top:

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

Then implement the collision callback as follows:

```
- (void)collidedWith:(SKPhysicsBody *)body contact:(SKPhysicsContact *)contact {
  if (body.categoryBitMask & EntityCategoryPlayerLaser) {
    Entity * other = (Entity *)body.node;
```

```
    MyScene *scene = (MyScene *)self.scene;
    [scene playExplosionLargeSound];
    [other removeFromParent];
    [self removeFromParent];
  }
}
```

This checks to see if the body that collided with the asteroid is a player laser. If it is, it plays an explosion sound and removes both bodies from the scene.

Build and run, and have fun blasting some asteroids!

Parallax Scrolling

Your game is starting to look pretty awesome: you have a hero, enemies and blasting! But in the Space Game Starter Kit, you don't just want awesome—you want überly awesome.

That's why you're going to add a parallax scrolling space background.

If you aren't familiar with parallax scrolling, it's just a fancy way of saying, "move parts of the background at different speeds than the other parts." If you've ever played SNES games like ActRaiser, you've often seen parallax scrolling in the background of the action levels to give a sense of depth.

Unlike Cocos2D, Sprite Kit does not come with parallax scrolling built-in. However, it's easy to implement yourself.

Create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **ParallaxNode** for Class, **SKNode** for Subclass of, click **Next** and then click **Create**.

Open **ParallaxNode.h** and replace the contents with the following:

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

@interface ParallaxNode : SKNode

@property (nonatomic, assign) CGPoint velocity;

- (id)initWithVelocity:(CGPoint)velocity;
- (void)addChild:(SKSpriteNode *)node parallaxRatio:(float)parallaxRatio;
- (void)update:(float)deltaTime;

@end
```

The idea behind `ParallaxNode` is that you will create one of these nodes with a velocity and a set of children sprites, like space dust, planetoids and so on. The

node will move each of the children sprites at some ratio of its velocity, so some sprites may move faster or slower than others for a parallax effect.

Next switch to **ParallaxNode.m** and replace the contents with the following:

```
#import "ParallaxNode.h"

@implementation ParallaxNode

- (id)initWithVelocity:(CGPoint)velocity {
  if ((self = [super init])) {
    _velocity = velocity;
  }
  return self;
}

- (void)addChild:(SKSpriteNode *)node parallaxRatio:(float)parallaxRatio {
  if (!node.userData) {
    node.userData = [NSMutableDictionary dictionary];
  }
  node.userData[@"ParallaxRatio"] = @(parallaxRatio);
  [super addChild:node];
}

- (void)update:(float)deltaTime {
  [self.children enumerateObjectsUsingBlock:^(SKSpriteNode * node, NSUInteger idx,
BOOL *stop) {
    float parallaxRatio = [(NSNumber *)node.userData[@"ParallaxRatio"] floatValue];
    CGPoint childVelocity = CGPointMultiplyScalar(self.velocity, parallaxRatio);
    CGPoint offset = CGPointMultiplyScalar(childVelocity, deltaTime);
    node.position = CGPointAdd(node.position, offset);
  }];
}

@end
```

Let's go over each of these methods:

- `initWithVelocity:`: This is a simple initializer that stores the passed-in velocity for later reference.
- `addChild:parallaxRatio:`: This adds a child sprite node to the parallax node. Note that the method needs to link the parallax ratio (how fast to move the sprite compared to the parallax node itself) to the sprite node. One easy way to do this is to store the ratio in the `userData` dictionary on the sprite node.
- `update:`: This method will be called each frame. It enumerates through all the node's children and moves them based on the velocity times the parallax ratio.

Now let's try this out. Open **MyScene.m** and import the header file:

```
#import "ParallaxNode.h"
```

Then add some new private instance variables for the parallax node and the various background sprite nodes you'll place inside:

```
ParallaxNode *_parallaxNode;
SKSpriteNode *_spacedust1;
SKSpriteNode *_spacedust2;
SKSpriteNode *_planetsunrise;
SKSpriteNode *_galaxy;
SKSpriteNode *_spatialanomaly;
SKSpriteNode *_spatialanomaly2;
```

Next add this new method to create the parallax node and backgrounds, right after `setupPhysics`:

```
- (void)setupBackground {
  _spacedust1 = [SKSpriteNode spriteNodeWithImageNamed:@"bg_front_spacedust"];
  _spacedust1.position = CGPointMake(0, self.size.height/2);
  _spacedust2 = [SKSpriteNode spriteNodeWithImageNamed:@"bg_front_spacedust"];
  _spacedust2.position = CGPointMake(_spacedust2.size.width, self.size.height/2);
  _planetsunrise = [SKSpriteNode spriteNodeWithImageNamed:@"bg_planetsunrise"];
  _planetsunrise.position = CGPointMake(600, 0);
  _galaxy = [SKSpriteNode spriteNodeWithImageNamed:@"bg_galaxy"];
  _galaxy.position = CGPointMake(0, self.size.height * 0.7);
  _spatialanomaly = [SKSpriteNode spriteNodeWithImageNamed:@"bg_spacialanomaly"];
  _spatialanomaly.position = CGPointMake(900, self.size.height * 0.3);
  _spatialanomaly2 = [SKSpriteNode spriteNodeWithImageNamed:@"bg_spacialanomaly2"];
  _spatialanomaly2.position = CGPointMake(1500, self.size.height * 0.9);

  _parallaxNode = [[ParallaxNode alloc] initWithVelocity:CGPointMake(-100, 0)];
  _parallaxNode.position = CGPointMake(0, 0);
  [_parallaxNode addChild:_spacedust1 parallaxRatio:1];
  [_parallaxNode addChild:_spacedust2 parallaxRatio:1];
  [_parallaxNode addChild:_planetsunrise parallaxRatio:0.5];
  [_parallaxNode addChild:_galaxy parallaxRatio:0.5];
  [_parallaxNode addChild:_spatialanomaly parallaxRatio:0.5];
  [_parallaxNode addChild:_spatialanomaly2 parallaxRatio:0.5];
  _parallaxNode.zPosition = -1;
  [_gameLayer addChild:_parallaxNode];
}
```

First you create each of the background objects—space dust and other things you might find out there—and position them in various places. I chose these positions based on what I thought looked best.

You then create a parallax node that moves each of its children 100 points per second to the left along the x-axis, if the `parallaxRatio` is 1. A lower `parallaxRatio`

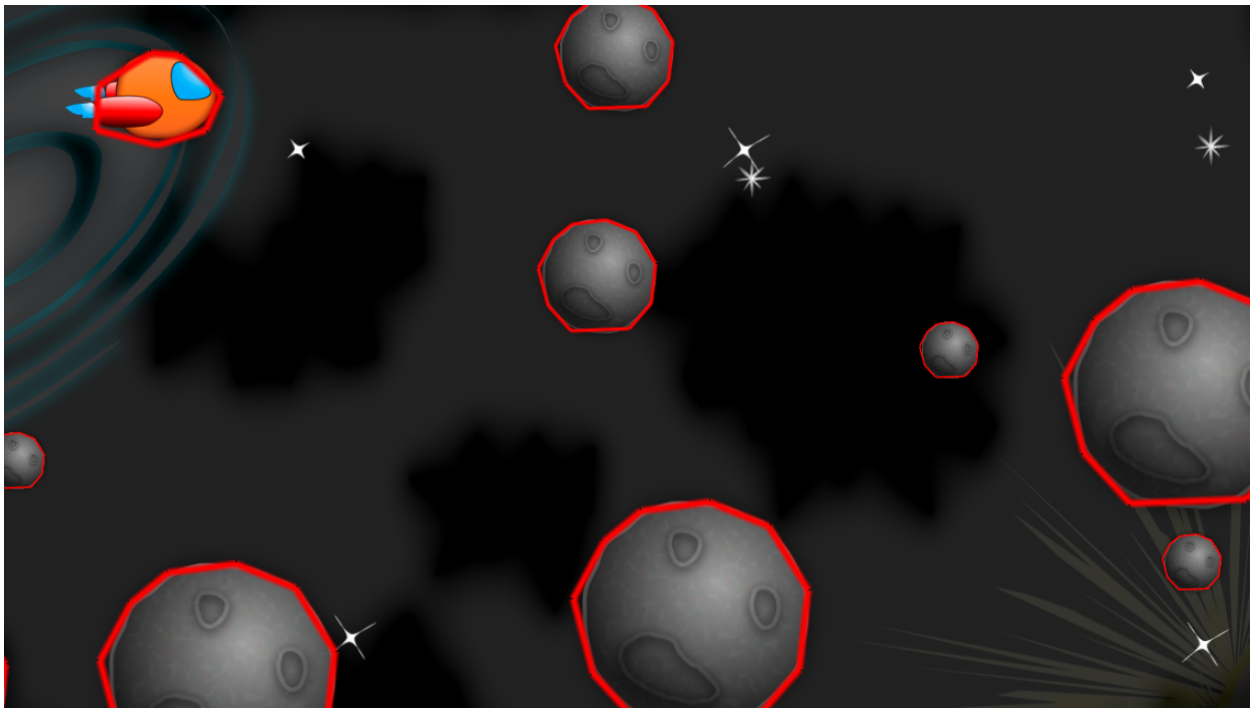
results in slower movement—for example, a `parallaxRatio` of 0.5 makes the non-spacedust backgrounds move at 50 points per second.

Finally, the method sets the z-position of the parallax node to -1 so that it's behind everything else and adds it to the layer.

Call this method from `initWithSize:`, after the call to `setupPhysics`:

```
[self setupBackground];
```

Build and run your project, and you should see an enhanced space scene:



Already there's a feeling of greater depth, but nothing is moving!

To fix this, you need to call the `update:` method on the parallax node each frame so that the node can move each of its children appropriately. To do this, add the following method after `updatePlayer`:

```
- (void)updateBg {
  [_parallaxNode update:_deltaTime];
}
```

Then call this method in `update:`, right *before* the check to see if you are in `GameStatePlay`. You want this called in every state, not just the play state:

```
[self updateBg];
```

Build and run your project, and you should begin to see the galaxy scroll by:

However, after a few seconds pass you'll notice a major problem: you run out of things to scroll through, leaving you staring at a plain background again! Your ship didn't just jump to intergalactic space (where, mysteriously, there are still asteroids), so let's see what you can do about this.

You want the background to keep scrolling endlessly, so you're going to employ a simple strategy: move the background back to the right once it has moved offscreen to the left.

To do this, add the following code to the bottom of `updateBg`:

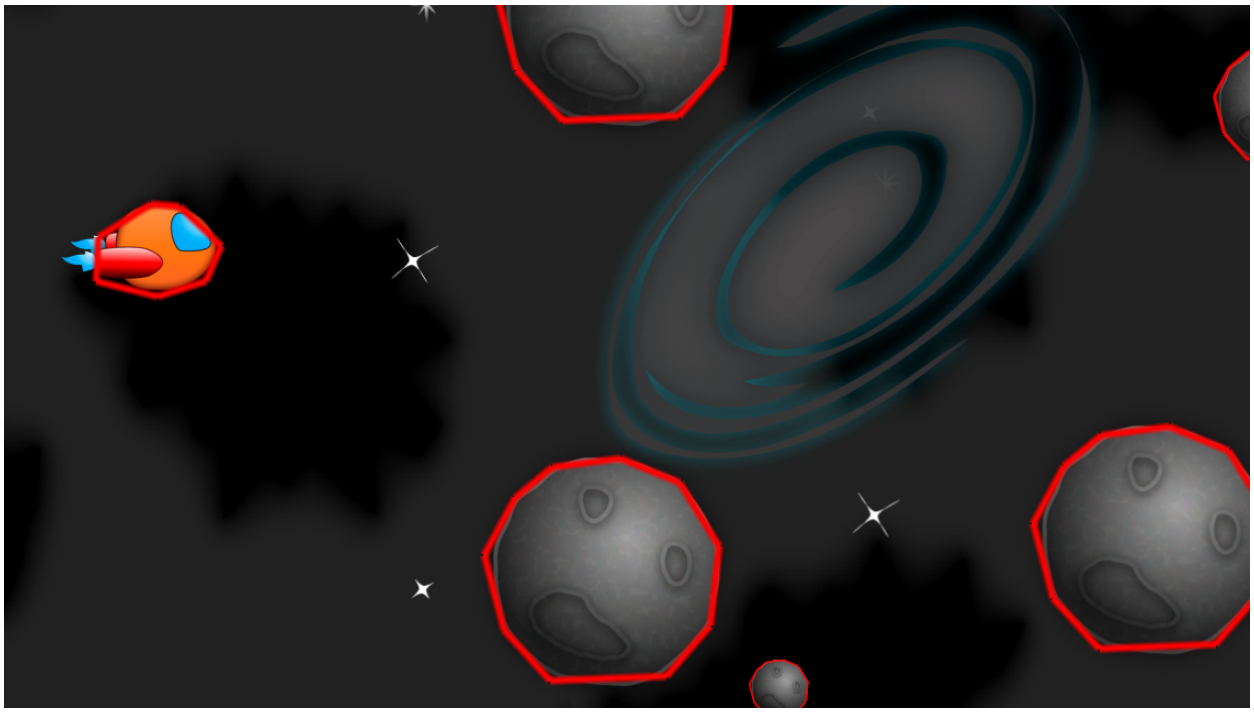
```
// 1
NSArray *bgs = @[_spacedust1, _spacedust2, _planetsunrise, _galaxy, _spatialanomaly,
_spatialanomaly2];
for (SKSpriteNode *bg in bgs) {
  // 2
  CGPoint scenePos = [bg convertPoint:bg.position toNode:self];
  // 3
  if (scenePos.x < -bg.size.width) {
    bg.position = CGPointAdd(bg.position, CGPointMake(_spacedust1.size.width*2, 0));
  }
}
```

Let's go over this section by section:

1. The code creates an array of the background sprites so that it's easy to iterate through them.

2. Remember that the position of a node is its position inside its parent node's coordinate system. You want to figure out if the node is offscreen, so you need the position in the scene's coordinate system. You can get this with the `convertPoint:toNode:` helper method. For more information about this, check out Chapter 5 of *iOS Games by Tutorials*, "Scrolling."
3. If the sprite is offscreen, the code moves the sprite two times the space dust's width to the right. This is because the sprites have been placed as if there is a large repeating background of double the space dust's width.

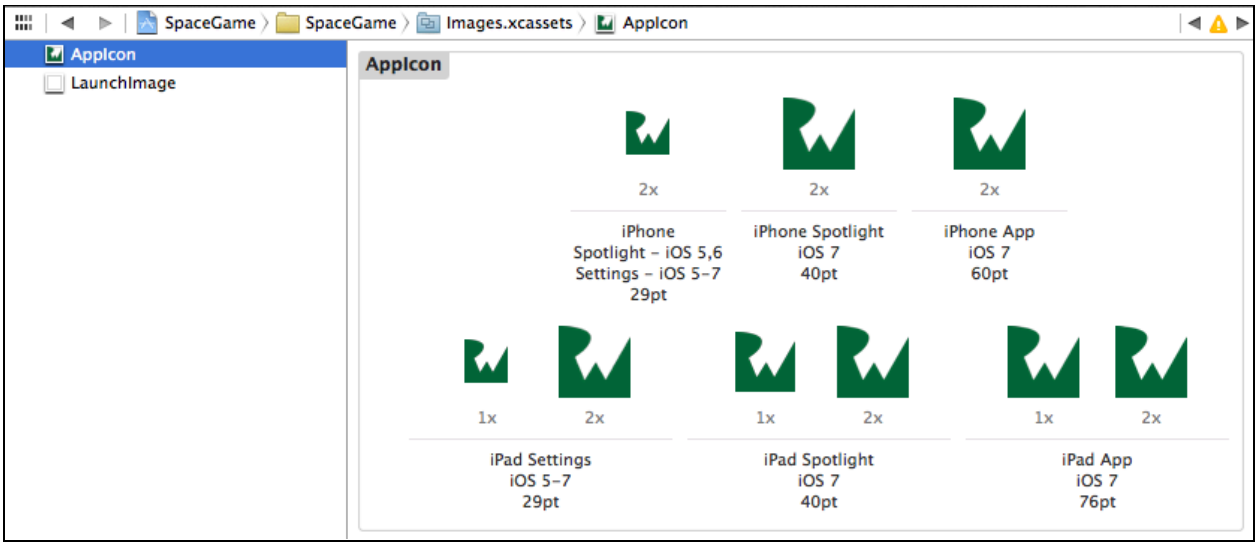
Build and run your project, and now the background should scroll continuously, to über-awesome effect!



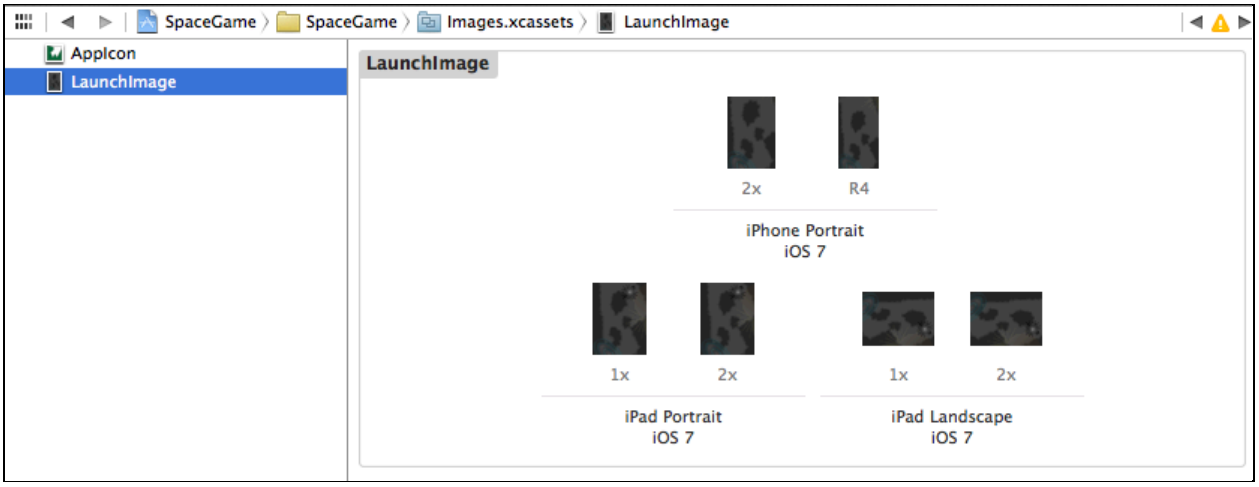
Finishing Touches

This game is almost ready to ship—no pun intended.

However, there are a few remaining tidbits to take care of first. Open **Images.xcassets** and click on your **AppIcon** entry. You will find some icons for the game under **Art\Other**. Drag them one at a time into the appropriate box on the screen. When you're done, it should look like the following:



Then do the same for **LaunchImage**—I have put some launch images in the folder as well. When you're done, it should look like the following:



> **Note**: At the time of writing this starter kit, there appears to be a bug where launch images set in this way don't show up properly on the iPhone, although they do on the iPad. If anyone knows a good workaround for this, let me know. ☺

Finally, open **SKTUtils/SKTNode+DebugDraw.m** and set the `kDebugDraw` constant at the top to `NO` to turn off debug drawing:

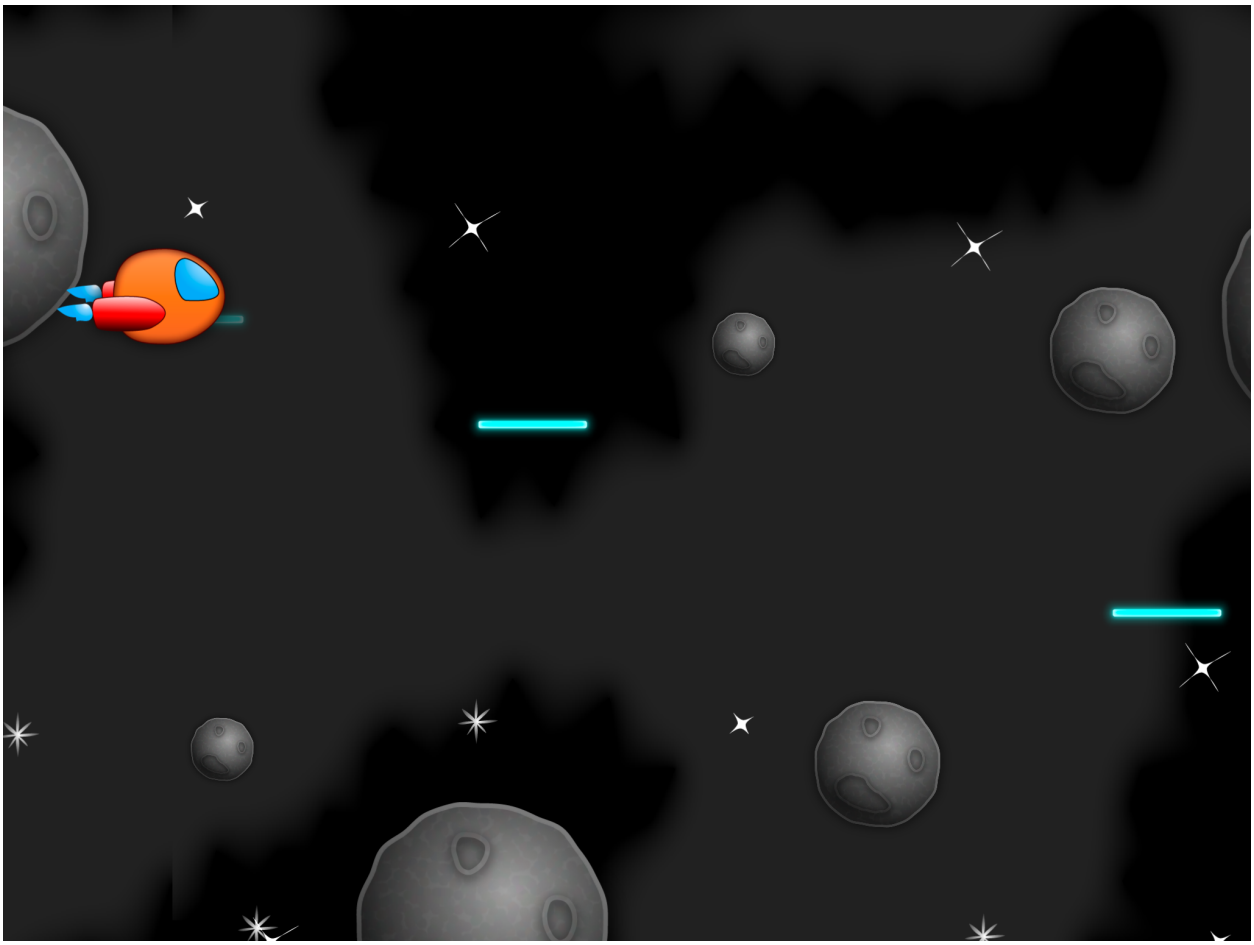
```
static BOOL kDebugDraw = NO;
```

You can turn this back on in subsequent chapters if you need to debug the shapes you are creating, but for now this makes the game look nice.

Delete the app from your device, do a **Product\Clean** and re-run, and enjoy a better splash screen, a better icon and no debug drawing!

What About the iPad?

If you have an iPad, try running the game on it—you'll see it works perfectly already. ☺



That's because you've been very careful to design the game so that it works on all kinds of devices: iPhones and iPod Touches (with or without retina display, on both 3.5" and 4" screens), and iPads (also with or without retina display).

Here's what you did:

- You chose the "Universal app" option when creating a new project with the Sprite Kit template.
- For every piece of art in your project, you made sure to add a double-sized version for the iPhone/iPod Touch retina (with the @2x extension), an iPad version (with the ~ipad extension, same as the @2x version in this game) and a quadruple-sized version for the iPad retina (with the @2x~ipad extension). Sprite Kit automatically loads the correct image based on the extension.
- You made your background images big enough so they'd fill the entire iPad screen (and overlap a bit on the iPhone).
- You didn't hardcode positions based on the iPhone screen size. Instead, you based them on multiples of the window size.

Pretty easy, eh? ☺

Where to Go From Here?

If you've made it this far, you've earned yourself impressive space game chops—you're shooting lasers, blowing up asteroids and looking pretty slick while you're at it. One could say you're a bit like Luke Skywalker piloting an X-wing… except you're piloting Xcode instead. ☺

If you're happy with how things are so far, you can stop here and use this project as a starting point for your own game. Here are a few features you might want to add:

- Right now there's no way to win—or lose! Come up with some criteria for winning and losing the game and add some game logic.
- Do you want to keep a score for the game? Consider giving points each time the player shoots an asteroid and displaying that in a label on the screen.
- If you are an artist, have an artist friend or have money to hire one, why not replace the artwork with some of your own?

Or you can continue reading the next chapter, where you, like Luke, will crash your Xcode 5 into the swamp of DagoSpriteKit and awaken your latent space-game-making powers!

Chapter 2: Hit Points and Explosions

Welcome back, brave explorer!

In this chapter, you're going to make new features rise out of the code swamp with some newly-learned Xcode mind tricks!

This is a short chapter, which will give you a much needed break after your previous exploration. You'll start by modifying your project so that an asteroid's hit points correspond to its size. You'll then add some spectacular explosions for asteroid destruction and hits to the player ship. Finally, you'll add some win/lose logic into the game so that you can see if you have what it takes to survive the vicious asteroid belt!

You will pick up where you left off in Chapter 1. If you skipped that chapter, you can continue on with the **SpaceGame1** project that comes with the starter kit.

Wars do not make one great, but adding massive space explosions to your game does, so let's get coding. ☺

> **Note:** In previous versions of this starter kit, readers spent the majority of this chapter integrating the Box2D physics engine into the Cocos2D game. However, with Sprite Kit this has been done for you already, so you've been using the physics engine in your game since Chapter 1! Gotta love Sprite Kit.

Using Hit Points

Right now, all of the asteroids are destroyed with a single shot. It would be much more interesting if it took more shots to destroy the larger asteroids than the smaller asteroids.

You've actually already added two properites for `hp` and `maxHp` to your `Entity` class and have set them to the appropriate value for each object—you just aren't using them yet. Let's fix that.

Open **Entity.m** and add these new methods:

