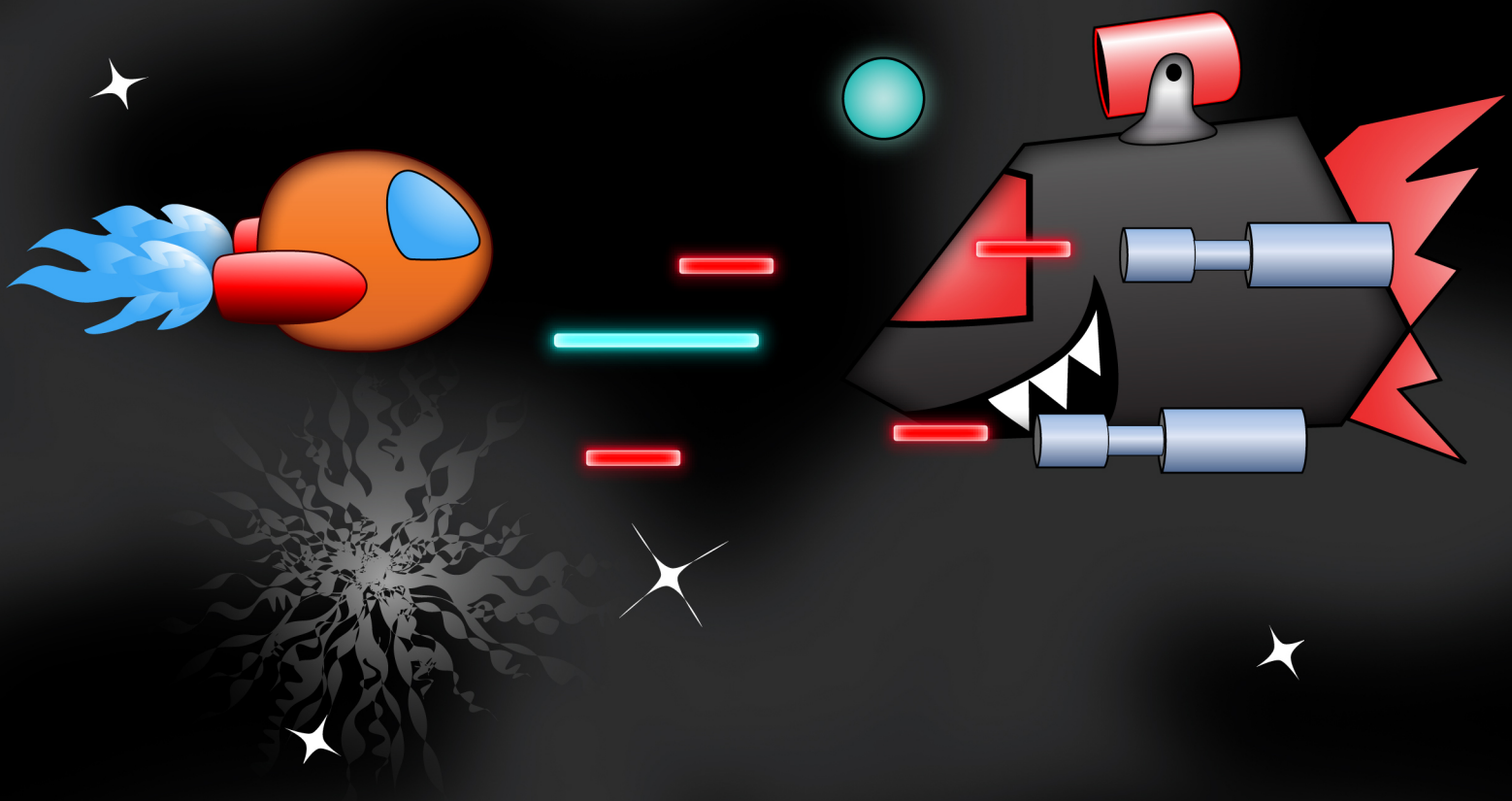
Space Game Starter Kit



Full source code and tutorials for creating a side-scrolling space shooter game for the iPhone and iPad!

by Ray Wenderlich
of raywenderlich.com

Version 1.1

Space Game Starter Kit for iPhone and iPad

By Ray Wenderlich of raywenderlich.com

This Starter Kit is dedicated to the readers of my blog.

Thank you for your continued readership and support!

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 that 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 tutorials in this guide are for your own personal use only. You are NOT allowed to distribute or sell the tutorials in this guide without prior authorization.

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

Please understand that there are some links contained in this guide that I may benefit from financially.

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

Table of Contents

Introduction

Welcome to the Space Game Starter Kit for iPhone and iPad!

The Space Game Starter Kit includes full source code for a complete side-scrolling space game for the iPhone and iPad, complete 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—but you also get four epic-length tutorials 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 will give you some great benefits, such as:

- It gives you an example of a fully functional game to study and learn from.
- It gives you code, art, sound effects, music, and particle systems that you can directly reuse in your own games.
- It gives you 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 heads first, or reinforce your existing knowledge.
- It will teach you tips and tricks that you might not have come across before, such as: modifying Box2D shapes based on sprite scale, defining levels in a property list, resizing textures dynamically, using Bezier paths for enemy movement and debug drawing and much more.
- It gives back to raywenderlich.com, making future tutorials, forum support, and Starter Kits possible!
- It is really fun and relaxing to go through the step-by-step tutorial, 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 test with, because the game will be using 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 read the book Programming in Objective-C 2.0 by Stephen Kochan. This is the book I used to learn, and it's also a great reference to look up certain syntax that might confuse you. Alternatively, there's an online tutorial here.

This starter kit also assumes you have some basic familiarity with Cocos2D, which is the framework we'll be using to make the game. If you are new to Cocos2D, I recommend you read a book that Rod Strougo and I wrote called Learning Cocos2D. It covers the most important aspects of using Cocos2D system by system, and along the way you create a game called Space Viking.

That said, if you are completely new to Objective-C and Cocos2D, you can still follow along with this tutorial because everything is presented step-by-step. It's just that there will be some missing pieces in your knowledge that the above books will fill in.

How To Use the Space Game Starter Kit

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

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

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

A second way of using the Space Game Starter Kit is you can go through these tutorials one by one and build up the Space Game from scratch. This way you'll learn …

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

Blast Off!

And with this introduction complete, it's time to blast off into our space game!

Grab your favorite caffeinated beverage and some snacks, prop up your feet and get ready for some fun—it's time 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!

Tutorial 1: A Basic Space Shooter

Get ready to strap in!

In this first tutorial, we're going to strap right in and make a simple version of our space game, where we can pilot a cool spaceship and blast through a dangerous asteroid belt!

You may notice that this tutorial is somewhat similar to the How To Make A Space Shooter iPhone Game 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, better sprite preloading, and much more.

So even if you've done that tutorial before, you should still go through this from the beginning!

So without further ado, let's bravely go where no tutorial has gone before! :]

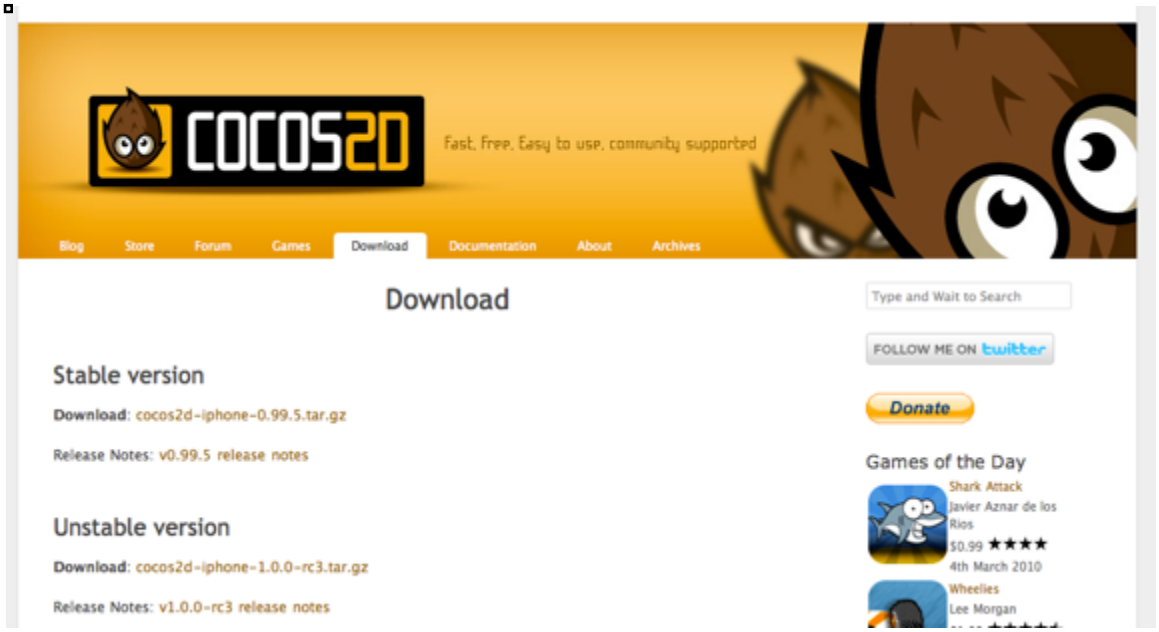
Installing Cocos2D

To make this game, we're going to make use of a free and popular open-source iPhone and iPad game programming framework called Cocos2D.

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

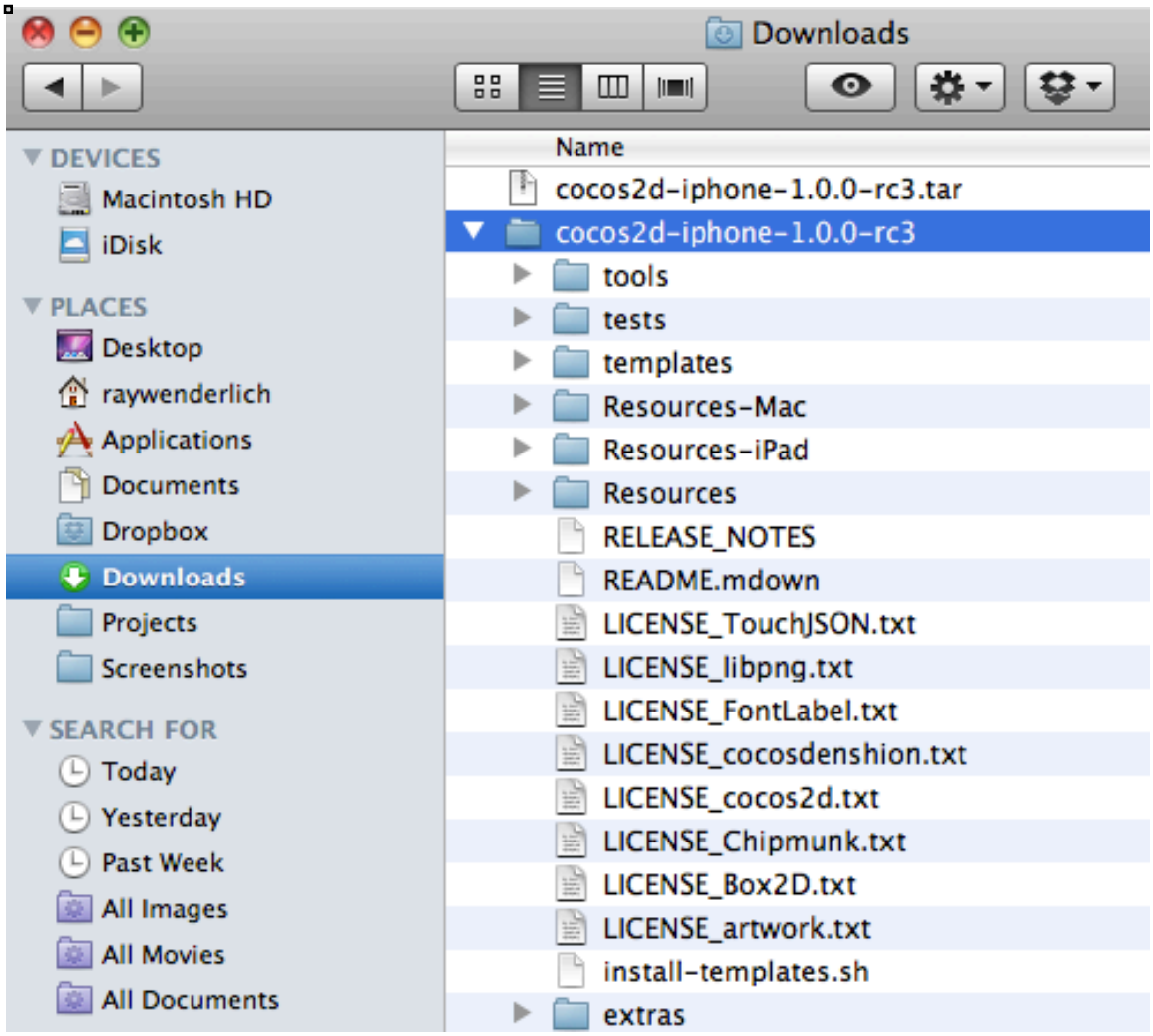
Load up your web browser and navigate to this page: http://www.cocos2d-iphone.org/download.

You'll see something that looks like the following:



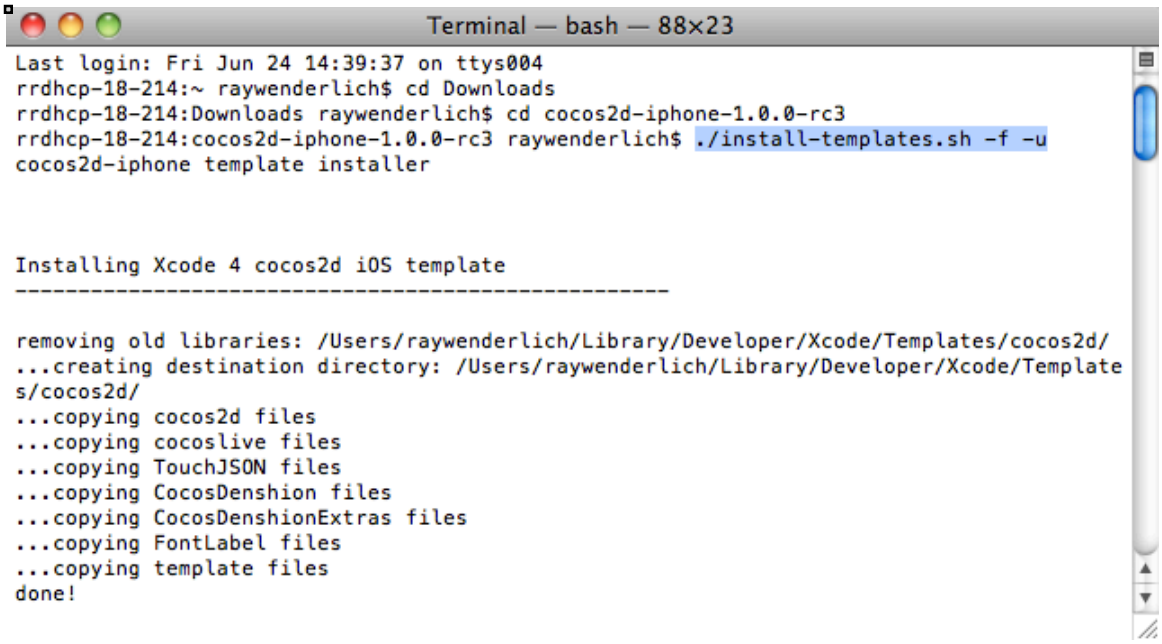
You should download the latest version of Cocos2D in the **Unstable version** section. Don't worry that it says unstable—it actually works quite well!

Once you've downloaded the file, locate the file in Finder and double click the file to unzip it. You'll see a folder that looks something like this:



Next, you need to install the Cocos2D templates so you can easily create a new project that uses Cocos2D with Xcode.

To do this, go to Applications\Utilities and run your Terminal app. Use the **cd** command to switch directories to where you downloaded cocos2d, and then install it with the following command as you can see in the screenshot below: **./install-templates.sh –f -u**



If you get a permission error when you try this, run the command with **sudo** in front, remove the **–u**, and enter your password when prompted.

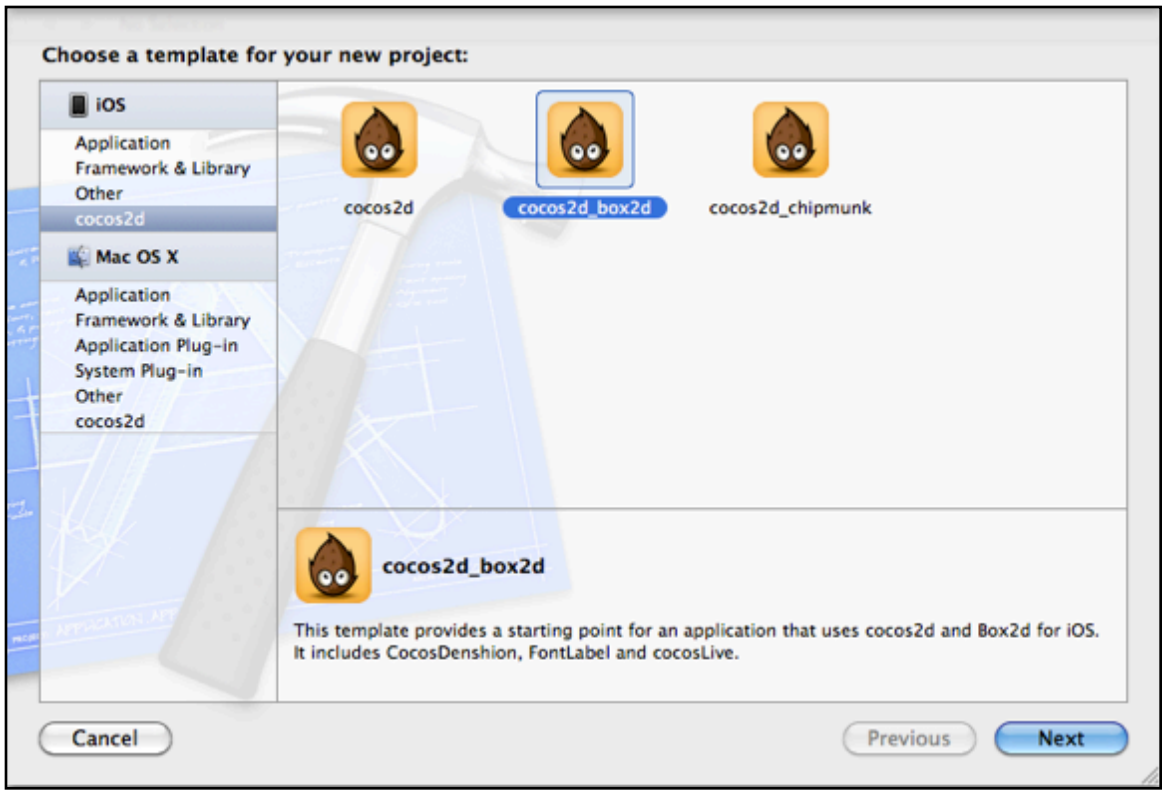
Finally, close Xcode if it is currently running, so that when it starts back up next time it can find the new Cocos2D templates you just installed.

That's it! Now you can move on to the next section where we'll start making our space game!

Displaying the Game Title

Let's get started by creating a "Hello, World" Cocos2D project, and then modifying it to display the title of the Space Game Starter Kit!

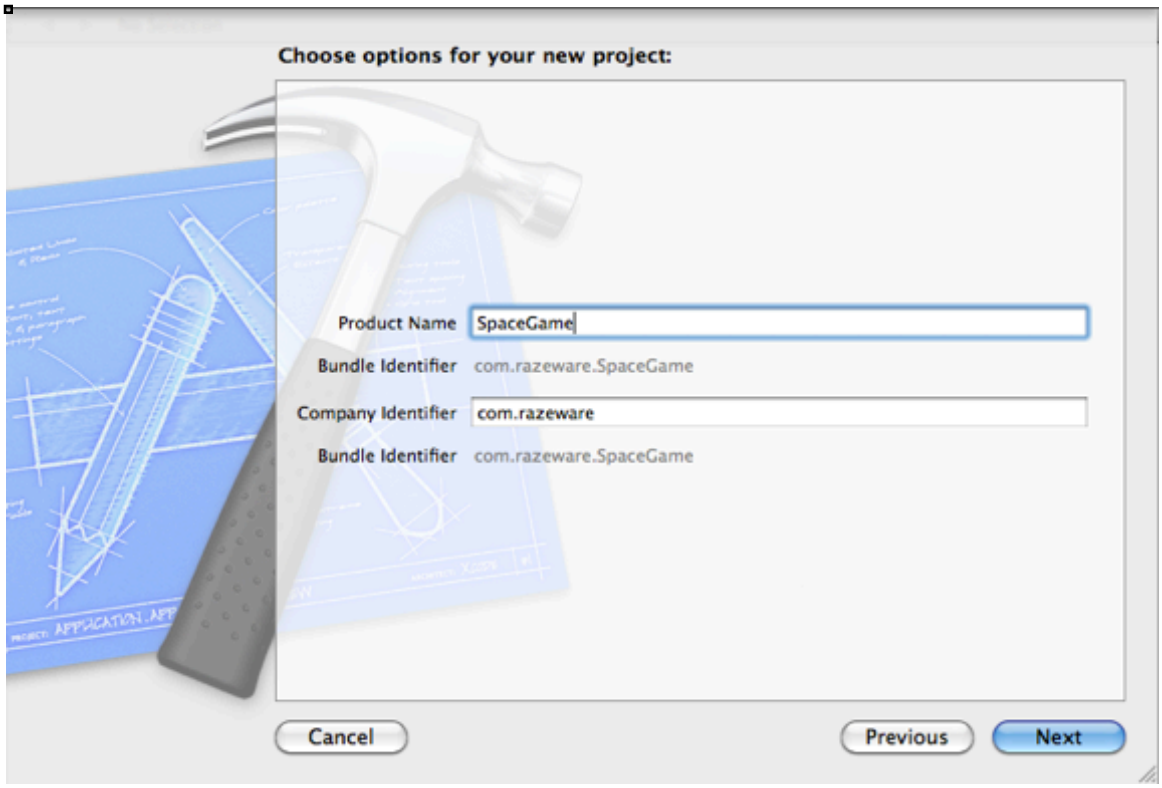
Start up Xcode, go to **File\New\New Project**, and choose the **iOS\cocos2d\cocos2d_box2d** template, as you can see below:



Note: You may wonder why we're choosing the Cocos2D and Box2D template, instead of the plain old Cocos2D template.

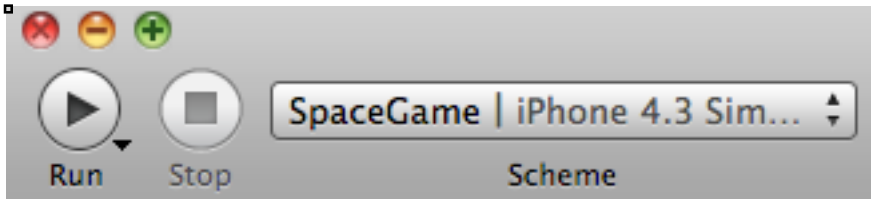
Well, Box2D is one of the physics libraries that comes included with Cocos2D. It includes some nice shape definition and collision detection features that we'll need later on in this game, so we'll choose this so we're set up to use Box2D when we need it.

Once you've selected the cocos2d_box2d template, click **Next**, enter **SpaceGame** for the Product Name, and click **Next** again.

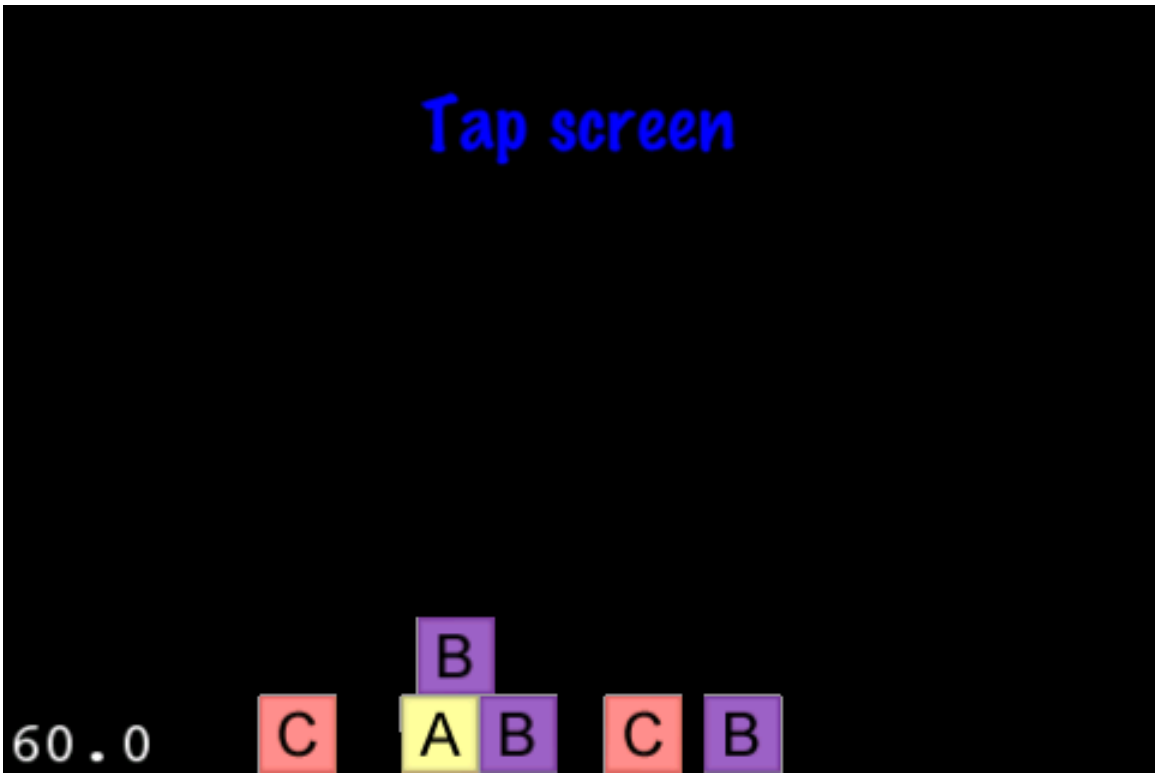


In the final step, choose a folder on your hard drive to save the project, and click **Create**.

You now have a "Hello, World" Cocos2D and Box2D project that the template has set up for you. You can compile and run this project if you'd like—just choose your **iPhone simulator** from the **Scheme** dropdown and click the **Run** button:

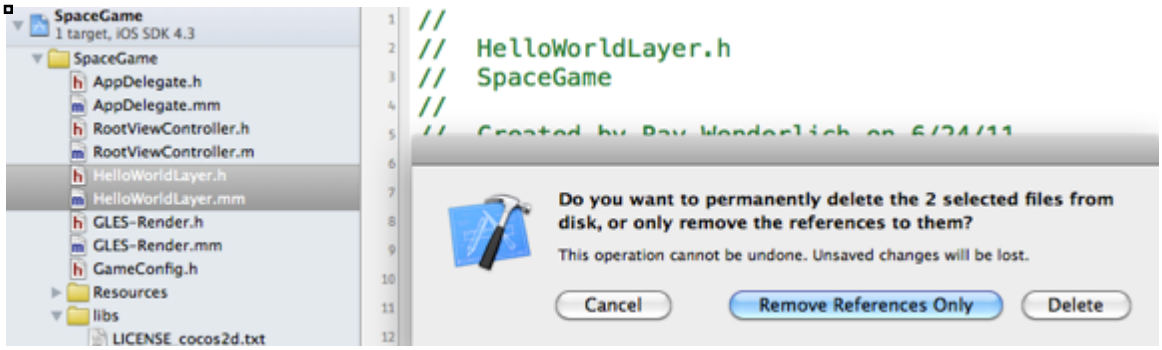


At this point you'll see a brick fall down onto the screen, and if you click you can create more bricks, demonstrating some of the capabilities of Cocos2D and the Box2D physics library.



The code for this is all in **HelloWorldLayer.h** and **HelloWorldLayer.mm**. It's interesting stuff that you might want to look at sometime.

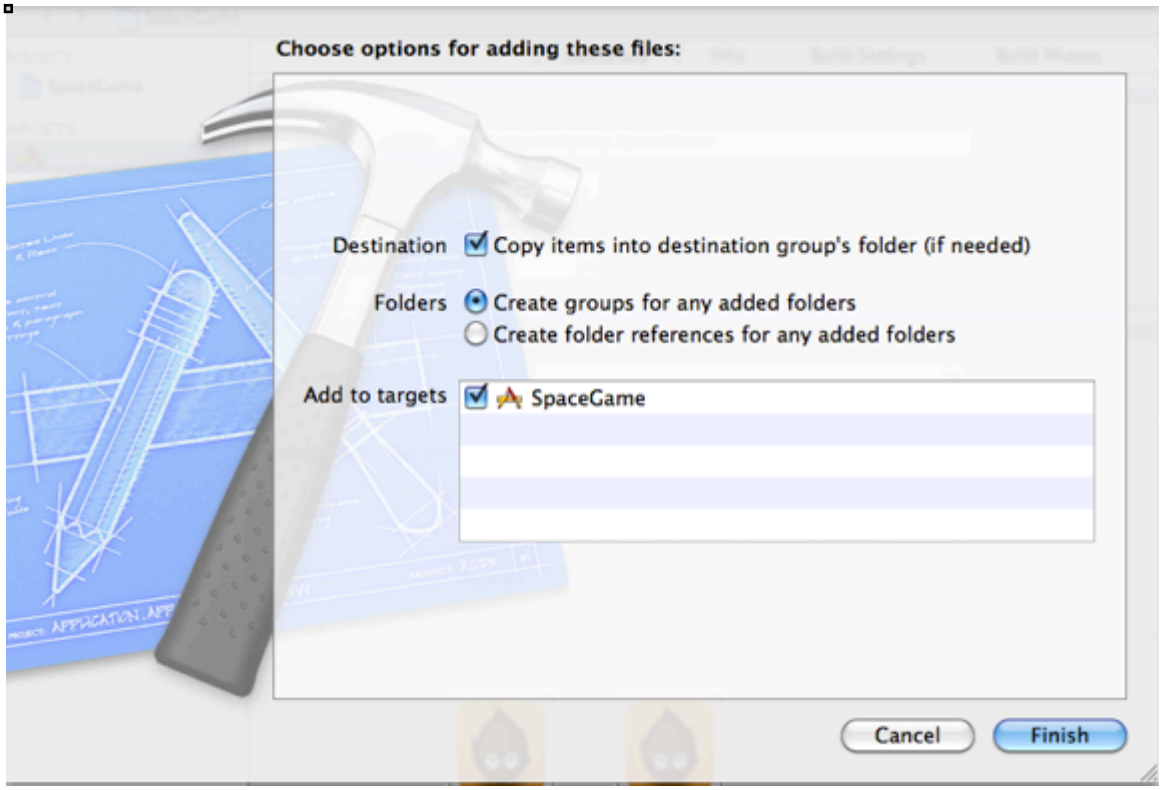
But we're making a space game, not a game about little bricks! So the first step is to select **HelloWorldLayer.h** and **HelloWorldLayer.mm** in Xcode, control-click them, and click **Delete**. When the warning comes up, click **Delete** again to blow those suckers away!