```
- (BOOL)isDead {
  return _hp <= 0;
}

- (void)takeHit {
  if (_hp > 0) {
    _hp--;
  }
  if ([self isDead]) {
    [self destroy];
  }
}

- (void)cleanup {
  [self removeFromParent];
}

- (void)destroy {
  _hp = 0;
  self.physicsBody = nil;
  [self removeAllActions];
  [self runAction:
    [SKAction sequence:@[
      [SKAction fadeAlphaTo:0 duration:0.2],
      [SKAction performSelector:@selector(cleanup) onTarget:self]
    ]]
  ];
}
```

Let's go over these methods one-by-one:

- `isDead`: This is a helper method that checks to see if the entity is dead. An entity is dead if its hit points are equal to or less than 0.
- `takeHit`: This is a helper method that you'll call when the entity takes a hit. It decreases the hit points by one and if the Entity is dead as a result, calls the `destroy` method.
- `cleanup`: This is a helper method that you should call when you want to remove the object from the scene. Right now it simply calls `removeFromParent`, but later some of the subclasses may override this for other behavior.
- `destroy`: You'll call this method when an object is dead. Rather than destroy the object right away, to add a little polish the method fades the object out before removing it. Note that since `physicsBody` is set to `nil`, the object will no longer receive physics contact callbacks while in the process of fading out.

Next open **Entity.h** and declare these methods:

```
- (BOOL)isDead;
- (void)takeHit;
```

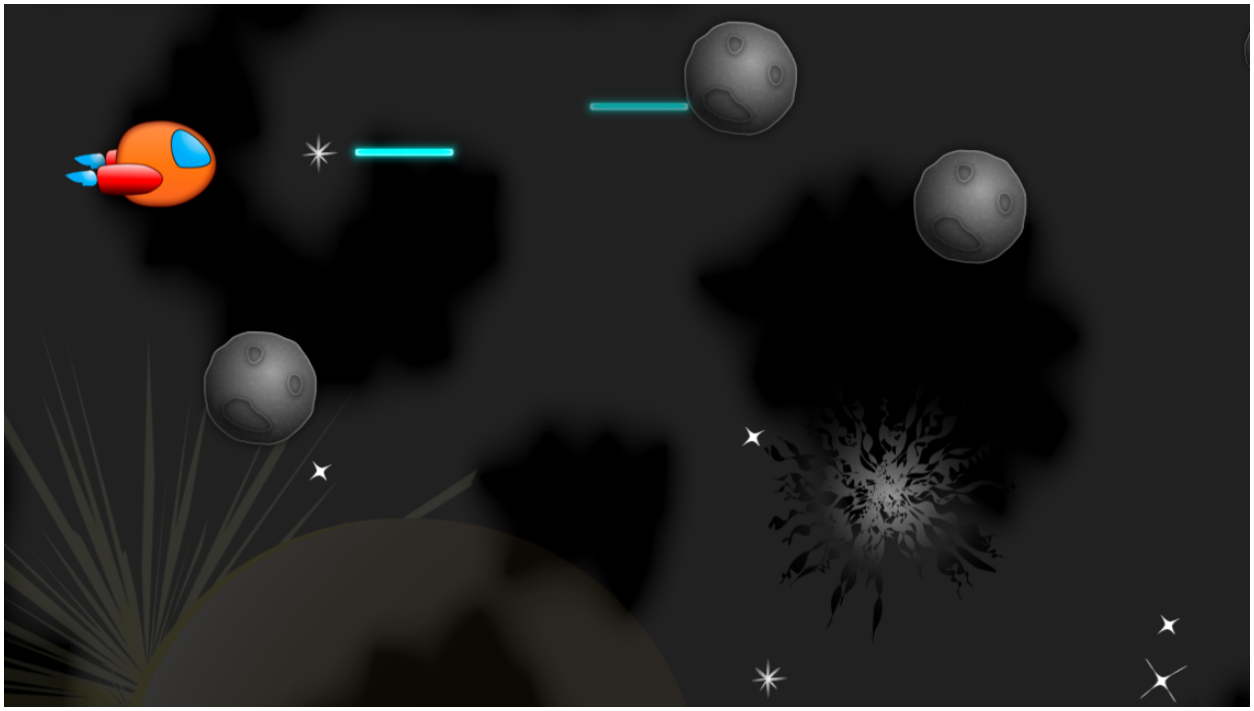
```
- (void)cleanup;
- (void)destroy;
```

Now that you have these new methods, you need to use them. To do so, switch to **Asteroid.m** and replace `collidedWith:contact:` with the following:

```
- (void)collidedWith:(SKPhysicsBody *)body contact:(SKPhysicsContact *)contact {
  if (body.categoryBitMask & EntityCategoryPlayerLaser) {
    Entity * other = (Entity *)body.node;
    [other destroy];
    [self takeHit];
    MyScene *scene = (MyScene *)self.scene;
    [scene playExplosionLargeSound];
  }
}
```

Instead of simply removing both objects from the scene, you destroy the other object—the laser—and have the asteroid take a hit. Remember, `takeHit` checks to see if the asteroid's HP is zero after decreasing the HP and if so calls `destroy` on the asteroid.

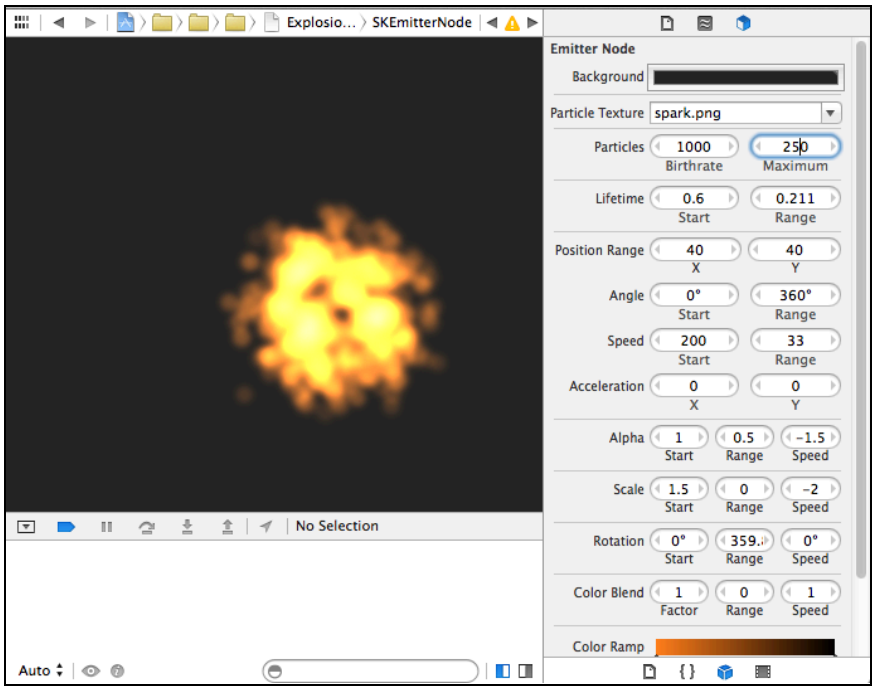
Build and run, and observe how it now takes more than one shot to destroy the medium and large asteroids:



Those asteroids just disappear, though—not in keeping with the `destroy` method name or with cherished space game aesthetics. You know what that means… it's time for some explosions!

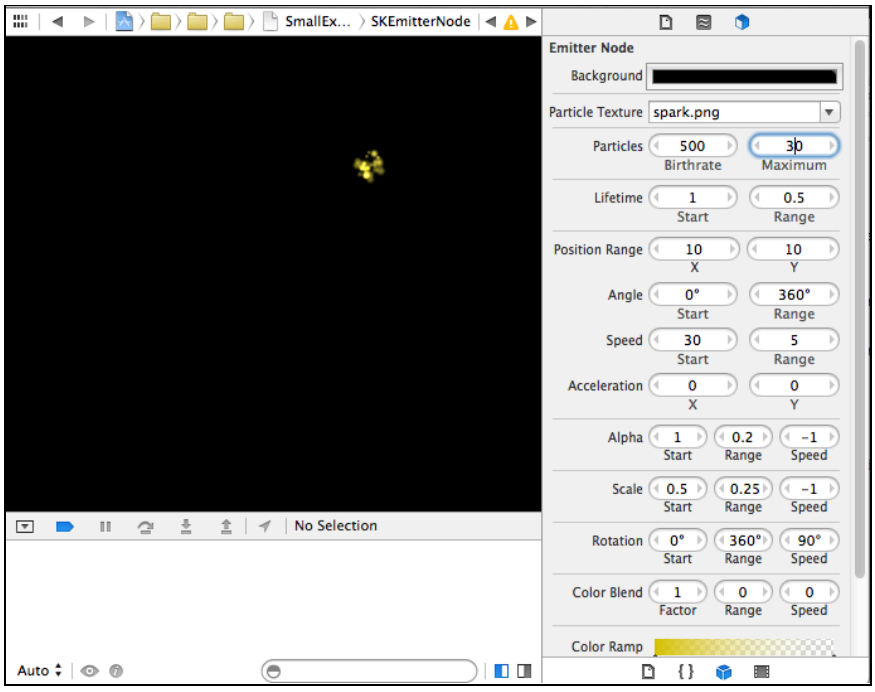
Explosions and Destruction!

I have already created a couple of explosion particle systems for you. There's a large explosion effect in **Art\Particles\Explosion.sks**:



Note that by default the effect will play only once, but if you click inside the **Maximum** box in the SKNode Inspector, it will loop continuously.

Similarly, there is a small explosion effect in **Art\Particles\SmallExplosion.sks**:



I came up with the values for these explosion effects through experimentation and personal preference. Feel free to play around with the configurations in the SKNode Inspector to fashion the explosions more to your liking!

Next you're going to add a helper method to spawn an explosion at a certain point. Then you'll call this method from the `Asteroid` class when an asteroid is hit by a laser.

Open **MyScene.m** and add this method right after `playExplosionLargeSound`:

```
- (void)spawnExplosionAtPosition:(CGPoint)position scale:(float)scale
  large:(BOOL)large {
  SKEmitterNode *emitter;

  if (large) {
    emitter = [NSKeyedUnarchiver unarchiveObjectWithFile:
      [[NSBundle mainBundle] pathForResource:@"Explosion" ofType:@"sks"]];
  } else {
    emitter = [NSKeyedUnarchiver unarchiveObjectWithFile:
      [[NSBundle mainBundle] pathForResource:@"SmallExplosion" ofType:@"sks"]];
  }
  emitter.position = position;
  emitter.particleScale = scale;
  emitter.numParticlesToEmit *= scale;
  emitter.particleLifetime /=scale;
  emitter.particlePositionRange = CGVectorMake(
    emitter.particlePositionRange.dx * scale,
    emitter.particlePositionRange.dy * scale);
  [emitter runAction:[SKAction skt_removeFromParentAfterDelay:1.0]];
```

```
  [_gameLayer addChild:emitter];

  if (large) {
    [self runAction:_soundExplosionLarge];
  } else {
    [self runAction:_soundExplosionSmall];
  }
}
```

This method spawns an explosion at a certain position with a certain scale, allowing you to make bigger or smaller explosions based on the size of the asteroid.

Unfortunately, `SKEmitterNode` does not have a simple scale property you can set on the particle system at the time of writing this starter kit, so instead you tweak some of the values, like `particleScale` and `particlePositionRange`, to decrease the size of the effect.

When the particle system ends, you want to remove it from the scene. The easiest way to do this is to remove the particle system after a delay, where the delay is the length of time you expect the particle system to run—1 second in this case.

Finally, you play a different sound effect based on whether or not the explosion is considered "large". A hit that merely damages an asteroid will be "small", but if an asteroid is completely annihilated the explosion will be "large".

Next switch to **MyScene.h** and add a declaration for the method so that you can call it from other classes:

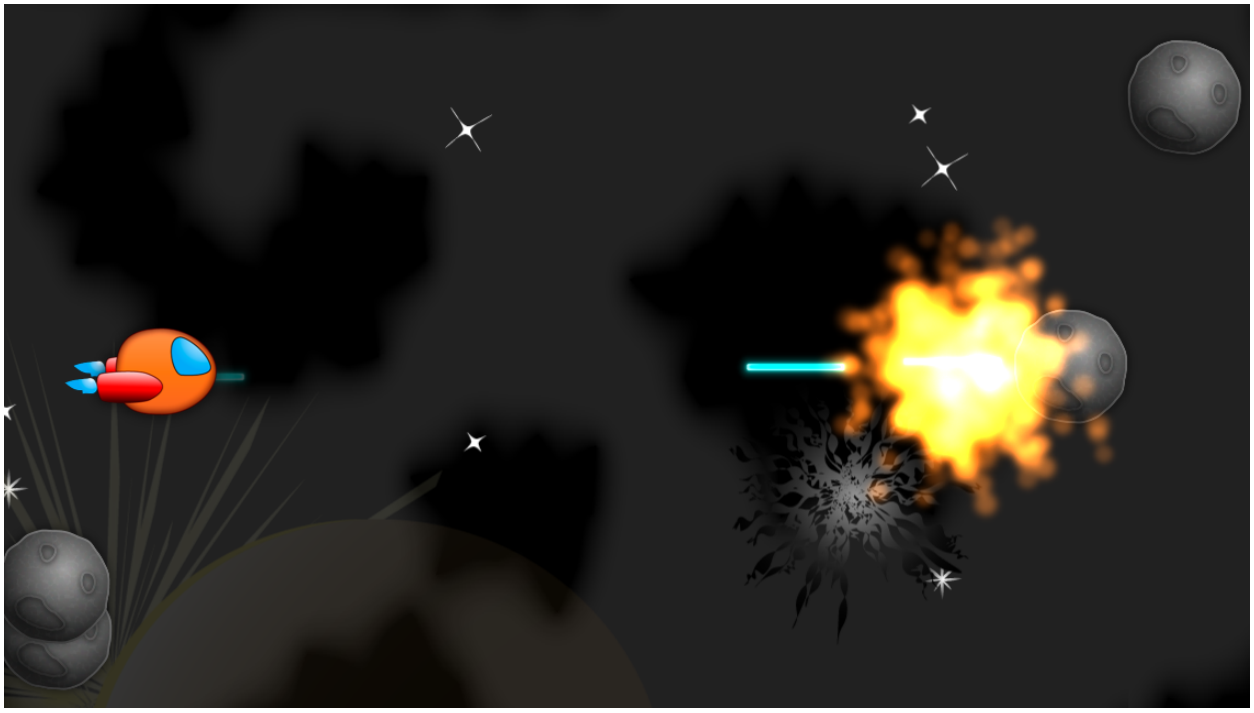
```
- (void)spawnExplosionAtPosition:(CGPoint)position scale:(float)scale
  large:(BOOL)large;
```

Finally, switch to **Asteroid.m** and, inside `collidedWith:contact:`, delete the line that calls `playExplosionLargeSound`. In its place, put these lines of code:

```
if ([self isDead]) {
  [scene spawnExplosionAtPosition:contact.contactPoint scale:self.xScale large:YES];
} else {
  [scene spawnExplosionAtPosition:contact.contactPoint scale:self.xScale large:NO];
}
```

Note that the position you pass in is the `contactPoint` on the physics contact. This makes sure that the explosion spawns right where the asteroid and the laser collide, for a more realistic effect.

That's it! Build and run your code, and you bask in the warm glow of engineered destruction.



Shaking the Screen

The explosions are impressive and all, but you could make them even better. What if the screen were to shake when the player destroys an asteroid?

Add this helper method to **MyScene.m**, right after `playExplosionLargeSound`:

```
- (void)shakeScreen:(int)oscillations {
  SKAction *action =
    [SKAction skt_screenShakeWithNode:_gameLayer amount:CGPointMake(0, 10.0)
                         oscillations:oscillations duration:0.1*oscillations];
  [_gameLayer runAction:action];
}
```

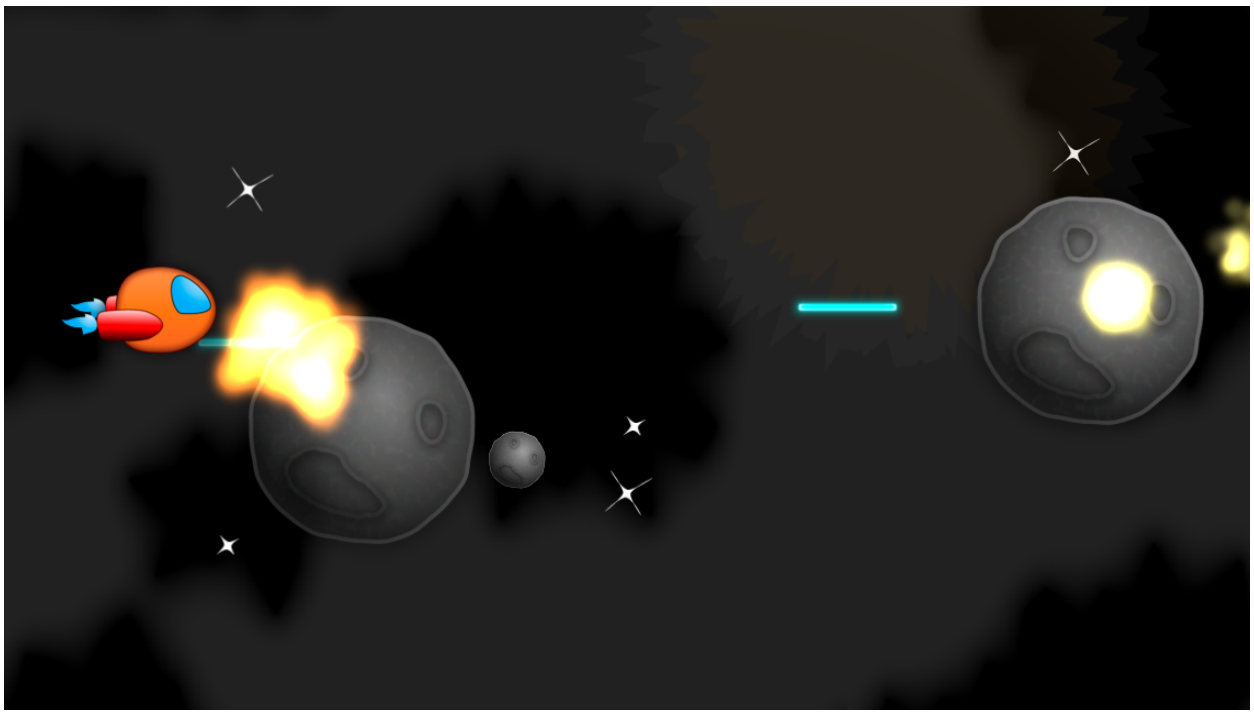
You could implement a screen shake effect with a sequence of `moveBy:duration:` actions—move up a bit, move down a bit and repeat—but here you're using a helper method from `SKTUtils` that creates an even better-looking screen shake via some subtle timing effects. You may remember this as one of the tricks from Chapters 17 and 18 of *iOS Games by Tutorials*, "Juice Up Your Game, Parts 1 and 2". To learn more, check out those chapters or take a peek at the code.

Next call this method from `spawnExplosionAtPosition:scale:large:`, right after the line that plays the `_soundExplosionLarge`:

```
  [self shakeScreen:10*scale];
```

Note that the bigger the asteroid, the more oscillations in the shake.

Build and run, and now you can destroy asteroids in style.



Taking Damage

So far, your ship has led an easy life. It's cruised through space blowing up asteroids without facing any threat whatsoever!

Well, things are about to get dangerous for your spaceship, because you're channeling your dark side and are going to begin detecting collisions for the ship. If the ship loses all its hit points, the player could even—*gasp*—lose the game!

Start by implementing a method that will display "Game Over" to the user. To do this, open **MyScene.m** and add this private instance variable:

```
BOOL _okToRestart;
```

This variable will let you know when it's OK for the user to tap the screen to restart the game. You don't want the user to be able to restart the game the second the "Game Over" text appears, because the user will likely be tapping like mad trying to shoot and might not see the text before the game restarts! Instead, you'll add a small delay, and use the above variable to keep track of when it's safe.

Next, add this new method to display the "Game Over" text right below `spawnPlayer`:

```
- (void)endScene:(BOOL)win {

  if (_levelManager.gameState == GameStateGameOver) return;
```

```
  _levelManager.gameState = GameStateGameOver;

  NSString *fontName = @"Avenir-Light";

  NSString *message;
  if (win) {
    message = @"You win!";
  } else {
    message = @"You lose!";
  }

  // Message Label
  SKLabelNode *messageLabel = [SKLabelNode labelNodeWithFontNamed:fontName];
  [messageLabel setScale:0];
  messageLabel.text = message;
  messageLabel.fontSize = [self fontSizeForDevice:96.0];
  messageLabel.fontColor = [SKColor colorWithRed:0.7 green:0.7 blue:0.7 alpha:1.0];
  messageLabel.position = CGPointMake(self.size.width/2, self.size.height * 0.6);
  messageLabel.verticalAlignmentMode = SKLabelVerticalAlignmentModeCenter;
  [_hudLayer addChild:messageLabel];

  SKAction *scaleAction = [SKAction scaleTo:1 duration:0.5];
  scaleAction.timingMode = SKActionTimingEaseOut;
  [messageLabel runAction:scaleAction];

  // Restart Label
  SKLabelNode *restartLabel = [SKLabelNode labelNodeWithFontNamed:fontName];
  [restartLabel setScale:0];
  restartLabel.text = @"Tap to Restart";
  restartLabel.fontSize = [self fontSizeForDevice:32.0];
  restartLabel.fontColor = [SKColor colorWithRed:0.7 green:0.7 blue:0.7 alpha:1.0];
  restartLabel.position = CGPointMake(self.size.width/2, self.size.height * 0.3);
  restartLabel.verticalAlignmentMode = SKLabelVerticalAlignmentModeCenter;
  [_hudLayer addChild:restartLabel];

  SKAction *waitAction = [SKAction waitForDuration:1.5];
  SKAction *scaleUpAction = [SKAction scaleTo:1.1 duration:0.5];
  scaleUpAction.timingMode = SKActionTimingEaseInEaseOut;
  SKAction *scaleDownAction = [SKAction scaleTo:0.9 duration:0.5];
  scaleDownAction.timingMode = SKActionTimingEaseInEaseOut;
  SKAction *okToRestartAction = [SKAction runBlock:^{
    _okToRestart = YES;
  }];
  SKAction *throbAction = [SKAction repeatActionForever:
    [SKAction sequence:@[scaleUpAction, scaleDownAction]]];
  SKAction *displayAndThrob = [SKAction sequence:
    @[waitAction, scaleAction, okToRestartAction, throbAction]];
  [restartLabel runAction:displayAndThrob];
}
```

This is a long method, but it should be an easy review. The first thing it does is change the game state to `GameStateGameOver`. This stops much of the code in `update` from running, like moving the player or spawning asteroids, since those behaviors check that the game state is `GameStatePlay`.

Next, it displays a label on the screen that says "You win!" or "You lose!", depending on the Boolean it receives, and then uses a few actions to make the label zoom in and throb on the screen.

After the label zooms onto the screen, the method sets `_okToRestart` to `YES`. Note that this part of the code is wrapped in an `SKAction` block; this way, you can add it to the sequence of events.

Open **MyScene.h** and declare this method so other classes can call it:

```
- (void)endScene:(BOOL)win;
```

Next, implement the collision callbacks on the player object so that the player gets hurt after colliding with something. Open **Player.m** and import the header for `MyScene` at the top of the file to let you call methods on the scene:

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

Then implement the collision callbacks as follows:

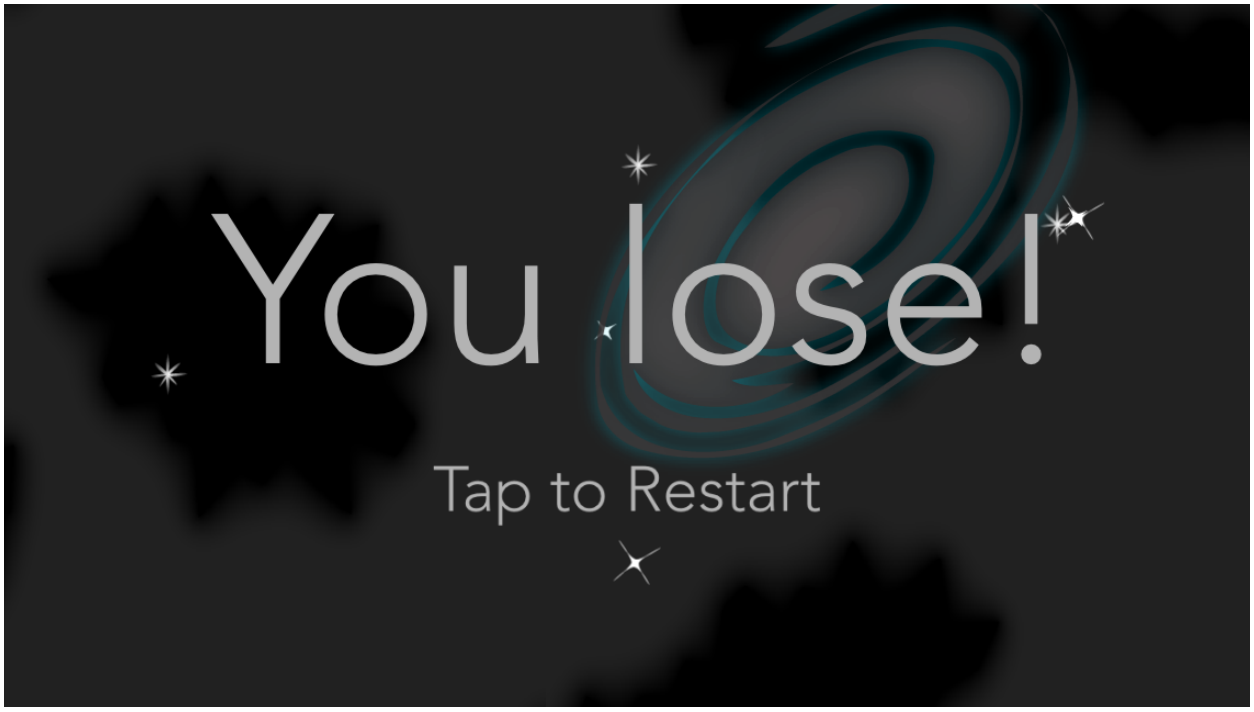
```
- (void)collidedWith:(SKPhysicsBody *)body contact:(SKPhysicsContact *)contact {
  // 1
  Entity * other = (Entity *)body.node;
  [other destroy];

  // 2
  if (body.categoryBitMask & EntityCategoryAsteroid ||
      body.categoryBitMask & EntityCategoryAlienLaser ||
      body.categoryBitMask & EntityCategoryAlien) {
    [self contactWithObstacle:contact];
  }
}

- (void)contactWithObstacle:(SKPhysicsContact *)contact {
  MyScene *scene = (MyScene *)self.scene;

  // 3
  [self takeHit];
  if ([self isDead]) {
    [scene spawnExplosionAtPosition:contact.contactPoint scale:1.0 large:YES];
    [scene endScene:NO];
  } else {
    [scene spawnExplosionAtPosition:contact.contactPoint scale:0.5 large:YES];
  }
```

```
}
```

Let's go over this section by section:

1. You immediately destroy any object that collides with the player.
2. Then you checks to see what has collided with the player. If it's an asteroid, alien laser or alien, you call a secondary method. You do this because later you'll want to do something different if the player collides with a power-up!
3. This secondary method is very similar to the code you put in the asteroid class. The ship takes a hit and spawns the appropriate type of explosion. If the ship is destroyed, you also call the `endScene:` method that you just wrote to end the game.

Build and run your code, lose the game on purpose and feel the power of the dark side!



When you tap to restart, though, nothing happens. That's because you haven't added any code to handle a tap when the game is in `GameStateGameOver`.

To fix this, switch back to **MyScene.m** and add the following code to the bottom of `touchesBegan:withEvent:`:

```
if (_levelManager.gameState == GameStateGameOver && _okToRestart) {
  MyScene * myScene = [MyScene sceneWithSize:self.size];
  SKTransition * reveal = [SKTransition flipHorizontalWithDuration:0.5];
  [self.view presentScene:myScene transition:reveal];
  return;
}
```

When the user taps the screen in this state and it's OK to restart, this code simply creates a new instance of `MyScene` and transitions to it, effectively "resetting" the game.

Build and run your code, and now your game should be better behaved!

Winning the Game

As Yoda would say, although the quick rewards of the dark side are seductive, true Xcode Jedis use their powers for good.

So it's time to give your spaceship a fighting chance to win the game! As a first step, let's define "winning the game" as the spaceship surviving for 30 seconds.

This is really easy. Add the following private instance variables to the top of **MyScene.m**:

```
NSTimeInterval _timeSinceGameStarted;
NSTimeInterval _timeForGameWon;
```

The first variable will track how much time has elapsed since the game began and the second how much time needs to pass for the player to win the game.

Next, initialize these variables at the end of `startSpawn`:

```
_timeSinceGameStarted = 0;
_timeForGameWon = 30;
```

Finally, add this code to the bottom of `update:`:

```
_timeSinceGameStarted += _deltaTime;
if (_timeSinceGameStarted > _timeForGameWon) {
  [self endScene:YES];
}
```

This keeps the time since the game began up to date. If the elapsed time is greater than the time needed to win, you call `endScene`, with the `win` flag set to `YES`.

Build and run the game, survive for 30 seconds—and may the accelerometer force be with you!

Where to Go From Here?

No more do I have to teach you in the swamp of DagoSpriteKit—at least in this chapter. ☺ At this point, you have a simple but complete space game where your player navigates through a dangerous asteroid belt with hit points, win/lose logic and awesome explosions!

Just like last time, you may stop here and use what you've built so far as a starting point for your own game. Here are a few features you might want to add:

- Begin the game with the asteroids spawning and moving slowly, and as time goes by make them spawn and move faster and faster. The challenge could be, "How long can your spaceship survive?"
- Every once in a while, spawn an über "boss" asteroid that is huge (maybe as big as the entire screen!) and has tons of hit points.
- Add different types of objects that fly toward the starship—instead of asteroids, what about angry cows or space birds?

Or you can continue reading the next chapter, where you'll learn how to add multiple levels, power-ups and alien swarms!

Chapter 3: Multiple Levels and Aliens!

Let's take it to the next level!

Right now, you have just one "level" in your game—your spaceship blasting through the asteroid belt for 30 seconds.

However, for a truly complete space game, you want multiple levels. Even better, you want to be able to define the levels by editing a simple text file. This way, you can easily change the levels without having to modify code—and you can even give the file to someone else so they can make the levels for you!

So in this chapter, you're going to modify the game so that it supports multiple levels, defined in an external property list file. After the asteroid belt, the second level will be a swarm of aliens, and the final level will end with a big boss fight. Each level will have multiple stages, and for each stage you'll decide whether you want to spawn asteroids, aliens, text—or all three!

You'll pick up where you left off in the last chapter. If you skipped the last chapter, you can continue on by using the **SpaceGame2** project that comes with this starter kit.

It's time to take your game to the next level—and beyond.

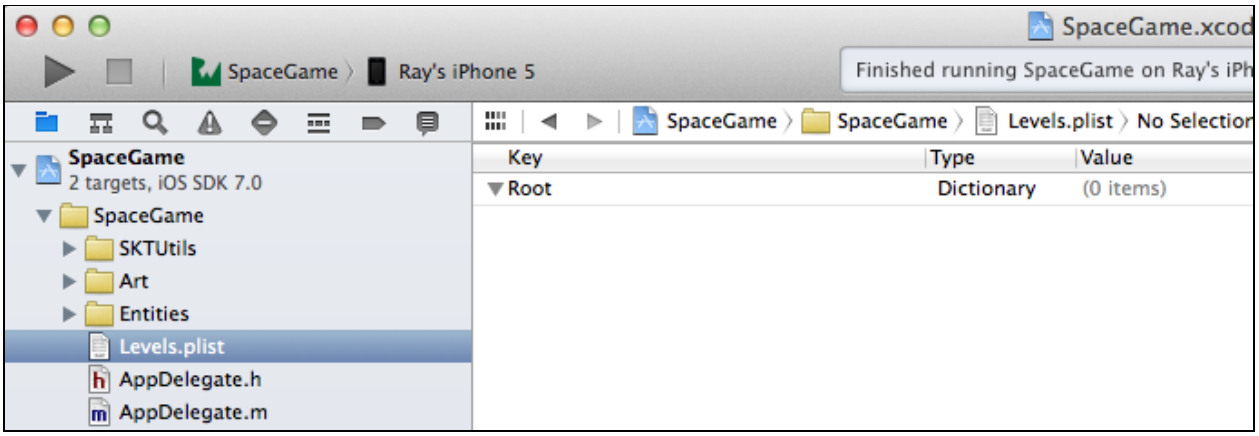
Creating a Property List for the Levels

Before you write any code, let's create the first version of the property list file you'll use to define the levels.

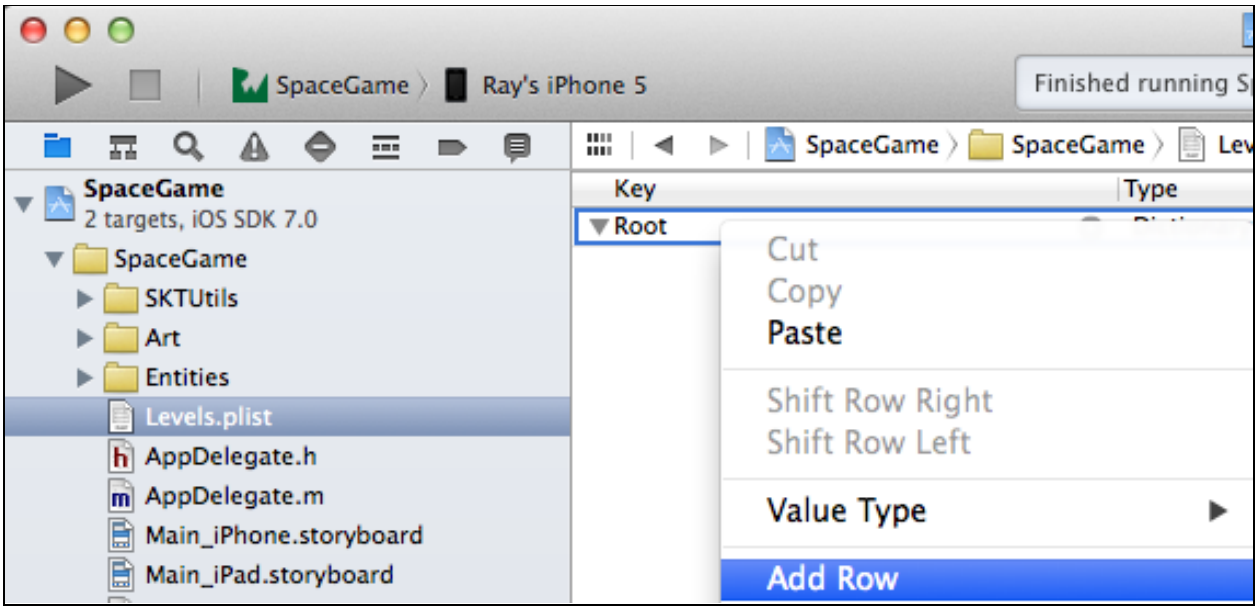
A property list is an XML file with a particular format that allows you to easily store common Objective-C types such as dictionaries, arrays, strings and numbers. Xcode has a built-in editor to work with these files, and iOS includes support to read them from your code, so it's very easy to work with this format.

Let's create the file. In Xcode, go to **File\New\File**, choose **iOS\Resource\Property List** and click **Next**. Name the file **Levels.plist** and click **Create**.

Select **Levels.plist** and the property list will appear, with a single dictionary element called "Root":



Control-click on the root key and select **Add Row**, as shown below:

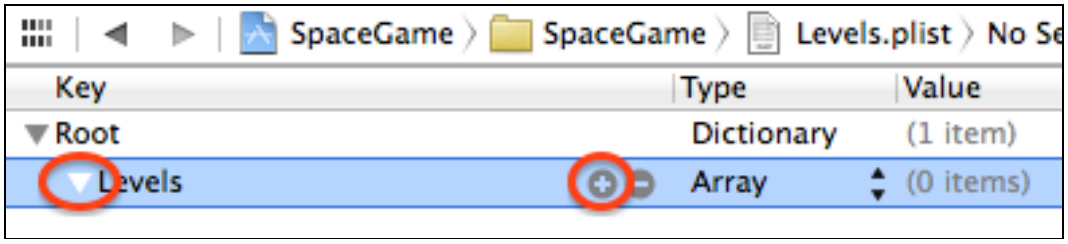


The new row that appears has three columns:

- **Key**: The root entry in the property list is a dictionary, so the **Key** is the string you'll use in code to pull out this particular row's data.
- **Type**: The type of data that this row contains. The ones you'll use most often are Array, Dictionary, Number and String.
- **Value**: This is the value of the entry. It really only applies to value types such as Number or String. You can simply double-click this to edit it and type in your own value.

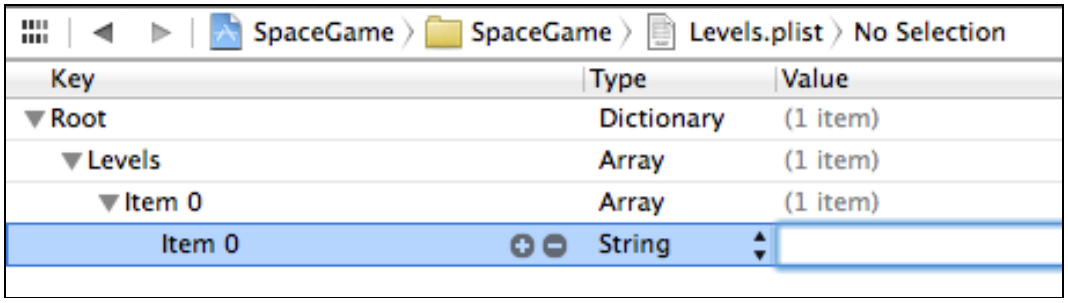
You want this first row to be a list of all the levels, so double-click the **Key** column and change it to read **Levels**. Then click the **Type** column and change it to **Array**.

To add a row for the first level, first click the arrow next to the **Levels** key so that the arrow points down and then click the plus (+) button next to the **Levels** key to create a new sub-item:

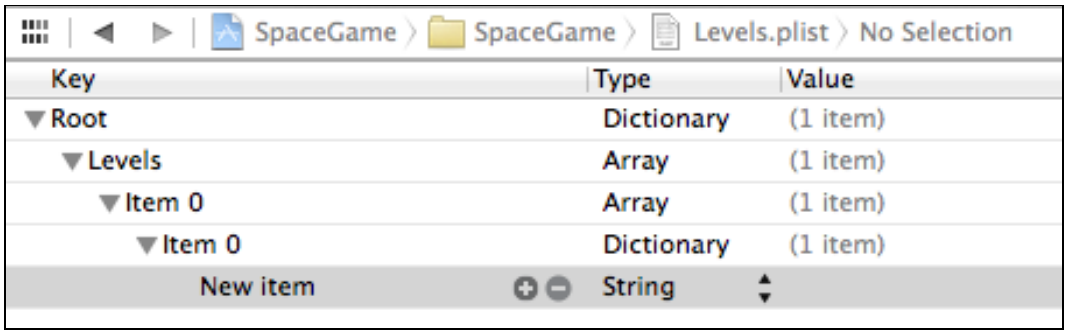


For this sub-item, notice that you can't change the **Key**. That's because Key only applies if the item is contained within a dictionary, while this sub-item is contained within an array.

Your going to break down your level into a series of stages, so change the **Type** of this entry to **Array** also. Then follow the same steps as you did earlier to create a new sub-item for the new array.



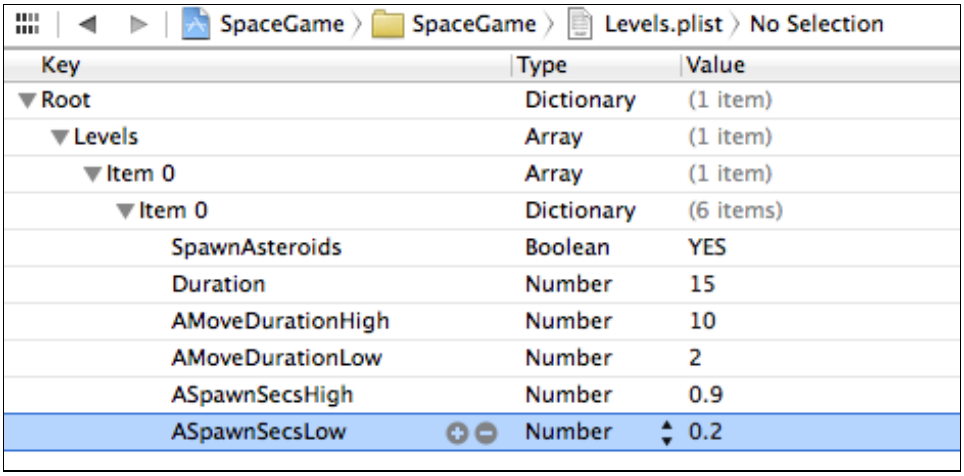
This new sub-item represents a single stage. You want to add a bunch of different properties to the stage, so change the **Key** to **Dictionary** and then add a new sub-item.



Notice that you can change the **Key** of this new sub-item because this item is a child of a dictionary. For the first entry, set the **Key** to **SpawnAsteroids**, the **Type** to **Boolean** and the **Value** to **YES**. This is the value you're going to use to tell the game whether or not to spawn asteroids during this stage.

Now repeat this process to set up several more properties for this stage, as listed and shown below. These remaining properties are all Number types.

- **Duration: 15**
- **AMoveDurationHigh: 10**
- **AMoveDurationLow: 2**
- **ASpawnSecsHigh: 0.9**
- **ASpawnSecsLow: 0.2**



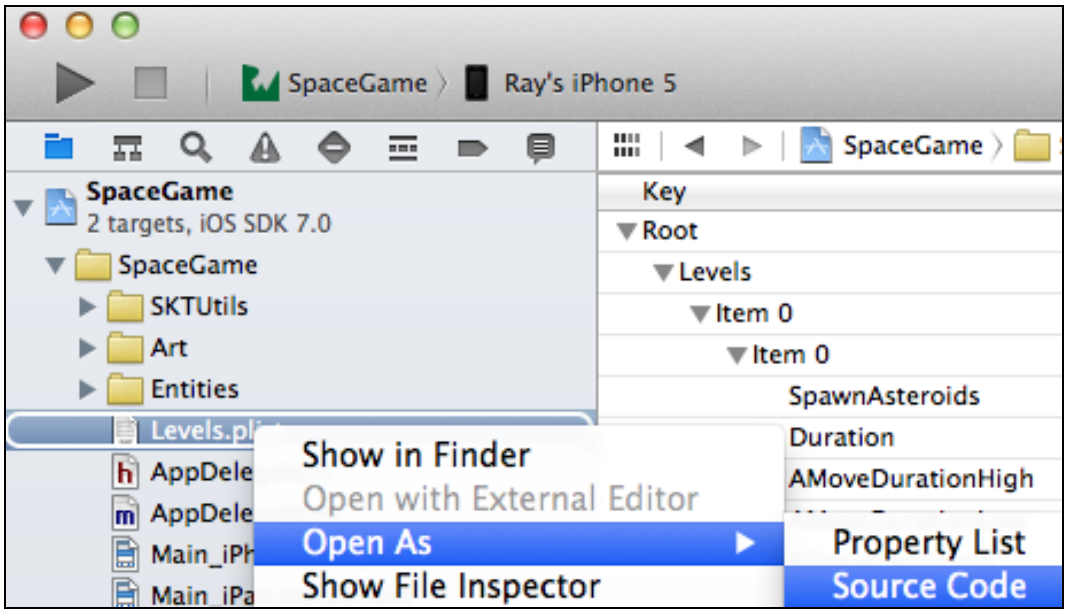
Note that the order of the items in the dictionary does not matter—your order can be different from what's pictured and it will work just the same.

You're going to use this data in your game as follows:

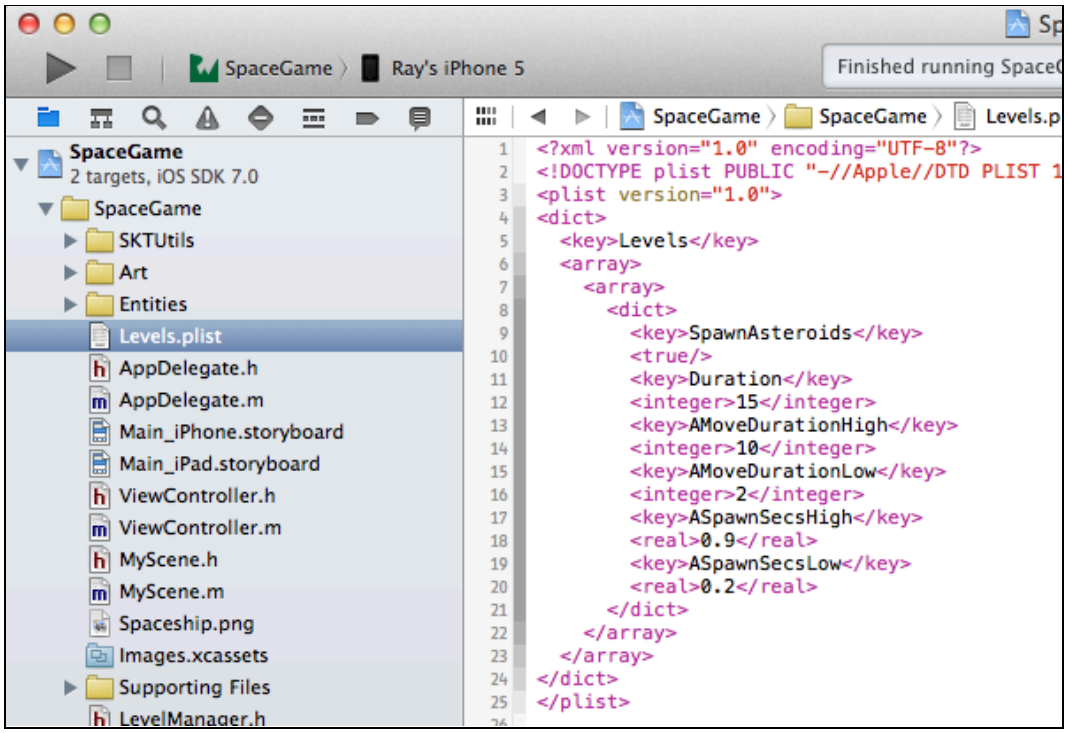
- **Duration** specifies how long this stage should last, in seconds.
- **AMoveDurationHigh** and **AMoveDurationLow** specify the range of time an asteroid should take to move across the screen in this stage.
- **ASpawnSecsHigh** and **ASpawnSecsLow** specify the range of time between asteroids spawning.

These keys and values are based on what you need for this game. When you're making your own game, you can add anything else you need!

Let's learn a bit more about how this property list works. Control-click **Levels.plist** and select **Open As\Source Code**:



You should see **Levels.plist** in XML format, as shown below:



As you can see, your property list file is human-readable and makes a lot of sense—there's an array with an inner array, with a dictionary inside that, and then a bunch of keys and values inside the dictionary.

I'm showing you this because sometimes it's more convenient to edit the file in a text editor like this rather than in the property list editor. The property list editor is great for small tweaks, but sometimes it's a lot faster to use a text editor to copy/paste or perform a bulk find and replace.

As this chapter continues, you're going to make a bunch of changes to **Levels.plist**. Rather than listing each and every change and having you continuously edit the file, I've made some pre-edited versions of **Levels.plist** that you can use for each stage of development by overwriting your version of the file with the updated version.

You can find this first version under **Levels\V1** in case you need it.

Now that you have a file with data about your levels, let's modify your game to use it!

Adding Multi-Level Support

Rather than clutter up your `MyScene` with a bunch of extra code to read this file, you're going to add it to the `LevelManager` class you created earlier. Right now this class only contains the `gameState`, but you're going to modify it to read this level file, keep track of the current stage and advance the stage at the proper time.

Open **LevelManager.m** and add the following import:

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

You'll need to use some Core Graphics functions declared in that header file later.

Next, replace the `@implementation` line near the top of the file with the following block to add some needed instance variables:

```
@implementation LevelManager {
  double _stageStart;
  double _stageDuration;

  NSDictionary *_data;
  NSArray *_levels;
  NSInteger _curLevelIdx;
  NSArray *_curStages;
  NSInteger _curStageIdx;
  NSDictionary *_curStage;
}
```

This declares some private instance variables to keep track of the time the current stage began and how long it should run in seconds.

The rest of the instance variables are references to the data pulled from **Levels.plist**. They will, one by one, store the overall dictionary of data from the file, the list of levels and the current level index, the list of stages within the level and the current stage index, and finally the dictionary of data for the current stage.

Next, replace the `init` method with this implementation:

```
- (id)init {
  if ((self = [super init])) {
    NSString *levelDefsFile = [[NSBundle mainBundle]
      pathForResource:@"Levels" ofType:@"plist"];
    _data = [NSDictionary dictionaryWithContentsOfFile:levelDefsFile];
    NSAssert(_data != nil, @"Couldn't open Levels file");

    _levels = (NSArray *) _data[@"Levels"];
    NSAssert(_levels != nil, @"Couldn't find Levels entry");

    _curLevelIdx = -1;
    _curStageIdx = -1;
    _gameState = GameStateMainMenu;
  }
  return self;
}
```

This gets the path for **Levels.plist** inside the main bundle. It then uses a helper method called `dictionaryWithContentsOfFile` to create a new `NSDictionary` based on the contents of the property list.

Then it gets the entry in the root dictionary named **Levels**, which is the first and only entry you made in the dictionary—that array you created with all of the level data.

Finally, add this set of helper methods:

```
- (NSInteger)curLevelIdx {
  return _curLevelIdx;
}

- (BOOL)hasProp:(NSString *)prop {
  NSString * retval =  (NSString *) _curStage[prop];
  return retval != nil;
}

- (NSString *)stringForProp:(NSString *)prop {
  NSString * retval =  (NSString *) _curStage[prop];
  NSAssert(retval != nil, @"Couldn't find prop %@", prop);
  return retval;
}

- (float)floatForProp:(NSString *)prop {
  NSNumber * retval = (NSNumber *) _curStage[prop];
  NSAssert(retval != nil, @"Couldn't find prop %@", prop);
  return retval.floatValue;
}

- (BOOL)boolForProp:(NSString *)prop {
```

```
  NSNumber * retval =  (NSNumber *) _curStage[prop];
  return [retval boolValue];
}

- (void)nextLevel {
  _curLevelIdx++;
  if (_curLevelIdx >= _levels.count) {
    _gameState = GameStateDone;
    return;
  }
  _curStages = (NSArray *) _levels[_curLevelIdx];
  [self nextStage];
}

- (void)nextStage {
  _curStageIdx++;
  if (_curStageIdx >= _curStages.count) {
    _curStageIdx = -1;
    [self nextLevel];
    return;
  }

  _gameState = GameStatePlay;
  _curStage = _curStages[_curStageIdx];

  _stageDuration = [self floatForProp:@"Duration"];
  _stageStart = CACurrentMediaTime();

  NSLog(@"Stage ending in: %f", _stageDuration);
}

- (BOOL)update {
  if (_gameState != GameStatePlay) return NO;
  if (_stageDuration == -1) return NO;

  double curTime = CACurrentMediaTime();
  if (curTime > _stageStart + _stageDuration) {
    [self nextStage];
    return YES;
  }

  return NO;
}
```

There's a lot of code here, but luckily most of it is quite simple.

`hasProp`, `stringForProp`, `floatForProp` and `boolForProp` are helper methods that look inside the `_curStage` dictionary, which you'll see initialized later, for particular

properties. Some of these assert if the keys don't exist, which shouldn't happen unless you make a mistake setting up the plist file.

`nextLevel` and `nextStage` contain the smarts to advance through the levels contained in the plist. `nextLevel` tries to get the next entry in the levels array, but if there are no more, it sets the game state to done.

`nextStage` tries to get the next entry in the stages array, but if there are no more it advances to the next level. You can see that `nextStage` also sets up the `_curStage` dictionary. Finally, it looks for a special key that must exist for each stage—the **Duration**, which specifies how long the stage should last in seconds—and squirrels that away, along with the current time as `_stageStart`.

`MyScene` will call `update` each frame, and its job is to check if it's time to advance to the next stage and to call `nextStage` if so. Note that `update` returns a `BOOL` that indicates if the game has advanced to a new stage. You'll need this in `MyScene`, because sometimes you'll want to perform special actions when the game first enters a new stage.

Finally, open **LevelManager.h** and add these declarations for the new methods:

```
- (NSInteger)curLevelIdx;
- (void)nextStage;
- (void)nextLevel;
- (BOOL)update;
- (float)floatForProp:(NSString *)prop;
- (NSString *)stringForProp:(NSString *)prop;
- (BOOL)boolForProp:(NSString *)prop;
- (BOOL)hasProp:(NSString *)prop;
```

These are the helper methods that you'll be calling from `MyScene`.

OK—the hard part is done, and now you just need to make use of it! Open up **MyScene.m** and comment out the old `_timeSinceGameStarted` and `_timeForGameWon` variables, since you won't be using them anymore:

```
// NSTimeInterval _timeSinceGameStarted;
// NSTimeInterval _timeForGameWon;
```

Next, add these two new methods right before `startSpawn`:

```
- (void)nextStage {
  [_levelManager nextStage];
  [self newStageStarted];
}

- (void)newStageStarted {
  if (_levelManager.gameState == GameStateDone) {
    [self endScene:YES];
  }
```

```
}
```

You call `nextStage` to advance the game to the next stage. It simply calls the method you already wrote in `LevelManager` to do this and then calls `newStageStarted`.

`newStageStarted` simply checks to see if the state has advanced to `GameStateDone`. If so, it ends the scene. Later, you'll add some more logic here.

Next, go to `startSpawn`, comment out the lines that include `_timeSinceGameStarted` and `_timeForGameWon` and add a call to `nextStage` in its place:

```
//_timeSinceGameStarted = 0;
//_timeForGameWon = 30;
[self nextStage];
```

Remember, calling `nextStage` will in turn call `nextStage` from `LevelManager`. The level manager has already read the property list and is keeping track of the game's current level and stage. When you call `nextStage`, it advances the stage and/or the level appropriately and keeps track of how long the game should stay in the new stage. You might want to take a peek at the `nextStage` method in `LevelManager` to refresh your memory of what's going on here.

Now you just need to call `update` from the level manager every frame so that it can detect when it is time to advance to the next stage, based on the settings in **Levels.plist**. To do so, add this new method after `updateBg`:

```
- (void)updateLevel {
  BOOL newStage = [_levelManager update];
  if (newStage) {
    [self newStageStarted];
  }
}
```

You will call this method each frame. It, in turn, calls update from the level manager, which returns a Boolean that indicates whether the level manager has moved to the next stage. If it has, you call the `newStageStarted` method you wrote earlier. This way, you can check for conditions like "Has the game ended?"

Finally, at the end of `update:`, comment out the old code that uses the `_timeSinceGameStarted` and `_timeForGameWon` variables and replace it with a call to `updateLevel`:

```
//_timeSinceGameStarted += _deltaTime;
//if (_timeSinceGameStarted > _timeForGameWon) {
//  [self endScene:YES];
//}
[self updateLevel];
```

So far, so good—you've updated the game to use the level manager to advance through the stages rather than hard-coding a certain time into the game. However, you still need to update the game to read the asteroid spawn settings from the level file.

To do this, find `updateAsteroids` and replace the first two lines with the following:

```
if (![_levelManager boolForProp:@"SpawnAsteroids"]) return;

float spawnSecsLow = [_levelManager floatForProp:@"ASpawnSecsLow"];
float spawnSecsHigh = [_levelManager floatForProp:@"ASpawnSecsHigh"];
```

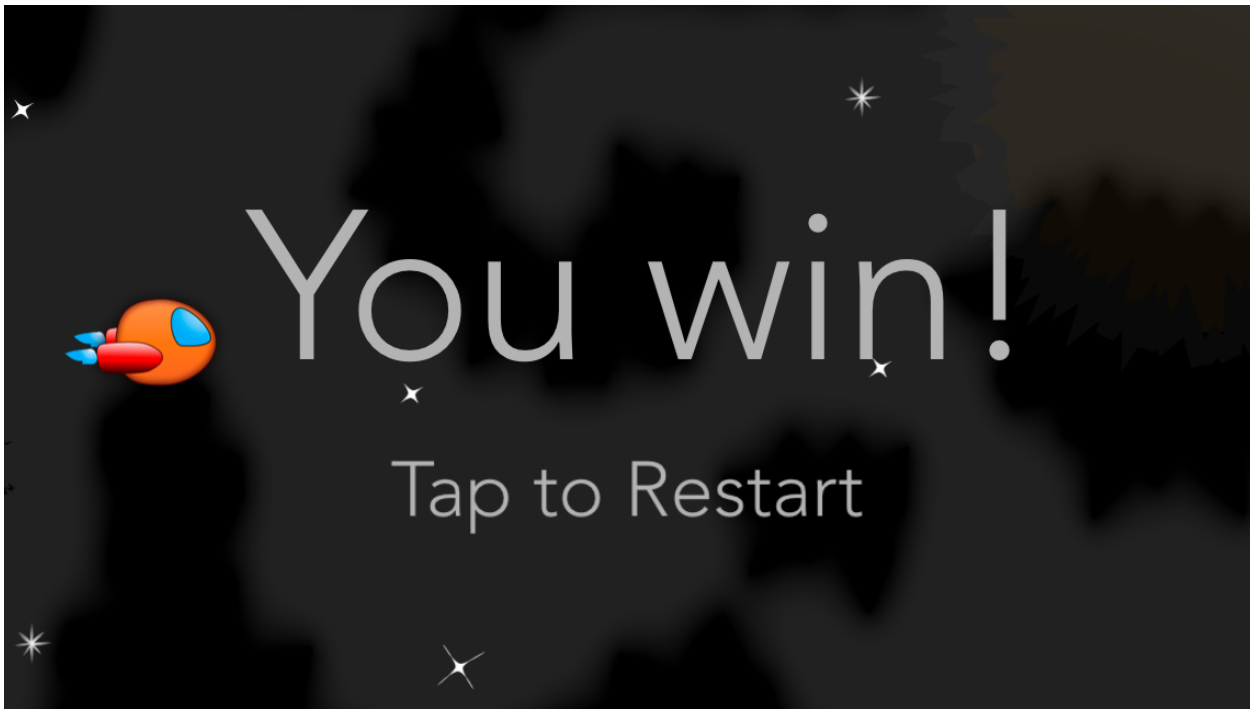
You don't want to spawn asteroids if the current stage doesn't have the **SpawnAsteroids** property. You also want to use the asteroid spawn settings that come from the current stage in Levels.plist rather than use hard-coded values.

Next, replace the first two lines of `spawnAsteroid` with the following:

```
float moveDurationLow = [_levelManager floatForProp:@"AMoveDurationLow"];
float moveDurationHigh = [_levelManager floatForProp:@"AMoveDurationHigh"];
```

Now instead of hardcoding these values, they come from whatever you put into Levels.plist!

w00t—you're finally done! Build and run your game, and you should see it work normally, except now it should take only 15 seconds to win because that's what you defined in Levels.plist.



You might not quite appreciate the beauty of this yet, since right now everything looks the same.

Let me show you why what you've done is über awesome. I created a new version of **Levels.plist** with some additional stages that you can find under **Levels\V2**. Copy this into your project directory, overwriting your current copy of **Levels.plist**. Also, go to **Product\Clean** so that the compiler uses the new version of Levels.plist rather than the old one, and do this whenever replacing Levels.plist in the future.

Then run your game again—and you'll have three stages of asteroids, with varying spawn rates and speeds:

- In the first wave, asteroids spawn normally.
- In the second wave, asteroids move especially quickly.
- In the third wave, asteroids move slowly but spawn quickly.

Build and run your code, and see if you can make it through!

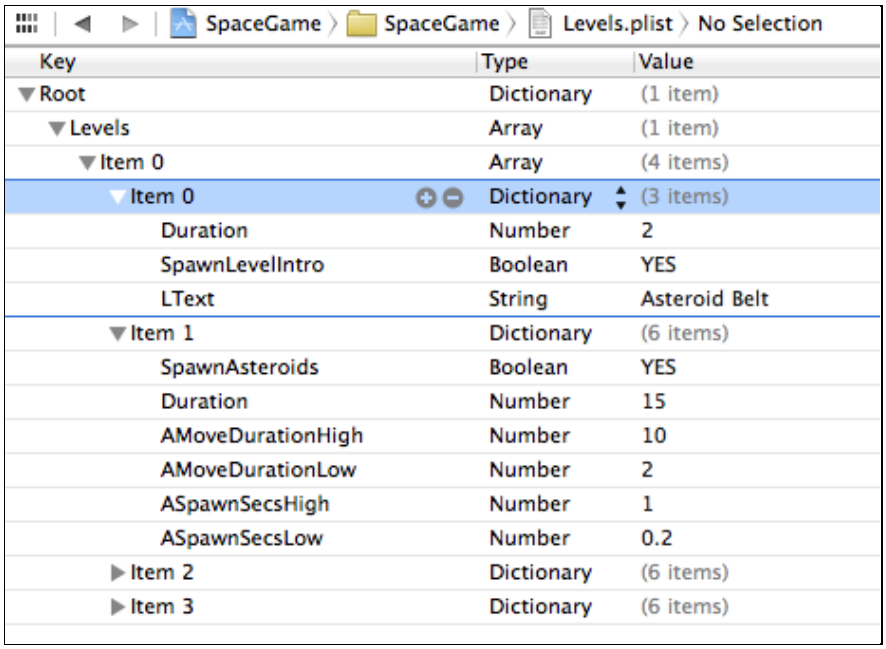
Think about how cool this is—by modifying a single file, you've completely changed the behavior of your game without having to touch a line of code!

Adding Level Intro Text

Since your game is going to have multiple levels, it would be nice if you displayed a level title on the screen as each one begins.

Now that you have a level management system in place, this will be a breeze!

Start by replacing your **Levels.plist** with the version in **Levels\V3**—and don't forget to **Product\Clean**. If you open this new version with the property list editor, you'll see that it contains a new level stage:



This has the key **SpawnLevelIntro**. You will look for this key to see if you should spawn some text to the screen for this stage. It also contains the text to display in **LText** and sets the Duration of the stage to two seconds.

Now watch how easy it is to use this! Begin by adding two instance variables for the labels you'll use in **MyScene.m**:

```
SKLabelNode *_levelIntroLabel1;
SKLabelNode *_levelIntroLabel2;
```

Then add this new method to spawn the text right after `endScene:`:

```
- (void)doLevelIntro {
  NSString *fontName = @"Avenir-Light";

  NSString *message1 = [NSString stringWithFormat:@"Level %d",
    (int)_levelManager.curLevelIdx+1];
  NSString *message2 = [_levelManager stringForProp:@"LText"];

  // Level Intro Label 1
  _levelIntroLabel1 = [SKLabelNode labelNodeWithFontNamed:fontName];
  [_levelIntroLabel1 setScale:0];
  _levelIntroLabel1.text = message1;
  _levelIntroLabel1.fontSize = [self fontSizeForDevice:48.0];
  _levelIntroLabel1.fontColor =
    [SKColor colorWithRed:0.7 green:0.7 blue:0.7 alpha:1.0];
  _levelIntroLabel1.position = CGPointMake(self.size.width/2, self.size.height * 0.6);
  _levelIntroLabel1.verticalAlignmentMode = SKLabelVerticalAlignmentModeCenter;
  [_hudLayer addChild:_levelIntroLabel1];
```

```
  SKAction *scaleUpAction1 = [SKAction scaleTo:1 duration:0.5];
  scaleUpAction1.timingMode = SKActionTimingEaseOut;
  SKAction *delayAction1 = [SKAction waitForDuration:3.0];
  SKAction *scaleDownAction1 = [SKAction scaleTo:0 duration:0.5];
  scaleDownAction1.timingMode = SKActionTimingEaseOut;
  SKAction *removeAction = [SKAction removeFromParent];
  SKAction *scaleUpDown = [SKAction sequence:
    @[scaleUpAction1, delayAction1, scaleDownAction1, removeAction]];
  [_levelIntroLabel1 runAction:scaleUpDown];

  // Level Intro Label 2
  _levelIntroLabel2 = [SKLabelNode labelNodeWithFontNamed:fontName];
  [_levelIntroLabel2 setScale:0];
  _levelIntroLabel2.text = message2;
  _levelIntroLabel2.fontSize = [self fontSizeForDevice:48.0];
  _levelIntroLabel2.fontColor =
    [SKColor colorWithRed:0.7 green:0.7 blue:0.7 alpha:1.0];
  _levelIntroLabel2.position = CGPointMake(self.size.width/2, self.size.height * 0.4);
  _levelIntroLabel2.verticalAlignmentMode = SKLabelVerticalAlignmentModeCenter;
  [_hudLayer addChild:_levelIntroLabel2];

  [_levelIntroLabel2 runAction:scaleUpDown];
}
```

This creates two strings to display—one with the level number retrieved from `LevelManager` and one with the value of the `LText` property for the current stage.

It then creates two labels and uses several actions to make them zoom into the screen, wait a few seconds, zoom back out and then remove themselves from the layer.

Now that this is in place, simply add the following code at the end of `newStageStarted` to call it if `SpawnLevelIntro` is set:

```
else if ([_levelManager boolForProp:@"SpawnLevelIntro"]) {
  [self doLevelIntro];
}
```

Build and run, and now when the level starts you'll see some intro text!

An Alien Swarm

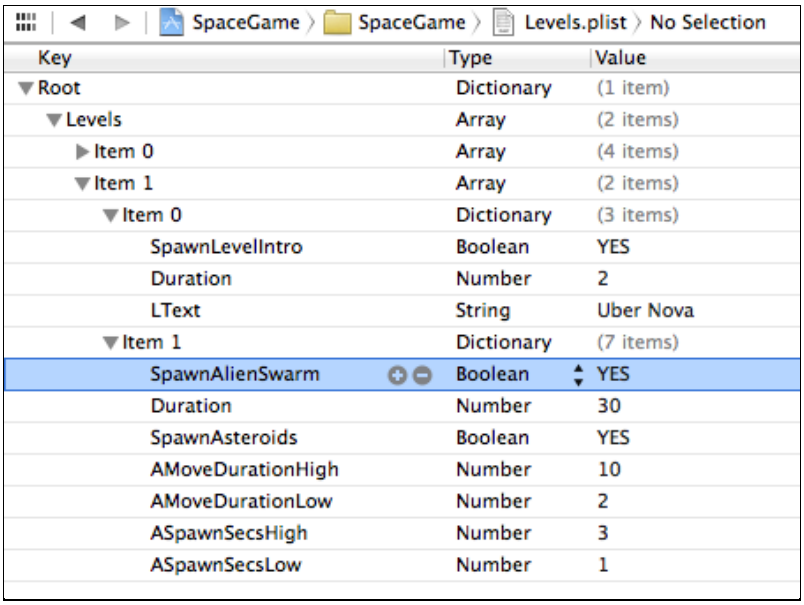
Just when your spaceship has made it through the asteroid belt to safety—here comes an alien swarm!

You're going to make your game able to spawn aliens in your game stages at your command. The aliens will try their best to shoot your spacecraft down with deadly laser beams.

In this section, you're just going to focus on getting the aliens to show up. They'll appear in the second level of the game and will be trickier to evade and destroy than some rocky asteroids, because they'll move in a curved path.

Let's start by once again replacing **Levels.plist** with a new version that has the definitions you need for this level. You'll find this in **Levels\V4**—and don't forget to **Product\Clean**.

Open this new version and you'll see an array for a new level with two stages:



The first stage displays the text for Level 2, which you're calling Uber Nova. Why? Because I like the way it sounds! ☺

The second stage has a new key called **SpawnAlienSwarm**, which you'll be using to tell if you should spawn aliens. Notice that the stage is set to spawn asteroids occasionally, too—it's pretty cool how you can combine these however you want, right?

You may also notice that I've shortened the asteroid stages to five seconds each so that you can quickly get to the aliens for testing.

OK, so let's add some aliens! First things first—you need to create an alien class. To do this, create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **Alien** for Class, **Entity** for Subclass of, click **Next** and then click **Create**.

You don't need to make any changes to the header, but you do need to make some changes to the implementation, so open **Alien.m** and replace the contents with the following:

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

@implementation Alien

- (instancetype)init {
  if ((self = [super initWithImageNamed:@"enemy_spaceship" maxHp:1])) {
    [self setupCollisionBody];
  }
  return self;
}
```

```
- (void)setupCollisionBody {
  CGPoint offset = CGPointMake(
    self.size.width * self.anchorPoint.x, self.size.height * self.anchorPoint.y);
  CGMutablePathRef path = CGPathCreateMutable();
  [self moveToPoint:CGPointMake(11, 25) path:path offset:offset];
  [self addLineToPoint:CGPointMake(42, 40) path:path offset:offset];
  [self addLineToPoint:CGPointMake(86, 40) path:path offset:offset];
  [self addLineToPoint:CGPointMake(114, 25) path:path offset:offset];
  [self addLineToPoint:CGPointMake(79, 11) path:path offset:offset];
  [self addLineToPoint:CGPointMake(44, 11) path:path offset:offset];
  CGPathCloseSubpath(path);

  self.physicsBody = [SKPhysicsBody bodyWithPolygonFromPath:path];
  [self attachDebugFrameFromPath:path color:[SKColor redColor]];

  self.physicsBody.categoryBitMask = EntityCategoryAlien;
  self.physicsBody.collisionBitMask = 0;
  self.physicsBody.contactTestBitMask = EntityCategoryPlayerLaser;
}

- (void)collidedWith:(SKPhysicsBody *)body contact:(SKPhysicsContact *)contact {

  Entity * other = (Entity *)body.node;
  if (body.categoryBitMask & EntityCategoryPlayerLaser) {
    [other destroy];
    [self takeHit];
    MyScene *scene = (MyScene *)self.scene;
    if ([self isDead]) {
      [scene spawnExplosionAtPosition:contact.contactPoint
        scale:self.xScale large:YES];
    } else {
      [scene spawnExplosionAtPosition:contact.contactPoint
        scale:self.xScale large:NO];
    }
  }

}

@end
```

This should be review by this point. It specifies the image to use for the sprite and creates a collision body based on the values I got when I traced the sprite. Then it implements the collision callback, making the alien ship take a hit when it's shot and spawning the appropriate explosion.

Switch to **MyScene.m** and import this new class:

```
#import "Alien.h"
```

Then add the following new instance variables:

```
NSInteger _numAlienSpawns;
NSTimeInterval _timeSinceLastAlienSpawn;
NSTimeInterval _timeForNextAlienSpawn;
UIBezierPath *_alienPath;
SKShapeNode *_dd1;
SKShapeNode *_dd2;
SKShapeNode *_dd3;
```

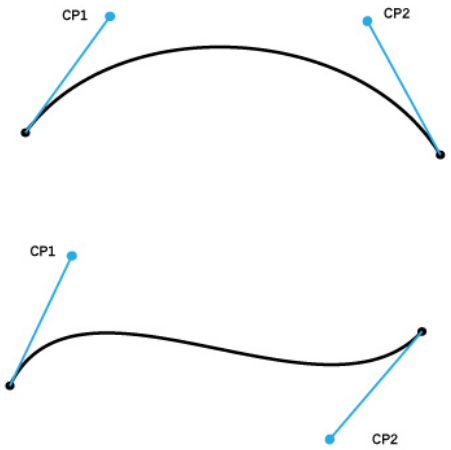
The first variable is to help you keep track of how many aliens are left to spawn in the current wave. You'll be generating a random number of aliens per wave later on.

The next two variables keep track of the times between waves of aliens. You'll be setting these to random values later on as well.

The next variable will store the path on which the aliens should move. `UIBezierPath` is a handy class that makes it easy to define different types of paths, including lines, circles and elipses. In this game, you are going to create a curved path for the aliens called a **bezier path**.

Huh? What in the heck is a bezier path?

A bezier path is a curved line that you can create by defining a start and end point (black dots) and two control points (CP1 and CP2 in blue), as you can see below:



Think of the control points as "pulling" the curve in their direction. The best way to figure out how to set them is via experimentation and debug drawing, which you'll learn how to do as you continue!

That's why you have the last three instance variables—every time you add a path, you will debug draw the path and its control points so that you can visualize what's going on.

Of course, for the paths to show up, you have to turn on debug drawing again! Open **SKTUtils\SKNode+DebugDraw.m** and turn debug drawing back on:

```
static BOOL kDebugDraw = YES;
```

Now for the fun part—the code to spawn the aliens! Add the following new method to **MyScene.m**, right after `spawnPlayerLaser`:

```
#pragma mark - Aliens

- (void)spawnAlien {
  // 1
  if (_numAlienSpawns == 0) {
    // 2
    CGPoint alienPosStart = CGPointMake(
      self.size.width * 1.3,
      RandomFloatRange(self.size.height*0.9, self.size.height * 1.0));
    // 3
    CGPoint cp1 = CGPointMake(
      RandomFloatRange(-self.size.width * 0.1, self.size.width * 0.6),
      RandomFloatRange(self.size.height * 0.7, self.size.height * 1.0));
    // 4
    CGPoint alienPosEnd = CGPointMake(
      self.size.width * 1.3,
      RandomFloatRange(0, self.size.height * 0.1));
    // 5
    CGPoint cp2 = CGPointMake(
      RandomFloatRange(-self.size.width * 0.1, self.size.width * 0.6),
      RandomFloatRange(0, self.size.height * 0.3));

    // 6
    _alienPath = [[UIBezierPath alloc] init];
    [_alienPath moveToPoint:alienPosStart];
    [_alienPath addCurveToPoint:alienPosEnd controlPoint1:cp1 controlPoint2:cp2];

    // 7
    _numAlienSpawns = RandomFloatRange(1, 20);
    _timeForNextAlienSpawn = 1.0;

    // 8
    [_dd1 removeFromParent];
    [_dd2 removeFromParent];
    [_dd3 removeFromParent];

    _dd1 = [self attachDebugFrameFromPath:_alienPath.CGPath color:
```

```
      [SKColor greenColor]];
    _dd2 = [self attachDebugFrameFromPoint:alienPosStart toPoint:cp1 color:
      [SKColor blueColor]];
    _dd3 = [self attachDebugFrameFromPoint:alienPosEnd toPoint:cp2 color:
      [SKColor blueColor]];
  } else {
    // 9
    _numAlienSpawns -= 1;

    // 10
    Alien *alien = [[Alien alloc] init];
    alien.name = @"alien";
    SKAction *pathAction = [SKAction followPath:_alienPath.CGPath asOffset:NO
      orientToPath:NO duration:3.0];
    SKAction *removeAction = [SKAction removeFromParent];
    [alien runAction:[SKAction sequence:@[pathAction, removeAction]]];
    [_gameLayer addChild:alien];
  }
}
```

There's a lot of code here and several new ideas, so let's go over this section by section:

1. `_numAlienSpawns` keeps track of the number of aliens to spawn in the current wave of aliens. If this is 0, that means that it's time to configure the settings for the next wave of aliens.
2. `alienPosStart`: Create a point offscreen (x-axis) in the top 90% the screen (y-axis).
3. `cp1`: Create a point on the left side of the screen (x-axis) in the top 70% of the screen (y-axis).
4. `alienPosEnd`: Create a point offscreen (x-axis) in the bottom 10% of the screen (y-axis).
5. `cp2`: Create a point on the left side of the screen (x-axis) in the bottom 30% of the screen (y-axis).
6. Use these points to construct a bezier curve using `moveToPoint` and `addCurveToPoint` from `UIBezierPath`. If you set the start to `alienPosStart`, the end to `alienPosEnd` and the two control points to `cp1` and `cp2`, the aliens will spawn offscreen to the top right, curve toward the middle of the screen and go back out to the right. If you reverse `alienPosStart`/`alienPosEnd` and the control points, they'll go bottom to top instead.
7. Chose a random number of aliens to spawn and a random amount of time before the next alien spawn.
8. Add debug drawings for the path itself and for the lines from the start/end to their control points, so the bezier path is easy to visualize and debug. You store

the drawings in variables so you can remove the old paths every time a new set of aliens is spawned.

9. In the case where `_numAlienSpawns` is not zero, decrease the value by one.

10. Create an alien and use `followPath:` to make the alien move along the path that you created earlier.

Next, add this new method right after `spawnAlien`:

```
- (void)updateAliens {
  if (![_levelManager boolForProp:@"SpawnAlienSwarm"]) return;

  _timeSinceLastAlienSpawn += _deltaTime;
  if (_timeSinceLastAlienSpawn > _timeForNextAlienSpawn) {
    _timeSinceLastAlienSpawn = 0;
    _timeForNextAlienSpawn = 0.3;
    [self spawnAlien];
  }
}
```

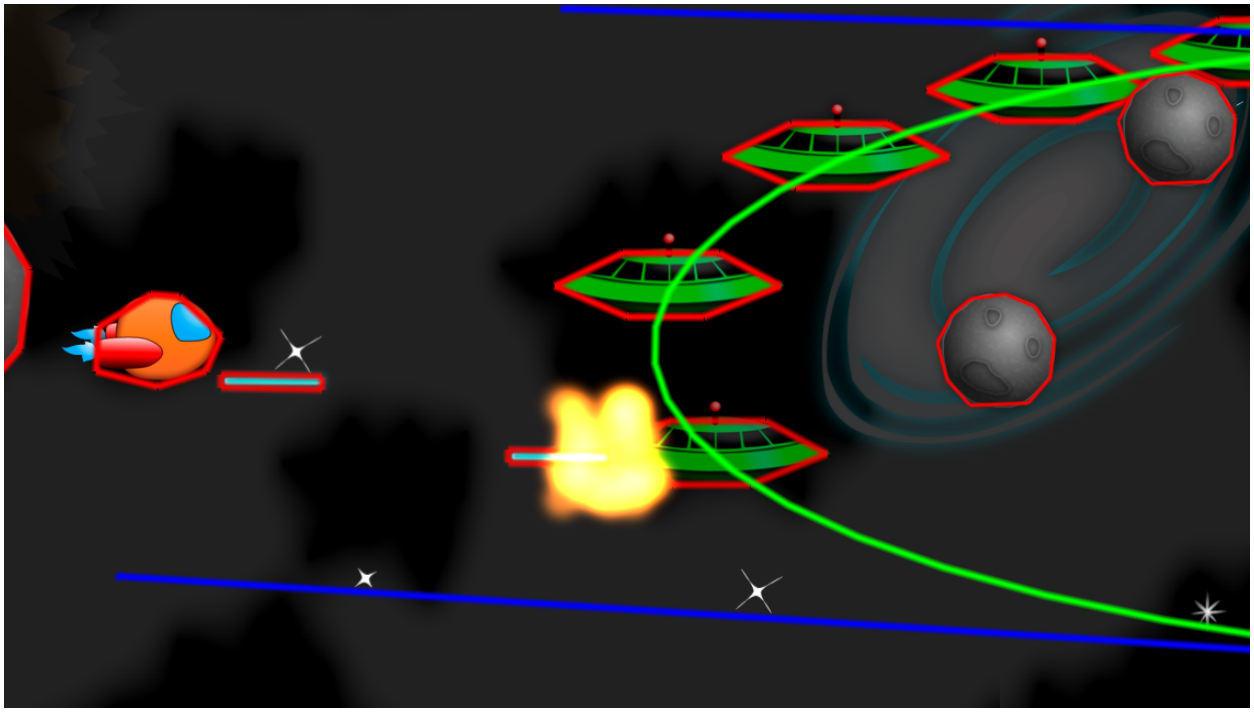
This method returns immediately if "SpawnAlienSwarm" is not set to YES on the current stage in Levels.plist.

It then checks to see if it's time to spawn the next alien and if so, calls `spawnAlien`.

Finally, call this method at the bottom of `update:`:

```
[self updateAliens];
```

Build and run your code, and now you can blast some aliens!



Remember, the green line here is the debug drawing for the path the ships are moving along, and the blue lines are drawn to the control points.

Aliens Shooting Lasers

It's definitely fun to see the aliens fly in, but right now there's no challenge. The way you've set up the Bezier curves, the aliens will never hit your spaceship, so there's no way the aliens can hurt the player.

That will all change in this section—because the aliens are going to open fire!

Rather than put the shooting logic inside `MyScene`, you're going to have each ship keep track of when it wants to fire. This keeps your code a bit cleaner.

But to do this, you need to give each `Entity` time to run each update. So open **Entity.m** and add a stub implementation that the alien class will override:

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

Switch to **Entity.h** and declare this method:

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

Open **MyScene.m** and add this method right after `updateLevel`:

```
- (void)updateChildren {
```

```
  [_gameLayer.children enumerateObjectsUsingBlock:^(id obj, NSUInteger idx, BOOL
*stop) {
    if ([obj isKindOfClass:[Entity class]]) {
      Entity *entity = (Entity *)obj;
      [entity update:_deltaTime];
    }
  }];
}
```

This enumerates through all of the children of the game layer. If any of them are an `Entity` class or subclass, it calls the update method on it. Right now it's just an empty stub method, but later you'll be overriding this in the alien class to periodically shoot lasers.

Also call this method at the end of `update:`:

```
 [self updateChildren];
```

It's almost time to shoot lasers, but first you need a laser to shoot! Create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **AlienLaser** for Class, **Entity** for Subclass of, click **Next** and then click **Create**.

You don't need to make any changes to the header, but you do need to make some changes to the implementation, so open **AlienLaser.m** and replace the contents with the following:

```
#import "AlienLaser.h"

@implementation AlienLaser

- (instancetype)init {
  if ((self = [super initWithImageNamed:@"laserbeam_red" maxHp:1])) {
    [self setupCollisionBody];
  }
  return self;
}

- (void)setupCollisionBody {
  CGPoint offset = CGPointMake(
    self.size.width * self.anchorPoint.x, self.size.height * self.anchorPoint.y);
  CGMutablePathRef path = CGPathCreateMutable();
  [self moveToPoint:CGPointMake(4, 8) path:path offset:offset];
  [self addLineToPoint:CGPointMake(24, 8) path:path offset:offset];
  [self addLineToPoint:CGPointMake(24, 3) path:path offset:offset];
  [self addLineToPoint:CGPointMake(4, 3) path:path offset:offset];
  CGPathCloseSubpath(path);

  self.physicsBody = [SKPhysicsBody bodyWithPolygonFromPath:path];
```

```
  [self attachDebugFrameFromPath:path color:[SKColor redColor]];

  self.physicsBody.categoryBitMask = EntityCategoryAlienLaser;
  self.physicsBody.collisionBitMask = 0;
  self.physicsBody.contactTestBitMask = EntityCategoryPlayer;
}

@end
```

Again, you should be a pro at this by now—you simply create the physics body for the laser and initialize its other properties.