So instead of this starter code, we're going to create our own code to display the first part of our game—the title of the game!

To display the title, we're going to need a file that contains the styled font that we're going to use for the text. So find the resources that come along with this Starter Kit—you should have a directory named **Art**.

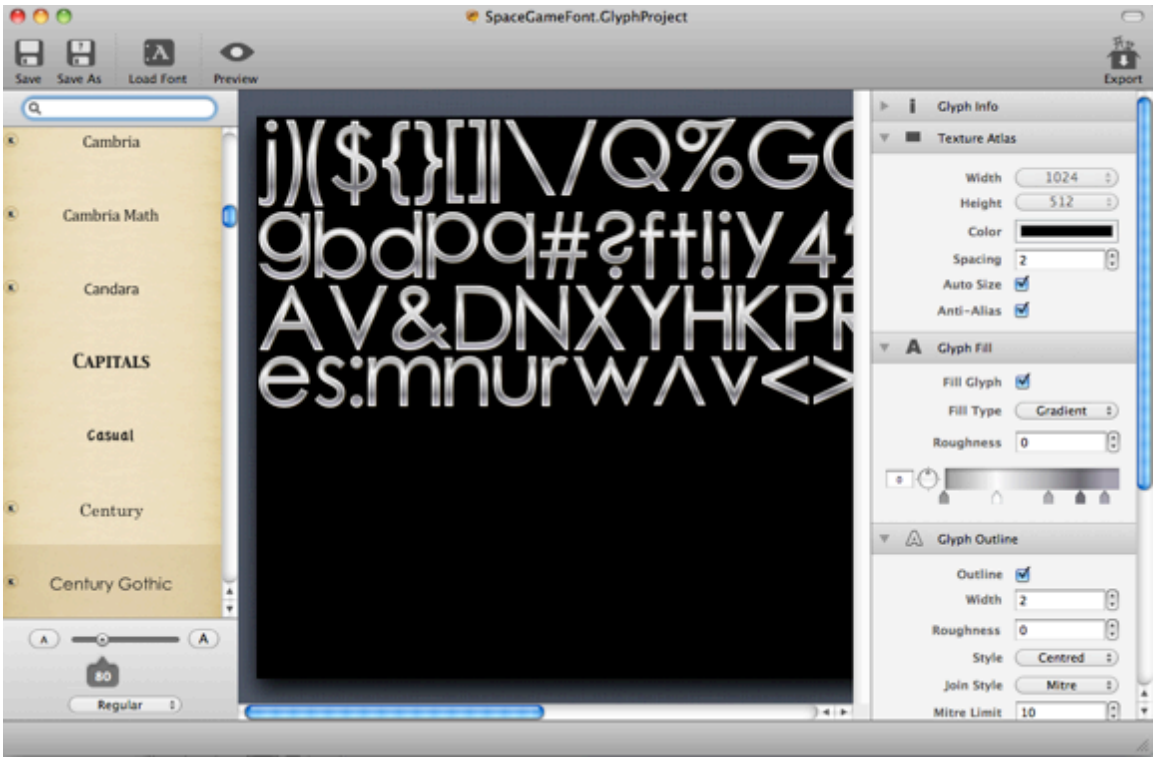
Once you find the the **Art** folder, drag it into your Xcode project. A dialog will pop up—make sure that **Copy items into destination group's folder (if needed)** is checked, **Create groups for any added folders** is selected, and click **Finish**.



You can look through the contents of the Art folder if you'd like—there's a lot of cool stuff in there. Note there's even some code in there (under the **Classes** folder) that you'll need later. We'll go over each piece as we need to use it.

For right now, we're about to use the files in **Art\Fonts**. These files contain the fonts that we need to use to display the title of the game. We created these files for you using the tool Glyph Designer, which lets you create these with a nice GUI.

(Optional) If you're curious how the font was made or would like to tweak the colors or font style, you can download Glyph Designer and open up the **SpaceGameFont.GlyphProject** that comes in the **Raw** folder included with your Space Game Starter Kit. From there you can look at the settings and change anything you'd like!



Note that I just used a built-in font on the Mac (Century Gothic) and used Glyph Designer to change the coloring of the font to have a cool metal effect (by using a series of gradients).

If you don't have Glyph Designer, don't worry - for the purposes of this tutorial, you can use the fonts I made as-is.

Now that you've deleted the starter code and added the resources you need, it's finally time to code!

Select the **SpaceGame** group in Xcode, go to **File\New\New File**, choose **iOS\Cocoa Touch\Objective-C class**, and click **Next**. Enter **NSObject** for Subclass of, click **Next**, name the new file **ActionLayer.m**, and click Save.

Then open up **ActionLayer.h** and replace it with the following:

```
#import "cocos2d.h"

@interface ActionLayer : CCLayer {
    CCLabelBMFont * _titleLabel1;
    CCLabelBMFont * _titleLabel2;
}

+ (id)scene;

@end
```

The first thing this does is import cocos2d.h, so you can use the Cocos2D classes and functions.

It then declares a new subclass of **CCLayer** named **ActionLayer**. In Cocos2D, each "screen" of your app typically consists of one Cocos2D **scene**, and each scene contains one or more **layers**.

In this space game, we're just going to have one scene, and one layer. The single layer will contain everything we're going to add into the game. We also create a static helper method (scene) to create the scene that contains the **ActionLayer**.

Note: A lot of times you'll see games use one scene for the title, a second scene for the gameplay, a third scene for the game over, and so on.

That is a good approach if you want different screens separated by transitions, but for this game I think the effect is cooler if we don't transition away, and keep everything within the same scene. That way it feels like the action is nonstop and dynamic.

Inside the **ActionLayer** class definition, we declare two instance variables – **_titleLabel1**, and **_titleLabel2**. These are both **CCLabelBMFont** classes, which is the class you use to display text to the screen in Cocos2D.

Note: A common question I get is why I like to name my instance variables with underscores like this.

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 we're working on an instance variable and not something else (like a local variable or property).

Now that you've set up the header file, switch to **ActionLayer.m** and replace the contents with the following:

```
#import "ActionLayer.h"

@implementation ActionLayer

+ (id)scene {

    CCScene *scene = [CCScene node];
    ActionLayer *layer = [ActionLayer node];
    [scene addChild:layer];
    return scene;

}

- (void)setupTitle {

    CGSize winSize = [CCDirector sharedDirector].winSize;

    NSString *fontName = @"SpaceGameFont.fnt";
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        fontName = @"SpaceGameFont-hd.fnt";
    }

    _titleLabel1 = [CCLabelBMFont labelWithString:@"Space Game" fntFile:fontName];
    _titleLabel1.scale = 0.5;
    _titleLabel1.position = ccp(winSize.width/2, winSize.height * 0.8);
    [self addChild:_titleLabel1 z:100];

    _titleLabel2 = [CCLabelBMFont labelWithString:@"Starter Kit" fntFile:fontName];
    _titleLabel2.scale = 1.25;
    _titleLabel2.position = ccp(winSize.width/2, winSize.height * 0.6);
    [self addChild:_titleLabel2 z:100];

}

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

@end
```

In this listing, the **scene** method creates a default **CCScene**, and adds our **ActionLayer** as a child of the scene. When we tell Cocos2D which scene to run on startup, we'll use this method.

The **setupTitle** is called by the **ActionLayer**'s **init**, and is the most important part of this listing. It first needs to get the size of the window, so we can place the text in the right spot. This is simple to get with Cocos2D—the **CCDirector** singleton class has a property called **winSize** that you can use.

Next, it figures out the name of the file that contains the name of the font file to display. We have two different versions of the font file in the project—one that is sized for the iPhone (**SpaceGameFont.fnt**), and a double sized one to use on the iPhone Retina display and iPad (**SpaceGameFont-hd.fnt**).

It turns out that Cocos2D will automatically look for files ending with "-hd" and use them if you're running on the Retina display (as long as you uncomment a particular line in **AppDelegate.mm** which we'll do soon). But there's nothing that's built-in to do this for the iPad, so we have to put a manual check if it's running on the iPad here.

Next, it creates the two labels to display on the screen. Note it sets their positions as a multiple of the window size. This is better than hardcoding specific offsets, because if we hardcoded offsets we'd have to have different offsets for both the iPhone and the iPad. By basing it on the window size, the same code will work on both devices!

The labels are also scaled differently, so the second line appears bigger than the first line.

We're almost ready to try this out—just need to tell Cocos2D to enable retina display support, and run our new scene on startup!

Open up **AppDelegate.mm**, and at the top of the file, delete the line that reads **#import "HelloWorldScene.h"** and replace it with the following:

```
#import "ActionLayer.h"
```

Next, find the two lines of code that enable Retina display support (inside **applicationDidFinishLaunching**), and uncomment them:

```
if( ! [director enableRetinaDisplay:YES] )
    CCLOG(@"Retina Display Not supported");
```

Finally, at the end of **applicationDidFinishLaunching** replace the final line to run **ActionLayer**'s scene instead of **HelloWorldLayer**'s scene:

```
[[CCDirector sharedDirector] runWithScene:[ActionLayer scene]];
```

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

Note you may have a few warnings due to some files you included in the Art folder – but don't worry about them for now.

Making the Title Sexy

Our title has a cool metal feel to it so far, but in the Space Game Starter Kit, we're committed to making things gratuitously awesome.

Start by modifying the **setupTitle** method inside **ActionLayer.m** to make the title zoom in, instead of just appearing on the screen. The modifications from last time are highlighted in the listing below.

```
- (void)setupTitle {

    CGSize winSize = [CCDirector sharedDirector].winSize;

    NSString *fontName = @"SpaceGameFont.fnt";
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        fontName = @"SpaceGameFont-hd.fnt";
    }

    _titleLabel1 = [CCLabelBMFont labelWithString:@"Space Game" fntFile:fontName];
    _titleLabel1.scale = 0;
    _titleLabel1.position = ccp(winSize.width/2, winSize.height * 0.8);
    [self addChild:_titleLabel1 z:100];
    [_titleLabel1 runAction:
     [CCSequence actions:
      [CCDelayTime actionWithDuration:1.0],
      [CCScaleTo actionWithDuration:1.0 scale:0.5],
     nil]];

    _titleLabel2 = [CCLabelBMFont labelWithString:@"Starter Kit" fntFile:fontName];
    _titleLabel2.position = ccp(winSize.width/2, winSize.height * 0.6);
    _titleLabel2.scale = 0;
    [self addChild:_titleLabel2 z:100];
    [_titleLabel2 runAction:
     [CCSequence actions:
      [CCDelayTime actionWithDuration:2.0],
      [CCScaleTo actionWithDuration:1.0 scale:1.25],
      nil]];

}
```

So now we start out each label with a scale of 0, and use Cocos2D actions to make them zoom in onto the screen.

If you're new to Cocos2D actions, they're really easy to use. You just create an action based on what you want to do—for example jump, rotate, or scale—and pass in the appropriate parameters.

Then you just run the action on the Cocos2D node you want to perform the action with the **runAction** method!

The labels each wait a little bit with **CCDelayTime**, then zoom in for a cool effect. We run both actions in a row by using a special action **CCSequence**.

CCSequence is easy to use—you just list out all the actions you want to run separated by commas, and put nil when you're done. It will then run them one after another, waiting for each to complete before moving to the next one.

So the label run a **CCSequence**, where the first action is to wait for a bit (via **CCDelayTime**), and the second is to zoom into a given scale (via **CCScaleTo**).

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

It's looking much cooler, but isn't quite cool enough. 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, then slows down before stopping. However this crazy text just goes full-speed the entire time!

To get a more natural feel, we're going to set up the text so it moves really quickly when it starts zooming in, then slows down at the end. You can easily do this in Cocos2D by wrapping the action in a **CCEaseOut** action. This makes the action slow down at the end of the action.

So modify **setupTitle** one last time, replacing the **CCScaleTo** lines to make use of the **CCEaseOut** actions, as highlighted below:

```
- (void)setupTitle {

    CGSize winSize = [CCDirector sharedDirector].winSize;

    NSString *fontName = @"SpaceGameFont.fnt";
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        fontName = @"SpaceGameFont-hd.fnt";
    }

    _titleLabel1 = [CCLabelBMFont labelWithString:@"Space Game" fntFile:fontName];
    _titleLabel1.scale = 0;
    _titleLabel1.position = ccp(winSize.width/2, winSize.height * 0.8);
    [self addChild:_titleLabel1 z:100];
    [_titleLabel1 runAction:
     [CCSequence actions:
      [CCDelayTime actionWithDuration:1.0],
      [CCEaseOut actionWithAction:
       [CCScaleTo actionWithDuration:1.0 scale:0.5] rate:4.0],
      nil]];

    _titleLabel2 = [CCLabelBMFont labelWithString:@"Starter Kit" fntFile:fontName];
    _titleLabel2.position = ccp(winSize.width/2, winSize.height * 0.6);
    _titleLabel2.scale = 0;
    [self addChild:_titleLabel2 z:100];
    [_titleLabel2 runAction:
     [CCSequence actions:
      [CCDelayTime actionWithDuration:2.0],
      [CCEaseOut actionWithAction:
       [CCScaleTo actionWithDuration:1.0 scale:1.25] rate:4.0],
      nil]];

}
```

Note that **CCEaseOut** takes two parameters—the action to modify, and the rate at which to ease the action out. You can play around with the rate to get the feeling you like, but often I start with 4.0 and tweak from there.

If you run the app again, you should see a subtle improvement in how the text zooms in!

Gratuitous Music and Sound Effects

We're going to start our game out with a bang by adding some awesome music and sound effects in, right from the beginning!

I've already created some sound effects for you, which you can find under the **Art\Sounds** folder. You've already added these to your project, so now you just have to add the code to play them!

> ***Note:*** If you're wondering how I made these sound effects, 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.

Luckily, playing sound effects is extremely simple with Cocos2D, so let's dive right in. Start by adding this import to the top of **ActionLayer.m** so you can use **SimpleAudioEngine** to play sounds:

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

Then add this new method right above init to preload all of the sound effects, and start the background music playing:

```
- (void)setupSound {
    [[SimpleAudioEngine sharedEngine] playBackgroundMusic:@"SpaceGame.caf" loop:YES];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"explosion_large.caf"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"explosion_small.caf"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"laser_enemy.caf"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"laser_ship.caf"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"shake.caf"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"powerup.caf"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"boss.caf"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"cannon.caf"];
    [[SimpleAudioEngine sharedEngine] preloadEffect:@"title.caf"];
}
```

Now that you've written this method, be sure to call it by adding the following line to the *beginning* of your **init** method:

```
[self setupSound];
```

And finally, we want to play a sound effect when the title appears, add one more action to the sequence of actions on **titleLabel1** in **setupTitle**:

```
[_titleLabel1 runAction:
     [CCSequence actions:
      [CCDelayTime actionWithDuration:1.0],
      [CCCallFunc actionWithTarget:self selector:@selector(playTitleEffect)],
      [CCEaseOut actionWithAction:
       [CCScaleTo actionWithDuration:1.0 scale:0.5] rate:4.0],
     nil]];
```

Here we set things up so that after the delay, it calls a method that should play the Space Game Starter Kit title sound effect. But of course we have to actually write that method, so add the following new method next:

```
- (void)playTitleEffect {
    [[SimpleAudioEngine sharedEngine] playEffect:@"title.caf"];
}
```

Pretty easy, eh? Compile and run your code, and now your game is starting to have some style!

Shooting Stars

However right now our game has a big problem. How can we have a space game without stars?!

We want to have some stars shooting from the right hand side of the screen to the left side of the screen, to make it feel like you're flying through space.

There are a couple ways we could implement this:

- We could have a background with a lot of stars drawn on it and move it from right to left. This would be easy, but the disadvantage is we'd need a large background, taking up texture memory.
- We could create a sprite for each star, and make it move from right to left. This would save a lot of texture memory since we'd only need one small texture per type of star, but our FPS would suffer because having a lot of sprites on the screen at once is expensive.
- Or we 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 we need!

The easiest way to create particle systems is to use a tool called Particle Designer. I've used particle designer to create a particle system that shoots stars from the right to left for you, in **Art\Particles**.

Note that there are three different particle systems to shoot stars (**Stars1-3.plist**), because there are three different types of star images, and each particle system can only use one image.

(Optional) If you're curious how these works or would like to tweak the settings, you can download Particle Designer and play around with them. If you need them, the raw images of the stars are located in **Raw\Particle System Art**.

For the purposes of this tutorial, you can use the particle system as-is. It's really easy! Start by adding the following new method right above **init** in **ActionLayer.m**:

```
- (void)setupStars {
    CGSize winSize = [CCDirector sharedDirector].winSize;
    NSArray *starsArray = [NSArray arrayWithObjects:@"Stars1.plist", @"Stars2.plist",
@"Stars3.plist", nil];
    for(NSString *stars in starsArray) {
        CCParticleSystemQuad *starsEffect = [CCParticleSystemQuad
particleWithFile:stars];
        starsEffect.position = ccp(winSize.width*1.5, winSize.height/2);
        starsEffect.posVar = ccp(starsEffect.posVar.x, (winSize.height/2) * 1.5);
        if (UI_USER_INTERFACE_IDIOM() != UIUserInterfaceIdiomPad) {
            starsEffect.scale = 0.5;
            starsEffect.posVar = ccpMult(starsEffect.posVar, 2.0);
        }
        [self addChild:starsEffect];
    }
}
```

So this method first creates an array with the three files defining the particle systems we want to use. Then it iterates through the array, and for each file it creates a **CCParticleSystemQuad** with that file.

It sets the position of the particle system to be offscreen to the right (1.5X the width) and in the middle of the screen. It also modifies the Y-variance of how each particle (i.e. star) spawns to be half the height of the screen, times 1.5.

This makes it so that the star field is still visible even if we zoom the layer out a bit, which we'll be doing later on in this game to create a neat effect. If we made the particle system just match the exact screen dimensions, it would look really weird, like stars were only in one particular rectangle in space!

Finally, note that the particle system was made with the iPad screen size in mind. So if we're not running on an iPad, it halves the scale of the particle system—and has to double the position variance to make up for that.

One last step! Simply add the line to call this to the bottom of your **init** method:

```
[self setupStars];
```

Compile and run, and now it's starting to look like an actual space game!

Adding a Main Menu

Our game has a cool title and some cool sound effects and music, but right now there's absolutely nothing to do!

So we'll add a simple main menu to the game with a single item that says "Play". When the user taps it, the game can begin.

Start by opening **ActionLayer.h**, and add an instance variable to keep track of the menu item for Play:

```
CCMenuItemLabel *_playItem;
```

Then open up **ActionLayer.m** and add the following code to the bottom of **setupTitle**:

```
CCLabelBMFont *playLabel = [CCLabelBMFont labelWithString:@"Play" fntFile:fontName];
_playItem = [CCMenuItemLabel itemWithLabel:playLabel target:self
selector:@selector(playTapped:)];
_playItem.scale = 0;
_playItem.position = ccp(winSize.width/2, winSize.height * 0.3);

CCMenu *menu = [CCMenu menuWithItems:_playItem, nil];
menu.position = CGPointZero;
[self addChild:menu];

[_playItem runAction:
 [CCSequence actions:
  [CCDelayTime actionWithDuration:2.0],
  [CCEaseOut actionWithAction:
   [CCScaleTo actionWithDuration:0.5 scale:0.5] rate:4.0],
  nil]];
```

This creates a **CCLabelBMFont** that reads "Play", and creates a **CCMenuItemLabel** based on this label. It positions the menu item to be in the middle of the screen along the x-axis, and slightly below the title text along the y-axis. It also sets the scale to 0, because at the bottom of the method it runs an action to make it zoom in, just like we did for the title.

Next, it makes a **CCMenu** with the single menu item we just created. It sets the menu at **CGPointZero** so that the coordinates of menu items are with respect to the bottom left of the screen.

The **CCMenuItemLabel** is set up to call a method called **playTapped** on the current object (**ActionLayer**) when it's tapped. So add the code for that next, right above **setupTitle**:

```
- (void)playTapped:(id)sender {

    [[SimpleAudioEngine sharedEngine] playEffect:@"powerup.caf"];

    NSArray * nodes = [NSArray arrayWithObjects:_titleLabel1, _titleLabel2, _playItem,
nil];
    for (CCNode *node in nodes) {
        [node runAction:
         [CCSequence actions:
          [CCEaseOut actionWithAction:
           [CCScaleTo actionWithDuration:0.5 scale:0] rate:4.0],
          [CCCallFuncN actionWithTarget:self selector:@selector(removeNode:)],
          nil]];
    }

}
```

For now, when the play button is tapped we're going to play a sound effect (**powerup.caf**) and make the title and menu item zoom out.

To avoid duplicating code, it puts each label and the menu item in 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, we need to remove it from the layer. Otherwise, we'd have these invisible Cocos2D nodes sitting around consuming resources. So the last thing we put in the **CCSequence** is a **CCCallFuncN** action, which will call the **removeNode** method as each node runs an action to scale the menu items down to nothing, passing in the node that just finished scaling down as a parameter.

So the last step is to add this method (right above **playTapped:**), which is very simple:

```
- (void)removeNode:(CCNode *)sender {
    [sender removeFromParentAndCleanup:YES];
}
```

This removes the passed-in Cocos2D node from the layer so it's no longer consuming resources, since it's no longer needed.

Compile and run, and you should see your new Play menu item. Tap it, and everything should zoom out. You're now ready to start work on the main action!

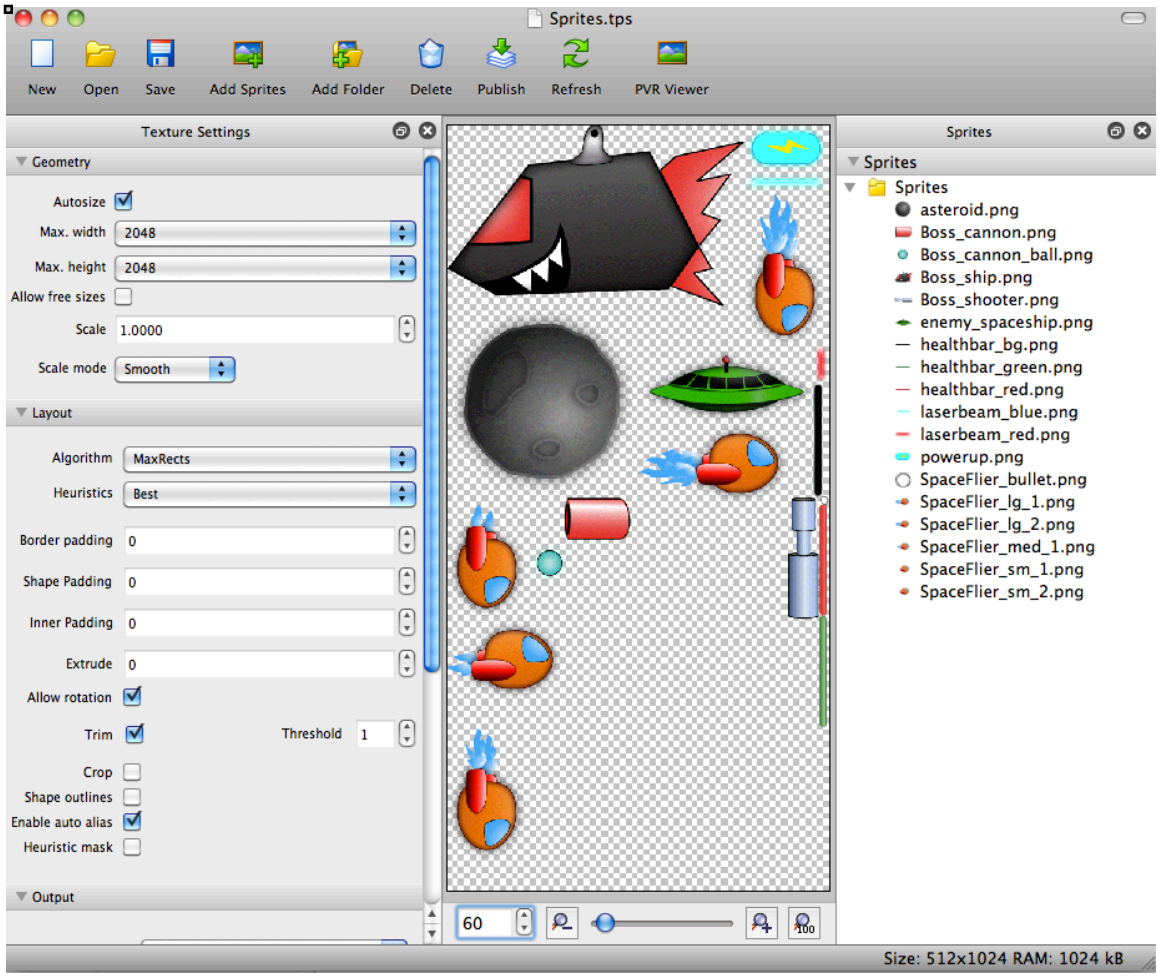
Adding Our Space Ship

The first thing we need in a space game is a space ship to pilot around! So in this section, you'll add our hero into the scene.

I put all of the art you need for the game (except for some of the background space objects) into a sprite sheet that I created with a tool called Texture Packer. You can find the sprite sheets under **Art\Spritesheets**.

> ***Note:*** The sprite sheets are saved in the .pvr.ccz format. This is my favorite format to use when adding images for Cocos2D games these days. I like the format because it's compressed, which means your app is a smaller size (which helps for over the air downloads and launch time). Also the PVR image format loads quickly on the iPhone, and with it you don't have to specify the pixel format you use because it's part of the file.
>
> The drawback is you can't preview a pvr.ccz from Xcode itself. The only way to look at it (that I know of) is to use Texture Packer.

(Optional) If you're curious how these were made, you can open up the Texture Packer file from **Raw\Sprites.tps** and play around with it yourself:



For the purposes of this tutorial, you can use the sprite sheet as-is. There are three steps to use a sprite sheet in Cocos2D:

1. Create a **CCSpriteBatchNode**, specifying the image generated by TexturePacker (**Sprites.pvr.ccz** in this case).
2. Use the **CCSpriteFrameCache** to load the plist generated by TexturePacker (**Sprites.plist** in this case).
3. When adding sprites, you should add them as children of the **CCSpriteBatchNode** (not children of the layer!) This way, you will get the performance benefit that **CCSpriteBatchNode** offers.

Let's try this out. Start by opening **ActionLayer.h** and add the following instance variables:

```
CCSpriteBatchNode *_batchNode;
CCSprite *_ship;
```

Here we're storing a reference to the **CCSpriteBatchNode** and the **CCSprite** for the space ship, since we'll need to use those in a bunch of different places.

Next, switch to **ActionLayer.m** and add the following method right above **init**:

```
- (void)setupBatchNode {
    NSString *spritesPvrCcz = @"Sprites.pvr.ccz";
    NSString *spritesPlist = @"Sprites.plist";
    if (UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) {
        spritesPvrCcz = @"Sprites-hd.pvr.ccz";
        spritesPlist = @"Sprites-hd.plist";
    }

    _batchNode = [CCSpriteBatchNode batchNodeWithFile:spritesPvrCcz];
    [self addChild:_batchNode z:-1];
    [[CCSpriteFrameCache sharedSpriteFrameCache]
addSpriteFramesWithFile:spritesPlist];
}
```

The first few lines figure out which version of the pvr.ccz and plist to load. On the iPhone/iPod touch, it loads the normal version, but on the iPad it runs the –hd version.

The last few lines create the **CCSpriteBatchNode** and add it to the layer (as in step 1 above) and load the plist into the **CCSpriteFrameCache** (as in step 2 above).

Now that you have the method to setup the batch node, you need to call it. Add the following line to the bottom of **init**:

```
[self setupBatchNode];
```

You could actually compile and run at this point if you wanted, but it wouldn't be very interesting because nothing would look different. Although you've added a **CCSpriteBatchNode** to the layer, nothing will show up until you add a sprite from the sprite sheet.

And the first sprite we're going to add is the space ship! Add the following method right above **playTapped**:

```
- (void)spawnShip {

    CGSize winSize = [CCDirector sharedDirector].winSize;

    _ship = [CCSprite spriteWithSpriteFrameName:@"SpaceFlier_sm_1.png"];
    _ship.position = ccp(-_ship.contentSize.width/2,
                         winSize.height * 0.5);
    [_batchNode addChild:_ship z:1];

    [_ship runAction:
     [CCSequence actions:
      [CCEaseOut actionWithAction:
       [CCMoveBy actionWithDuration:0.5
                           position:ccp(_ship.contentSize.width/2 + winSize.width*0.3,
0)]
                               rate:4.0],
      [CCEaseInOut actionWithAction:
       [CCMoveBy actionWithDuration:0.5
                           position:ccp(-winSize.width*0.2, 0)]
                               rate:4.0],
      nil]];

}
```

To create a sprite that is part of a sprite sheet, it needs to use the spriteWithSpriteFrameName method to initialize the sprite. Then it sets the position of the sprite to be offscreen to the left along the x-axis, and in the middle of the screen along the y-axis.

> ***Note:*** If you're confused how –ship.contentSize.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 where the center of the sprite is. So 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 it fully offscreen, you move it to the left half the width of the sprite. And you can get the width of the ship by ship.contentSize. Voila!

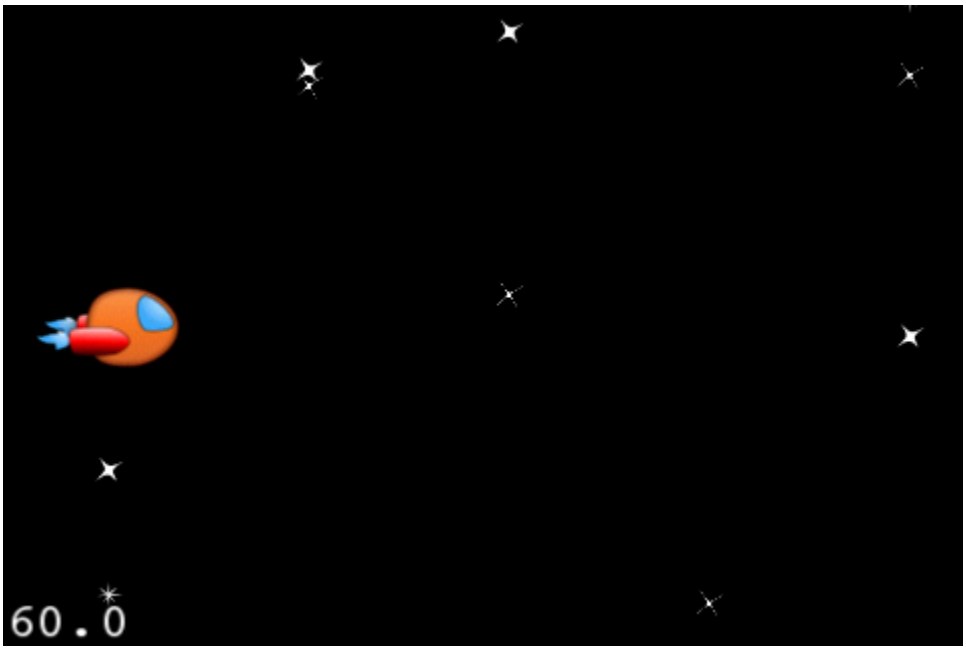
Also note that the sprite is added as a child of the batch node (not a child of the layer). This gives the performance benefits of using a **CCSpriteBatchNode**, which can run batch drawing commands from multiple sprites using the same texture.

After creating the sprite and adding it to the batch node, it runs a sequence of actions to move the ship into the scene by moving it forward, and then back a bit. It uses **CCEaseOut** and **CCInOut** actions to make this a bit smoother, as discussed earlier in this tutorial.

Almost done—just need to call this method when the user taps the Play button. So add the following to the bottom of **playTapped**:

```
[self spawnShip];
```

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



Animating the Ship

If you look at the art for the game under **Raw\Sprites**, 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 just displaying the first image like we're doing right now, it would be a lot cooler if we had it animating!

Luckily animating a sprite is ridiculously easy in Cocos2D. There are just two steps:

1. Create a **CCAnimation**, specifying the images that make up the animation.
2. Create a **CCAnimate** action and run it on the sprite, specifying the **CCAnimation** you created earlier.

Note that the **CCAnimate** action runs the animation only once, so a lot of times you want to wrap it in a **CCRepeatForever** action so it keeps going until you tell it to stop.

Let's try this out! Add the following code to the end of the **spawnShip** method:

```
CCSpriteFrameCache * cache =
    [CCSpriteFrameCache sharedSpriteFrameCache];

CCAnimation *animation = [CCAnimation animation];
[animation addFrame:
    [cache spriteFrameByName:@"SpaceFlier_sm_1.png"]];
[animation addFrame:
    [cache spriteFrameByName:@"SpaceFlier_sm_2.png"]];
animation.delay = 0.2;

[_ship runAction:
 [CCRepeatForever actionWithAction:
  [CCAnimate actionWithAnimation:animation]]];
```

This creates a new animation and adds both sprite frames to it, according to step 1 above. Then it runs a new **CCAnimate** action on the ship, wrapped in a **CCRepeatForeverAction**, according to step 2 above.

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

Moving with the Accelerometer

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

So let's make the ship move based on how we tilt our device. If we tilt it downwards (so it's more flat), we'll have the ship move up, and if we tilt the device upwards (so it's move vertical) we'll have the ship move down.

To receive data from the accelerometer, add the following line at the bottom of your **init** method:

```
self.isAccelerometerEnabled = YES;
```

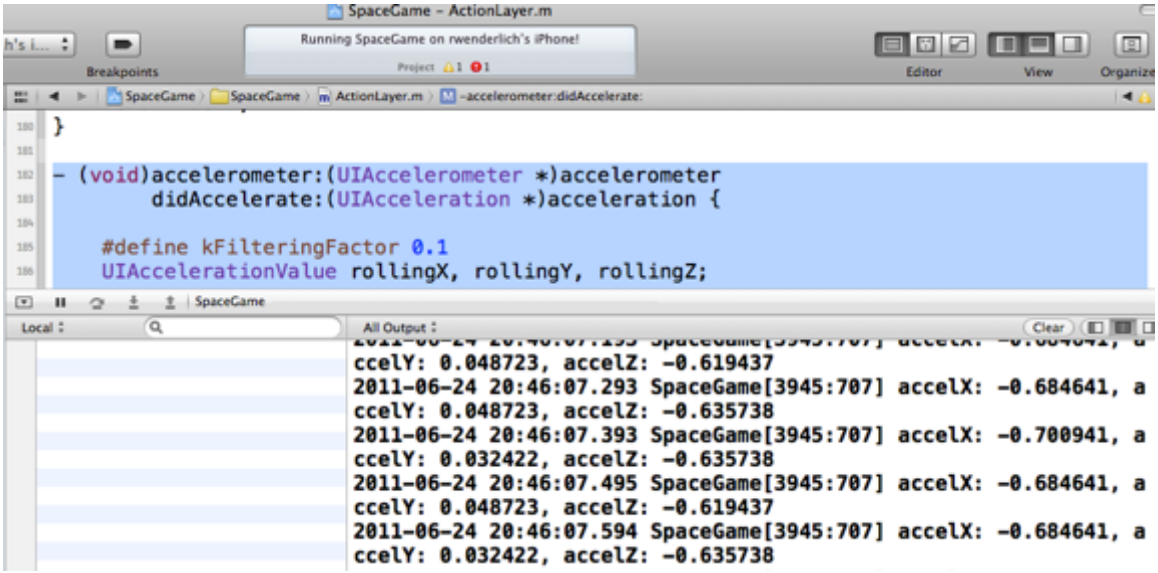
By adding this line, you can now implement the **accelerometer:didAccelerate** method and receive information from the accelerometer periodically. For now, let's just implement this to display what we get:

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
        didAccelerate:(UIAcceleration *)acceleration {

    #define kFilteringFactor 0.75
    static UIAccelerationValue rollingX = 0, rollingY = 0, rollingZ = 0;

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

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

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

This code comes directly from Apple sample code, to filter the accelerometer values so it's not so "jiggly". Don't worry if you don't understand this, all you really need to know is that it makes things more smooth. 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, we just print out the (filtered) values for x, y, and z so we can see what we get so far.

Compile 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:



You'll notice that as you tilt your iPhone up and down, accelX goes up and down. This is what we're going to base the ship movement on.

We'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.

The strategy we're going to use is just update a variable for how fast the ship should be moving as we get accelerometer input, and then use this variable to move the ship the appropriate amount every frame later.

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! First switch to ActionLayer.h and add the instance variable we need to keep track of how fast the ship should move up or down:

```
float _shipPointsPerSecY;
```

Then add the code to set this at the bottom of **accelerometer:didAccelerate**:

```
CGSize winSize = [CCDirector sharedDirector].winSize;

#define kRestAccelX 0.6
#define kShipMaxPointsPerSec (winSize.height*0.5)
#define kMaxDiffX 0.2

float accelDiffX = kRestAccelX - ABS(accelX);
float accelFractionX = accelDiffX / kMaxDiffX;
float pointsPerSecX = kShipMaxPointsPerSec * accelFractionX;

_shipPointsPerSecY = pointsPerSecX;
```

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 will be be near 0 when it's close to 0.6, and near 1.0 when it's close to 0.8, for example.

Finally, we set the points per second to move equal to the maximum speed to move (half the height of the screen per second) times the acceleration fraction. Voila!

Now you need to add the code to move the spaceship according to **_shipPointsPerSecY**. Start by adding the following to the bottom of your init method:

```
[self scheduleUpdate];
```

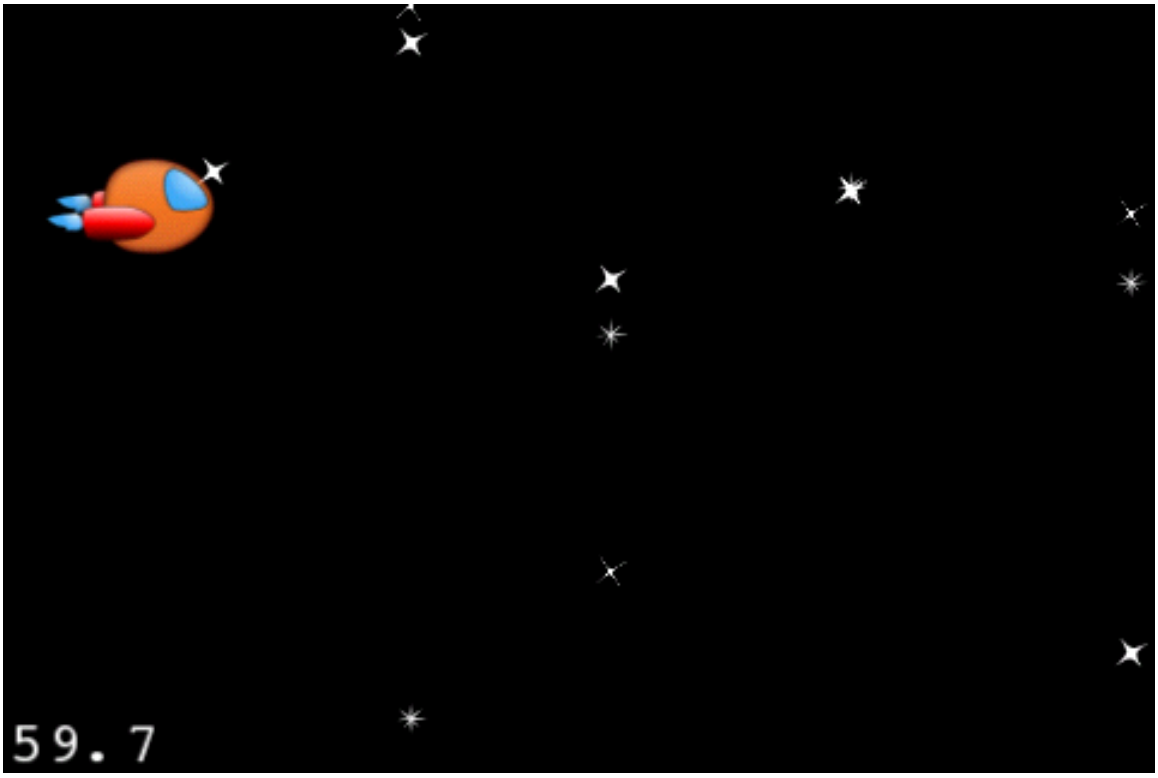
This makes it so that your **update** method will be called every frame, with a parameter of how much time has passed since last frame (delta time, or **dt**). Let's implement this to move the spaceship:

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

    CGSize winSize = [CCDirector sharedDirector].winSize;

    float maxY = winSize.height - _ship.contentSize.height/2;
    float minY = _ship.contentSize.height/2;

    float newY = _ship.position.y + (_shipPointsPerSecY * dt);
    newY = MIN(MAX(newY, minY), maxY);
    _ship.position = ccp(_ship.position.x, newY);

}

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

This simply multiplies the **_shipPointsPerSecY** times the delta time to figure out how much to move the space ship for this frame, and adds it to the ship's position.

It also has some code to prevent the ship from moving past the top or bottom edge of the screen.

You're done! Compile and run the code, and now you should be able to move your space ship 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

We have tunes, we have space ships, but we are missing something major—things to smash!

Now it's time to add the first enemy type to our game—asteroids!

We're going to take a very simple strategy:

- Every so often, we'll spawn an asteroid.

- We'll start it offscreen to the right, at a random y-value between the top and bottom of the screen.
- We'll move the asteroid from the right to the left, at a random speed.
- For kicks, we'll also make the asteroids different sizes!

Before we can begin, we need to add a helper method to help us get a random number between two float values. We'll be needing this to figure out random times to spawn asteroids, etc.

Since this is a helper method that more than one class might want to use, we're going to define this in a separate file. Create it by going to **File\New\New File**, choose **iOS\Cocoa Touch\Objective-C class**, and click **Next**. Enter **NSObject** for Subclass of, click **Next**, name the new file **Common.m**, and click Save.

Once you've created the file, open up **Common.h** and replace the contents with the following:

```
float randomValueBetween(float low, float high);
```

If you're wondering why this function signature looks different than you're used to, that's because this is a C function, not an Objective-C function. But you can call C functions from Objective-C, so we can use this in our code with no problems.

Next move to **Common.m** and replace the contents with the following:

```
#import "Common.h"

float randomValueBetween(float low, float high) {
    return (((float) arc4random() / 0xFFFFFFFFu)
        * (high - low)) + low;
}
```

You don't need to worry about how this works—just know that you can use this function to give you a random number between two floats. So you could pass in **randomValueBetween**(0, 1.0) and it might give you 0.475 for example.

Now open **ActionLayer.h** and add a new instance variable we'll use to keep track of the next time we should spawn an asteroid:

```
double _nextAsteroidSpawn;
```

Then switch to **ActionLayer.m** and import **Common.h** so you can use this new method:

```
#import "Common.h"
```

Finally, it's time for the fun part! In **ActionLayer.m**, add a new method right above **update**:

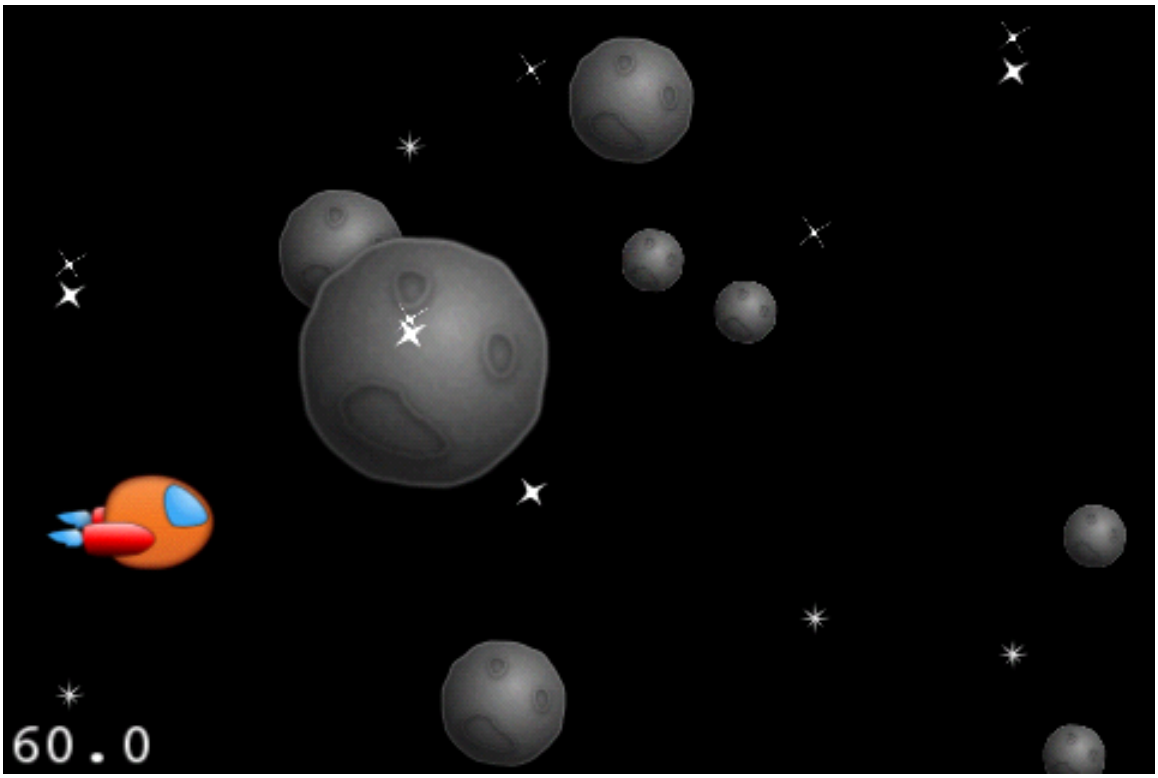
```
- (void)updateAsteroids:(ccTime)dt {
    CGSize winSize = [CCDirector sharedDirector].winSize;
    // Is it time to spawn an asteroid?
    double curTime = CACurrentMediaTime();
    if (curTime > _nextAsteroidSpawn) {

        // Figure out the next time to spawn an asteroid
        float randSecs = randomValueBetween(0.20, 1.0);
        _nextAsteroidSpawn = randSecs + curTime;

        // Figure out a random Y value to spawn at
        float randY = randomValueBetween(0.0,
            winSize.height);

        // Figure out a random amount of time to move
        // from right to left
        float randDuration = randomValueBetween(2.0, 10.0);

        // Create a new asteroid sprite
        CCSprite *asteroid = [CCSprite spriteWithSpriteFrameName:@"asteroid.png"];
        [_batchNode addChild:asteroid];

        // Set its position to be offscreen to the right
        asteroid.position = ccp(winSize.width+asteroid.contentSize.width/2, randY);

        // Set it's size to be one of 3 random sizes
        int randNum = arc4random() % 3;
        if (randNum == 0) {
            asteroid.scale = 0.25;
        } else if (randNum == 1) {
            asteroid.scale = 0.5;
        } else {
            asteroid.scale = 1.0;
        }

        // Move it offscreen to the left, and when it's
        // done call removeNode
        [asteroid runAction:
         [CCSequence actions:
          [CCMoveBy actionWithDuration:randDuration position:ccp(-winSize.width-
asteroid.contentSize.width, 0)],
          [CCCallFuncN actionWithTarget:self selector:@selector(removeNode:)],
          nil]];
    }
}
```

I've added comments above each section to explain what it's doing. By now, this code should look familiar to you, because it's used several concepts we've already covered before!

One last step—just add the following to the bottom of **update** to call your new method:

```
[self updateAsteroids:dt];
```

Compile and run your code, and now you can navigate your ship through a random asteroid belt!



Sprite Caching

There's a subtle problem with the above code, that probably doesn't make a big difference now, but will the more complicated your game gets.

The problem is the game is currently continually creating and destroying asteroids as they come into the scene and leave the scene, over and over.

Each time it creates a new asteroid, it has to allocate fresh memory—and this is an expensive operation on the iPhone.

It would be more efficient to simply pre-create a certain number of asteroids, and when one goes offscreen we can re-use it by moving it to the right, and sending it across again.

We're going to create a little helper class to help us manage this array of sprites called—you guessed it—**SpriteArray**. It will have a method to create an array of sprites and add them to the batch node, and another to get the next available sprite.

Not only is this simple to write, but it also makes our life easier later on since we'll have a collection of all of the asteroids we can iterate through.

So let's get started! Go to **File\New\New File**, choose **iOS\Cocoa Touch\Objective-C class**, and click **Next**. Enter **NSObject** for Subclass of, click **Next**, name the new file **SpriteArray.m**, and click Save.

Open up **SpriteArray.h** and replace the contents with the following:

```
#import "cocos2d.h"

@interface SpriteArray : NSObject {
    CCArray * _array;
    int _nextItem;
}

@property (readonly) CCArray * array;

- (id)initWithCapacity:(int)capacity spriteFrameName:(NSString *)spriteFrameName
batchNode:(CCSpriteBatchNode *)batchNode;
- (id)nextSprite;

@end
```

This class derives from **NSObject** and has two instance variables—one for the array of sprites, and one that keeps track of the next available sprite.

Note that this uses **CCArray** for the array of sprites, instead of what you might be used to (**NSMutableArray**). **CCArray** is a helper class that comes with Cocos2D that has an API very similar to **NSMutableArray**, but it's optimized for speed.

We also declare a property for the array (so other classes can get access to it), an initializer, and a helper method to get the next available sprite.

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

```
#import "SpriteArray.h"

@implementation SpriteArray
@synthesize array = _array;

- (id)initWithCapacity:(int)capacity spriteFrameName:(NSString *)spriteFrameName
batchNode:(CCSpriteBatchNode *)batchNode {

    if ((self = [super init])) {

        _array =
            [[CCArray alloc] initWithCapacity:capacity];
        for(int i = 0; i < capacity; ++i) {
            CCSprite *sprite = [CCSprite
                spriteWithSpriteFrameName:spriteFrameName];
            sprite.visible = NO;
            [batchNode addChild:sprite];
            [_array addObject:sprite];
        }

    }
    return self;

}

- (id)nextSprite {
    id retval = [_array objectAtIndex:_nextItem];
    _nextItem++;
    if (_nextItem >= _array.count) _nextItem = 0;
    return retval;
}

- (void)dealloc {
    [_array release];
    _array = nil;
    [super dealloc];
}

@end
```

The initializer creates a new array and fills it with a number of sprites, specified via the **capacity** variable. For each sprite, it sets it to initially invisible and adds it to the batch node and the array.

nextSprite is a simple helper method that gets the next available sprite and returns it—advancing the **_nextItem** variable along the way. It returns the item as an **id** type (which

means "any object") just to avoid having to specify the type of item in the array in more than one place.

Now let's make use of this to preload our asteroids! Start by importing the helper class at the top of **ActionLayer.h**:

```
#import "SpriteArray.h"
```

Then add an instance variable for the array of asteroids:

```
SpriteArray * _asteroidsArray;
```

Switch to **ActionLayer.m** and add a method to initialize this array right above the **init** method:

```
- (void)setupArrays {
    _asteroidsArray = [[SpriteArray alloc] initWithCapacity:30
spriteFrameName:@"asteroid.png" batchNode:_batchNode];
}
```

This creates an array of 30 asteroids. We want to make this as small as we can, but as large as we need. We probably won't have more than 30 asteroids on the screen at a time so this is a good number to use.

Next, call this method at the bottom of **init**:

```
[self setupArrays];
```

Now that we have an array of asteroids set up, we just need to modify our existing code to use it instead of always creating a new sprite. Start by modifying **updateAsteroids**—replace the lines under "Create a new asteroid sprite" with the following:

```
CCSprite *asteroid = [_asteroidsArray nextSprite];
[asteroid stopAllActions];
asteroid.visible = YES;
```

Here we use the helper method to get the next available sprite from the array, stop any actions that may be currently running on it, and set it to visible.

Still in **updateAsteroids**, replace the **[asteroid runAction:...]** line to run **invisNode**: instead of **removeNode**:

```
[asteroid runAction:
 [CCSequence actions:
  [CCMoveBy actionWithDuration:randDuration position:ccp(-winSize.width-
asteroid.contentSize.width, 0)],
  [CCCallFuncN actionWithTarget:self selector:@selector(invisNode:)],
  nil]];
```

We're making this replacement because we don't want to remove the sprite from the batch node now that we're reusing sprites—we just want to make it invisible. Add the new **invisNode** method next to do exactly that:

```
- (void)invisNode:(CCNode *)sender {
    sender.visible = FALSE;
}
```

One final step! In **setupArrays** you created a variable with **alloc/init**, so you need to **release** it in **dealloc**:

```
- (void)dealloc {
    [_asteroidsArray release];
    [super dealloc];
}
```

Phew—finally done! Compile and run your code, and you won't notice any visual difference—but there's a lot less allocation going on.

We'll be reusing this method for many other arrays of items in our space game in the future—including lasers, which come next!

Lasers Go Pew, Pew!

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

As mentioned in the last section, rather than continuously creating laser sprites each time the user taps, we'll precreate an array of lasers and just reuse old ones when we need them.

You know the drill by now! Start by declaring an instance variable for the lasers array in **ActionLayer.h**:

```
SpriteArray * _laserArray;
```

Then add this line to the bottom of **setupArrays** to initialize the array with 15 laser sprites:

```
_laserArray = [[SpriteArray alloc] initWithCapacity:15
spriteFrameName:@"laserbeam_blue.png" batchNode:_batchNode];
```

And before we forget, add a line to **dealloc** to release the array:

```
[_laserArray release];
```

OK, onto the fun stuff. We want to shoot lasers whenever the user taps the screen, so the first thing we need to do is turn on receiving touch events by adding the following line to the bottom of **init**:

```
self.isTouchEnabled = YES;
```

By adding this line, we'll start to receive touch callbacks such as **ccTouchesBegan**, **ccTouchesMoved**, etc. All we need for this game is **ccTouchesBegan**—when the user first begins to tap down:

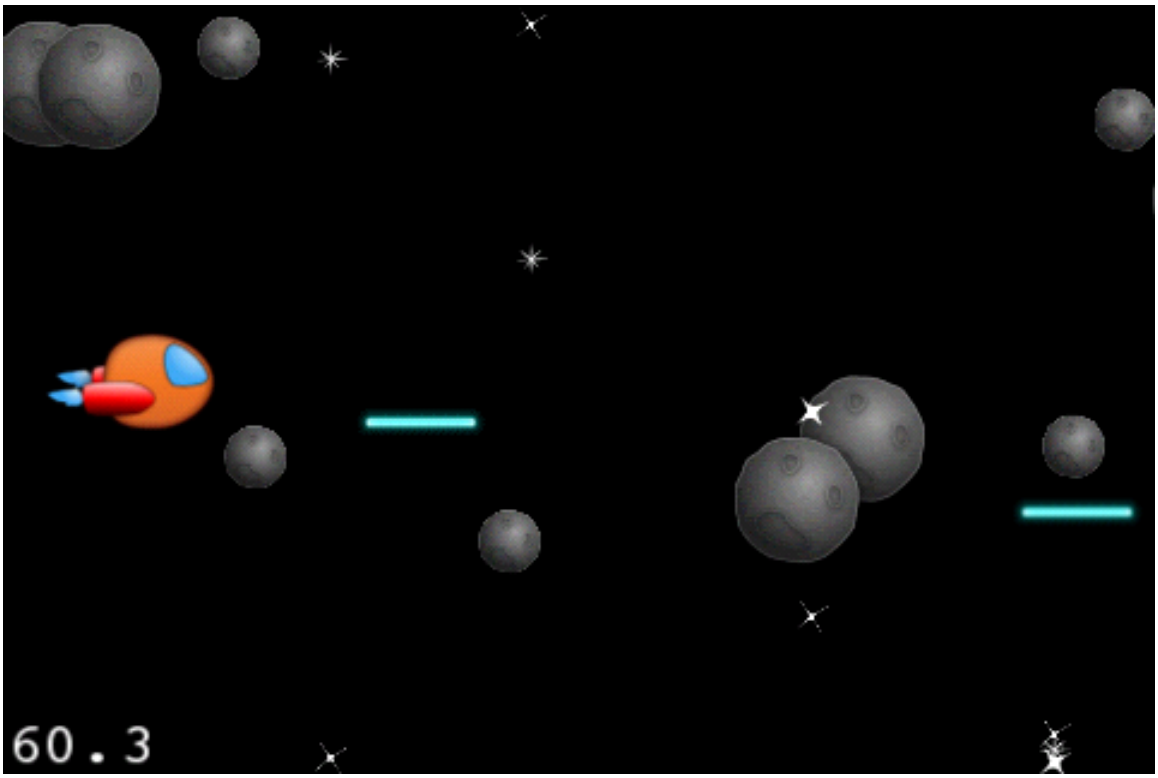
```
- (void)ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

    CGSize winSize = [CCDirector sharedDirector].winSize;

    [[SimpleAudioEngine sharedEngine]
        playEffect:@"laser_ship.caf" pitch:1.0f pan:0.0f
        gain:0.25f];

    CCSprite *shipLaser = [_laserArray nextSprite];
    [shipLaser stopAllActions];
    shipLaser.visible = YES;

    shipLaser.position = ccpAdd(_ship.position,
        ccp(shipLaser.contentSize.width/2, 0));
    [shipLaser runAction:
     [CCSequence actions:
      [CCMoveBy actionWithDuration:0.5
           position:ccp(winSize.width, 0)],
      [CCCallFuncN actionWithTarget:self
           selector:@selector(invisNode:)],
      nil]];

}
```

This plays a laser sound effect when the user taps, gets the next available laser sprite, stops any running actions, and sets it to visible.

It then sets its position to be to the right of the space ship, 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 only important thing is that it goes past the edge of the screen.

When the laser is done moving, it calls **invisNode** passing itself as a parameter so it will be set to **invisible**.

That's it! Pretty easy since we had that reusable SpriteArray class, eh? Compile and run your code, and go pew-pew!



Basic Collision Detection

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

So let's add some basic collision detection to tell if a laser hits any asteroids, and if so destroy both the laser and the asteroid.

For this first tutorial, we're going to use the bounding box of the sprite for collision detection. The bounding box is a square around the sprite, which includes transparent space.

So sometimes it will register a collision when there isn't really one. That's annoying so in the second tutorial we'll cover a better (but more advanced) method.