Next switch to **MyScene.m** and import the new class:

```
#import "AlienLaser.h"
```

Then add this helper method to spawn a laser right after `spawnPlayerLaser`:

```
- (void)spawnAlienLaserAtPosition:(CGPoint)position {
  AlienLaser * laser = [[AlienLaser alloc] init];
  laser.position = position;
  [_gameLayer addChild:laser];

  SKAction *moveAction = [SKAction moveBy:CGVectorMake(-self.size.width, 0)
    duration:2.0];
  SKAction *removeAction = [SKAction removeFromParent];
  [laser runAction:[SKAction sequence:@[moveAction, removeAction]]];

  [self runAction:_soundLaserEnemy];
}
```

This creates a laser at the specified position and makes it move to the left at a set speed.

You will be calling this method periodically from the alien class, so declare this method in **MyScene.h**:

```
- (void)spawnAlienLaserAtPosition:(CGPoint)position;
```

Finally, let's get those lasers spawning! Open **Alien.m** and add these two private instance variables:

```
@implementation Alien {
  NSTimeInterval _timeSinceLastLaserShot;
  NSTimeInterval _timeForNextLaserShot;
}
```

These will keep track of the time elapsed since the last laser shot and how much time should pass before the next laser should be shot.

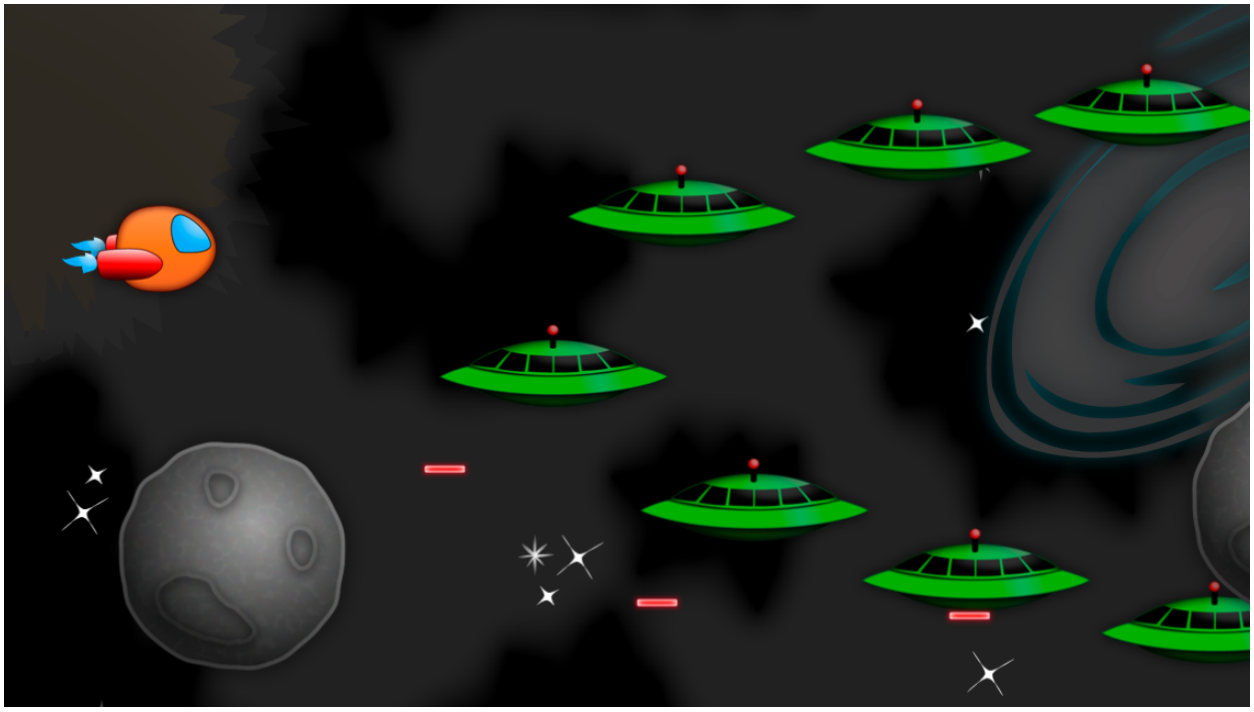
Add these line to `init`, right after the call to `setupCollisionBody`, to initialize these values:

```
_timeForNextLaserShot = RandomFloatRange(0.1, 4);
_timeSinceLastLaserShot = 0;
```

As the final step, override `update:` to shoot lasers periodically:

```
- (void)update:(CFTimeInterval)deltaTime {
  [super update:deltaTime];

  _timeSinceLastLaserShot += deltaTime;
  if (_timeSinceLastLaserShot > _timeForNextLaserShot) {
    _timeSinceLastLaserShot = 0;
    _timeForNextLaserShot = RandomFloatRange(0.1, 4);
    MyScene *myScene = (MyScene *)self.scene;
    [myScene spawnAlienLaserAtPosition:self.position];
  }
}
```

Build and run, and now you have a challenging second level with an alien swarm firing lasers! You may also want to turn off debug drawing in **SKNode+DebugDraw.m** since you don't need it anymore.



Adding a Power-Up

You can't have a space game without adding some kind of cool power-up your ship can collect!

In this section, you'll just focus on adding the power-up itself—in the next section, you'll make it do something.

First of all, you need to update your **Levels.plist** to the version in **Levels\V5**—and don't forget to **Product\Clean**. This version contains a new **SpawnPowerups** Boolean, which you set to true in Level 1, Stage 2 and Level 2, Stage 1. There's also a variable called **PSpawnSecs**, which will determine the time between spawning power-ups.

Next, you need a class for the power-up. Create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **Powerup** for Class, **Entity** for Subclass of, click **Next** and then click **Create**.

Open **Powerup.m** and replace the contents with the following:

```
#import "Powerup.h"

@implementation Powerup

- (instancetype)init {
  if ((self = [super initWithImageNamed:@"powerup" maxHp:1])) {
    [self setupCollisionBody];
  }
  return self;
}

- (void)setupCollisionBody {
  CGPoint offset = CGPointMake(self.size.width * self.anchorPoint.x,
    self.size.height * self.anchorPoint.y);
  CGMutablePathRef path = CGPathCreateMutable();
  [self moveToPoint:CGPointMake(9, 30) path:path offset:offset];
  [self addLineToPoint:CGPointMake(54, 29) path:path offset:offset];
  [self addLineToPoint:CGPointMake(54, 8) path:path offset:offset];
  [self addLineToPoint:CGPointMake(8, 9) path:path offset:offset];
  CGPathCloseSubpath(path);

  self.physicsBody = [SKPhysicsBody bodyWithPolygonFromPath:path];
  [self attachDebugFrameFromPath:path color:[SKColor redColor]];

  self.physicsBody.categoryBitMask = EntityCategoryPowerup;
  self.physicsBody.collisionBitMask = 0;
  self.physicsBody.contactTestBitMask = EntityCategoryPlayer;
}
```

```
@end
```

As usual, this simply creates the physics shape for collision detection on the power-up.

Now open **MyScene.m** and import your new header:

```
#import "Powerup.h"
```

Add these two instance variables:

```
  NSTimeInterval _timeSinceLastPowerup;
  NSTimeInterval _timeForNextPowerup;
```

And add this new method after `updateAliens` to spawn a new power-up:

```
#pragma mark - Powerups

- (void)spawnPowerup {
  Powerup *powerup = [[Powerup alloc] init];
  powerup.position = CGPointMake(self.size.width,
    RandomFloatRange(0, self.size.height));
  SKAction *moveAction = [SKAction moveByX:-self.size.width*1.5 y:0 duration:5.0];
  SKAction *removeAction = [SKAction removeFromParent];
  [powerup runAction:[SKAction sequence:@[moveAction, removeAction]]];
  [_gameLayer addChild:powerup];
}
```

This creates a new power-up object to the right side of the screen and makes it move to the left and then remove itself from the scene.

You'll want to call this periodically whenever the "SpawnPowerups" flag is set to YES in the current stage, so add this new method right after `spawnPowerup` to do so:

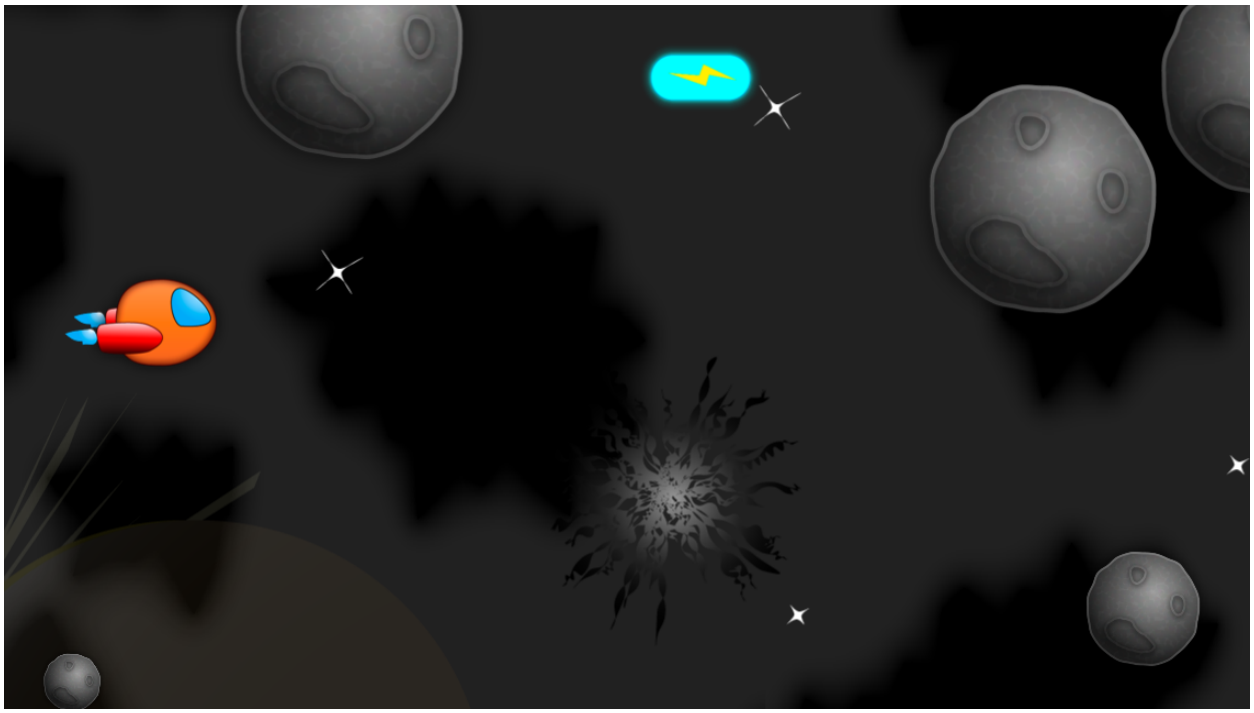
```
- (void)updatePowerups {
  if (![_levelManager boolForProp:@"SpawnPowerups"]) return;

  _timeSinceLastPowerup += _deltaTime;
  if (_timeSinceLastPowerup > _timeForNextPowerup) {
    _timeSinceLastPowerup = 0;
    _timeForNextPowerup = [_levelManager floatForProp:@"PSpawnSecs"];
    [self spawnPowerup];
  }
}
```

You should be able to follow along with this, as it follows the same pattern as the rest of the update routines.

Don't forget to add the line to call this at the end of `update:`:

```
[self updatePowerups];
```

Build and run, and you'll have power-ups appear in the game:



But they don't do anything yet! For now, let's just make the power-ups play a sound effect when the spaceship collides with one. To do this, add this new method in **MyScene.m**, right after `updatePowerups`:

```
- (void)applyPowerup {
  [self runAction:_soundPowerup];
}
```

Then open **MyScene.h** and declare this method so that you can call it from another class:

```
- (void)applyPowerup;
```

Finally, you want to call this from the player class when it collides with a power-up. Open **Player.m** and add this to the bottom of `collidedWith:contact::`

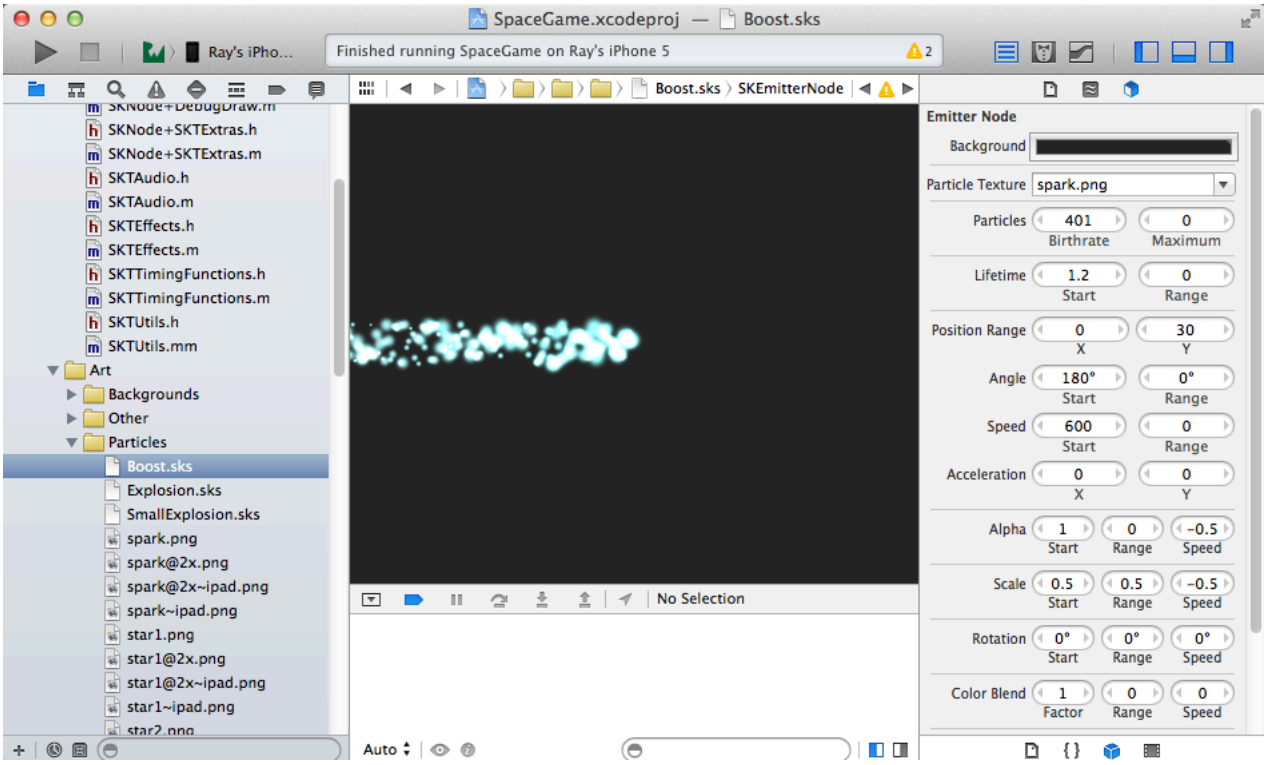
```
else if (body.categoryBitMask & EntityCategoryPowerup) {
  MyScene *scene = (MyScene *)self.scene;
  [scene applyPowerup];
}
```

Build and run your code. After a time, a power-up will appear. Try to collect it and, if all works well, a power-up sound effect will play!

Full Thrusters Ahead!

For the final part of this chapter, you're going to make the power-up live up to its name. When the player gathers a power-up, you'll make the ship zoom ahead with powerful thrusters and smash through asteroids and aliens with full invincibility!

You're going to use a kick-ass special effect for this that you can find under **Art\Particles\Boost.sks**. Optionally, you can open it with the built-in particle editor and tweak it if you'd like:



But the effect is pretty cool as-is, so there's no need to modify it unless you want to!

First, you need to add support to the player class for invincibility. To do this, open **Player.h** and add a new property for invincibility:

```
@property (assign) BOOL invincible;
```

Then open **Player.m** and, inside `constactWithObstacle:`, find the line that calls `[self takeHit]`. Replace that line with the following:

```
if (!self.invincible) {
  [self takeHit];
}
```

Now whenever the ship is invincible, `takeHit` won't be called—but everything the ship collides with will be destroyed as usual.

That's it for invincibility—now you just need to turn it on and apply the cool boost effect whenever the player scores a power-up. Open **MyScene.m** and add this to the bottom of `applyPowerup`:

```
// 1
SKEmitterNode *emitter = [SKEmitterNode skt_emitterNamed:@"Boost"];
emitter.zPosition = -1;
[_player addChild:emitter];

// 2
float const scaleDuration = 1.0;
float const waitDuration = 5.0;

// 3
_player.invincible = YES;
SKAction *moveForwardAction = [SKAction moveByX:self.size.width * 0.6 y:0
duration:scaleDuration];
SKAction *waitAction = [SKAction waitForDuration:waitDuration];
SKAction *moveBackAction = [moveForwardAction reversedAction];
SKAction *boostDoneAction = [SKAction runBlock:^{
  _player.invincible = NO;
  [emitter removeFromParent];
}];
[_player runAction:[SKAction sequence:
  @[moveForwardAction, waitAction, moveBackAction, boostDoneAction]]];

// 4
float const scale = 0.75;
float const diffX = (_spacedust1.size.width - (_spacedust1.size.width * scale))/2;
float const diffY = (_spacedust1.size.height - (_spacedust1.size.height * scale))/2;
SKAction *moveOutAction = [SKAction moveByX:diffX y:diffY duration:scaleDuration];
SKAction *moveInAction = [moveOutAction reversedAction];
SKAction *scaleOutAction = [SKAction scaleTo:scale duration:scaleDuration];
SKAction *scaleInAction = [SKAction scaleTo:1.0 duration:scaleDuration];
SKAction *outAction = [SKAction group:@[moveOutAction, scaleOutAction]];
SKAction *inAction = [SKAction group:@[moveInAction, scaleInAction]];
[_gameLayer runAction:[SKAction sequence:@[outAction, waitAction, inAction]]];
```

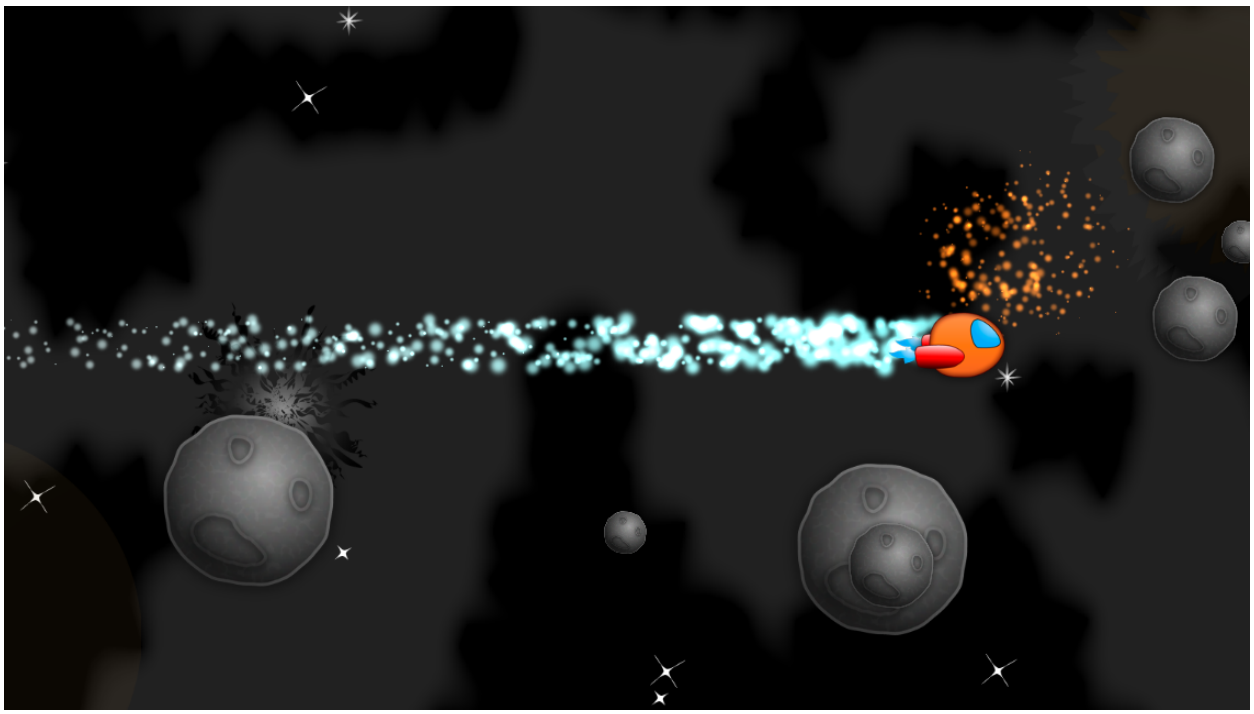
Let's go over this section by section:

1. This creates a new particle system for the boost effect and adds it to the player. Note that you're using the helper method `skt_emitterNamed:` from `SKTUtils`, which is a handy one-liner shortcut to create a particle system.
2. As you'll see shortly, you're going to zoom out from the screen when you apply the boost. This code sets the duration of the zoom out and of the ship's invincibility. Feel free to tweak these to your liking.

3. This sets up a sequence of actions that takes place when the player grabs the power-up. It sets the ship to invincible, moves the ship forward by 60% of the screen width, waits a bit and then reverses the actions.

4. For an extra cool effect, this bit of code zooms the screen out to 75% of its original size. This is why you've been making the stars spawn beyond the height of the screen, and why you've been creating ships offscreen more than just past the width of the screen—so that when you're zoomed out, everything will still look OK. Note the calculation to move the game layer a bit to offset the fact that when you scale a node, it scales from the bottom left of the node, rather than what you might consider the "center" of the node.

You did it! Build and run your code, and grab a power-up so you can smash through waves of enemies!



Where to Go From Here?