But for now, onwards to the simple solution! Add the following new method right above the **update** method:

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

    for (CCSprite *laser in _laserArray.array) {
        if (!laser.visible) continue;

        for (CCSprite *asteroid in _asteroidsArray.array) {
            if (!asteroid.visible) continue;

            if (CGRectIntersectsRect(asteroid.boundingBox, laser.boundingBox)) {

                [[SimpleAudioEngine sharedEngine] playEffect:@"explosion_large.caf"
pitch:1.0f pan:0.0f gain:0.25f];
                asteroid.visible = NO;
                laser.visible = NO;
                break;
            }
        }
    }

}
```

This loops through each laser, and each asteroid, and checks if their bounding boxes collide (and both are visible).

If so, it plays an explosion sound effect, and makes both of them invisible to "destroy" them.

It's as simple as that! Just add the line to call this new method to the bottom of **update**:

```
[self updateCollisions:dt];
```

Compile and run, and have fun blasting some asteroids!

Parallax Scrolling

Our game is starting to look great, we have a hero, enemies, and blasting! But in the Space Game Starter Kit, we don't just want great—we want uberly awesome.

And what could be more uberly awesome than a parallax scrolling space background?!

If you aren't familiar with parallax scrolling, is just a fancy way of saying, "move some parts of the background more slowly than the other parts." If you've ever played SNES games like ActRaiser, you'll often see this in the background of the action levels.

It's really easy to use parallax scrolling in Cocos2D. You just have to do three steps:

1. Create a **CCParallaxNode**, and add it to the layer.
2. Create items you wish to scroll, and add them to the **CCParallaxNode** with **addChild:parallaxRatio:positionOffset**.
3. Move the **CCParallaxNode** to scroll the background. It will scroll the children of the **CCParallaxNode** more quickly or slowly based on what you set the **parallaxRatio** to.

Let's see how this works. Start by opening **ActionLayer.h**, and add the following inside the @interface:

```
CCParallaxNode *_backgroundNode;
CCSprite *_spacedust1;
CCSprite *_spacedust2;
CCSprite *_planetsunrise;
CCSprite *_galaxy;
CCSprite *_spacialanomaly;
CCSprite *_spacialanomaly2;
```

Then switch to **ActionLayer.m**, and add the following new method right before **init**:

```
- (void)setupBackground {

    CGSize winSize = [CCDirector sharedDirector].winSize;

    // 1) Create the CCParallaxNode
    _backgroundNode = [CCParallaxNode node];
    [self addChild:_backgroundNode z:-2];

    // 2) Create the sprites we'll add to the
    // CCParallaxNode
    if (UI_USER_INTERFACE_IDIOM() ==
      UIUserInterfaceIdiomPad) {
        _spacedust1 = [CCSprite
            spriteWithFile:@"bg_front_spacedust-hd.png"];
        _spacedust2 = [CCSprite
            spriteWithFile:@"bg_front_spacedust-hd.png"];
        _planetsunrise = [CCSprite
            spriteWithFile:@"bg_planetsunrise-hd.png"];
        _galaxy = [CCSprite
            spriteWithFile:@"bg_galaxy-hd.png"];
        _spacialanomaly = [CCSprite
            spriteWithFile:@"bg_spacialanomaly-hd.png"];
        _spacialanomaly2 = [CCSprite
            spriteWithFile:@"bg_spacialanomaly2-hd.png"];
        _spacedust1.scale = 1.5;
        _spacedust2.scale = 1.5;
    } else {
```

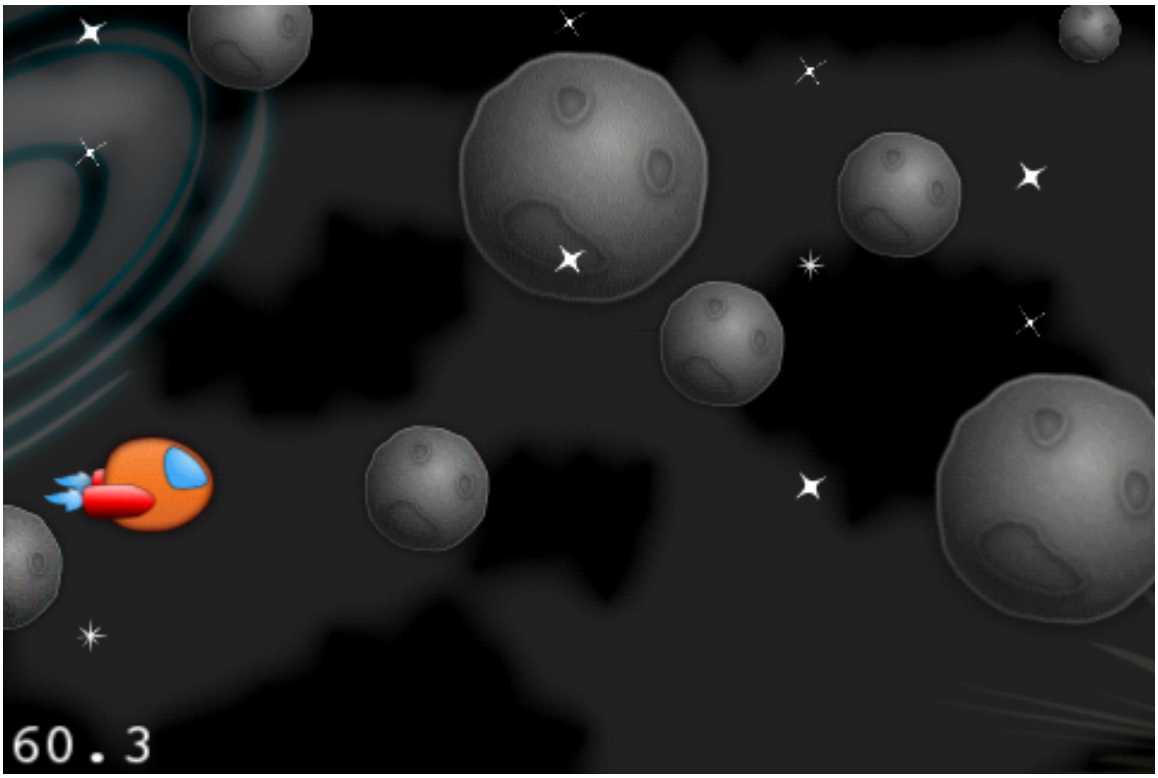
```
        _spacedust1 = [CCSprite
            spriteWithFile:@"bg_front_spacedust.png"];
        _spacedust2 = [CCSprite
            spriteWithFile:@"bg_front_spacedust.png"];
        _planetsunrise = [CCSprite
            spriteWithFile:@"bg_planetsunrise.png"];
        _galaxy = [CCSprite
            spriteWithFile:@"bg_galaxy.png"];
        _spacialanomaly = [CCSprite
            spriteWithFile:@"bg_spacialanomaly.png"];
        _spacialanomaly2 = [CCSprite
            spriteWithFile:@"bg_spacialanomaly2.png"];
    }

    // 3) Determine relative movement speeds for space dust
    // and background
    CGPoint dustSpeed = ccp(0.1, 0.1);
    CGPoint bgSpeed = ccp(0.05, 0.05);

    // 4) Add children to CCParallaxNode
    [_backgroundNode addChild:_spacedust1 z:0
        parallaxRatio:dustSpeed
        positionOffset:ccp(0,winSize.height/2)];
    [_backgroundNode addChild:_spacedust2 z:0
        parallaxRatio:dustSpeed
        positionOffset:ccp(_spacedust1.contentSize.width*
            _spacedust1.scale, winSize.height/2)];
    [_backgroundNode addChild:_galaxy z:-1
        parallaxRatio:bgSpeed
        positionOffset:ccp(0,winSize.height * 0.7)];
    [_backgroundNode addChild:_planetsunrise z:-1
        parallaxRatio:bgSpeed
        positionOffset:ccp(600,winSize.height * 0)];
    [_backgroundNode addChild:_spacialanomaly z:-1
        parallaxRatio:bgSpeed
        positionOffset:ccp(900,winSize.height * 0.3)];
    [_backgroundNode addChild:_spacialanomaly2 z:-1
        parallaxRatio:bgSpeed
        positionOffset:ccp(1500,winSize.height * 0.9)];
}
```

The comments explain each section above. The only extra thing to point out is that we have to choose the –hd sprite versions for the iPad, and we also have to scale the space dust image a bit so it's sure to fill the entire screen on the iPad.

Then add the line to call this to the bottom of **init**:

```
[self setupBackground];
```

Compile and run your project, and you should see the start of a space scene:



However this isn't very interesting yet, since nothing is moving!

To move the space dust and backgrounds, all you need to do is move the parallax node itself. For every Y points we move the parallax node, the dust will move 0.1Y points, and the backgrounds will move 0.05Y points.

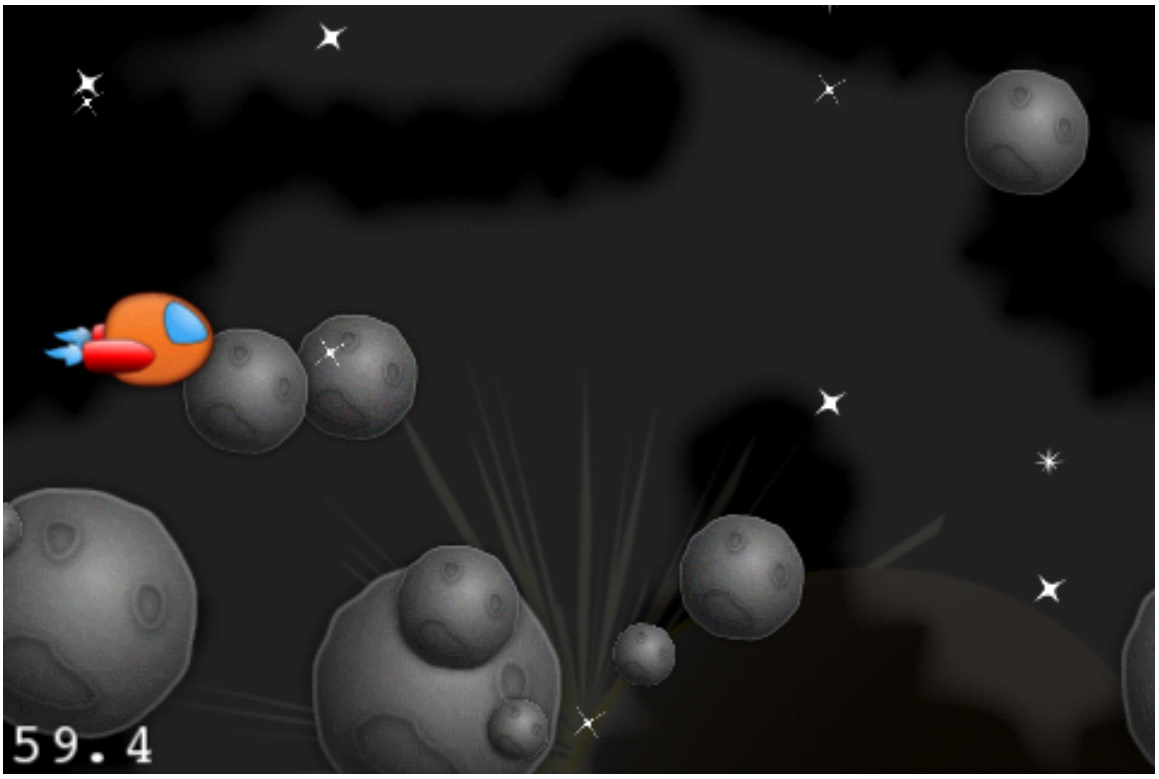
To move the parallax node, you'll simply update the position every frame according to a set velocity. Try this out for yourself by adding the following new method to **ActionLayer.m**, right above **update**:

```
- (void)updateBackground:(ccTime)dt {
    CGPoint backgroundScrollVel = ccp(-1000, 0);
    _backgroundNode.position =
        ccpAdd(_backgroundNode.position,
               ccpMult(backgroundScrollVel, dt));
}
```

And call this new method from **update**:

```
[self updateBackground:dt];
```

Compile and run your project, and things should start to scroll pretty neatly with parallax scrolling!



However, after a few seconds goes by, you'll notice a major problem: we run out of things to scroll through, and you end up with a blank screen! That would be pretty boring, so let's see what we can do about this.

Continuous Scrolling

We want the background to keep scrolling endlessly. The strategy we're going to take to do this is to simply move the background to the right once it has moved offscreen to the left.

One minor problem is that **CCParallaxNode** currently doesn't have any way to modify the offset of a child node once it's added. You can't simply update the position of the child node itself, because the **CCParallaxNode** overwrites that each update.

However, I've created a category on **CCParallaxNode** that you can use to solve this problem. You already added this file to your project when you dragged in the **Art** folder—it's under **Art\Classes\CCParallaxNode-Extras.h/m.**

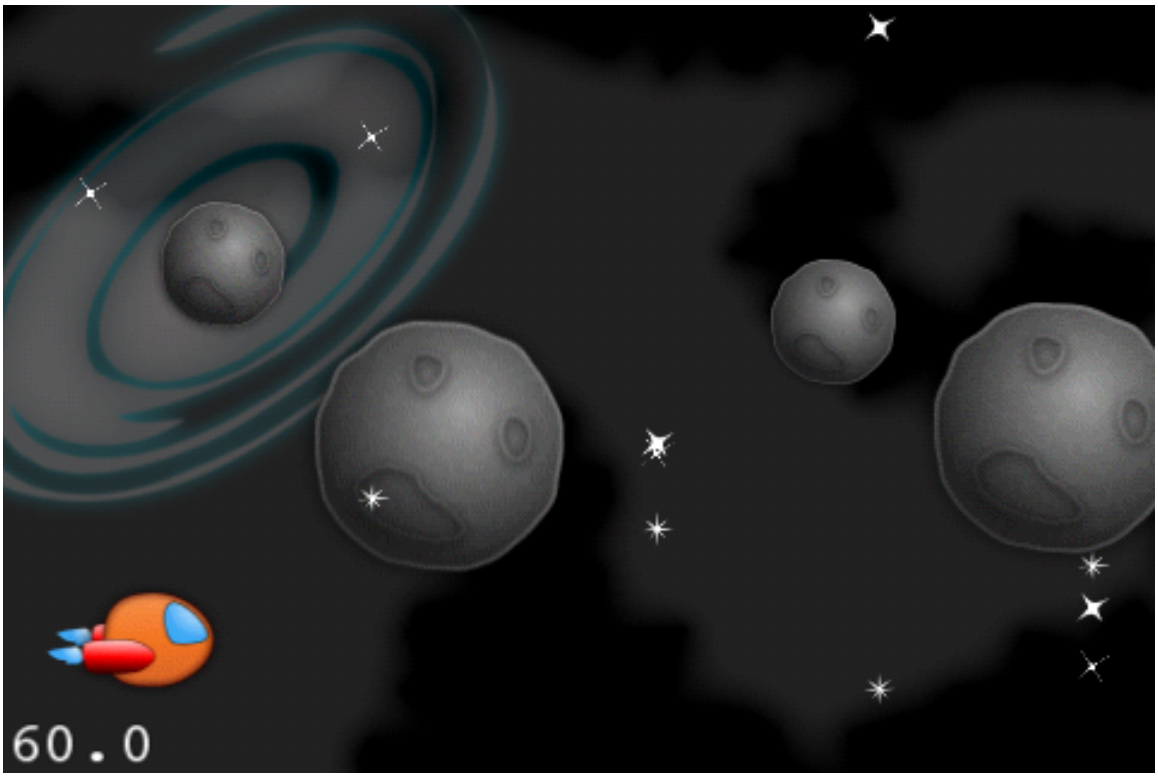
To make use of this, first import the header at the top of **ActionLayer.m**:

```
#import "CCParallaxNode-Extras.h"
```

And then add this code to the bottom of **updateBackground** to reset the positions of each sprite as they go offscreen:

```
NSArray *spaceDusts = [NSArray arrayWithObjects:_spacedust1, _spacedust2, nil];
for (CCSprite *spaceDust in spaceDusts) {
    if ([_backgroundNode
        convertToWorldSpace:spaceDust.position].x < -
        spaceDust.contentSize.width*self.scale) {
        [_backgroundNode
         incrementOffset:ccp(2*spaceDust.contentSize.width*
                             spaceDust.scale,0)
         forChild:spaceDust];
    }
}

NSArray *backgrounds = [NSArray arrayWithObjects:_planetsunrise, _galaxy,
_spacialanomaly, _spacialanomaly2, nil];
for (CCSprite *background in backgrounds) {
    if ([_backgroundNode
        convertToWorldSpace:background.position].x < -
        background.contentSize.width*self.scale) {
        [_backgroundNode incrementOffset:ccp(2000,0)
          forChild:background];
    }
}
```

Compile and run your project, and now the background should scroll continuously through a cool space scene!



Finishing Touches

This game is almost ready to ship (no pun intended)!

However, there are a few small tidbits you should take care of first. First, go to AppDelegate.mm, find the line that says **[director setDisplayFPS:YES]**, and change it to:

```
[director setDisplayFPS:NO];
```

That will turn off the FPS display at the bottom left of the screen, something you don't want when shipping!

Also, go to your **Resources** group and delete **Default.png**, which is the Cocos2D splash image. I've included some substitutes under **Art\Other** for this that look like the first part of the space game. You can delete **blocks.png** if you want too, since you aren't using it.

Also delete all of the icons in **Resources**—I've included a substitute for that as well.

Delete the app from your simulator, do a **Product\Clean** and re-run, and you should enjoy no FPS, a better splash screen, and a better icon!

What About the iPad?

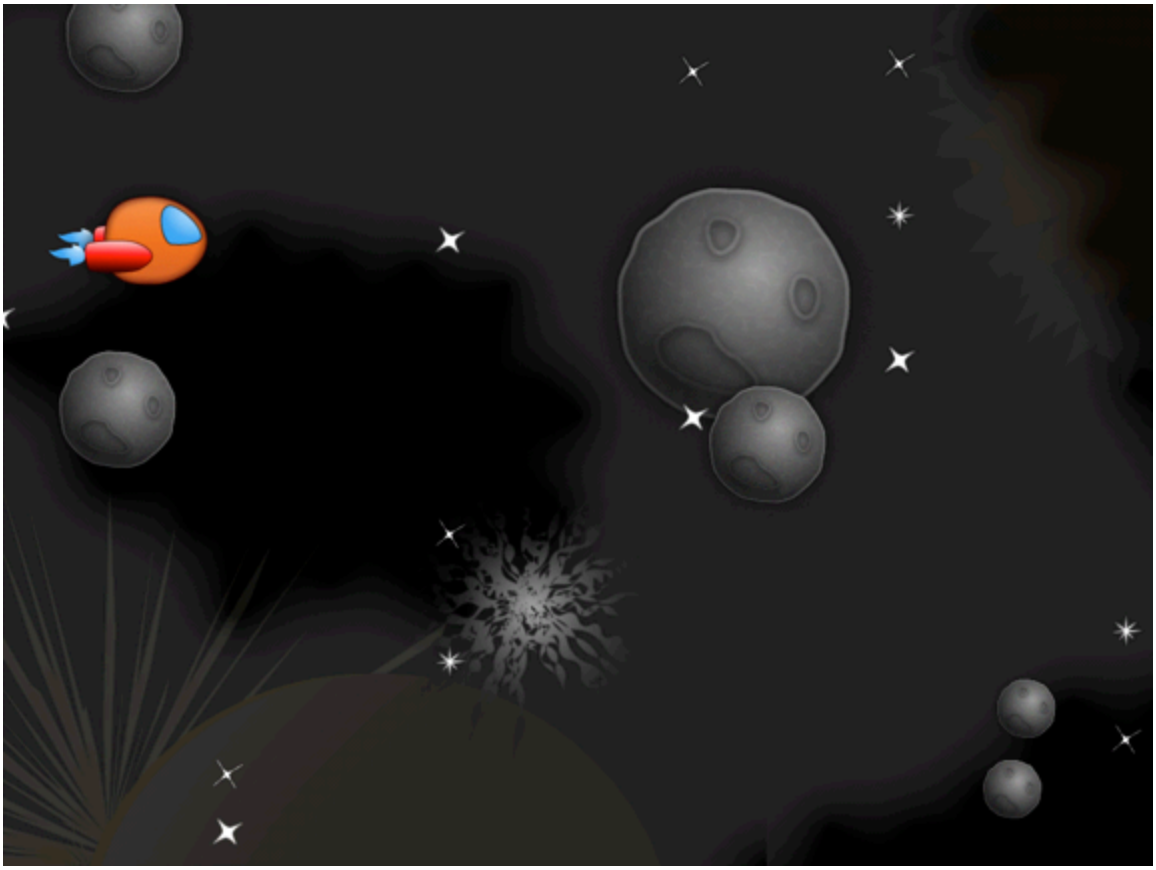
If you have an iPad you'd like to run this game on—don't worry, we haven't forgotten about you!

In fact, we've been very careful to write all of the code so that it works on the iPad already:

- Whenever loading an image, we added code to load the –hd version for the iPad.
- We made our background images big enough so they'd fill the entire iPad screen (and overlap a bit on the iPhone).
- We didn't hardcode positions based on the iPhone screen size. Instead we based them on multiples of the window size.

So all we need to do to get it to work on the iPad is to set up the app as a Universal app! This is extremely easy to do just click on the SpaceGame project entry in the Groups and Files tree, select the SpaceGame target, go to Build Settings, search for Targeted Device Family, and set it to iPhone/iPad.

It's as easy as that! Compile and run your code, and enjoy your space game full-screen on the iPad!



Where To Go From Here?

If you've made it this far, you're getting to be pretty kick-ass at making space games—you're shooting lasers, blowing up asteroids, and looking pretty slick while you're at it. In fact you're kinda 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 code as a starting point for your own game! Here are a few features you might want to add:

- Right now the game has no way to win (or lose!) Figure out what should cause the player to win or lose the game and add some game logic in.
- Do you want to keep a score for the game? Maybe give points each time the player shoots an asteroid and display 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 tutorial, where you (like Luke) will crash your Xcode 4 into the swamp of De-go-cocos2D, and you'll learn to awaken your latent space-game-making powers!

Tutorial 2: Collisions and Explosions

Welcome back, brave explorer!

In this tutorial, you're going to awaken your latent space-making powers and make new features rise out of the code swamp with Xcode mind tricks!

You'll start by modifying your project to use Box2D for collision detection for better accuracy. You'll then add some cool explosions for when you blow up asteroids or the ship gets hit. And finally, you'll add some win/lose logic into the game so you can see if you have what it takes to survive the vicious asteroid belt!

We're going to pick up where we left off last tutorial. If you skipped last tutorial, you can continue on with the **SpaceGame1** project that comes with the Starter Kit.

So, although wars do not make one great, adding massive space explosions to our game does, so let's get coding! :]

Adding Box2D Support

In the last tutorial, we used a very simple method of collision detection where we just used the bounding box of the sprites to check if they collided.

As you play the game, you'll notice that this causes some odd results. Sometimes asteroids get destroyed even though it looks like the lasers don't actually hit them, since it's counting the transparent space. And even though we haven't added code to check for asteroid-to-ship collisions yet, if we did it would have the same problem. But worse, because if the game counted an asteroid hitting your ship even though it looked like you dodged it, you'd probably get really mad!

To add better collision detection into our game, we're going to leverage Box2D because it has a great library of polygon definition and collision detection code that we can use.

But of course, the first step is to set up our project to use Box2D! In this section we'll go through the basic steps I use whenever I want to add Box2D support to a project.

The first thing we need to do is define the point-to-meter ratio, or **PTM_RATIO**. In Cocos2D we measure objects in points, but Box2D wants to deal with objects in meters (of an average size between 1-10 meters). So we need to have a conversion factor to convert between these two.

The average size of our objects in the Space Game Starter kit is around 100 pixels or so, so we'll just say that 100 pixels equals 1 meter. And of course, objects are double sized on the iPad, so it will be 200 pixels equals 1 meter on the iPad.

So just add this conversion ratio to the top of **Common.h** like this:

```
#define PTM_RATIO ((UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad) ? 200.0 : 100.0)
```

OK, now we can start adding Box2D support to our **ActionLayer**. Switch to **ActionLayer.h** and start by adding the following imports to the top of the file:

```
#import "Box2D.h"
#import "GLES-Render.h"
```

This imports Box2D and the code that does Box2D debug drawing.

Still in **ActionLayer.h**, add the following instance variables to the class definition:

```
b2World * _world;
GLESDebugDraw * _debugDraw;
```

This declares the variables we'll use to keep track of the Box2D world, and the class to perform Box2D debug drawing.

Next, switch to **ActionLayer.m** and add the method to create the Box2D world right above **init**:

```
- (void)setupWorld {
    b2Vec2 gravity = b2Vec2(0.0f, 0.0f);
    bool doSleep = false;
    _world = new b2World(gravity, doSleep);
}
```

Here we set up the Box2D world to have no gravity, and not to let any objects sleep. We don't need gravity because we'll be moving the Box2D objects ourselves, so we don't want anything external trying to move them.

Now add the following method right after **setupWorld**:

```
- (void)setupDebugDraw {
    _debugDraw = new GLESDebugDraw(PTM_RATIO*
        [[CCDirector sharedDirector] contentScaleFactor]);
    _world->SetDebugDraw(_debugDraw);
    _debugDraw->SetFlags(b2DebugDraw::e_shapeBit |
        b2DebugDraw::e_jointBit);
}
```

This creates the class to perform Box2D debug drawing, so we can see a representation in the Cocos2D world of what's in the Box2D world. Note that this class takes a pixel-to-meter ratio, not a point-to-meter ratio, so we need to multiply the point-to-meter ratio by the Cocos2D **contentScaleFactor** for it to work correctly on the Retina display.

Next we're going to add a method that we're going to use to add a test shape into the Box2D world, just so we can make sure everything is working OK:

```
- (void)testBox2D {

    CGSize winSize = [CCDirector sharedDirector].winSize;

    b2BodyDef bodyDef;
    bodyDef.type = b2_dynamicBody;
    bodyDef.position = b2Vec2(winSize.width/2/PTM_RATIO,
                              winSize.height/2/PTM_RATIO);
    b2Body *body = _world->CreateBody(&bodyDef);

    b2CircleShape circleShape;
    circleShape.m_radius = 25.0/PTM_RATIO;
    b2FixtureDef fixtureDef;
    fixtureDef.shape = &circleShape;
    fixtureDef.density = 1.0;
    body->CreateFixture(&fixtureDef);

    body->ApplyAngularImpulse(0.01);

}
```

This code performs the basic steps you need to take to add an object to the Box2D world:

1. Create a **body definition**, specifying the type of the body and its position. In this case, we choose a dynamic body, which means it can move (rather than a static body, which does not move). Also note that whenever we set the position of a Box2D object, we have to give it Box2D coordinates, so we use the **PTM_RATIO** to convert between Cocos2D and Box2D coordinates.
2. Tell the Box2D world to create a **body**, passing in the body definition.
3. Create a **shape**, specifying its size or vertices. In this case, we just want a simple circle shape, so we just set its radius.
4. Create a **fixture definition**, specifying parameters on the shape such as density (the higher the density the harder an object is to move), friction (how slippery an object is), or restitution (how bouncy an object is). Here we just set the density to 1.0.
5. Tell the Box2D body to create a **fixture**, passing in the fixture definition.

Note that a Box2D body can contain more than one fixture (for example you can have an object made up of a few circles and a box), but in this test code all we need is one.

One more note about the above method—you'll notice a call to **ApplyAngularImpulse** at the end. This is just some test code to make the circle rotate, so we can make sure everything is working OK.

Now that you have all of these methods, add the code to **init** to call them. **Important**: add them to the top of the method, because we want to make sure Box2D is set up first thing!

```
[self setupWorld];
[self setupDebugDraw];
[self testBox2D];
```

Next you need to give Box2D time to update its simulation. Even though you'll be updating the Box2D objects positions yourself and letting Cocos2D handle the movement, Box2D still needs time to process the collision detection. So add the following method right above **update**:

```
- (void)updateBox2D:(ccTime)dt {
    _world->Step(dt, 1, 1);
}
```

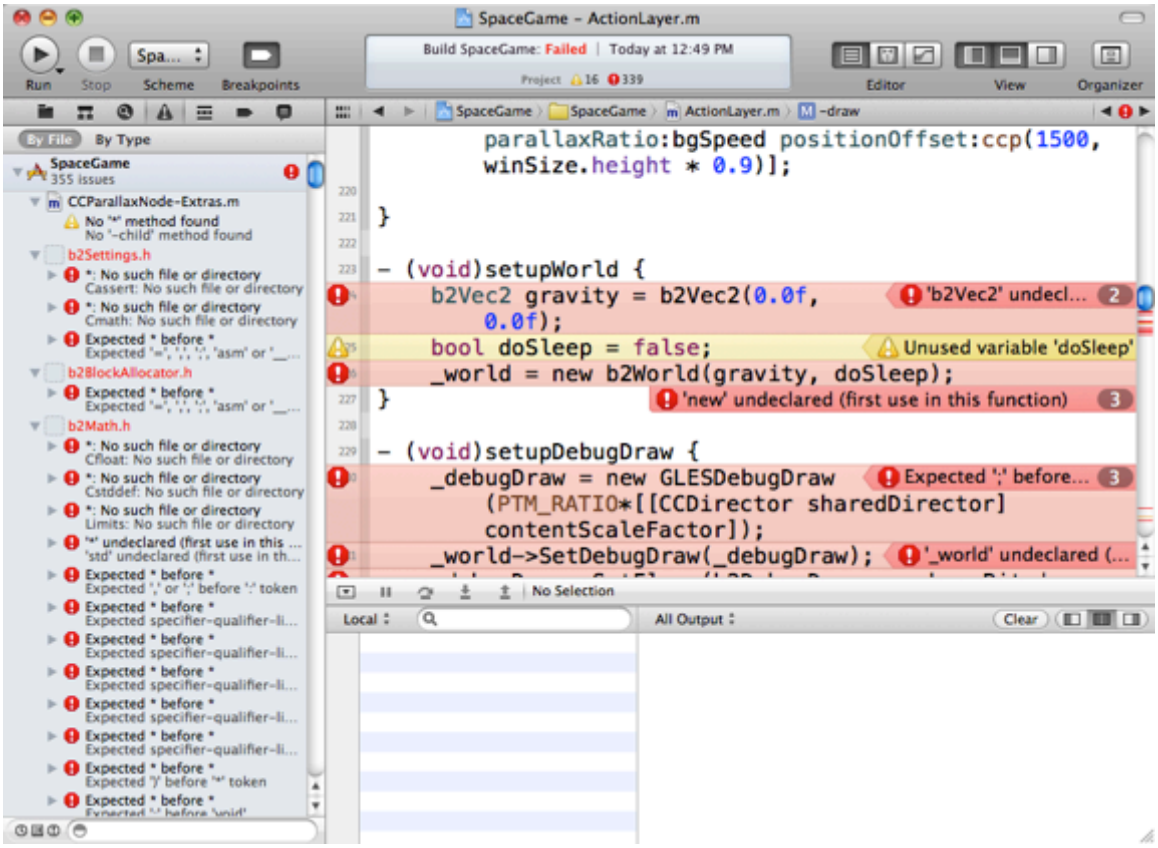
And then add this line at the end of **update** to call your new method:

```
[self updateBox2D:dt];
```

One last step—you still need to add the code to perform the debug drawing so you can visualize what's in the Box2D world. So add the following method to the end of the file:

```
-(void) draw {
    glDisable(GL_TEXTURE_2D);
    glDisableClientState(GL_COLOR_ARRAY);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);

    _world->DrawDebugData();

    glEnable(GL_TEXTURE_2D);
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
}
```

That's it! Compile and run and… BOOM!



Wow—what's all that about?!

Well, Box2D is written in C++, and if you want to call C++ code from Objective-C code, you need to make sure your files end in **.mm,** not **.m**.

So rename **ActionLayer.m** to **ActionLayer.mm**, **Common.m** to **Common.mm**, and **SpriteArray.m** to **SpriteArray.mm**.

Compile and run again, and this time it run OK, and if all works well you should see a strange rotating pink circle in the middle of your screen:

If this shows up, congrats—that means you've accomplished a lot! You've created a Box2D world, set up debug drawing, and have given Box2D time to update its simulation. You've also added a test object into the world, and made it rotate.

Now that you've got Box2D set up, you can start using it to add some better collision detection into the game!

Defining Box2D Shapes with Physics Editor

In the test code above, we manually created a Box2D body and shape by writing code. It was pretty straightforward to write, because we just had a simple circle shape.

But now we want to make shapes for the ship, asteroid, and other objects, which aren't necessarily simple circles. Instead, they are going to be polygons.

You can manually define polygon shapes with Box2D by writing code if you want—you can create an array where you define the point of each vertex on the polygon. However in practice this is very slow and tedious work!

One solution that I've covered in tutorials in the past like this one is to use a tool called Vertex Helper that lets you visually define the vertices by clicking around the objects in a nice GUI.

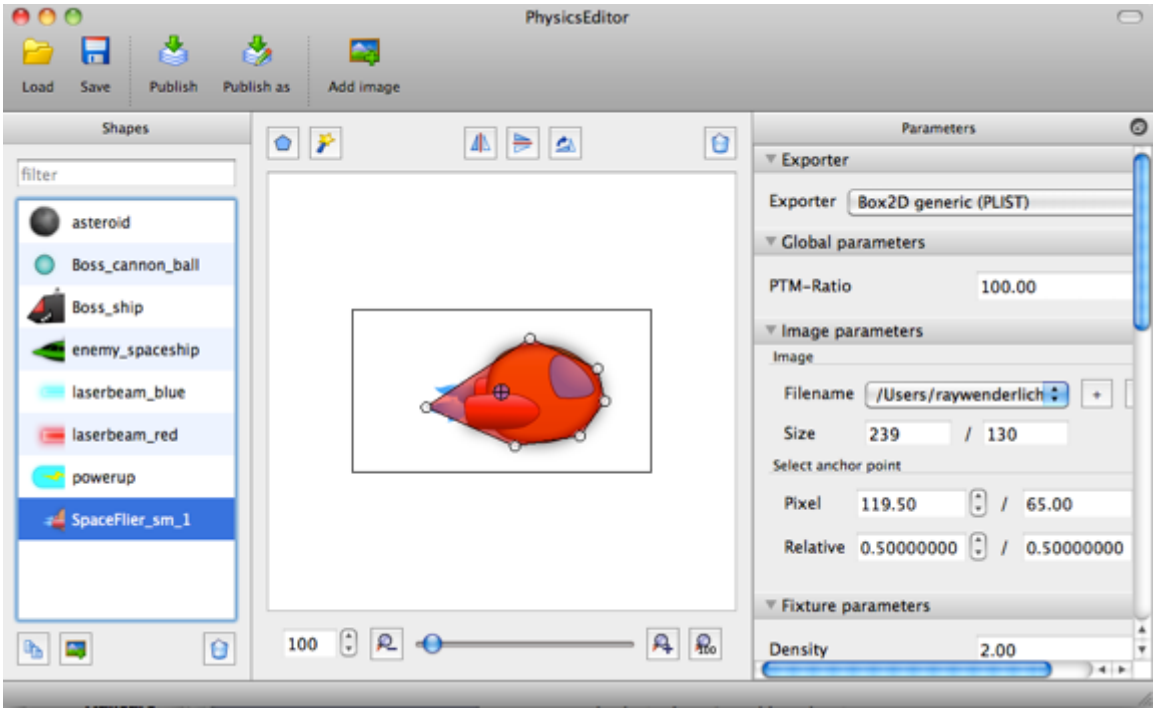
This method is a great "bare bones" way of defining shapes, but when shapes gets more complicated it becomes a bit more annoying to use.

So for this project, we use an alternative tool called Physics Editor. Physics Editor has a couple of cool features that Vertex Helper doesn't have, including:

- The ability to automatically trace shapes for sprites just by clicking on them with a magic wand!
- The ability to automatically break up a concave shape into multiple convex shapes behind the scenes.
- The ability to specify physics parameters (such as friction, density, or in our case—collision options) directly from within the tool.

If you don't have or don't want to purchase Physics Editor, don't worry— you can skip to the next section and continue with this tutorial with the shape files I already made with Physics Editor. When you make your own game or want to trace your own objects, you can create the shapes manually or with the free version of Vertex Helper.

If you already have Physics Editor or choose to purchase it, you can open **Raw\Shapes.pes** to see the physics shapes I traced for each sprite, and tweak them if you'd like:



I added the sprites for each object we're going to need collision detection for in the game, and started by using the magic wand tool to get a basic shape. I then adjusted the tolerance to get a low count of vertexes (around 8 or so per object)—because the less vertexes you have, the faster the collision detection will go. I also manually moved some of the vertexes around a bit to get a better match to the shape.

Feel free to look through the objects and see how the shapes are defined for each. Notice how the names that show up in the sidebar (such as "powerup") are the "shape names". We'll need these to pull out the information exported by Physics Editor back in the code. Right now, they pretty much map up 1-1 with the sprite names (but don't necessarily have to).

Mapping Box2D Shapes to Sprites

Now that we have defined some geometry for the Box2D shapes using Physics Editor, it's time to "attach" these shapes to our Cocos2D sprites.

We're going to associate a particular Box2D body with each Cocos2D sprite. Then every frame, as the Cocos2D sprites move, we'll also move their associated Box2D body to the same spot.

Because the sprites positions match up to the Box2D body's positions, we can then use Box2D collision detection to tell us when the Box2D shapes collide—and that will be the same as telling us the sprites collided!

To manage all of this (and some other cool features such as hit points!), we're going to make a subclass of **CCSprite** called **GameObject**. Let's go ahead and make this now.

Go to **File\New\New File**, choose **iOS\Cocoa Touch\Objective-C class**, and click **Next**. Enter **NSObject** for Subclass of, click **Next**, name the new file **GameObject.mm**, and click Save.

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

```
#import "cocos2d.h"
#import "Box2D.h"
#import "Common.h"

@interface GameObject : CCSprite {
    float _hp;
    float _maxHp;
    b2World* _world;
    b2Body* _body;
    NSString *_shapeName;
}

@property (assign) float maxHp;

- (id)initWithSpriteFrameName:(NSString *)spriteFrameName world:(b2World *)world
shapeName:(NSString *)shapeName maxHp:(float)maxHp;
- (BOOL)dead;
- (void)destroy;
- (void)revive;
- (void)takeHit;

@end
```

This creates a subclass of **CCSprite** that has instance variables for the object's current hit point, max hit points, a reference to the Box2D world, the Box2D shape that it is associated with, and the name of the shape made with Physics Editor.

It also has a few helper methods we're about to write—an initializer, a method to check if the object is "dead" (i.e. 0 hp), a method to destroy the object, a method to bring the object "back to life" (for when we're reusing a sprite), and a method to make the object take a hit (i.e. lose an hp).

Now switch over to **GameObject.mm** and replace it with the following implementation:

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

@implementation GameObject
@synthesize maxHp = _maxHp;

- (id)initWithSpriteFrameName:(NSString *)spriteFrameName world:(b2World *)world
shapeName:(NSString *)shapeName maxHp:(float)maxHp {

    if ((self = [super
        initWithSpriteFrameName:spriteFrameName])) {
        _hp = maxHp;
        _maxHp = maxHp;
        _world = world;
        _shapeName = [shapeName retain];
    }
    return self;
}

- (void) destroyBody {
    if (_body != NULL) {
        _world->DestroyBody(_body);
        _body = NULL;
    }
}

- (void) createBody {

    [self destroyBody];

    b2BodyDef bodyDef;
    bodyDef.type = b2_dynamicBody;
    bodyDef.position.Set(self.position.x/PTM_RATIO,
                         self.position.y/PTM_RATIO);
    bodyDef.userData = self;
    _body = _world->CreateBody(&bodyDef);
    [[ShapeCache sharedShapeCache]
     addFixturesToBody:_body
     forShapeName:_shapeName
     scale:self.scale];
    [self setAnchorPoint:
     [[ShapeCache sharedShapeCache] anchorPointForShape:_shapeName]];
```

```
}

- (void)setNodeInvisible:(CCNode *)sender {
    sender.position = CGPointZero;
    sender.visible = NO;
    [self destroyBody];
}

- (void)revive {
    _hp = _maxHp;
    [self stopAllActions];
    self.visible = YES;
    self.opacity = 255;
    [self createBody];
}

- (BOOL)dead {
    return _hp == 0;
}

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

- (void)destroy {

    _hp = 0;
    [self stopAllActions];
    [self runAction:
     [CCSequence actions:
      [CCFadeOut actionWithDuration:0.1],
      [CCCallFuncN actionWithTarget:self
                           selector:@selector(setNodeInvisible:)],
      nil]];

}

- (void)dealloc {
    [_shapeName release];
    _shapeName = nil;
```

```
    [super dealloc];
}

@end
```

Wow—that's a lotta code! Luckily most of it is pretty simple, though. Let's go over it method by method.

- **initWithSpriteFrameName:world:shapeName:maxHp** starts by calling the superclass's (**CCSprite**) **initWithSpriteFrameName** method, to initialize the sprite with a particular image. It then saves off the parameters for use later.
- **destroyBody** destroys any existing Box2D body. We'll destroy the Box2D body whenever it isn't needed, to conserve resources.
- **createBody** creates a new Box2D body to associate to the sprite. It sets its position to the sprites current position, and sets the spite as the user data of the Box2D body (so that when we have a reference to the Box2D body, we can easily get access to the sprite). To create the shapes, it uses some helper code called the **ShapeCache**, which reads the file exported by Physics Editor to add the appropriate shapes to the Box2D body along with their parameters set up in the tool. Note that you also need to add the line to set the anchor point of the sprite based on the values set up in Physics Editor. If you forget this, you'll often get the sprites not matching up right to the shapes!
- **setNodeInvisible** will be called whenever an object is destroyed (such as an asteroid exploding). It simply sets the node to invisible and calls the method to destroy the Box2D body.
- **revive** will be called whenever we want to reuse a sprite from our sprite array. It basically resets everything. It sets the hp back to the max, stops any actions, sets it as fully opaque (non-transparent), and creates a fresh Box2D body.
- **dead** is a little helper method that checks if the hp is 0 or not.
- **takeHit** is another helper method that subtracts an hp. When the hp reaches 0, it calls the **destroy** method to destroy the shape.
- **destroy** is a method I added for fun. Instead of just immediately removing the object when it's destroyed, this method makes it fade out first. When the fade out is complete, it calls the **setNodeInvisible** method discussed earlier, which completes the process.
- Last but not least, in **dealloc** we free the **_shapeName** variable that was retained in the initializer.

You might be wondering where that **ShapeCache** helper code came from. Well, you already added it to your project earlier—it's under **Art\Classes**, as **ShapeCache.h** and **ShapeCache.mm**.

The original version of this file was written by Andreas Löw, the author of Physics Editor. However, I modified this version to let you specify the scale of the sprite as a parameter to **addFixturesToBody**, because in this game we want to have asteroids of different sizes.

Feel free to use this in future projects of yours where you want to have Box2D shapes resized based on the scale of your sprites. However note that once a Box2D body is created, you can't resize it—you have to destroy the body and recreate it if you want to resize it. This is one of the reasons we destroy the body when the sprite is destroyed, and create a new one every time we spawn a new sprite.

Take a deep breath—you've finished the hard part! Now you just need to make use of this new class in the **SpriteArray** (so it will contain **GameObjects** instead of **CCSprites**) and **ActionLayer**.

We'll start with **SpriteArray**. Switch **to SpriteArray.h** and add the following import to the top of the file:

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

Then modify the initializer to read as follows:

```
- (id)initWithCapacity:(int)capacity spriteFrameName:(NSString *)spriteFrameName
batchNode:(CCSpriteBatchNode *)batchNode world:(b2World *)world shapeName:(NSString
*)shapeName maxHp:(int)maxHp;
```

We basically just added some extra parameters, since **GameObject** takes more parameters than **CCSprite** did.

Switch to **SpriteArray.mm** and replace the initializer signature there as well:

```
- (id)initWithCapacity:(int)capacity spriteFrameName:(NSString *)spriteFrameName
batchNode:(CCSpriteBatchNode *)batchNode world:(b2World *)world shapeName:(NSString
*)shapeName maxHp:(int)maxHp {
```

And replace the line that creates the **CCSprite** inside the **for** loop with the following (but don't change the lines that add the sprite to the batchNode and array underneath!)

```
GameObject *sprite = [[[GameObject alloc] initWithSpriteFrameName:spriteFrameName
world:world shapeName:shapeName maxHp:maxHp] autorelease];
```

This just creates a **GameObject** instead of a **CCSprite** and passes the parameters through.

Done with **SpriteArray**—onto **ActionLayer**! Start with **ActionLayer.h** and add this import to the top of the file:

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

Also change the type of **_ship** from a **CCSprite** to a **GameObject**:

```
GameObject *_ship;
```

Then switch to **ActionLayer.mm** and add an import to the top of that file also:

```
#import "ShapeCache.h"
```

Remember that when we want to destroy an asteroid or laser we've been calling invisNode to set the node to invisible. We need to replace this to call the destroy method on GameObject instead since it does a bit more work including destroying the Box2D body, so replace invisNode with the following:

```
- (void)invisNode:(GameObject *)sender {
    [sender destroy];
}
```

Next, find the **spawnShip** method and replace the line that creates the ship with the following:

```
_ship = [[[GameObject alloc] initWithSpriteFrameName:@"SpaceFlier_sm_1.png"
world:_world shapeName:@"SpaceFlier_sm_1" maxHp:10] autorelease];
```

Here we're creating the ship as a **GameObject** instead of **CCSprite**, so we can have a Box2D body associated with it—and finally a set of hit points for the ship!

Still in **spawnShip**, add a method to **revive** the ship after setting the ship's position:

```
[_ship revive];
```

Note that it is important to call **revive** after setting the position, because **revive** sets the initial position of the Box2D body based on the sprite's position.

Next, find the **setupArrays** method and replace it with the following:

```
- (void)setupArrays {
    _asteroidsArray = [[SpriteArray alloc] initWithCapacity:15
spriteFrameName:@"asteroid.png" batchNode:_batchNode world:_world
shapeName:@"asteroid" maxHp:1];
    _laserArray = [[SpriteArray alloc] initWithCapacity:15
spriteFrameName:@"laserbeam_blue.png" batchNode:_batchNode world:_world
shapeName:@"laserbeam_blue" maxHp:1];
}
```

This uses the new version of the **SpriteArray** initializer, passing in the additional parameters. We initialize all the asteroids as 1 hp, but don't worry—later on we'll be modifying the hp so the bigger asteroids have more hit points!

Remember how we are using the **ShapeCache** to read in the shape definitions inside **GameObject**? Well we still need to add the code to load the shapes we made with Physics Editor into the cache. So add this method next, right above **init**:

```
- (void)setupShapeCache {
    [[ShapeCache sharedShapeCache] addShapesWithFile:@"Shapes.plist"];
}
```

And then add the line to call this in **init**, right after **testBox2D** (in fact comment that out while you're at it):

```
//[self testBox2D];
[self setupShapeCache];
```

Now move to the **updateAsteroids** method. Replace the line that creates a new asteroid with the following:

```
GameObject *asteroid = [_asteroidsArray nextSprite];
```

Then find the code that makes the asteroids different size and modify the code to set their HP based on their size too—and call **revive** once it's all set. The modifications are highlighted below:

```
int randNum = arc4random() % 3;
if (randNum == 0) {
    asteroid.scale = 0.25;
    asteroid.maxHp = 2;
} else if (randNum == 1) {
    asteroid.scale = 0.5;
    asteroid.maxHp = 4;
} else {
    asteroid.scale = 1.0;
    asteroid.maxHp = 6;
}
[asteroid revive];
```

Almost done—just two more steps! First, add the following to the end of **updateBox2D**:

```
for(b2Body *b = _world->GetBodyList(); b; b=b->GetNext()) {
    if (b->GetUserData() != NULL) {
        GameObject *sprite =
            (GameObject *)b->GetUserData();

        b2Vec2 b2Position =
            b2Vec2(sprite.position.x/PTM_RATIO,
                   sprite.position.y/PTM_RATIO);
        float32 b2Angle =
            -1 * CC_DEGREES_TO_RADIANS(sprite.rotation);

        b->SetTransform(b2Position, b2Angle);
    }
}
```

This loops through each Box2D object, and look at their user data to see if they have an associated **GameObject**. If it does, it sets the Box2D position and rotation to be the same as the **GameObject**'s position and rotation.

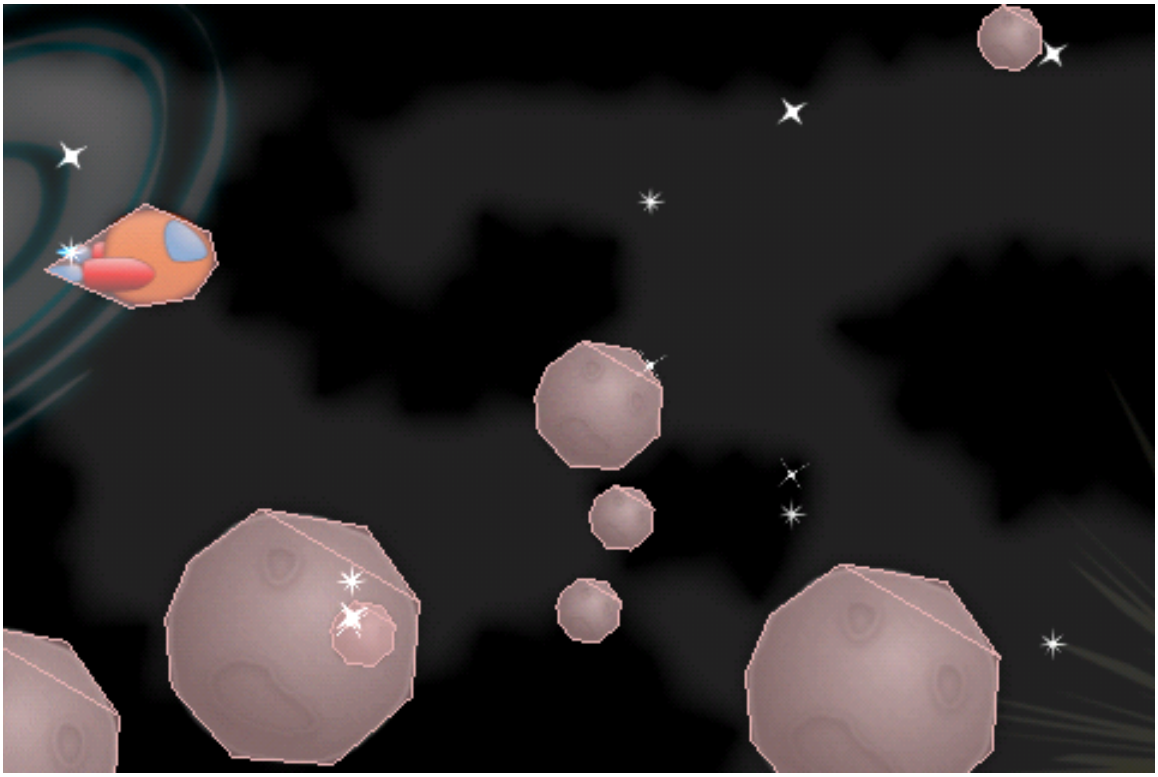
Finally, in **ccTouchesBegan:**, replace the four lines that create the ship laser with the following:

```
GameObject *shipLaser = [_laserArray nextSprite];
[shipLaser stopAllActions];
shipLaser.position = ccpAdd(_ship.position, ccp(shipLaser.contentSize.width/2, 0));
[shipLaser revive];
```

This is just the modification we need to use the GameObject rather than CCSprite.

Phew—you're finally done! Compile and run your project, and now you should see pink Box2D shapes following the sprites, that we'll be able to leverage for much improved collision detection!

While you're playing with the game, don't worry that the Box2D shapes aren't destroyed right now when the asteroids are destroyed—we're going to completely replace that collision detection code next.



Better Collision Detection

Finally, it's time to add the better collision detection!

Now that we have Box2D set up and Box2D shapes mapped to sprites, this is going to be very easy.

The way you detect collisions with Box2D is to register a special object called a contact listener. So switch to **ActionLayer.h** and declare an instance variable for a contact listener like so:

```
b2ContactListener * _contactListener;
```

b2ContactListener is just a generic class—you need to subclass it with your own implementation. I've written a very simple implementation for you located under **Art\Classes\SimpleContactListener.h**, that just forwards the **beginContact** and **endContact** callbacks (that are called when objects begin and end colliding) onto the **ActionLayer**.

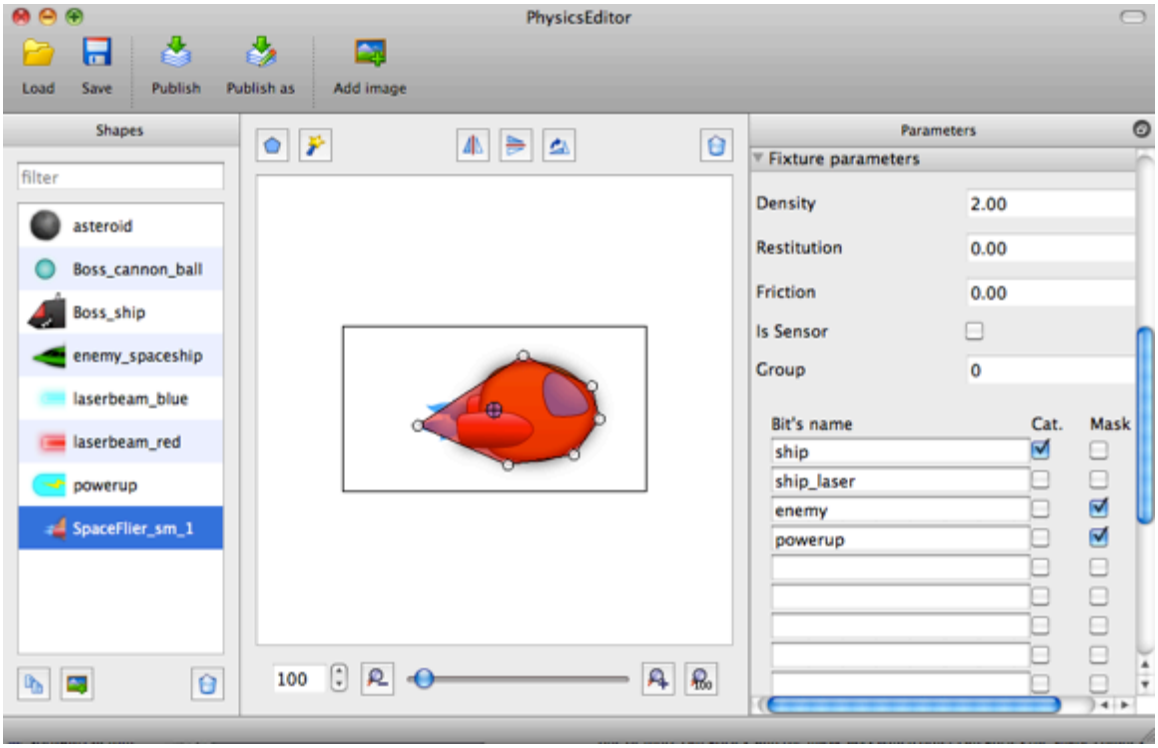
So switch to **ActionLayer.mm** and import **SimpleContactListener.h** at the top of the file:

```
#import "SimpleContactListener.h"
```

OK, before we go any further, let's talk a bit more about how collision detection works in Box2D. When you define shapes, you set some flags to decide which shapes collide with which.

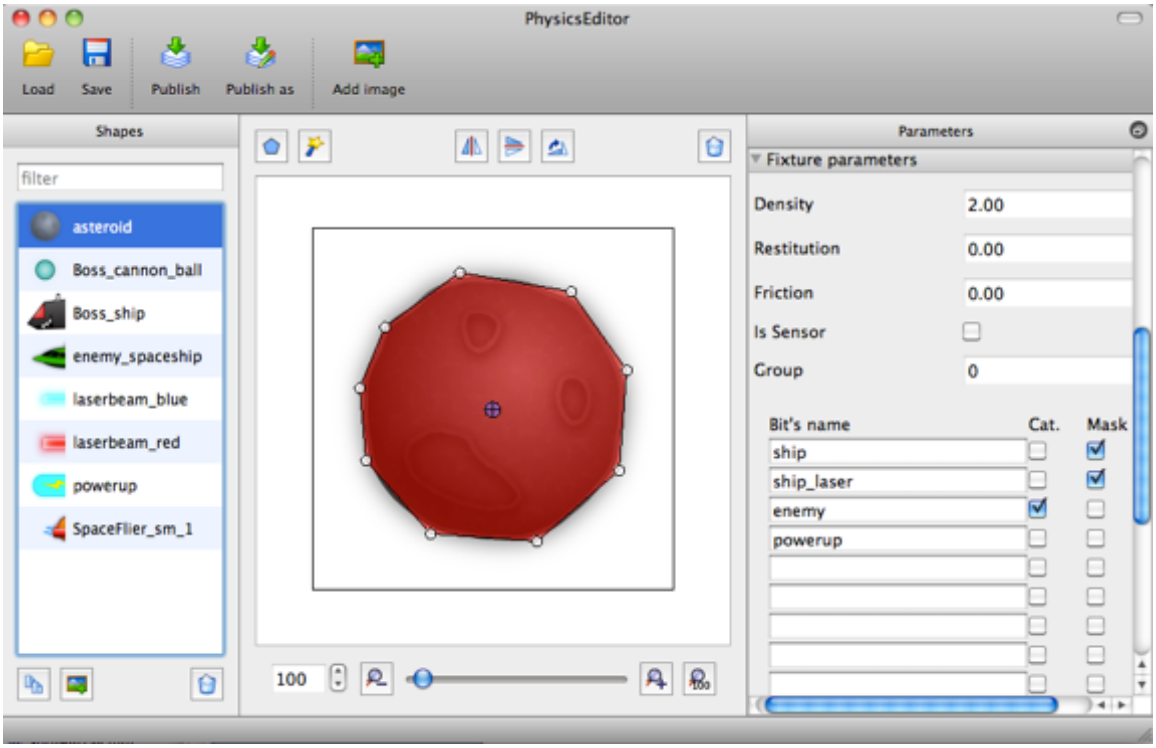
You do this by setting a shape's **category**, and collision **mask**. A shape can belong to one or more categories, and the mask says which other categories the shape collides with.

You can set this up with Physics Editor—in fact I already have done this for you! If you take a look at the shape for the ship you'll see the category and mask have been set up like the following:



If you look at the panel to the far right, you'll see that there are four categories set up: **ship**, **ship_laser**, **enemy**, and **powerup**. This shape has a checkmark in the **Cat** column for ship, which means it's in the **ship** category. And it has checkmarks for **enemy** and **powerup** under the **Mask** column, which means it collides with shapes in the **enemy** and **powerup** categories.

Let's look at another example. Here's the asteroid shape:



As you can see, this is marked as being in the **enemy** category, and colliding with shapes in the **ship** and **ship_laser** categories.

Back in code, we can't refer to categories by the names they're set up as in Physics Editor—instead we have to use numeric values. Categories and masks are stored as a bit field, so the first category has a bitmask of 0x1, the second as 0x2, and so on.

Let's make some constants to make referring to these easier in code. Add the following to the top of **ActionLayer.mm**:

```
#define kCategoryShip       0x1
#define kCategoryShipLaser  0x2
#define kCategoryEnemy      0x4
#define kCategoryPowerup    0x8
```

Then, add these lines to the end of **setupWorld** to initialize and register the collision handler:

```
_contactListener = new SimpleContactListener(self);
_world->SetContactListener(_contactListener);
```

Now that the collision handler is set up, the **SimpleContactListener** will call the **beginContact** and **endContact** methods on our **ActionLayer** whenever shapes begin and end colliding. So let's implement these next:

```
- (void)beginContact:(b2Contact *)contact {

    b2Fixture *fixtureA = contact->GetFixtureA();
    b2Fixture *fixtureB = contact->GetFixtureB();
    b2Body *bodyA = fixtureA->GetBody();
    b2Body *bodyB = fixtureB->GetBody();
    GameObject *spriteA = (GameObject *) bodyA->GetUserData();
    GameObject *spriteB = (GameObject *) bodyB->GetUserData();

    if (!spriteA.visible || !spriteB.visible) return;

    CGSize winSize = [CCDirector sharedDirector].winSize;

    if ((fixtureA->GetFilterData().categoryBits &
            kCategoryShipLaser &&
         fixtureB->GetFilterData().categoryBits &
            kCategoryEnemy) ||
        (fixtureB->GetFilterData().categoryBits &
            kCategoryShipLaser &&
         fixtureA->GetFilterData().categoryBits &
            kCategoryEnemy)) {

        // Determine enemy ship and laser
        GameObject *enemyShip = (GameObject*) spriteA;
        GameObject *laser = (GameObject *) spriteB;
        if (fixtureB->GetFilterData().categoryBits &
            kCategoryEnemy) {
            enemyShip = (GameObject*) spriteB;
            laser = (GameObject*) spriteA;
        }

        // Make sure not already dead
        if (!enemyShip.dead && !laser.dead) {

            [enemyShip takeHit];
            [laser takeHit];
```

```
            if ([enemyShip dead]) {

                [[SimpleAudioEngine sharedEngine] playEffect:@"explosion_large.caf"
pitch:1.0f pan:0.0f gain:0.25f];

            } else {

                [[SimpleAudioEngine sharedEngine] playEffect:@"explosion_small.caf"
pitch:1.0f pan:0.0f gain:0.25f];

            }

        }

    }
}

- (void)endContact:(b2Contact *)contact {

}
```

Since **beginContact** will get called whenever any shapes collide (that are marked as collidable with the **category** and **mask** bitfields), the first thing we need to do is check what is colliding.

That is where those constants we defined earlier come in handy. We check here to see if a laser collides with an enemy.

If it does, we call the method on the enemy and laser **GameObjects** to take a hit. This is better than our old method of just destroying the sprites, because now we have objects with multiple hp!

We also play a different sound effect based on whether it got damaged, or completely destroyed.