If you've made it this far, you're a true code warrior. Think about all you've accomplished—you have a cool space game with multiple levels, multiple types of enemies with different sizes and hit points, power-ups, lasers and explosions. Best of all—you can configure the levels via a simple text file!

As usual, you're welcome to stop at this point and begin extending this to make your own game. Here are some features you might want to add:

- More levels! You could start editing **Levels.plist** to create your own levels with different combinations of asteroids and aliens, ramping up the difficulty higher and higher as the space ship travels through the system!

- Now that you know how easy it is to add different types of things to spawn to **Levels.plist**, follow the examples of adding spawning aliens, text, power-ups and asteroids to add something of your own creation. Maybe you'll want to add another type of alien that moves differently—perhaps in the style of *Space Invaders* rather than in curves?
- Why not add some more power-ups? The first one that comes to mind is a power-up that improves the player's laser in some way, perhaps by making it fire multiple shots successively or three beams at once that spread in different directions?

Or you can continue reading the next chapter, where you'll wrap up the Space Game Starter Kit with a bad-ass boss fight!

Chapter 4: The Final Boss Fight

A Dark Nemesis Awaits…

Your spaceship is having a great adventure so far—it made it through the asteroid belt and battled the alien swarms at Uber Nova.

But meanwhile, in a galaxy not so far away, a dark nemesis has been plotting your spaceship's demise.

Since boss battles are meant to be quite epic, the first thing you're going to do is add a health bar to the game so that you can see your current hit points—and that of the boss you're about to do battle with.

Then, you'll add the big boss to the game. He's a lot more complicated than enemies you've added so far, because he's broken into multiple parts that can move independently, and he can move around in a complicated manner and shoot multiple weapons at a time.

You're going to pick up right where you left off in the last chapter. If you skipped the last chapter, you can continue on with the **SpaceGame3** project that comes with this starter kit.

Get ready for the final showdown—your spaceship is about to meet its match!

Creating a Health Bar

You're going to modify `Entity` so that it can optionally display a health bar representing the number of hit points the object has remaining. It will be really helpful in a long fight to know how badly your ship—and the boss—have been damaged.

Note that there are simpler ways of creating a health bar than what you're going to do here. It could be as simple as drawing a line above the entity to show its health using a shape node.

The method you're going to use results in a better-looking health bar, though, because you can customize it with any artwork you'd like and animate decreasing

or increasing health. Plus, this method will demonstrate a cool API call you might find useful in other ways in the future.

Start by adding a new enum to the top of **Entity.h**:

```
typedef NS_ENUM(NSInteger, HealthBarType) {
  HealthBarTypeNone = 0,
  HealthBarTypeGreen,
  HealthBarTypeRed
};
```

This declares the type of health bar to display for the entity. The default will be none, but if you do want to display a health bar you have two variants—green, which you'll use for the ship, and red, which you'll use for enemies.

Next, add a new property inside the `@interface` to keep track of the health bar type for the entity:

```
@property (nonatomic, assign) HealthBarType healthBarType;
```

Also, modify the initializer declaration to take a new parameter for the health bar type:

```
- (instancetype)initWithImageNamed:(NSString *)name maxHp:(NSInteger)maxHp
  healthBarType:(HealthBarType)healthBarType;
```

Switch to **Entity.m** and replace the initializer with the following (changes highlighted):

```
- (instancetype)initWithImageNamed:(NSString *)name maxHp:(NSInteger)maxHp
  healthBarType:(HealthBarType)healthBarType {

  if ((self = [super initWithImageNamed:name])) {
    _maxHp = maxHp;
    _hp = maxHp;
    _healthBarType = healthBarType;
    [self setupHealthBar];
  }
  return self;

}

- (void)setupHealthBar {
}
```

This simply stores the health bar type and calls `setupHealthBar`, which you will implement soon.

But first, now that you've updated the initializer to take a new parameter, you need to update each of `Entity`'s subclasses to pass in this new parameter. You'll start with the asteroid class.

Open **Asteroid.m** and find the line that calls `initWithImageNamed:maxHp:`. Then replace it with the following:

```
if ((self = [super initWithImageNamed:@"asteroid" maxHp:maxHp
healthBarType:HealthBarTypeRed])) {
```

This sets the health bar to red for the asteroid.

Now repeat this for the rest of `Entity`'s subclasses, according to the following list:

- `Player`: `HealthBarTypeGreen`
- `PlayerLaser`: `HealthBarTypeNone`
- `Alien`: `HealthBarTypeRed`
- `AlienLaser`: `HealthBarTypeNone`
- `Powerup`: `HealthBarTypeNone`

At this point, build your project and make sure it still compiles. Great—you've set your health bar type and now you just need to add some code to display it.

Open **Entity.m** and add a few private instance variables:

```
@implementation Entity {
  SKSpriteNode *_healthBarBg;
  SKSpriteNode *_healthBarProgress;
  CGFloat _fullWidth;
  CGFloat _displayedWidth;
}
```

The health bar is made up of two sprites: a background sprite (`_healthBarBg`) and a colored bar that indicates the health remaining (`_healthBarProgress`).

`_fullWidth` keeps track of the maximum possible width of the health bar and `_displayedWidth` keeps track of the currently displayed portion of the width. If you're confused about why you need both, read on.

It's important to note that at a given moment, the health bar might not actually display the ship's current health. For example, say the ship has 50% health, then gets hit down to 25% health. Rather than make the health bar immediately jump from 50% to 25%, you will gradually decrease the health bar over several frames until it reaches the desired value—48%, 46% and so on until it reaches 25%. This is what `_displayedWidth` tracks, and it will make the movement of the health bar nice and smooth.

Next add this code inside `setupHealthBar`:

```
// 1
if (_healthBarType == HealthBarTypeNone) return;

// 2
_healthBarBg = [SKSpriteNode spriteNodeWithImageNamed:@"healthbar_bg"];
_healthBarBg.position = CGPointMake(0, self.size.height * 0.5);
[self addChild:_healthBarBg];

// 3
NSString * progressSpriteName;
if (_healthBarType == HealthBarTypeGreen) {
  progressSpriteName = @"healthbar_green";
} else {
  progressSpriteName = @"healthbar_red";
}

// 4
_healthBarProgress = [SKSpriteNode spriteNodeWithImageNamed:progressSpriteName];
_healthBarProgress.anchorPoint = CGPointZero;
_healthBarProgress.position = _healthBarBg.position;
_healthBarProgress.position = CGPointMake(
  _healthBarBg.position.x - _healthBarProgress.size.width/2,
  _healthBarBg.position.y - _healthBarProgress.size.height/2);
[self addChild:_healthBarProgress];

// 5
_fullWidth = _healthBarBg.size.width;
_displayedWidth = _fullWidth;
```

Let's go over this section by section:

1. If the health bar type is set to `HealthBarTypeNone`, there's no need to do anything, so you return immediately.
2. Here you create and position the background for the health bar. Remember that a sprite's anchor point is set to the middle of the sprite by default, so this code positions the background centered on the entity sprite along the x-axis, and right above the entity sprite on the y-axis.
3. You determine which color foreground sprite to use for the health bar, based on the health bar's type.
4. As the ship takes damage, you want to set the size of the foreground appropriately—for example, when the ship is at 60% of its health, you want to set the size of the foreground to 60% along the x-axis. So, to make the foreground align with the background properly, this code sets the anchor point to the lower left and sets the position so that it covers the background sprite precisely. That way, when you reduce the size of the foreground sprite along its x-axis, the left side of the foreground still lines up with the left side of the background.

5. Here you keep track of the full width and the displayed width. Right now they are the same.

There's one last step. Replace the `update:` method with the following:

```
- (void)update:(CFTimeInterval)deltaTime {
  // 1
  if (_healthBarType == HealthBarTypeNone) return;

  // 2
  float percentage = (float) _hp / (float) _maxHp;
  percentage = MIN(percentage, 1.0);
  percentage = MAX(percentage, 0);
  float desiredWidth = _fullWidth * percentage;

  // 3
  float POINTS_PER_SEC = 50;
  if (desiredWidth < _displayedWidth) {
    _displayedWidth = MAX(desiredWidth, _displayedWidth - POINTS_PER_SEC * deltaTime);
  } else {
    _displayedWidth = MIN(desiredWidth, _displayedWidth + POINTS_PER_SEC * deltaTime);
  }
  _healthBarProgress.size =
    CGSizeMake(_displayedWidth, _healthBarProgress.size.height);
}
```

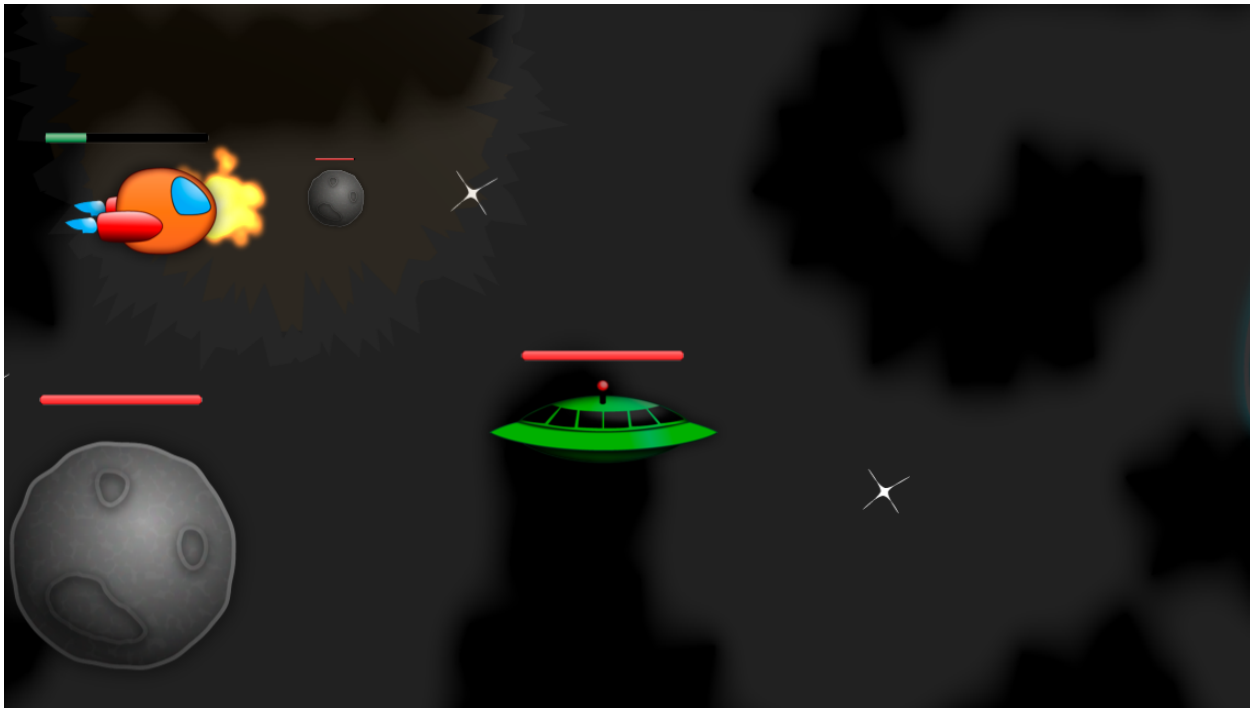
Remember, you are calling `update:` on each entity from `MyScene` each frame. Before, the default implementation did nothing; now it updates the health bar to match the ship's hit points in a neat, animated way. Here's how this works:

1. Again, if the health bar is `HealthBarTypeNone`, there's nothing to do, so you immediately return.

2. You calculate the fraction of hit points left and set the desired width of the health bar based on this.

3. Rather than set the health bar to the desired width right away, here you animate it gradually over time. Each second, the health bar can move 50 points toward its desired width. Finally, you set the size of the sprite to the updated (reduced) width.

That's it! Build and run, and you should now see a health bar above your ship displaying your current health—and smoothly animating as it decreases!



Auto-Fading the Health Bar

I don't know about you, but I find it annoying to have that health bar always hovering around. It ruins the effect of flying free in space!

Instead of erasing all that work you just did, let's modify the health bar so that it appears only when the entity's health changes—and fades away otherwise. That way, players will get a notification about their health when they are hurt, but the rest of the time, they can enjoy the beautiful scenery.

This is pretty easy to accomplish. First, add these lines to the bottom of **Entity.m**'s `setupHealthBar` method:

```
_healthBarProgress.hidden = YES;
_healthBarBg.hidden = YES;
```

This sets the health bar to be initially invisible.

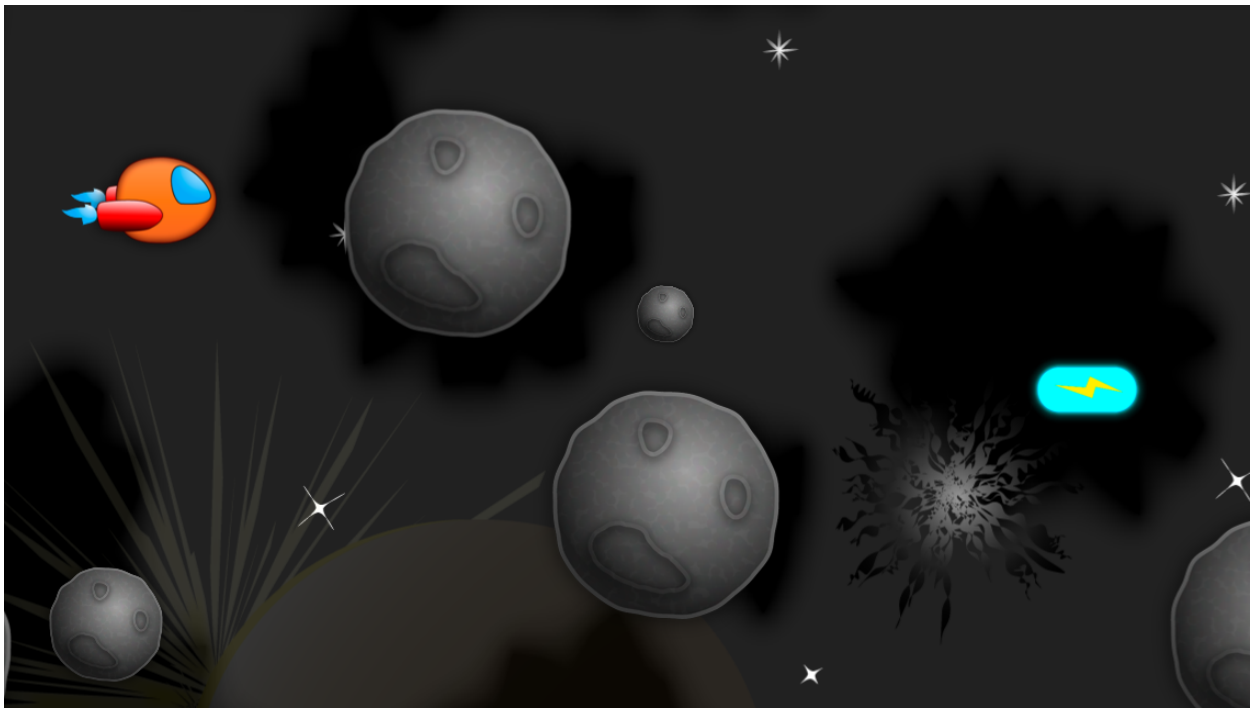
Next, add these lines to the bottom of `update:`:

```
if (desiredWidth != _displayedWidth) {
  NSArray * sprites = @[_healthBarBg, _healthBarProgress];
  for (SKSpriteNode * sprite in sprites) {
    sprite.hidden = NO;
    [sprite removeAllActions];
    [sprite runAction:
      [SKAction sequence:@[
```

```
        [SKAction fadeInWithDuration:0.25],
        [SKAction waitForDuration:2.0],
        [SKAction fadeOutWithDuration:0.25],
        [SKAction runBlock:^{
          sprite.hidden = YES;
        }]
      ]
    ]];
  }
}
```

This checks to see if the desired width is *not* equal to the displayed width. If so, it sets the health bar to visible and schedules a sequence of actions to fade it in, wait a bit, then fade it out. Note that this bit of code is called continuously as the health bar animates, and each time it cancels and reschedules these actions until the health bar is done animating—then a few seconds later, it fades out the health bar.

And you're done! Build and run, and now the health bar only shows up when necessary.



Adding the Big Boss

Just when you're distracted by a trivial thing like the health bar—out of nowhere, your nemesis strikes!

In this section, the Big Boss is going to join the game to spoil the party. There are a lot of steps to get the Big Boss fully functional. For now, you'll just make him roll into the scene looking foreboding, as a warning of what is to come!

As usual, begin by creating a class for the boss. Create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **Boss** for Class, **Entity** for Subclass of, click **Next** and then click **Create**.

Open **Boss.m** and replace the contents with the following:

```
#import "Boss.h"

@implementation Boss

- (instancetype)init {
  if ((self = [super initWithImageNamed:@"Boss_ship" maxHp:50
    healthBarType:HealthBarTypeRed])) {
    [self setupCollisionBody];
  }
  return self;
}

- (void)setupCollisionBody {
  CGPoint offset = CGPointMake(
    self.size.width * self.anchorPoint.x, self.size.height * self.anchorPoint.y);
  CGMutablePathRef path = CGPathCreateMutable();
  [self moveToPoint:CGPointMake(60, 98) path:path offset:offset];
  [self addLineToPoint:CGPointMake(142, 107) path:path offset:offset];
  [self addLineToPoint:CGPointMake(175, 42) path:path offset:offset];
  [self addLineToPoint:CGPointMake(155, 14) path:path offset:offset];
  [self addLineToPoint:CGPointMake(39, 9) path:path offset:offset];
  [self addLineToPoint:CGPointMake(6, 27) path:path offset:offset];
  CGPathCloseSubpath(path);

  self.physicsBody = [SKPhysicsBody bodyWithPolygonFromPath:path];
  [self attachDebugFrameFromPath:path color:[SKColor redColor]];

  self.physicsBody.categoryBitMask = EntityCategoryAlien;
  self.physicsBody.collisionBitMask = 0;
  self.physicsBody.contactTestBitMask = EntityCategoryPlayerLaser;
}

@end
```

As usual, this creates a collision shape for the boss. Note that the boss has a lot more HP than any entity you've seen so far! ☺

You're not done here yet—you're going to add a bit of code to make the boss move into the scene on startup. First, add this new private instance variable:

```
@implementation Boss {
  BOOL _initialMove;
}
```

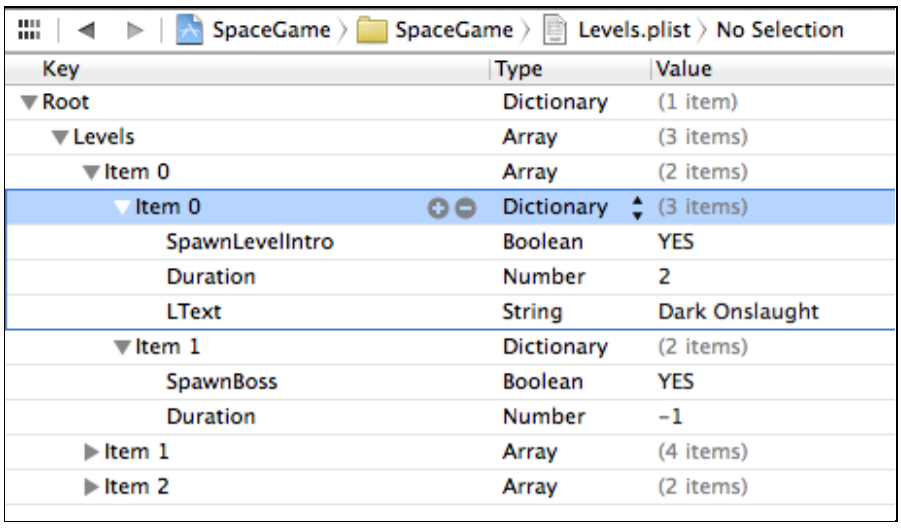
This keeps track of whether the boss has made his initial move yet. Next, override `update:` as follows:

```
- (void)update:(CFTimeInterval)dt {
  [super update:dt];

  if (!_initialMove) {

    _initialMove = YES;
    CGPoint midScreen = CGPointMake(
      self.scene.size.width/2, self.scene.size.height/2);
    SKAction * move = [SKAction moveTo:midScreen duration:4.0];
    [self runAction:move];
  }
}
```

This makes it so that if the boss hasn't made his initial move, `update:` runs an action to move the boss to the middle of the screen.

Now that you have a boss subclass, you need to modify your game to use it. The first step is to create a new level entry in **Levels.plist** with a stage that has a flag to spawn a boss.

I've already made a new version of **Levels.plist** for you with these settings—you can find it under **Levels\V6**. Replace your version of **Levels.plist** with this—and don't forget to **Product\Clean**! When you open it, you'll see the following:



You have an ominous new level title, and then in the second stage you have a new **SpawnBoss** flag set to YES. Also note that **Duration** is set to -1. This is a special

case you'll use to indicate that "the stage will have some special code to advance to the next stage itself when it's done, so don't base progress on time."

Instead, the only way this stage will end is when one of you is dead!

OK, let's make use of all of this. Open **MyScene.m** and add this import to the top of the file:

```
#import "Boss.h"
```

Then add the following new private instance variable:

```
Boss *_boss;
```

This variable keeps track of the boss. There can be only one at a time—after all, he is the boss!

Next, add a new method right after `applyPowerup` to create the boss:

```
#pragma mark - Boss

- (void)spawnBoss {
  _boss = [[Boss alloc] init];
  _boss.position = CGPointMake(self.size.width * 1.2, self.size.height * 1.2);
  [_gameLayer addChild:_boss];

  [self shakeScreen:100];
  [self runAction:_soundBoss];
}
```

This sets the boss to be initially offscreen to the upper right and adds him to the game layer. Remember, the boss's `update:` method will then run an action to make him move onscreen. The above method also shakes the screen and plays a cool sound effect.