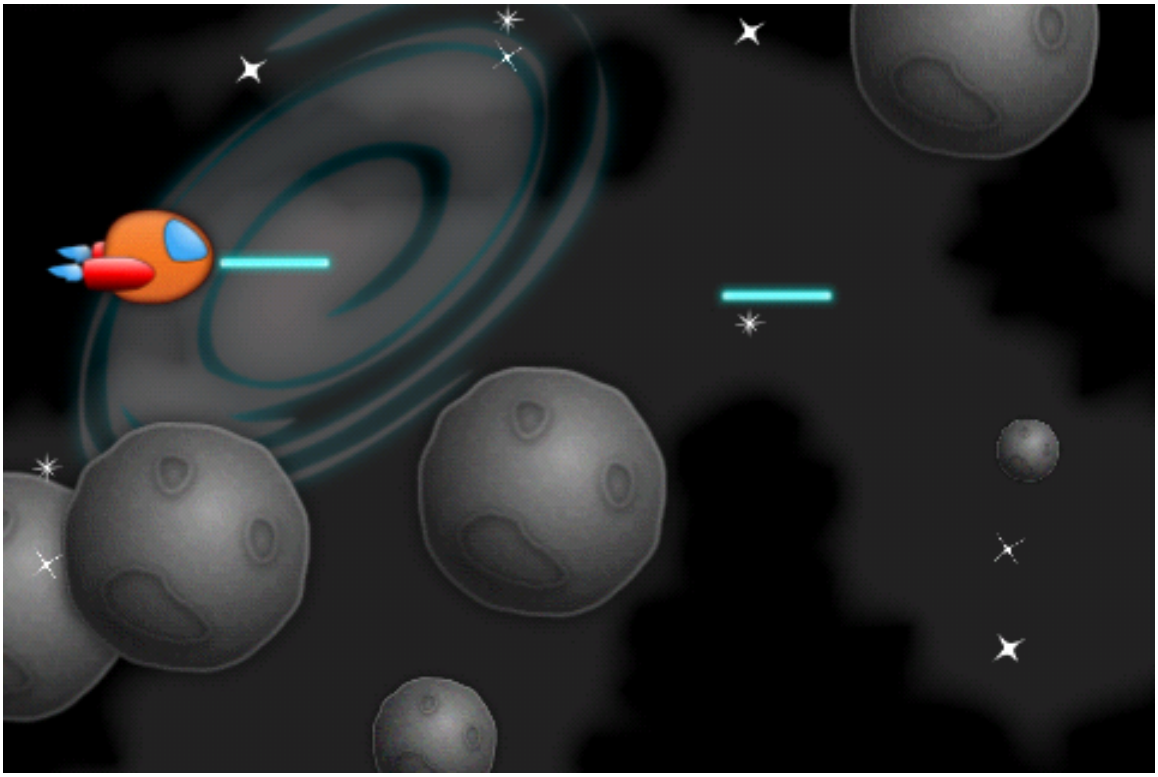
Two final steps: comment out that old collision detection routine in **update** now that you have your new uber awesome method:

```
//[self updateCollisions:dt];
```

And comment out the debug drawing while we're at it in the **draw** method, we don't really need that anymore:

```
//_world->DrawDebugData();
```

Compile and run, and finally you have collision detection the way it's meant to be—accurate and fun! You'll also notice now asteroids take more than one shot to be destroyed!



Explosions and Destruction!

In this section, we're going to add some cool explosions as we shoot down the asteroids.

Rather than creating a new explosion each and every time we destroy an asteroid (which would be inefficient), we're going to take the same strategy we did for reusing sprites. We'll preload an array of explosions, and just reuse them as we need them.

So let's create a helper class to help manager this, quite similar to **SpriteArray**. Go to **File\New\New File**, choose **iOS\Cocoa Touch\Objective-C class**, and click **Next**. Enter **NSObject** for Subclass of, click **Next**, name the new file **ParticleSystemArray.mm**, and click Save.

Open up **ParticleSystemArray.h** and replace the contents with the following:

```
#import "cocos2d.h"

@interface ParticleSystemArray : NSObject {
    CCArray * _array;
    int _nextItem;
}

@property (readonly) CCArray * array;

- (id)initWithFile:(NSString *)file capacity:(int)capacity parent:(CCNode *)parent;
- (id)nextParticleSystem;

@end
```

This is almost exactly like **SpriteArray** which we discussed earlier except for a different initializer, so no need to discuss further here.

Next switch to **ParticleSystemArray.mm** and replace the contents with the following:

```
#import "ParticleSystemArray.h"

@implementation ParticleSystemArray
@synthesize array = _array;

- (id)initWithFile:(NSString *)file capacity:(int)capacity parent:(CCNode *)parent {

    if ((self = [super init])) {

        _array = [[CCArray alloc]
            initWithCapacity:capacity];
        for(int i = 0; i < capacity; ++i) {

            CCParticleSystemQuad *particleSystem =
              [CCParticleSystemQuad particleWithFile:file];
            [particleSystem stopSystem];
            [parent addChild:particleSystem z:10];
            [_array addObject:particleSystem];

        }

    }
    return self;

}
```

```
- (id)nextParticleSystem {
    CCParticleSystemQuad * retval =
        [_array objectAtIndex:_nextItem];
    _nextItem++;
    if (_nextItem >= _array.count) _nextItem = 0;

    // Reset particle system scale
    if (UI_USER_INTERFACE_IDIOM() !=
        UIUserInterfaceIdiomPad) {
        retval.scale = 0.5;
    } else {
        retval.scale = 1.0;
    }

    return retval;
}

- (void)dealloc {
    [_array release];
    _array = nil;
    [super dealloc];
}

@end
```

Again this is very similar to **SpriteArray**, but instead of creating a **GameObject** it creates a **CCParticleSystemQuad**.

Note that the **nextParticleSystem** includes the code to half the size of the particle system if it's not running on the iPad. This is because we created the particle system with iPad dimensions in mind.

To use this, it's quite similar to using a **SpriteArray**. Start by adding the following import to the top of **ActionLayer.h**:

```
#import "ParticleSystemArray.h"
```

And add an instance variable for the array inside the class definition:

```
ParticleSystemArray * _explosions;
```

Then switch to **ActionLayer.mm** and add the following line to the end of **setupArrays** to initialize the array with 3 usable explosions:

```
_explosions = [[ParticleSystemArray alloc] initWithFile:@"Explosion.plist" capacity:3
parent:self];
```

As you can see, the particle system for the explosion is defined in **Explosions.plist**, which you can find under **Art\Particles**. Optionally, you can open this file up with Particle Designer and see how it's set up, and even tweak it if you'd like!

Since you allocated a new explosions array, don't forget to add this line to **dealloc**:

```
[_explosions release];
```

Now that our array is all set up, we can start adding explosions! Navigate to **beginContact**, and place the following code inside the **if ([enemyShip dead]) { }** case:

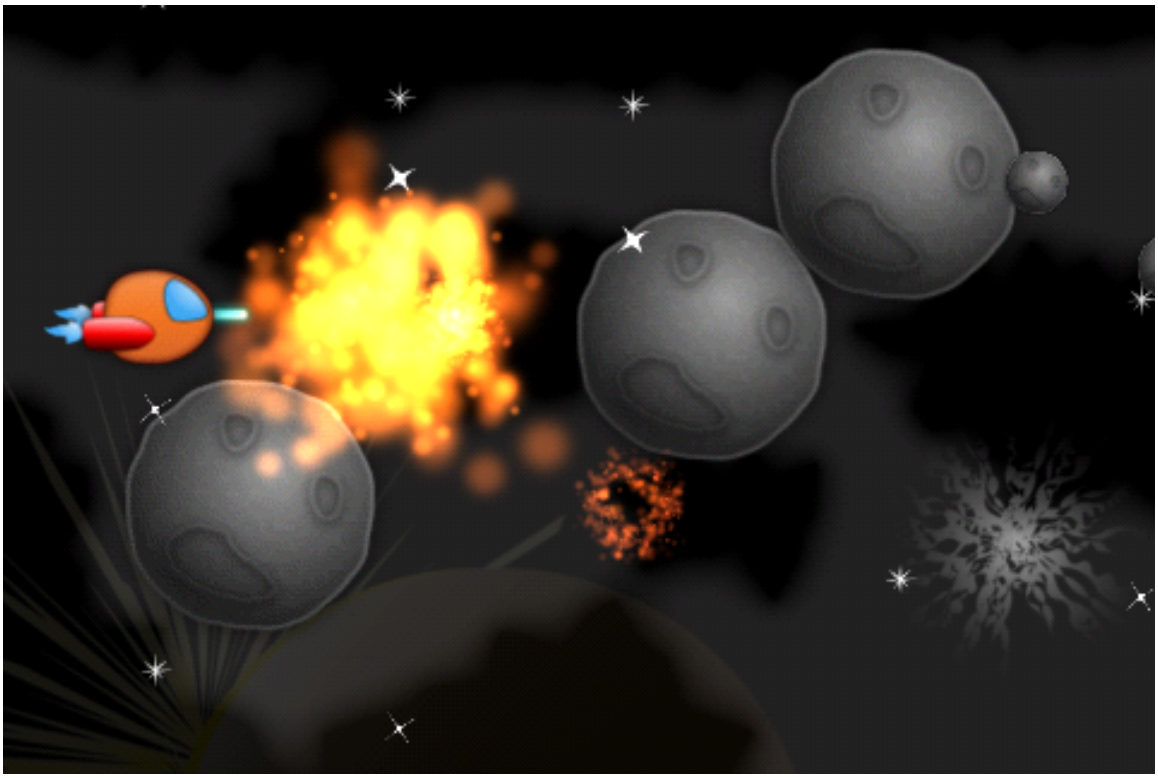
```
CCParticleSystemQuad *explosion =
    [_explosions nextParticleSystem];
explosion.position = enemyShip.position;
[explosion resetSystem];
```

This gets the next available particle system, places it at the enemy ship's position, and calls resetSystem (which makes it play from the beginning).

Similarly, add this code inside the **else {}** case to play a slightly smaller explosion:

```
CCParticleSystemQuad *explosion =
    [_explosions nextParticleSystem];
explosion.scale *= 0.25;
explosion.position = enemyShip.position;
[explosion resetSystem];
```

That's it! Compile and run your code, and you should see some explosions!



As you play the game however, you'll notice that the explosions don't look quite right. When a big asteroid is hit, a small explosion goes off in the center of the asteroid, rather than where you'd expect—where the laser hits the asteroid.

So it would be a lot better if we could place the explosion wherever the laser hits the asteroid. And while we're at it, let's make it even more awesome by making the explosion size based on the asteroid size, and shake the screen when an asteroid gets destroyed!

So go back to **beginContact**, and add the following to the beginning of the method, right after checking if the sprites are visible:

```
b2WorldManifold manifold;
contact->GetWorldManifold(&manifold);
b2Vec2 b2ContactPoint = manifold.points[0];
CGPoint contactPoint = ccp(b2ContactPoint.x * PTM_RATIO, b2ContactPoint.y *
PTM_RATIO);
```

When two shapes collide, Box2D stores some information about the collision in a special data structure called a **manifold**. Since the polygons could intersect at more than one point, it contains all of the points they might intersect.

Here we just want to get any of the contact points, so we pull out the first one in the array and convert it to Cocos2D coordinates.

Now go back to the **if ([enemyShip dead]) { }** case, and replace the line that sets the explosion's position with the following:

```
if (enemyShip.maxHp > 3) {
    [self shakeScreen:6];
    explosion.scale *= 1.0;
} else if (enemyShip.maxHp > 1) {
    [self shakeScreen:3];
    explosion.scale *= 0.5;
} else {
    [self shakeScreen:1];
    explosion.scale *= 0.25;
}
explosion.position = contactPoint;
```

So when an enemy dies, we look at it's max HP. The more HP it has, the more we make the screen shake, and the larger the explosion. Finally we set the position of the explosion to be the contact point (rather than the center of the enemy).

In the **else {}** case, replace the line that sets the explosion's position also:

```
explosion.position = contactPoint;
```

The last thing we need to add is that helper method to shake the screen. Add the **shakeScreen** method to the top of the file:

```
- (void)shakeScreen:(int)times {

    id shakeLow = [CCMoveBy
        actionWithDuration:0.025 position:ccp(0, -5)];
    id shakeLowBack = [shakeLow reverse];
    id shakeHigh =  [CCMoveBy
        actionWithDuration:0.025 position:ccp(0, 5)];
    id shakeHighBack = [shakeHigh reverse];
    id shake = [CCSequence actions:shakeLow, shakeLowBack,
        shakeHigh, shakeHighBack, nil];
    CCRepeat* shakeAction = [CCRepeat
        actionWithAction:shake times:times];

    [self runAction:shakeAction];
}
```

This is a simple method that quickly moves the entire layer down 5 pixels, then up 10 pixels, then back down 5 pixels to the original position. It also repeats this as many times as is specified in the input parameter.

Compile and run, and now you can explode asteroids in style!

Taking Damage

So far our ship has had an easy life. He's been able to fly through space blowing up asteroids, without any threat to his life!

Well things are about to get a lot more dangerous for our space ship, because we're about to show our dark side and add collision detection for when asteroids hit the space ship. If the ship loses all his hit points, he can even—*gasp*—lose the game!

We'll begin our evil plan by creating an enumeration at the top of **ActionLayer.h** to keep track of the two different game stages we're going to have—the asteroid spawning stage, and the game over stage.

```
enum GameStage {
    GameStageAsteroids = 0,
    GameStageDone
};
```

While you're there, also predeclare two variables—one to keep track of the game stage, and one to keep track of whether the game over menu has appeared:

```
GameStage _gameStage;
BOOL _gameOver;
```

Next, switch to **ActionLayer.mm** and add a method at the top of the file to display a "game over" label, along with a menu item you can tap to restart the game. This is a handy method I use in a lot of games, especially as I'm prototyping them.

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

    if (_gameOver) return;
    _gameOver = TRUE;
    _gameStage = GameStageDone;

    CGSize winSize = [CCDirector sharedDirector].winSize;

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

```
    CCLabelBMFont *label;
    if (UI_USER_INTERFACE_IDIOM() ==
        UIUserInterfaceIdiomPad) {
        label = [CCLabelBMFont
            labelWithString:message
            fntFile:@"SpaceGameFont-hd.fnt"];
    } else {
        label = [CCLabelBMFont
            labelWithString:message
            fntFile:@"SpaceGameFont.fnt"];
    }
    label.scale = 0.1;
    label.position = ccp(winSize.width/2,
                         winSize.height * 0.6);
    [self addChild:label];

    CCLabelBMFont *restartLabel;
    if (UI_USER_INTERFACE_IDIOM() ==
        UIUserInterfaceIdiomPad) {
        restartLabel = [CCLabelBMFont
            labelWithString:@"Restart"
            fntFile:@"SpaceGameFont-hd.fnt"];
    } else {
        restartLabel = [CCLabelBMFont
            labelWithString:@"Restart"
            fntFile:@"SpaceGameFont.fnt"];
    }

    CCMenuItemLabel *restartItem = [CCMenuItemLabel
        itemWithLabel:restartLabel target:self
        selector:@selector(restartTapped:)];
    restartItem.scale = 0.1;
    restartItem.position = ccp(winSize.width/2,
                               winSize.height * 0.4);

    CCMenu *menu = [CCMenu menuWithItems:restartItem, nil];
    menu.position = CGPointZero;
    [self addChild:menu];

    [restartItem runAction:[CCScaleTo
        actionWithDuration:0.5 scale:0.5]];
    [label runAction:[CCScaleTo actionWithDuration:0.5
        scale:0.5]];

}
```

Although this method looks like a lot of code, it's actually doing very simple stuff we've done several times before:

- Checks if the game over menu has already appeared, and bails if so.
- Creates a label saying "You win!" or "You lose!" depending on what's passed in to the method.
- Creates a label and menu item saying "Restart" and display it in a menu on the screen.
- Zooms in the label and menu item from 0 to 0.5 scale to make a neat zoom in effect.

When the "Restart" label is tapped it calls **restartTapped**, so add the code for that next:

```
- (void)restartTapped:(id)sender {

    // Reload the current scene
    CCScene *scene = [ActionLayer scene];
    [[CCDirector sharedDirector] replaceScene:
        [CCTransitionZoomFlipX transitionWithDuration:0.5
            scene:scene]];

}
```

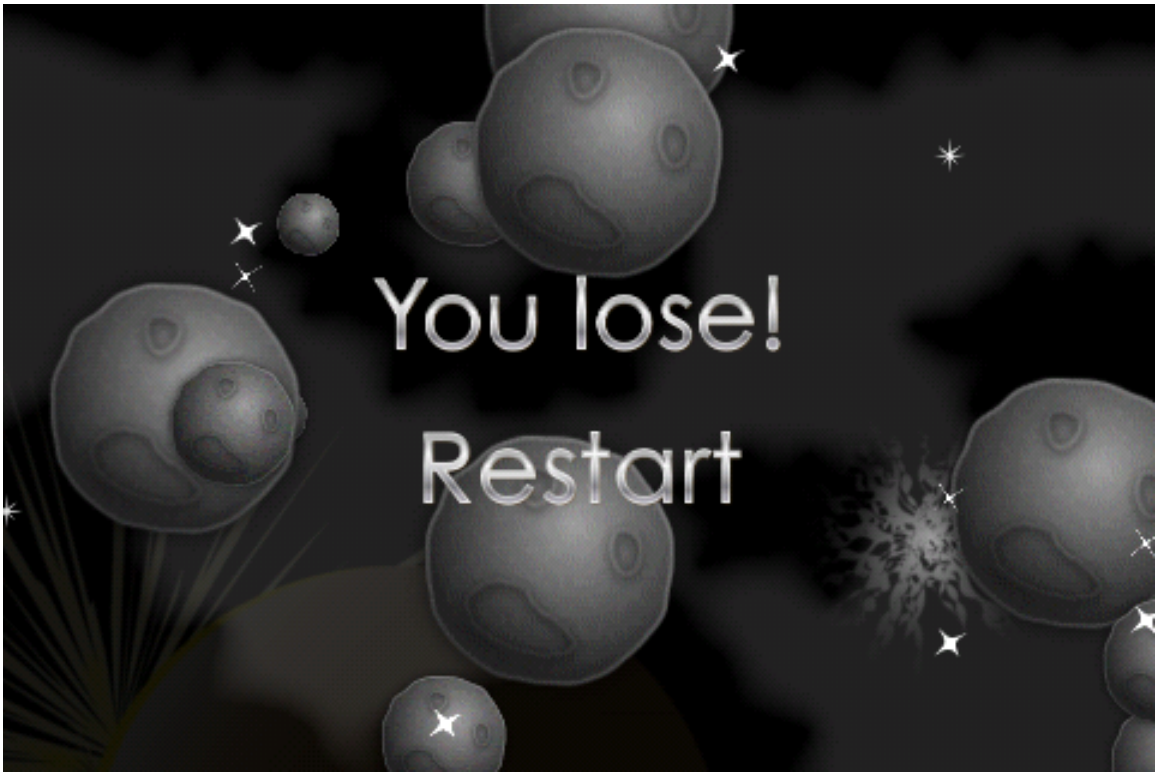
This creates a fresh copy of the **ActionLayer**'s scene (hence re-initializing everything) and replaces the current scene with this new scene. It also wrap things in a cool transition animation for style.

One last step—to add the code for checking for an enemy colliding with the space ship! Add the following to the end of **beginContact**:

```
if ((fixtureA->GetFilterData().categoryBits & kCategoryShip && fixtureB->GetFilterData().categoryBits & kCategoryEnemy) ||
    (fixtureB->GetFilterData().categoryBits & kCategoryShip && fixtureA->GetFilterData().categoryBits & kCategoryEnemy)) {

    // Determine enemy ship
    GameObject *enemyShip = (GameObject*) spriteA;
    if (fixtureB->GetFilterData().categoryBits & kCategoryEnemy) {
        enemyShip = spriteB;
    }

    if (!enemyShip.dead) {

        [[SimpleAudioEngine sharedEngine] playEffect:@"explosion_large.caf" pitch:1.0f pan:0.0f gain:0.25f];

        [self shakeScreen:1];
        CCParticleSystemQuad *explosion = [_explosions nextParticleSystem];
        explosion.scale *= 0.5;
        explosion.position = contactPoint;
        [explosion resetSystem];

        [enemyShip destroy];
        [_ship takeHit];

        if (_ship.dead) {
            [self endScene:NO];
        }

    }

}
```

This checks to see if an enemy collides with the ship, and if so plays an explosion, shakes the screen, destroys the enemy, and makes the ship take a hit. If the ship is dead after this, it displays the Game Over menu.

Compile and run your code, and feel the power of the dark side!



As you've been playing with the game, you might have noticed something strange. When the main menu shows up, if you tap somewhere other than "Play", it still shoots a laser even though there's no ship!

That's because right now we aren't keeping track of another game stage—when the main menu is up. When we're in that stage, we don't want to shoot or spawn asteroids. Go to **ActionLayer.h** and replace the **GameStage** enum with the following:

```
enum GameStage {
    GameStageTitle = 0,
    GameStageAsteroids,
    GameStageDone
};
```

Switch to **ActionLayer.mm**, and add the following to the beginning of **updateAsteroids**:

```
if (_gameStage != GameStageAsteroids) return;
```

This makes it so that asteroids don't spawn except if we're in the asteroids game stage. This is good because it prevents our ship from being hit as soon as he comes out, or asteroids to spawn endlessly even after the game is over.

When the Play button is tapped we want to switch to the asteroids stage, so add the following to the bottom of the **playTapped** method:

```
_gameStage = GameStageAsteroids;
```

One last step—add this line of code to the beginning of **ccTouchesBegan** to prevent the laser from shooting when it shouldn't!

```
if (_ship == nil || _ship.dead) return;
```

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

Winning the Game

As Yoda would say, although turning to the dark side is more quick and seductive, a true Xcode jedi uses his power for good.

So it's time to give our space ship a fighting chance to win the game! For now we'll simply make it so that if the space ship can survive for 30 seconds, he wins!

This is real easy. Add the following instance variable to the class definition in **ActionLayer.h**:

```
double _gameWonTime;
```

Switch to **ActionLayer.mm** and add the following lines to initialize this variable at the bottom of **init**:

```
double curTime = CACurrentMediaTime();
_gameWonTime = curTime + 30.0;
```

This sets the **_gameWonTime** variable to 30 seconds in the future.

Finally, add the following lines of code to the bottom of the **update** method:

```
if (CACurrentMediaTime() > _gameWonTime) {
    [self endScene:YES];
}
```

Compile and run the game, and may the accelerometer force be with you!

Where To Go From Here?

No more do we have to teach you in the swamp of De-go-cocos2D—at least in this tutorial :] At this point, you have a complete but simple space game where you navigate through a dangerous asteroid belt, complete with accurate collision detection and awesome explosions!

Just like last time, feel free to stop at this point and use this as a starting point for your own game. Here are a few features you might want to add:

- It might be cool to start the asteroids off spawning slow, and as time goes on make them spawn more and more often or go faster and faster. Then the challenge can be how long your space ship can survive.
- It might be fun to have an uber asteroid to spawn once in a while, that is huge (maybe as big as the entire screen!) but has tons of hp.
- You could add different types of objects that fly toward you—instead of asteroids, what about angry cows or space birds?

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

Tutorial 3: Multiple Levels and Aliens!

Let's take it to the next level!

So far, we just have one "level" in our game—our space ship blasting through the asteroid belt.

However, for our full game we want multiple levels. The first level will be the asteroid belt, the second level will be a swarm of aliens, and the final level will end with a big boss fight.

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

So in this tutorial, we're going to modify the game so that it supports multiple levels, defined in an external property list file. Each level will have multiple stages, and in each stage we'll let you decide whether you want to spawn asteroids, aliens, text—or all three!

We're going to pick up where we left off last tutorial. If you skipped last tutorial, you can continue on with the **SpaceGame2** project that comes with the Starter Kit.

It's time to take our game to the next level!

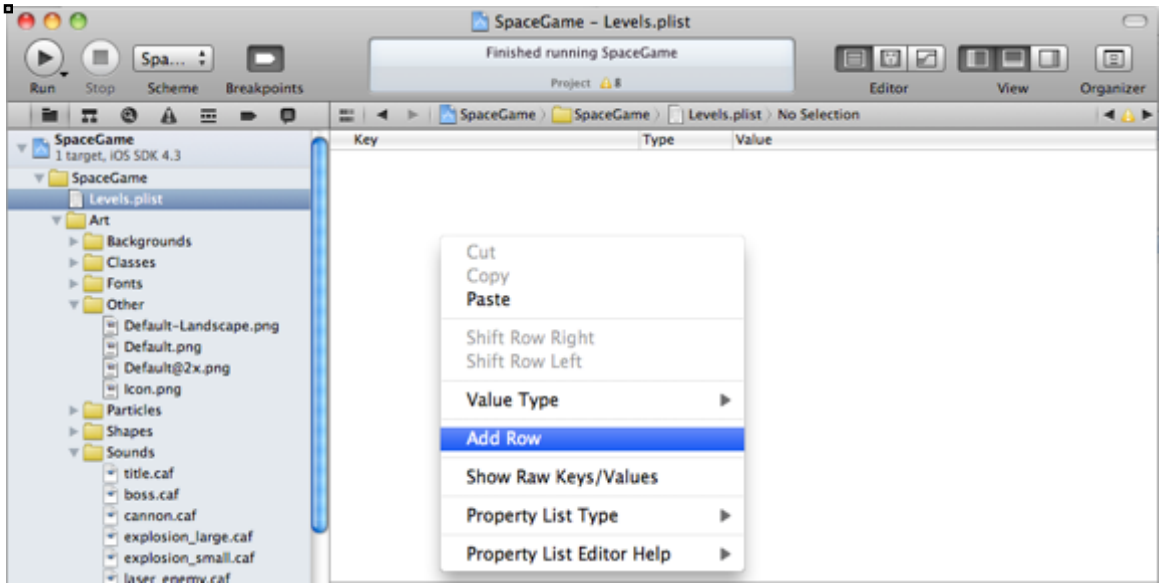
Creating a Property List for the Levels

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

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

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

Select **Levels.plist**, and a blank editor for the property list will show up. Control-click somewhere inside, and select **Add Row**, as shown below:

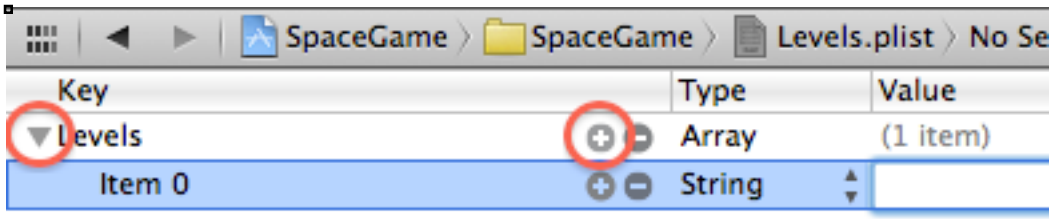


A new row will show up, and you'll see it has three columns:

- **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 is 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.

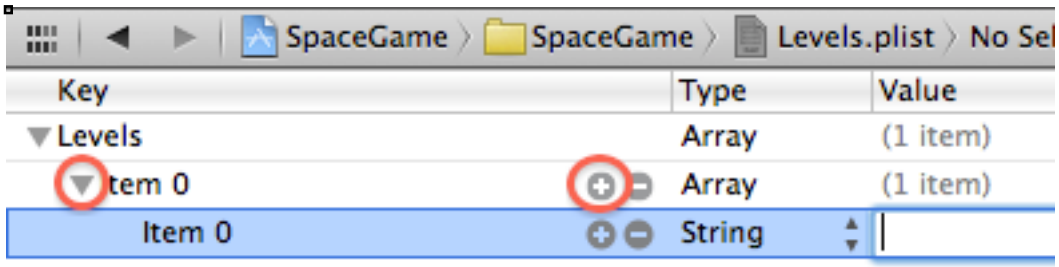
For this first row, we want it to be a list of all the levels. So double click the **Key** column and change it to read **Levels**. Then click the **Type** column and change it to **Array**.

We want to add a row for the first level, so first click the arrow next to the **Levels** key so the arrow points down, then click the plus button next to the **Levels** key to create a new sub-item:

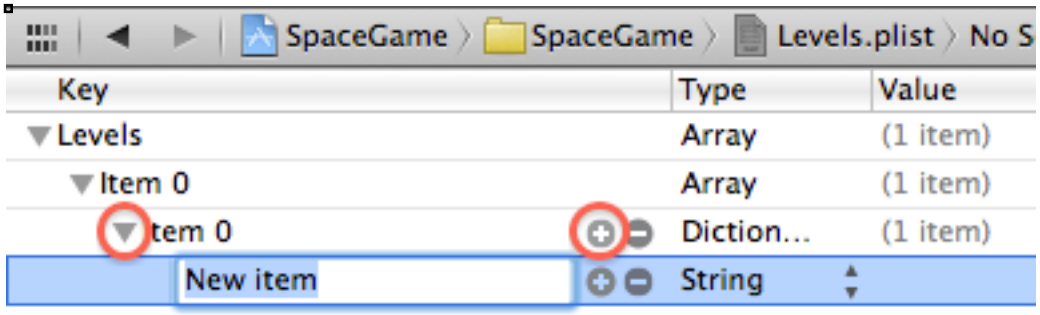


For this sub-item, notice that you can't change the **Key**—that's because Key only really applies if the item is contained within a dictionary (yet this is contained within an array).

Our level is going to be broken down 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. We want to add a bunch of different properties about the stage, so change the **Key** to **Dictionary**, and then add a new sub-item.



Notice how now you can change the **Key** 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 we're going to use in the game to tell whether this stage should spawn asteroids or not.

Now repeat this process to set up several more properties for this stage, as shown below:

Key	Type	Value
▼ Levels	Array	(1 item)
▼ Item 0	Array	(1 item)
▼ Item 0	Diction…	(6 items)
SpawnAsteroids	Boolean	YES
Duration	Number	15
AMoveDurationHigh	Number	10
AMoveDurationLow	Number	2
ASpawnSecsHigh	Number	0.9
ASpawnSecsLow	Number	0.2

We're going to use this data in our game as follows:

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

We just made up these strings and values based on what we need for this game. When you're making your own game, you can add anything else you need!

Let's see a bit more about how this property list works. Control-click **Levels.plist**, and select **Show in Finder**. Then control-click on **Levels.plist** in Finder, select **OpenWith\Other**, and select **TextEdit.app**. You should see **Levels.plist** as plaintext, as shown here:

Levels.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Levels</key>
        <array>
                <array>
                        <dict>
                                <key>SpawnAsteroids</key>
                                <true/>
                                <key>Duration</key>
                                <integer>15</integer>
                                <key>AMoveDurationHigh</key>
                                <integer>10</integer>
                                <key>AMoveDurationLow</key>
                                <integer>2</integer>
                                <key>ASpawnSecsHigh</key>
                                <real>0.9</real>
                                <key>ASpawnSecsLow</key>
                                <real>0.2</real>
                        </dict>
                </array>
        </array>
</dict>
</plist>
```

As you can see here, it's human readable and makes a lot of sense how it works—an array with an inner array, with a dictionary inside that. Then a bunch of keys and values inside the dictionary.

The reason I show you this is that sometimes it's more convenient to edit the file in a text editor like this rather than using the property list editor. The property list editor is great for small tweaks, but sometimes just copying and pasting with a text editor goes a lot faster.

As this tutorial continues, we're going to be making a bunch of changes to **Levels.plist**. Rather than having you continuously edit this file (and having to list each and every change here), I've made some pre-edited versions of **Levels.plist** that you can use for each stage of development (and overwrite 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 we have a file with data about our levels, let's modify our game to use it!

Adding Multi-Level Support

Rather than clutter up our **ActionLayer** with a bunch of extra code to read this file, we're going to make a helper class called **LevelManager** to contain all the code.

Go to **File\New\New File**, choose **iOS\Cocoa Touch\Objective-C class**, and click **Next**. Enter **NSObject** for Subclass of, click **Next**, name the new file **LevelManager.m**, and click Save.

Then open up **LevelManager.h** and replace it with the following:

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

typedef enum {
    GameStateTitle = 0,
    GameStateNormal,
    GameStateDone
} GameState;

@interface LevelManager : NSObject {

    GameState _gameState;
    double _stageStart;
    double _stageDuration;

    NSDictionary * _data;
    NSArray * _levels;
    int _curLevelIdx;
    NSArray * _curStages;
    int _curStageIdx;
    NSDictionary * _curStage;
}

@property (assign) GameState gameState;
@property (readonly) int curLevelIdx;

- (void)nextStage;
- (void)nextLevel;
- (BOOL)update;
- (float)floatForProp:(NSString *)prop;
```

```
- (NSString *)stringForProp:(NSString *)prop;
- (BOOL)boolForProp:(NSString *)prop;
- (BOOL)hasProp:(NSString *)prop;

@end
```

It's important to understand what's going on here, so let's go over it bit by bit.

First, the file contains an enum for the **GameStates** for the game. Note this is different than the stages inside a level—rather these are the main overall states such as "title screen showing", "going through levels and stages like normal", and "game done."

The **LevelManager** class has instance variables to keep track of the current game state, as well as the times the current stage has started (and how long it should run in seconds).

The rest of the instance variables are references to the data pulled from **Levels.plist**. It stores 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.

Finally it contains some helper methods that we'll be using from **ActionLayer**. There's a method to advance to the next stage or next level, a method to give **LevelManager** time to update itself, and some helper methods to return properties for the current stage (or check if they exist).

Switch to **LevelManager.m** and replace the file with its implementation:

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

@implementation LevelManager
@synthesize gameState = _gameState;
@synthesize curLevelIdx = _curLevelIdx;

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

        NSString *levelDefsFile =
            [[NSBundle mainBundle]
                pathForResource:@"Levels" ofType:@"plist"];
        _data =
            [[NSDictionary
                dictionaryWithContentsOfFile:levelDefsFile]
                    retain];
```

```
        NSAssert(_data != nil,
             @"Couldn't open Levels file");

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

        _curLevelIdx = -1;
        _curStageIdx = -1;
        _gameState = GameStateTitle;

    }
    return self;
}

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

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

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

- (BOOL)boolForProp:(NSString *)prop {
    NSNumber * retval =  (NSNumber *)
        [_curStage objectForKey:prop];
    if (!retval) return FALSE;
    return [retval boolValue];
}
```

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

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

    _gameState = GameStateNormal;
    _curStage = [_curStages objectAtIndex:_curStageIdx];

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

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

}

- (BOOL)update {
    if (_gameState == GameStateTitle ||
        _gameState == GameStateDone) return FALSE;
    if (_stageDuration == -1) return FALSE;

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

    return FALSE;
}

- (void)dealloc {
    [_data release];
```

```
    [super dealloc];
}

@end
```

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

The most important code is inside **init**. This gets the path for **Levels.plist** inside the main bundle (this is where all of the files that are included in your Xcode project are stored). 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 we made in the dictionary, which is that array we created with all of the level data.

hasProp, **stringForProp**, **floatForProp**, **boolForProp** are all 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 stage 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'll see that **nextStage** is where it sets up the **_curStage** dictionary. It also 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**).

update will be called by **ActionLayer** 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 this method returns a **BOOL** that indicates if the game has advanced to a new stage. We'll need this in **ActionLayer**, because sometimes we need to perform special actions when the game first enters a new stage.

OK—the hard part is done, now we just need to make use of this! Open up **ActionLayer.h** and add the following import to the top of the file:

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

Then go inside the class definition and comment out the old **_gameWonTime** variable since we won't be using them anymore, and add a new instance variable for the **LevelManager**:

```
//double _gameWonTime;
```

```
LevelManager * _levelManager;
```

Also add the following method declaration to the bottom of the file:

```
- (void)newStageStarted;
```

This is a new method we're going to write to contain the special code we need when a new stage starts up, and we need to call it from all over the file so we're predeclaring it here so we don't have to worry about method ordering.

Switch to **ActionLayer.mm**, and add a new method to initialize the level manager right above **init**:

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

Add the line to call this at the bottom of **init** and comment out the previous lines initializing the game won time:

```
//double curTime = CACurrentMediaTime();
//_gameWonTime = curTime + 30.0;
[self setupLevelManager];
```

Before we forget, add the line to **dealloc** to free the memory for this:

```
[_levelManager release];
```

Now that we have the **LevelManager** set up, we can start to use it. At the bottom of **playTapped**, comment out the old line setting the game stage and advance the stage with **LevelManager** instead:

```
//_gameStage = GameStageAsteroids;
[_levelManager nextStage];
[self newStageStarted];
```

Then switch to **updateAsteroids**, comment out the first line and add the **LevelManager** equivalent instead:

```
//if (_gameStage != GameStageAsteroids) return;
if (_levelManager.gameState != GameStateNormal) return;
if (![_levelManager boolForProp:@"SpawnAsteroids"]) return;
```

Basically we don't want to spawn asteroids if we're not in the normal game state, or if the current stage doesn't have the **SpawnAsteroids** property.

Also inside updateAsteroids, update the lines creating the randSecs, randY, and randDuration to use the values from the LevelManager as follows:

```
float spawnSecsLow =
    [_levelManager floatForProp:@"ASpawnSecsLow"];
float spawnSecsHigh =
    [_levelManager floatForProp:@"ASpawnSecsHigh"];
float randSecs =
    randomValueBetween(spawnSecsLow, spawnSecsHigh);
_nextAsteroidSpawn = randSecs + curTime;

float randY = randomValueBetween(0.0, winSize.height);

float moveDurationLow =
    [_levelManager floatForProp:@"AMoveDurationLow"];
float moveDurationHigh =
    [_levelManager floatForProp:@"AMoveDurationHigh"];
float randDuration = randomValueBetween(moveDurationLow, moveDurationHigh);
```

So now instead of having these values hardcoded, they come from whatever you put into **Levels.plist**!

Next, go to **beginContact** and inside the **if (ship.dead) {}** statement add the following to mark the game state as done:

```
 _levelManager.gameState = GameStateDone;
```

Also inside the **endScene** method, comment out the line of code that used to set the old **_gameStage** variable:

```
//_gameStage = GameStageDone;
```

Almost done! Next we just need to give the **LevelManager** time to update each frame, so add the following new methods right above **update**:

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

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

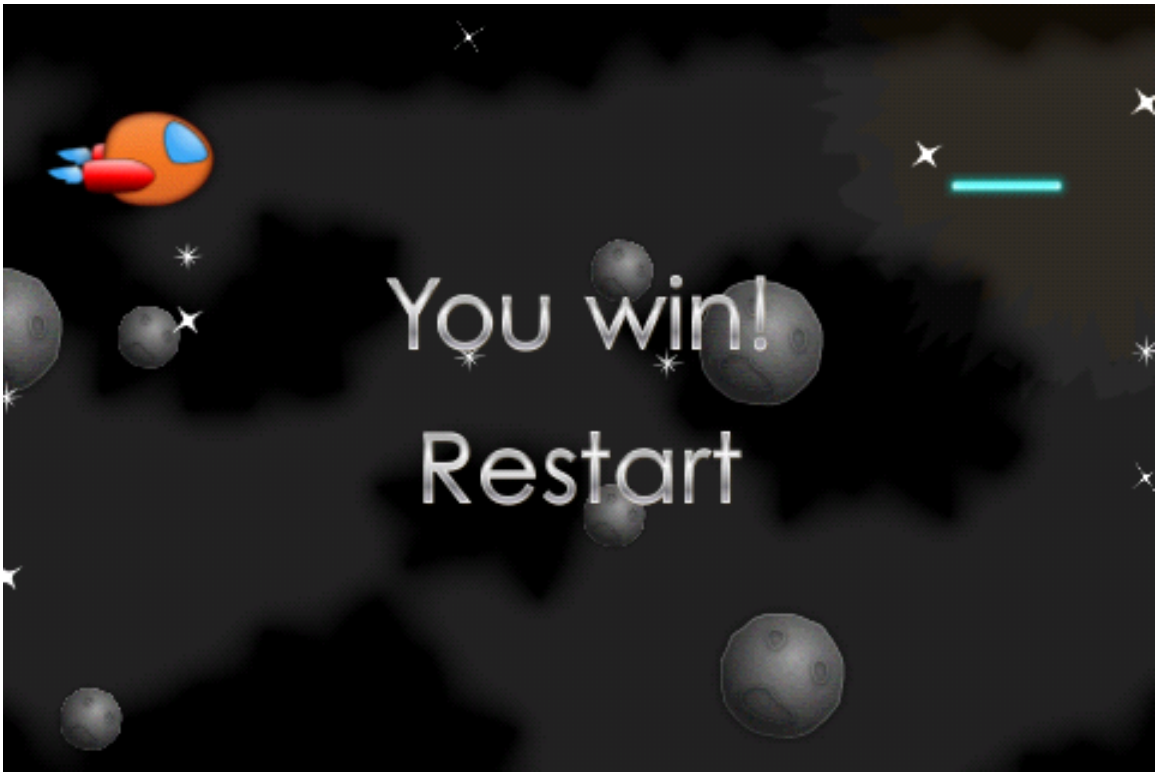
updateLevel calls update each frame, and if it returns TRUE (which means a new stage has started) it calls **newStageStarted**.

newStageStarted checks to see if the game has advanced to the **GameStateDone** state, and if so calls the **endScene** method to display the Game Over text and Restart menu.

The final step is just to add the line of code to the end of **update** to call this, and comment out the old game over check:

```
//if (CACurrentMediaTime() > _gameWonTime) {
//    [self endScene:YES];
//}
[self updateLevel:dt];
```

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



You might not quite appreciate the beauty of this yet, since right now things look kinda the same.

So let me show you how uber this is. 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 new Levels.plist is picked up (and do this whenever replacing Levels.plist in the future).

Then run your game again—and you'll now 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.

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

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

Adding Level Intro Text

Since we're going to have multiple levels in this game, it would be cool if we displayed a title of the level on the screen as each level begins.

Now that we have this 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 up with the property list editor you'll see that it contains a new level stage:

Key	Type	Value
▼ Levels	Array	(1 item)
▼ Item 0	Array	(4 items)
▼ Item 0	Diction…	(3 items)
Duration	Number	2
LText	String	Asteroid Belt
SpawnLevelIntro	Boolean	YES
▼ Item 1	Diction…	(6 items)
AMoveDurationHigh	Number	10
AMoveDurationLow	Number	2
ASpawnSecsHigh	Number	1
ASpawnSecsLow	Number	0.2
Duration	Number	15
SpawnAsteroids	Boolean	YES
▶ Item 2	Diction…	(6 items)
▶ Item 3	Diction…	(6 items)

This has the key **SpawnLevelIntro**, which we'll check for to see if we 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 2 seconds.

Now check out how easy this is to use! Start by adding two instance variables for the labels we'll use to the class definition in **ActionLayer.h**:

```
CCLabelBMFont *_levelIntroLabel1;
CCLabelBMFont *_levelIntroLabel2;
```

Then add this new method to spawn the text right above **newStageStarted**:

```
- (void)doLevelIntro {

    CGSize winSize = [CCDirector sharedDirector].winSize;
    NSString *fontName = @"SpaceGameFont.fnt";
    if (UI_USER_INTERFACE_IDIOM() ==
        UIUserInterfaceIdiomPad) {
        fontName = @"SpaceGameFont-hd.fnt";
    }

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

    _levelIntroLabel1 =
        [CCLabelBMFont labelWithString:message1
            fntFile:fontName];
    _levelIntroLabel1.scale = 0;
    _levelIntroLabel1.position =
        ccp(winSize.width/2, winSize.height * 0.6);
    [self addChild:_levelIntroLabel1 z:100];
    [_levelIntroLabel1 runAction:
     [CCSequence actions:
      [CCEaseOut actionWithAction:
       [CCScaleTo actionWithDuration:0.5 scale:0.5]
       rate:4.0],
      [CCDelayTime actionWithDuration:3.0],
      [CCEaseOut actionWithAction:
       [CCScaleTo actionWithDuration:0.5 scale:0]
       rate:4.0],
      [CCCallFuncN actionWithTarget:self
          selector:@selector(removeNode:)],
      nil]];

    _levelIntroLabel2 =
        [CCLabelBMFont labelWithString:message2
            fntFile:fontName];
    _levelIntroLabel2.position =
        ccp(winSize.width/2, winSize.height * 0.4);
    _levelIntroLabel2.scale = 0;
    [self addChild:_levelIntroLabel2 z:100];
    [_levelIntroLabel2 runAction:
```

```
    [CCSequence actions:
     [CCEaseOut actionWithAction:
      [CCScaleTo actionWithDuration:0.5 scale:0.5]
     rate:4.0],
     [CCDelayTime actionWithDuration:3.0],
     [CCEaseOut actionWithAction:
      [CCScaleTo actionWithDuration:0.5 scale:0]
     rate:4.0],
     [CCCallFuncN actionWithTarget:self
         selector:@selector(removeNode:)],
     nil]];

}
```

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

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

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

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

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

An Alien Swarm

Just when our space ship thinks he's made it through the asteroid belt to safety—he comes across an alien swarm!

We're going to make our game able to spawn aliens in our game stages. The aliens will move in a curved path through the screen, and (eventually) try their best to shoot our space craft down via deadly laser beams.

For this section, we're just going to focus on getting the aliens to show up in a second level of the game, have a bunch of them spawn in a curved path.

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

If you open this up, you'll see that there's an array for a new level, with two stages:

Key	Type	Value
▶ Item 2	Diction…	(6 items)
▶ Item 3	Diction…	(6 items)
▼ Item 1	Array	(2 items)
▼ Item 0	Diction…	(3 items)
Duration	Number	2
LText	String	Uber Nova
SpawnLevelIntro	Boolean	YES
▼ Item 1	Diction…	(7 items)
AMoveDurationHigh	Number	10
AMoveDurationLow	Number	2
ASpawnSecsHigh	Number	3
ASpawnSecsLow	Number	1
Duration	Number	30
SpawnAlienSwarm	Boolean	YES
SpawnAsteroids	Boolean	YES

The first stage displays the text for level 2, which we're calling Uber Nova. Why? Because it sounds cool! :]

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

You may also notice that I've shortened the asteroid stages to 5 seconds each so we can get quickly to the aliens for testing.

OK, so let's add some aliens! Switch to **ActionLayer.h** and add the following new instance variables:

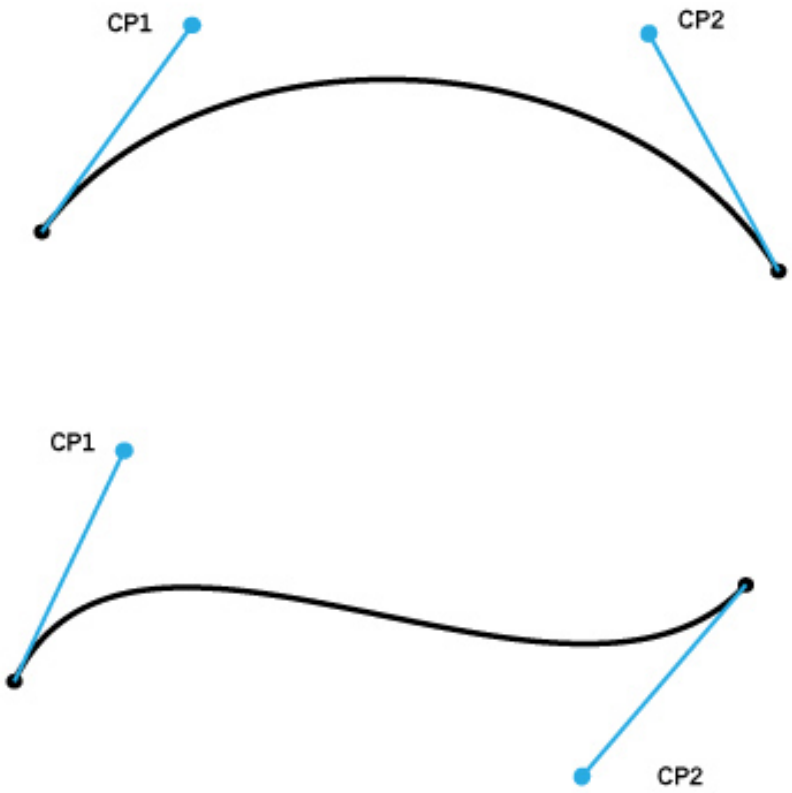
```
SpriteArray * _alienArray;
double _nextAlienSpawn;
double _numAlienSpawns;
CGPoint _alienSpawnStart;
ccBezierConfig _bezierConfig;
```

Since we're going to be creating a bunch of alien sprites, we once again use our trusty friend **SpriteArray** to help us reuse these sprites.

We also keep track of the next time we want to spawn an alien, and the number of remaining alien spawns in this "wave". We have a point at which we'll start spawning aliens for the "wave", and finally something weird called a **ccBezierConfig**.

Huh? What in the heck is a **ccBezierConfig**?

A **ccBezierConfig** is the data structure you use to create a Bezier path in Cocos2D. If you are not familiar with a Bezier path, it's a curved line that you can create by defining a start and end point, and two control points, as you can see below:



Think of the control points as "pulling" the curve in the direction of the control points. The best way to figure out what to set for them is via experimentation and debug drawing, which we'll cover as we continue on!

Switch to **ActionLayer.mm**, and add the code to initialize the array of aliens at the bottom of **setupArrays**:

```
_alienArray = [[SpriteArray alloc] initWithCapacity:15
spriteFrameName:@"enemy_spaceship.png" batchNode:_batchNode world:_world
shapeName:@"enemy_spaceship" maxHp:1];
```

While we're thinking about the alien array, add the code to release it in **dealloc** as well:

```
[_alienArray release];
```

Now for the fun part—the code to spawn the aliens! Add the following new method right above **update**:

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

    if (_levelManager.gameState != GameStateNormal) return;
    if (![_levelManager hasProp:@"SpawnAlienSwarm"])
        return;

    CGSize winSize = [CCDirector sharedDirector].winSize;

    double curTime = CACurrentMediaTime();
    if (curTime > _nextAlienSpawn) {

        if (_numAlienSpawns == 0) {
            CGPoint pos1 = ccp(winSize.width*1.3,
                randomValueBetween(0, winSize.height*0.1));
            CGPoint cp1 =
                ccp(randomValueBetween(winSize.width*0.1,
                                       winSize.width*0.6),
                randomValueBetween(0, winSize.height*0.3));
            CGPoint pos2 = ccp(winSize.width*1.3,
                randomValueBetween(winSize.height*0.9,
                                   winSize.height*1.0));
            CGPoint cp2 =
                ccp(randomValueBetween(winSize.width*0.1,
                                       winSize.width*0.6),
                randomValueBetween(winSize.height*0.7,
                                   winSize.height*1.0));
            _numAlienSpawns = arc4random() % 20 + 1;
            if (arc4random() % 2 == 0) {
                _alienSpawnStart = pos1;
                _bezierConfig.controlPoint_1 = cp1;
                _bezierConfig.controlPoint_2 = cp2;
```

```
                _bezierConfig.endPosition = pos2;
            } else {
                _alienSpawnStart = pos2;
                _bezierConfig.controlPoint_1 = cp2;
                _bezierConfig.controlPoint_2 = cp1;
                _bezierConfig.endPosition = pos1;
            }

            _nextAlienSpawn = curTime + 1.0;

        } else {

            _nextAlienSpawn = curTime + 0.3;

            _numAlienSpawns -= 1;

            GameObject *alien = [_alienArray nextSprite];
            alien.position = _alienSpawnStart;
            [alien revive];

            [alien runAction:
                [CCBezierTo actionWithDuration:3.0
                    bezier:_bezierConfig]];
        }
    }
}
```

This code takes the following strategy:

- Each "wave" of aliens, we'll choose a random number of aliens to spawn in the wave between 1 and 20 and will figure out their path they should move in by choosing four random points:
  - **pos1**: Create a point offscreen (x-axis) in the top half of the screen (y-axis).
  - **cp1**: Create a point on the left side of the screen (x-axis), in the top fourth (y-axis).
  - **pos2**: Create a point offscreen (x-axis) in the bottom half of the screen (y-axis).
  - **cp2**: Create a point on the left side of the screen (x-axis), in the bottom fourth (y-axis).
- We can use these points to construct a bezier curve. If we set the start to pos1, the end to pos2, 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 we reverse pos1/pos2 and the control points, they'll go bottom to top instead.

- Choose a random number so half the time the aliens spawn bottom to top, and half the time top to bottom.
- Every time we need to spawn an alien in the wave, we get the next available alien sprite and run an action to move it along the pre-created Bezier curve.

Next add the line to call this at the end of your **update** method:

```
[self updateAlienSwarm:dt];
```

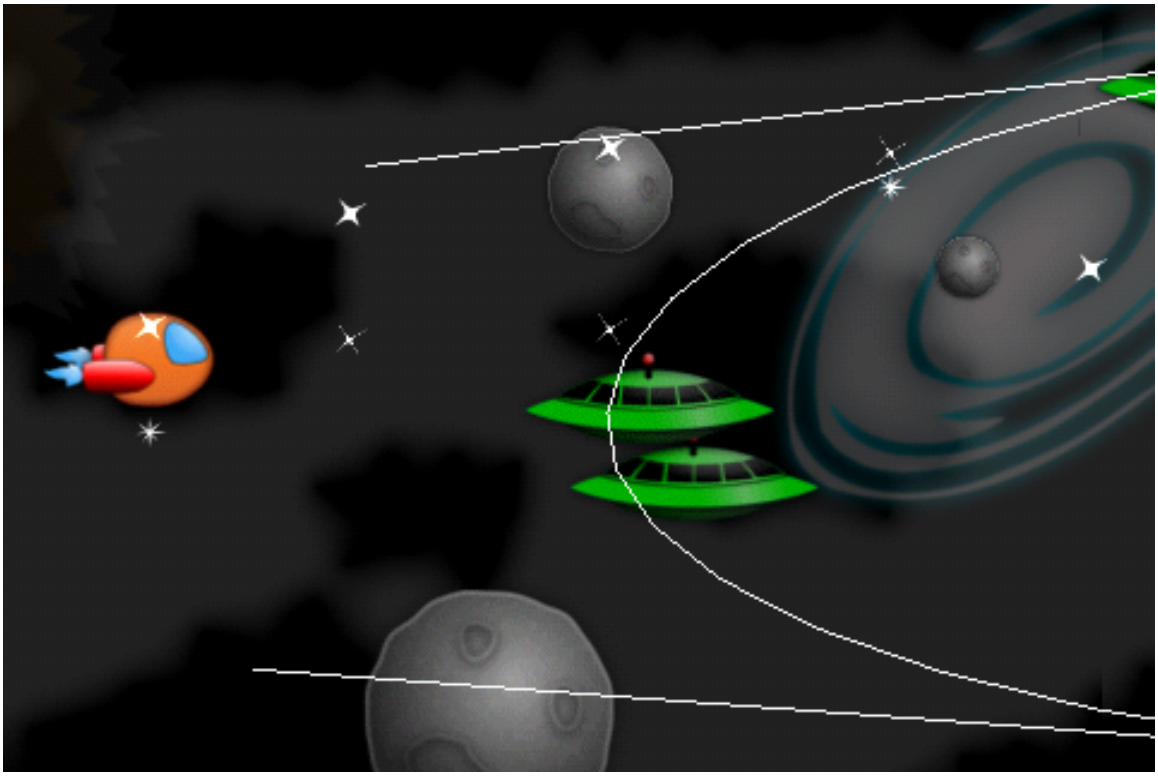
One final step. In your own games, you might want to tweak the paths the aliens move on, but it's notoriously hard to visualize what's going on with Bezier paths in your head. So here's some code you can add to debug draw the Bezier paths to the screen so you can modify them visually.

Add the following at the bottom of your **draw** method:

```
if (_levelManager.gameState == GameStateNormal &&
    [_levelManager boolForProp:@"SpawnAlienSwarm"]) {

    ccDrawCubicBezier(_alienSpawnStart,
        _bezierConfig.controlPoint_1,
        _bezierConfig.controlPoint_2,
        _bezierConfig.endPosition, 16);
    ccDrawLine(_alienSpawnStart,
        _bezierConfig.controlPoint_1);
    ccDrawLine(_bezierConfig.endPosition,
        _bezierConfig.controlPoint_2);

}
```

This uses some built-in Cocos2D functions to draw the Bezier path we created in **updateAlienSwarm**, as well as the lines between the start/end points and their control points. Seeing this visually makes it a lot easier to understand what's going on and tweak it.

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



Aliens Shooting Lasers

It's definitely cool seeing the aliens fly in, but right now there's no challenge. The way we've set up the Bezier curves, the aliens will never hit our space ship, so there's no way the aliens can hurt him!

But that will all change in this section—because they're going to open fire on our space ship!

Start by adding two new instance variables to **ActionLayer.h**:

```
double _nextShootChance;
SpriteArray * _enemyLasers;
```

This will keep track of the next time an alien might shoot, and an array of reusable enemy lasers.

Switch to **ActionLayer.mm**, and add the code to initialize the laser array at the end of **setupArrays**:

```
_enemyLasers = [[SpriteArray alloc] initWithCapacity:15
spriteFrameName:@"laserbeam_red.png" batchNode:_batchNode world:_world
shapeName:@"laserbeam_red" maxHp:1];
```

Also add the code to release this in **dealloc**:

```
[_enemyLasers release];
```

Now for the important stuff. Add the following code to the bottom of **updateAlienSwarm**:

```
if (curTime > _nextShootChance) {
    _nextShootChance = curTime + 0.1;

    for (GameObject *alien in _alienArray.array) {
        if (alien.visible) {
            if (arc4random() % 40 == 0) {
                [self shootEnemyLaserFromPosition:
                    alien.position];
            }
        }
    }
}
```

So every so often, we loop through all of the aliens, and give them a 1/40 chance at shooting a laser from their current position.

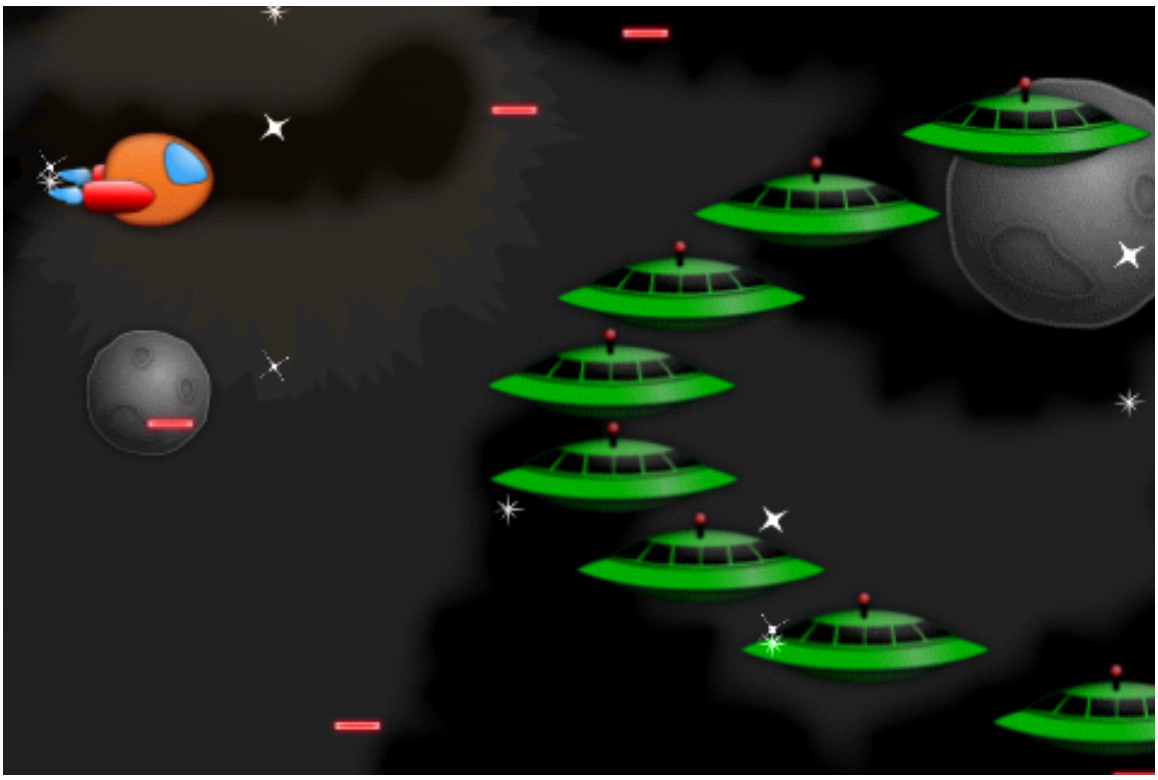
Finally add the **shootEnemyLaserFromPosition** method, right above **updateAlienSwarm**:

```
- (void)shootEnemyLaserFromPosition:(CGPoint)position {
    CGSize winSize = [CCDirector sharedDirector].winSize;
    GameObject *shipLaser = [_enemyLasers nextSprite];

    [[SimpleAudioEngine sharedEngine] playEffect:@"laser_enemy.caf" pitch:1.0f
pan:0.0f gain:0.25f];
    shipLaser.position = position;
    [shipLaser revive];
    [shipLaser stopAllActions];
    [shipLaser runAction:
     [CCSequence actions:
      [CCMoveBy actionWithDuration:2.0
          position:ccp(-winSize.width, 0)],
      [CCCallFuncN actionWithTarget:self
          selector:@selector(invisNode:)],
      nil]];
}
```

This simply gets the next available laser, plays a sound effect, and makes it move across the screen to the left, and disappears when it's done.

Since the shape for this laser that I created in Physics Editor is marked as belonging in the "enemy" category, if it collides with the ship it will cause the ship to lose an HP—so we don't even have to write any extra code for that!

Compile and run, and now you have a challenging second level with an alien swarm firing lasers! You may also want to comment out the Bezier path debug drawing since we don't need it anymore.



Adding a Power Up

We can't have a space game without adding some kind of cool power up our ship can collect!

In this section, we'll just focus on adding the power up itself—in the next section we'll do something cool with it.

First of all, you need to update your **Levels.plist** to the version in **Levels\V5** (and don't forget to **Product\Clean**). This contains a new **SpawnPowerups** boolean, which we set to true in

Level 1 Stage 2 and Level 2 Stage 1. There's also a variable **PSpawnSecs** which will determine the time between spawning powerups.

Next, open **ActionLayer.h** and add the following new instance variables:

```
SpriteArray * _powerups;
double _nextPowerupSpawn;
```

This is the reusable array of powerups, and the next time to spawn a powerup.

Switch to **ActionLayer.mm** and add the code to initialize the powerups array at the end of **setupArrays**:

```
_powerups = [[SpriteArray alloc] initWithCapacity:1 spriteFrameName:@"powerup.png"
batchNode:_batchNode world:_world shapeName:@"powerup" maxHp:1];
```

Also add the code to release this in **dealloc**:

```
[_powerups release];
```

Now for the important part—the code to spawn the powerups. Add the following new method right above **update**:

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

    if (_levelManager.gameState != GameStateNormal) return;
    if (![_levelManager boolForProp:@"SpawnPowerups"])
        return;

    CGSize winSize = [CCDirector sharedDirector].winSize;

    double curTime = CACurrentMediaTime();
    if (curTime > _nextPowerupSpawn) {
        _nextPowerupSpawn = curTime +
          [_levelManager floatForProp:@"PSpawnSecs"];

        GameObject * powerup = [_powerups nextSprite];
        powerup.position = ccp(winSize.width,
            randomValueBetween(0, winSize.height));
        [powerup revive];
        [powerup runAction:
         [CCSequence actions:
          [CCMoveBy actionWithDuration:5.0
              position:ccp(-winSize.width*1.5, 0)],
          [CCCallFuncN actionWithTarget:self
```

```
                selector:@selector(invisNode:)],
            nil]];
    }

}
```

This checks to see when it's time to spawn a power up, and when it's time it grabs the next available power up and moves it offscreen to the left.