Next, add these lines to the end of `newStageStarted` to spawn the boss when appropriate:

```
else if ([_levelManager hasProp:@"SpawnBoss"]) {
  [self spawnBoss];
}
```

If a stage has the "SpawnBoss" flag, this code spawns the boss at the start of the stage.

There's one last step. If the player destroys the boss, you want to advance the game to the next stage. So you need to implement the collision callback on the boss to take a hit and advance to the next stage when appropriate.

First, open **MyScene.h** and declare the `nextStage` method, which you wrote earlier:

```
- (void)nextStage;
```

Then open **Boss.m** and import this header:

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

And implement the collision callback as follows:

```
- (void)collidedWith:(SKPhysicsBody *)body contact:(SKPhysicsContact *)contact {
  Entity * other = (Entity *)body.node;
  if (body.categoryBitMask & EntityCategoryPlayerLaser) {
    [other destroy];
    [self takeHit];
    MyScene *scene = (MyScene *)self.scene;
    if ([self isDead]) {
      [scene spawnExplosionAtPosition:contact.contactPoint
        scale:self.xScale large:YES];
      [scene nextStage];
    } else {
      [scene spawnExplosionAtPosition:contact.contactPoint
        scale:self.xScale large:NO];
    }
  }
}
```

This is similar to the other collision callbacks you've written, except that, when the boss dies, it advances to the next stage.

Build and run the code—and finally, your nemesis appears!

Adding the Weapons

If you look at the boss right now, you might have to force back a chuckle. It looks like your Big Bad Boss left his house without pulling on his weapons!

Let's save your boss from further embarrassment by adding them for him. It turns out that he's well stocked—he's got two laser shooters and a cannon turret, as well!

Open up **Boss.m** and add the private instance variables for these weapons:

```
SKSpriteNode *_shooter1;
SKSpriteNode *_shooter2;
SKSpriteNode *_cannon;
```

Then add this new method to initialize the weapons:

```
- (void)setupWeapons {
  _shooter1 = [SKSpriteNode spriteNodeWithImageNamed:@"Boss_shooter"];
  _shooter1.position = CGPointMake(self.size.width * .15, 0);
  [self addChild:_shooter1];

  _shooter2 = [SKSpriteNode spriteNodeWithImageNamed:@"Boss_shooter"];
  _shooter2.position = CGPointMake(self.size.width * .05, -self.size.height * 0.4);
  [self addChild:_shooter2];

  _cannon = [SKSpriteNode spriteNodeWithImageNamed:@"Boss_cannon"];
  _cannon.position = CGPointMake(0, self.size.height * 0.45);
```

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

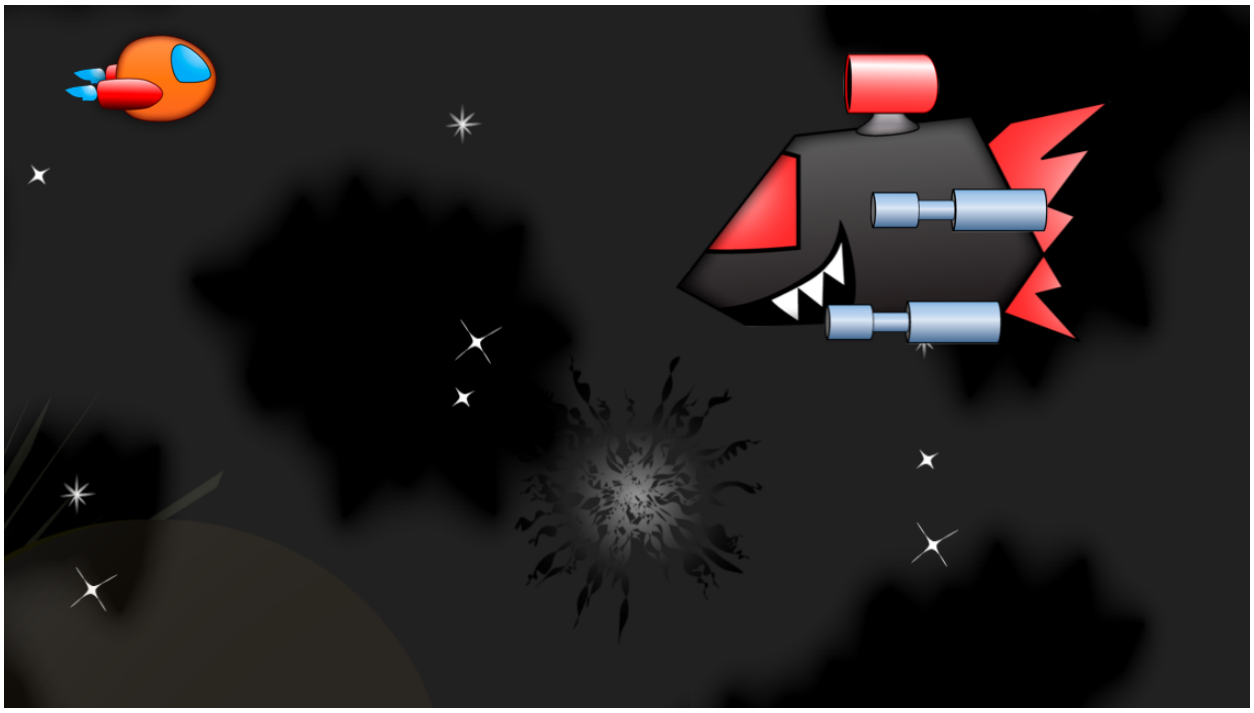
This creates two sprites using the **Boss_shooter.png** image and one with the **Boss_cannon.png** image. It adds them as children of the boss sprite and sets their positions based on the bottom-left corner of the boss sprite.

Note that you've set the positions as multiples of the boss sprite's size, rather than hardcoded offsets. This prevents you from having to write two different offsets—one for iPhone and one for iPad.

Call this method from `init`, right after the call to `setupCollisionBody`:

```
[self setupWeapons];
```

Build and run the code, and now the boss comes in fully dressed! ☺



Boss In Action

I'm willing to bet that one of those times you tested the above code, you blasted the poor boss into oblivion without him even having a chance to fight back!

Well, it's that kind of insolence that made the boss an evil overlord in the first place. You've just made him mad.

The boss's strategy will be to totally confuse you by doing a sequence of random actions. Sometimes he'll move around, sometimes he'll pause and sometimes he'll shoot.

Create a new method to do this in **Boss.m**, right above `update:`:

```
- (void)performRandomAction {
  int randomAction = arc4random() % 4;
  SKAction *action;

  if (randomAction == 0 || !_initialMove) {
    _initialMove = YES;
    float randWidth = RandomFloatRange(
      self.scene.size.width * 0.6, self.scene.size.width * 1.0);
    float randHeight = RandomFloatRange(
      self.scene.size.height * 0.1, self.scene.size.height * 0.9);
    CGPoint randDest = CGPointMake(randWidth, randHeight);

    CGPoint offset = CGPointSubtract(self.position, randDest);
    float length = CGPointLength(offset);
    float BOSS_POINTS_PER_SEC = 100.0;
    float duration = length / BOSS_POINTS_PER_SEC;

    NSLog(@"Moving to %@ over %0.2f", NSStringFromCGPoint(randDest), duration);

    action = [SKAction moveTo:randDest duration:duration];
  } else if (randomAction == 1) {
    action = [SKAction waitForDuration:0.2];
  } else if (randomAction >= 2 && randomAction < 4) {
    MyScene *scene = (MyScene *)self.scene;
    [scene spawnAlienLaserAtPosition:
      [self convertPoint:_shooter1.position toNode:self.parent]];
    [scene spawnAlienLaserAtPosition:
      [self convertPoint:_shooter2.position toNode:self.parent]];
    action = [SKAction waitForDuration:0.2];
  }

  [self runAction:
    [SKAction sequence:@[
      action,
      [SKAction performSelector:@selector(performRandomAction) onTarget:self]
    ]
  ]];
}
```

There's a lot of code here, but most of this should be review. I'll just point out the particularly interesting bits:

- You pick a random number and create an action to run based on that number. At the end of the method, you run this action and then call `randomAction` again to run another one.
- To determine a spot to move the boss, you get a random x-coordinate between 0.6–1.0 times the screen width, and a random y-coordinate between 0.1–0.9

times the screen height. That way the boss will never collide with the ship, but will move around in a random pattern.

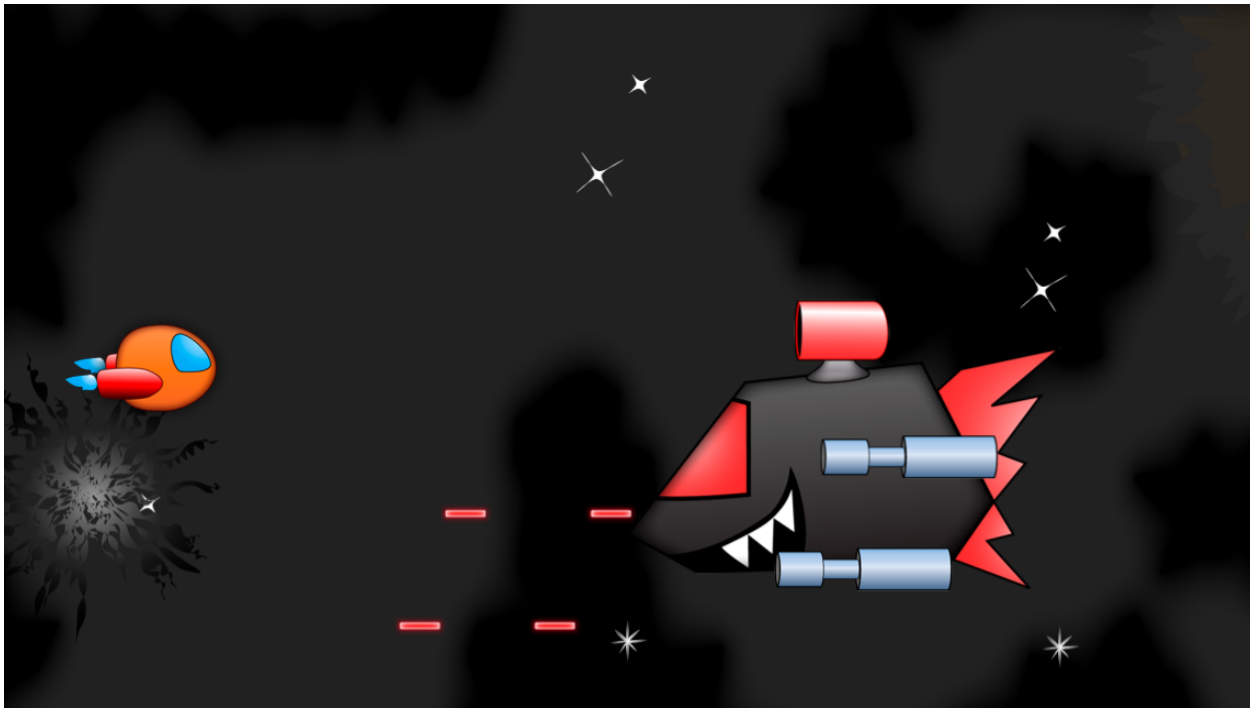
- You want the boss to move at a constant rate, but when you use `moveTo:`, you don't pass a rate—you pass a duration. To figure out the duration, you first determine how far the boss will move by subtracting the destination from the current position. You then use a helper function to get the length based on this vector (`CGPointLength`). Now that you have the length (in points), you divide it by a rate (in points per second) to get the seconds to move. The rate is hard-coded to 100 points per second here, but you could make this a random value if you'd like.
- You want the lasers to shoot from the positions of the shooters, but you can't use the shooters' positions as they are because, as children of the boss, their positions are relative to the boss. So you use `convertPoint:toNode:` to convert the shooters' positons to the game layer's coordinates.

One final step—replace `update:` with the following:

```
- (void)update:(CFTimeInterval)dt {
  [super update:dt];

  if (!_initialMove) {
    [self performRandomAction];
  }
}
```

This calls your new method rather than the old test code.

Build and run the code, and see if you have what it takes to defeat the boss!



The Hidden Weapon

As with most good boss fights, just when you think you've got him beat—he pulls out a hidden weapon!

And in this boss's case, it's that big cannon sitting on top that has been ominously quiet… until now.

Open **Boss.m** and import this header:

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

Then add this new method right before `update:`:

```
- (void)updateCannon {
  MyScene *scene = (MyScene *)self.scene;
  CGPoint cannonWorld = [self convertPoint:_cannon.position toNode:self.parent];
  CGPoint offsetToPlayer = CGPointSubtract(cannonWorld, scene.player.position);
  float cannonAngle = CGPointToAngle(offsetToPlayer);
  _cannon.zRotation = cannonAngle;
}
```

This code makes sure that the cannon always points at the player's ship. But how in the world does it work?! Well, let's go over it line by line.

• The first line sets up a pointer to the scene to eventually access the ship.

- The second line figures out the position of the cannon in the coordinate space of the game layer.
- The third line figures out the vector between the cannon's position and the ship's position by simply subtracting the two.
- The fourth line figures out the angle to turn the cannon so that it points at the ship. You're using a helper method from `SKTUtils` (`CGPointToAngle`) to get the angle of a vector.
- The final line sets the rotation of the cannon to match.

Call this method at the bottom of `update:`:

```
[self updateCannon];
```

Build and run the code, and—uh oh—that big cannon remains pointed at the ship!

And cannonballs are right about to follow!

First, you need a class for the cannonballs. Create a new file by going to **File\New\File**. Choose **iOS\Cocoa Touch\Objective-C class** and click **Next**. Enter **CannonBall** for Class, **Entity** for Subclass of, click **Next** and then click **Create**.

Open **CannonBall.m** and replace the contents with the following:

```
#import "CannonBall.h"

@implementation CannonBall
```

```
- (instancetype)init {
  if ((self = [super initWithImageNamed:@"Boss_cannon_ball" maxHp:1
    healthBarType:HealthBarTypeNone])) {
    [self setupCollisionBody];
  }
  return self;
}

- (void)setupCollisionBody {
  CGPoint offset = CGPointMake(
    self.size.width * self.anchorPoint.x, self.size.height * self.anchorPoint.y);
  CGMutablePathRef path = CGPathCreateMutable();
  [self moveToPoint:CGPointMake(9, 18) path:path offset:offset];
  [self addLineToPoint:CGPointMake(12, 24) path:path offset:offset];
  [self addLineToPoint:CGPointMake(19, 25) path:path offset:offset];
  [self addLineToPoint:CGPointMake(26, 18) path:path offset:offset];
  [self addLineToPoint:CGPointMake(25, 12) path:path offset:offset];
  [self addLineToPoint:CGPointMake(19, 7) path:path offset:offset];
  [self addLineToPoint:CGPointMake(11, 8) path:path offset:offset];
  CGPathCloseSubpath(path);

  self.physicsBody = [SKPhysicsBody bodyWithPolygonFromPath:path];
  [self attachDebugFrameFromPath:path color:[SKColor redColor]];

  self.physicsBody.categoryBitMask = EntityCategoryAlienLaser;
  self.physicsBody.collisionBitMask = 0;
  self.physicsBody.contactTestBitMask = EntityCategoryPlayer;
}

@end
```

As usual, this creates a collision shape for the cannonball.

Open **MyScene.m** and import this header:

```
#import "CannonBall.h"
```

Then add this new method right after `spawnAlienLaserAtPosition:`:

```
- (void)shootCannonBallAtPlayerFromPosition:(CGPoint)position {
  CannonBall *cannonBall = [[CannonBall alloc] init];
  cannonBall.position = position;
  [_gameLayer addChild:cannonBall];

  CGPoint offset = CGPointSubtract(_player.position, cannonBall.position);
  CGPoint shootVector = CGPointNormalize(offset);
  CGPoint shootTarget = CGPointMultiplyScalar(shootVector, self.size.width * 2);
```

```
  [cannonBall runAction:[SKAction sequence:@[
    [SKAction moveByX:shootTarget.x y:shootTarget.y duration:5.0],
    [SKAction removeFromParent]
  ]]];
}
```

This method takes the starting position of the cannonball, but it needs to figure out where to fire it—and how fast to move it.

It first finds the vector between the cannon and the ship's positions by subtracting the two. It also normalizes the vector, which is a fancy way of saying it makes the vector's length 1. This is handy because then you can multiply it by a number to get a vector in the same direction, but of a desired length.

And that's exactly what you do in the next line—you make the vector two times the width of the window, so that you can be sure the cannonball moves far enough to be offscreen.

Since the cannonball moves the same distance each time you call this method—only the direction changes—you can move it a set number of seconds each time (5 seconds here) and it will always move at the same speed.

Next, declare this method in **MyScene.h** so that you can call it from another file:

```
- (void)shootCannonBallAtPlayerFromPosition:(CGPoint)position;
```

You're almost done! Switch to **Boss.m** and, in `performRandomAction`, modify the line that creates the random action number as follows:

```
  int randomAction = arc4random() % 5;
```

This simply gives you a random number between 0-4 (inclusive) rather than between 0-3 (inclusive).

Then add the following to the end of the if/else statement to handle the extra random action case:

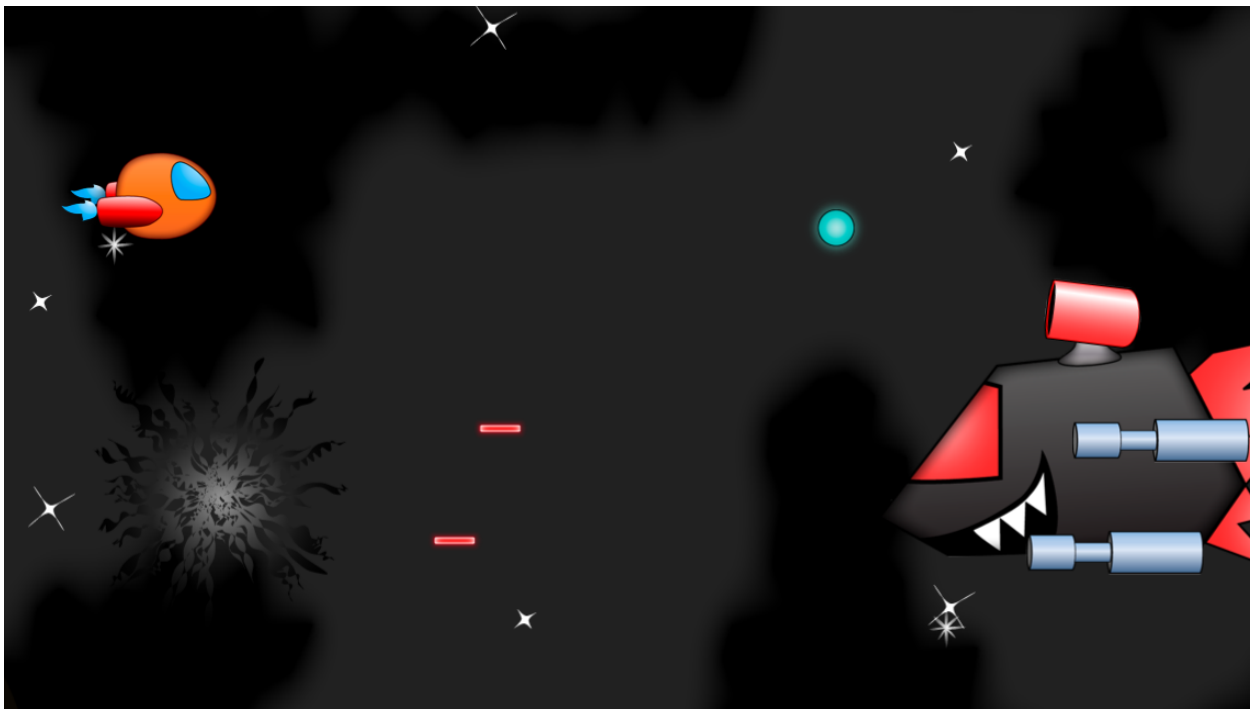
```
else if (randomAction == 4) {
  MyScene *scene = (MyScene *)self.scene;
  [scene shootCannonBallAtPlayerFromPosition:
    [self convertPoint:_cannon.position toNode:self.parent]];
  action = [SKAction waitForDuration:0.2];
}
```

This figures out the cannon's position and then tells the scene to shoot a cannonball at the ship from that position.

The last step is to replace your **Levels.plist** with the final version, located in **Levels\V7**, and—you guessed it—don't forget to **Product\Clean**. This final

version of **Levels.plist** has three full levels, demonstrating everything you've built in this starter kit.

Build and run the game, and see if you can get all the way through—including the now much more challenging boss fight at the end!



And no fair cheating by hacking the code! ☺

Where to Go From Here?

Guess what? You've done it—you've coded the entire Space Game Starter Kit!

You should be proud of everything you've accomplished. You've just made a complete game from scratch, with a lot of really cool features.

You've written every line of code on your own, so now you can take everything you've learned and make your own game!

It's up to you whether you want to keep building on top of this game, or take some ideas or code from it and make something completely different.

One piece of advice I'll offer is to *start simple*. Just like you did in these chapters, take the simplest idea you have and get it working. Then add features and effects one by one until you're satisfied.

And then, the most important part—submit your game to the App Store so others can enjoy your creation! And when you do, please drop me a line—I'd love to see what you've come up with!

C Conclusion

Thank you!

I hope you enjoyed the Space Game Starter Kit and had fun making this game. I can't thank you enough for your continued support of raywenderlich.com and everything our team does there.

I appreciate each and every one of you for taking the time to try out the Space Game Starter Kit. If you have an extra minute, I would really love to hear what you thought of this starter kit!

Please leave a comment at http://www.raywenderlich.com/forums, or send me an email if you'd rather reach me privately. Although sometimes it takes me a while to respond, I do read each and every email, so please don't hesitate to drop me a note!

Finally, if you haven't already, you can follow me on Twitter (@rwenderlich) and subscribe to my monthly iOS newsletter for my favorite iOS dev links and tutorials.

Please stay in touch, and best of luck with your iOS adventures!

Ray Wenderlich

ray@raywenderlich.com