Call this new method at the bottom of **update**:

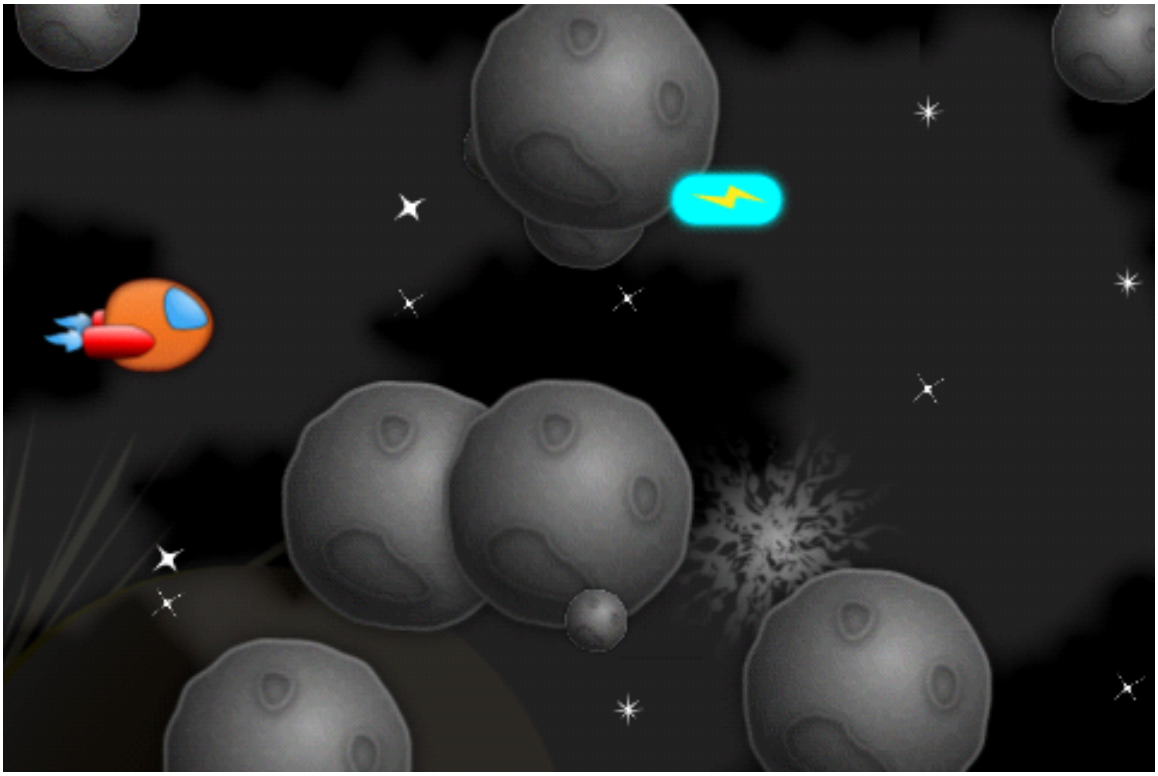
```
[self updatePowerups:dt];
```

Currently, if the ship collides with the power up nothing would happen, because we don't have any code checking for a collision between shapes with **kCategoryShip** and **kCategoryPowerup**. So add the code to check for that to the bottom of **beginContact** next:

```
if ((fixtureA->GetFilterData().categoryBits & kCategoryShip && fixtureB-
>GetFilterData().categoryBits & kCategoryPowerup) ||
    (fixtureB->GetFilterData().categoryBits & kCategoryShip && fixtureA-
>GetFilterData().categoryBits & kCategoryPowerup)) {

    // Determine power up
    GameObject *powerUp = (GameObject*) spriteA;
    if (fixtureB->GetFilterData().categoryBits & kCategoryPowerup) {
        powerUp = spriteB;
    }

    if (!powerUp.dead) {
        [[SimpleAudioEngine sharedEngine] playEffect:@"powerup.caf" pitch:1.0 pan:0.0
gain:1.0];

        [powerUp destroy];
        // TODO: Make the powerup do something!
    }
}
```

This is just some test code to make sure the collision detection code is working – when the ship and power up collide, we play a sound effect and destroy the power up.

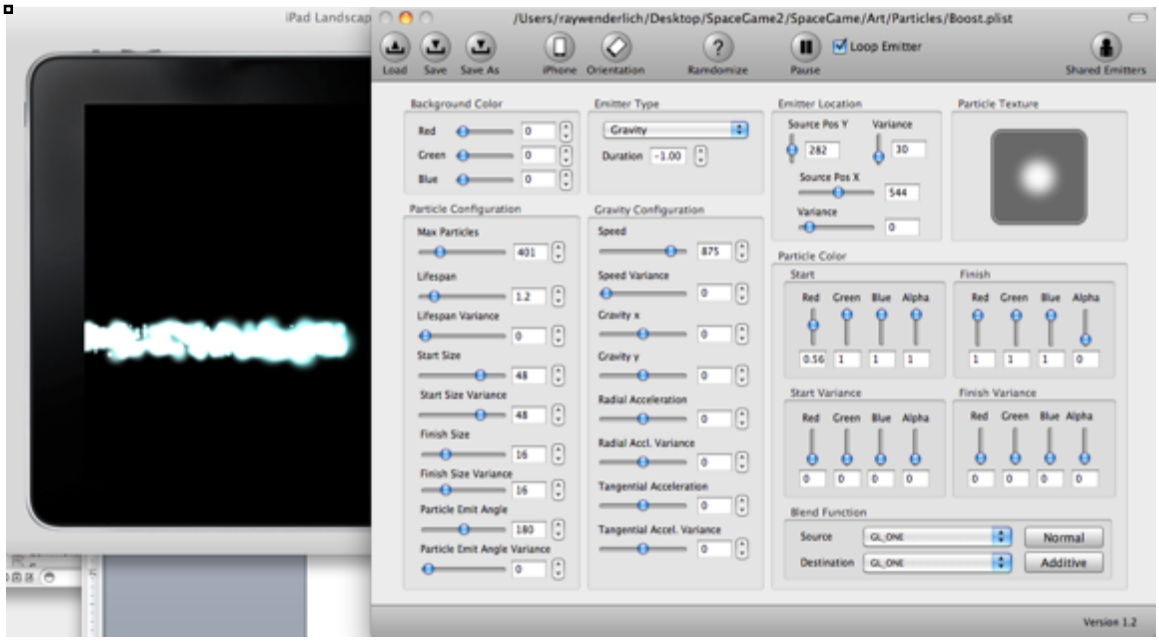
Compile and run your code, and after a time a power up will appear. Try to collect it, and if all works well a power up sound effect will play!



Full Thrusters Ahead!

For the final part of this tutorial, we're going to add a kick-ass special effect and ability when the ship picks up the power up. We'll make the ship zoom ahead with powerful thrusters and smash through asteroids and aliens with full invincibility!

We're going to use a special effect for this that you can find under **Art\Particles\Boost.plist**. Optionally, you can open it up with Particle Designer and tweak it if you'd like:



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

Open up **ActionLayer.h** and add the following instance variables:

```
BOOL _invincible;
ParticleSystemArray * _boostEffects;
```

The first is a flag to keep track if we're invincible or not, and the second is our array of reusable boost effects.

Switch to **ActionLayer.mm** and add the code to set up the array at the end of **setupArrays**:

```
_boostEffects = [[ParticleSystemArray alloc] initWithFile:@"Boost.plist" capacity:1
parent:self];
```

Also add the code to release the array in **dealloc**:

```
[_boostEffects release];
```

Next, add the following new method right above **update**:

```
- (void)updateBoostEffects:(ccTime)dt {
    for (CCParticleSystemQuad * particleSystem in _boostEffects.array) {
        particleSystem.position = _ship.position;
    }
}
```

We want the boost effect to shoot behind the ship, so each frame we update the boost effect's position to be the same as the ship's position. This way, as the ship moves the particle system will move with it.

Call this new method at the bottom of **update**:

```
[self updateBoostEffects:dt];
```

Now it's time to add the code that starts the boost effect. Not only are we going to start the special effect, but we're also going to move the ship forward and make the entire layer zoom out, to make it look totally awesome.

Inside **beginContact**, right after where you have the comment "TODO: Make the powerup do something!", add the following code:

```
float scaleDuration = 1.0;
float waitDuration = 5.0;
_invincible = YES;
CCParticleSystemQuad *boostEffect = [_boostEffects nextParticleSystem];
[boostEffect resetSystem];

[_ship runAction:
 [CCSequence actions:
  [CCMoveBy actionWithDuration:scaleDuration position:ccp(winSize.width * 0.6, 0)],
  [CCDelayTime actionWithDuration:waitDuration],
  [CCMoveBy actionWithDuration:scaleDuration position:ccp(-winSize.width * 0.6, 0)],
  nil]];

[self runAction:
 [CCSequence actions:
  [CCScaleTo actionWithDuration:scaleDuration scale:0.75],
  [CCDelayTime actionWithDuration:waitDuration],
  [CCScaleTo actionWithDuration:scaleDuration scale:1.0],
  [CCCallFunc actionWithTarget:self selector:@selector(boostDone)],
  nil]];
```

This marks the ship as invincible and starts up a particle system. It then moves the ship forward by 60% of the screen width, waits 5 seconds, then moves him back.

At the same time, it scales the entire layer to 75% of its original size. This is why we've been making the stars spawn beyond the height of the screen, and why we've been creating ships offscreen more than just past the width of the screen—so that when we're zoomed out everything will still look OK.

When the special effect is done, it calls **boostDone**. Add the code for this next:

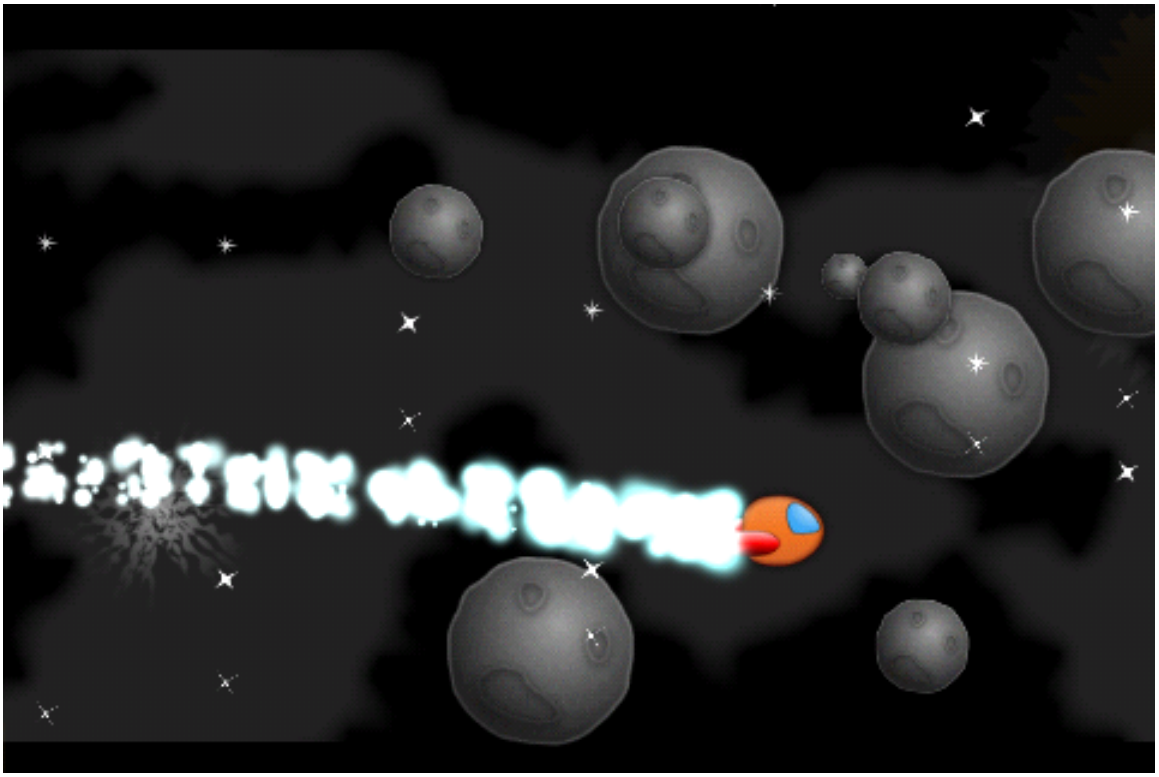
```
- (void)boostDone {
    _invincible = NO;
    for (CCParticleSystemQuad * boostEffect in _boostEffects.array) {
        [boostEffect stopSystem];
    }
}
```

This turns off invincibility and stops the boost effect.

One final step. Search for the **[_ship takeHit]** line and put a wrapper around it so the ship doesn't take a hit if it's invincible:

```
if (!_invincible) {
    [_ship takeHit];
}
```

You did it! Compile 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 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, explosions, and best of all—you can configure the levels via a simple text file!

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

- You could start editing **Levels.plist** to create your own levels. You could add different combinations of asteroids and aliens, ramping up the difficulty harder and harder as the space ship travels through the system!
- Now that you know how easy it is to add different types of things you're spawning into **Levels.plist**, follow the examples of adding spawning aliens, text, power ups, and asteroids to add something else that spawns. Maybe you'll want to add another type of alien that moves differently (perhaps in space invader style rather than in curves)?
- Why not add some more power ups? The first one that comes to mind is a power up that improves your laser in some way. Maybe makes it multi-fire, or makes it shoot three lasers at once?

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

Tutorial 4: The Final Boss Fight

A Dark Nemesis Awaits…

Our space ship is having a great adventure so far—he's made it through the asteroid belt, and he's battled the alien swarms at Uber Nova.

But meanwhile, in a galaxy far far away, a dark nemesis has been plotting an evil plan for our space ship's demise.

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

Then, we'll add the big boss into the game. He's a lot more complicated than enemies we'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.

We're going to pick up where we left off last tutorial. If you skipped last tutorial, you can continue on with the SpaceGame3 project that comes with the Starter Kit.

Get ready for the final showdown—our space ship is about to meet his match!

Creating a Health Bar

We're going to modify **GameObject** so it can optionally display a health bar that shows how much health the object has remaining. This will be really helpful to see how much our ship has been damaged—and the boss, too.

Note that there are simpler ways of creating a health bar than this (you can go as simple as drawing a line above the enemies to show their health, or using **CCProgressTimer / CCProgressTo**), but this method looks really good because we can customize it with any artwork we'd like, animate the health decreasing or increasing, plus demonstrate a cool API call you might find useful for other reasons in the future.

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

```
typedef enum {
    HealthBarTypeNone = 0,
    HealthBarTypeGreen,
    HealthBarTypeRed
} HealthBarType;
```

This declares the type of health bar to display for the GameObject. The default will be none , but if we do want to display a health bar we have two variants—green (for the ship), and red (for the boss).

Next add a bunch of instance variables to the class:

```
HealthBarType _healthBarType;
CCSprite * _healthBarBg;
CCSprite * _healthBarProgress;
CCSpriteFrame * _healthBarProgressFrame;
float _fullWidth;
float _displayedWidth;
```

_healthBarType keeps track of the type of health bar to display, as discussed above. The health bar is made of two sprites—a background sprite (**_healthBarBg**), and a sprite that lies on top, that fills up a portion of the health bar to show how much health remains (**_healthBarProgress**).

_healthBarProgressFrame is a reference to the sprite frame for the progress bar, and finally it also keeps track of the full width of the health bar, and the currently displayed portion of the progress (**_displayedWidth**).

It's important to note that the health bar might actually display something different than the ship's current health. For example, say the ship has 50% health, then gets hit down to 25% health. Rather than making the health bar immediately jump from 50% to 25%, over several frames it will gradually decrease the health bar until it reaches the desired value—i.e. 48%, 46%, and so on until it reaches 25%. This will make it look nice and smooth—and that's what **_displayedWidth** tracks!

Next, replace the initializer with the following to take an extra **healthBarType** parameter:

```
- (id)initWithSpriteFrameName:(NSString *)spriteFrameName world:(b2World *)world
shapeName:(NSString *)shapeName maxHp:(float)maxHp
healthBarType:(HealthBarType)healthBarType;
```

Switch to **GameObject.mm**, and replace the initializer there as well:

```
- (id)initWithSpriteFrameName:(NSString *)spriteFrameName world:(b2World *)world
shapeName:(NSString *)shapeName maxHp:(float)maxHp
healthBarType:(HealthBarType)healthBarType {
```

Then add the following code to the end of the method:

```
_healthBarType = healthBarType;
[self setupHealthBar];
[self scheduleUpdate];
```

This stores away the reference to the health bar type, calls a method we're about to write to set up the health bar, and finally schedules this object to receive a call to its **update** method each frame.

You might not have seen that before—it's important to realize that you can make any object receive a call to **update**—not just the layer! Here we want the game object to be updated each frame so we can animate the health bar if necessary.

Next, add the implementation of **setupHealthBar** right above the initializer:

```
- (void)setupHealthBar {

    if (_healthBarType == HealthBarTypeNone) return;

    _healthBarBg = [CCSprite spriteWithSpriteFrameName:
        @"healthbar_bg.png"];
    _healthBarBg.position = ccpAdd(self.position,
        ccp(self.contentSize.width/2,
            -_healthBarBg.contentSize.height));
    [self addChild:_healthBarBg];

    NSString *progressSpriteName;
    if (_healthBarType == HealthBarTypeGreen) {
        progressSpriteName = @"healthbar_green.png";
    } else {
        progressSpriteName = @"healthbar_red.png";
    }
    _healthBarProgressFrame =
        [[CCSpriteFrameCache sharedSpriteFrameCache]
            spriteFrameByName:progressSpriteName];
    _healthBarProgress =
        [[CCSprite spriteWithSpriteFrameName:
            progressSpriteName] retain];
    _healthBarProgress.position =
        ccp(_healthBarProgress.contentSize.width/2,
            _healthBarProgress.contentSize.height/2);
    _fullWidth = _healthBarProgress.textureRect.size.width;
    [_healthBarBg addChild:_healthBarProgress];

}
```

There are some subtle but important things to point out about this method:

- The health bar background is added as a child of the game object. That means that as the game object moves, the health bar background will move with it.
- The position of the health bar background is with respect to the bottom left of the game object. So we center it along the game object on the x-axis (by setting its x-coordinate to half the size of the game object) , and set it to show up slightly below the game object on the y-axis.
- The health bar progress is added as a child of the health bar background. So now it's position is relative to the background (not the game object)! So to center it, we just add half the size of the progress bar along the x and y axis.

- Why did we add it as a child of the health bar background instead of the game object? Either way would work fine, it's just that this way makes some of the math easier later.
- We also store away the full width of the texture for later usage and the sprite frame of the progress bar.

Next, add the **update** method that will be called every frame, whose job is to keep the health bar progress up to date:

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

    if (_healthBarType == HealthBarTypeNone) return;

    float POINTS_PER_SEC = 50;

    float percentage = _hp / _maxHp;
    percentage = MIN(percentage, 1.0);
    percentage = MAX(percentage, 0);
    float desiredWidth = _fullWidth *percentage;

    if (desiredWidth < _displayedWidth) {
        _displayedWidth = MAX(desiredWidth,
            _displayedWidth - POINTS_PER_SEC*dt);
    } else {
        _displayedWidth = MIN(desiredWidth,
            _displayedWidth + POINTS_PER_SEC*dt);
    }

    CGRect oldTextureRect = _healthBarProgress.textureRect;
    CGRect newTextureRect = CGRectMake(
        oldTextureRect.origin.x, oldTextureRect.origin.y,
        _displayedWidth, oldTextureRect.size.height);

    CGRect rectInPixels =
        CC_RECT_POINTS_TO_PIXELS( newTextureRect );
    [_healthBarProgress setTextureRectInPixels:rectInPixels
        rotated:_healthBarProgressFrame.rotated
        untrimmedSize:
            _healthBarProgressFrame.originalSizeInPixels];

    _healthBarProgress.position = ccp(_displayedWidth/2,
         _healthBarProgress.contentSize.height/2);

}
```

The **update** method first bails if there's no progress bar to display.

It then figures out the ship's health percentage by dividing the hp by the max hp. It restricts this to at least 0% and at most 100%.

It then figures out the desired width of the health bar by multiplying the full width of the health bar by the current health percentage.

If the current displayed width is less than or greater than the desired width, it incremements or decrements the displayedWidth at a certain rate – **POINTS_PER_SEC**. It uses **dt** (the delta time elapsed since the last frame) to determine the exact number of pixels to increment for this frame.

Now for an interesting part. Somehow, we need to make only a certain part of the progress bar texture show up, based on this desired width. How do you do that with Cocos2D?

Well, one simple way is you can use the **setTextureRectInPixels** method to specify a subset of the texture to display. We have to create a rectangle for this first—we make sure to use the old texture's origin when creating it because it comes from a sprite sheet and will likely be nonzero.

After setting the new texture size to display, we need to move the sprite so it's flush with the left side of the background. So we set its position to its current displayed width/2, which will align it to the left hand side.

One more step and then we're done with GameObject.mm. In **revive**, we need to set the displayed width back to the full width, in order to avoid the health bar animating from 0 back up to full.

```
_displayedWidth = _fullWidth;
```

Open **SpriteArray.h**, and modify the initializer to take the **healthBarType** parameter:

```
- (id)initWithCapacity:(int)capacity spriteFrameName:(NSString *)spriteFrameName
batchNode:(CCSpriteBatchNode *)batchNode world:(b2World *)world shapeName:(NSString
*)shapeName maxHp:(int)maxHp healthBarType:(HealthBarType)healthBarType;
```

Then switch to **SpriteArray.mm** and replace its initializer there too:

```
- (id)initWithCapacity:(int)capacity spriteFrameName:(NSString *)spriteFrameName
batchNode:(CCSpriteBatchNode *)batchNode world:(b2World *)world shapeName:(NSString
*)shapeName maxHp:(int)maxHp healthBarType:(HealthBarType)healthBarType {
```

And modify the line to create the **GameObject** to pass this new parameter through:

```
GameObject *sprite = [[[GameObject alloc] initWithSpriteFrameName:spriteFrameName
world:world shapeName:shapeName maxHp:maxHp healthBarType:healthBarType] autorelease];
```

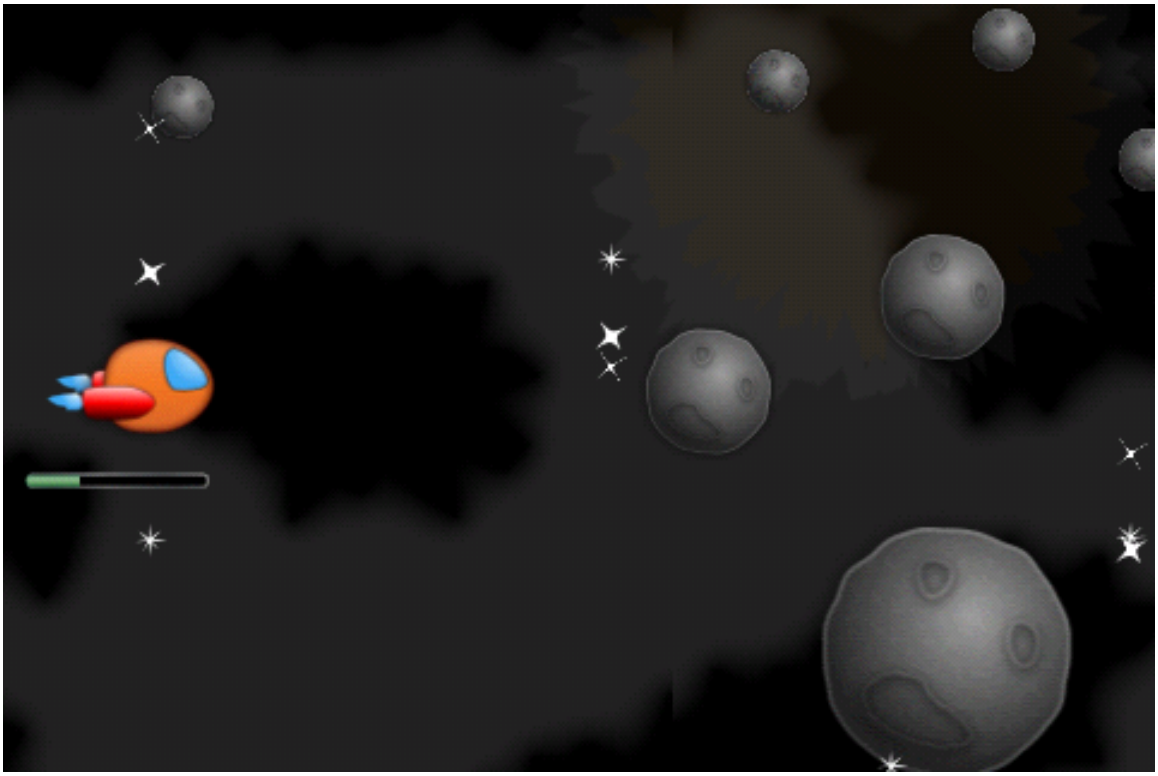
Then switch to **ActionLayer.mm** and modify the line in **spawnShip** that creates the ship to make the health bar type green:

```
_ship = [[[GameObject alloc] initWithSpriteFrameName:@"SpaceFlier_sm_1.png"
world:_world shapeName:@"SpaceFlier_sm_1" maxHp:10 healthBarType:HealthBarTypeGreen]
autorelease];
```

The final step is to modify **setupArrays** to set the **healthBarType** to none for every other type of object:

```
- (void)setupArrays {
    _asteroidsArray = [[SpriteArray alloc] initWithCapacity:30
spriteFrameName:@"asteroid.png" batchNode:_batchNode world:_world
shapeName:@"asteroid" maxHp:1 healthBarType:HealthBarTypeNone];
    _laserArray = [[SpriteArray alloc] initWithCapacity:15
spriteFrameName:@"laserbeam_blue.png" batchNode:_batchNode world:_world
shapeName:@"laserbeam_blue" maxHp:1 healthBarType:HealthBarTypeNone];
    _explosions = [[ParticleSystemArray alloc] initWithFile:@"Explosion.plist"
capacity:3 parent:self];
    _alienArray = [[SpriteArray alloc] initWithCapacity:15
spriteFrameName:@"enemy_spaceship.png" batchNode:_batchNode world:_world
shapeName:@"enemy_spaceship" maxHp:1 healthBarType:HealthBarTypeNone];
    _enemyLasers = [[SpriteArray alloc] initWithCapacity:15
spriteFrameName:@"laserbeam_red.png" batchNode:_batchNode world:_world
shapeName:@"laserbeam_red" maxHp:1 healthBarType:HealthBarTypeNone];
    _powerups = [[SpriteArray alloc] initWithCapacity:1 spriteFrameName:@"powerup.png"
batchNode:_batchNode world:_world shapeName:@"powerup" maxHp:1
healthBarType:HealthBarTypeNone];
    _boostEffects = [[ParticleSystemArray alloc] initWithFile:@"Boost.plist"
capacity:1 parent:self];
}
```

That's it! Compile and run, and you should now see a health bar on 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 kind of annoying always looking at that health bar. It ruins the beautiful free-space flying effect!

So let's modify the health bar to only show up when the health changes—and fade away otherwise. That way players can get an update of their health when they are hurt, but enjoy the beautiful scenery otherwise.

This is pretty easy to accomplish. In **GameObject.mm**'s update method, add the following to the bottom:

```
if (desiredWidth != _displayedWidth) {
    _healthBarBg.visible = TRUE;
    [_healthBarBg stopAllActions];
    [_healthBarBg runAction:
     [CCSequence actions:
      [CCFadeTo actionWithDuration:0.25 opacity:255],
      [CCDelayTime actionWithDuration:2.0],
      [CCFadeTo actionWithDuration:0.25 opacity:0],
      [CCCallFunc actionWithTarget:self selector:@selector(fadeOutDone)],
      nil]];
}
```

So every time the desired width doesn't equal the displayed width (i.e. it's animating), we run an action to fade in to full visibility, wait 2 seconds, then fade out again.

If it's already visible, this won't change anything until 2 seconds later, because when you call **CCFadeTo** specifying full opacity, and it's already full opacity, nothing will change. Also note that it calls **stopAllActions** at the beginning, which is important because it needs to cancel any previous actions that might be trying to fade out after a while.

When it's fully faded out it calls **fadeOutDone**, so add its code:

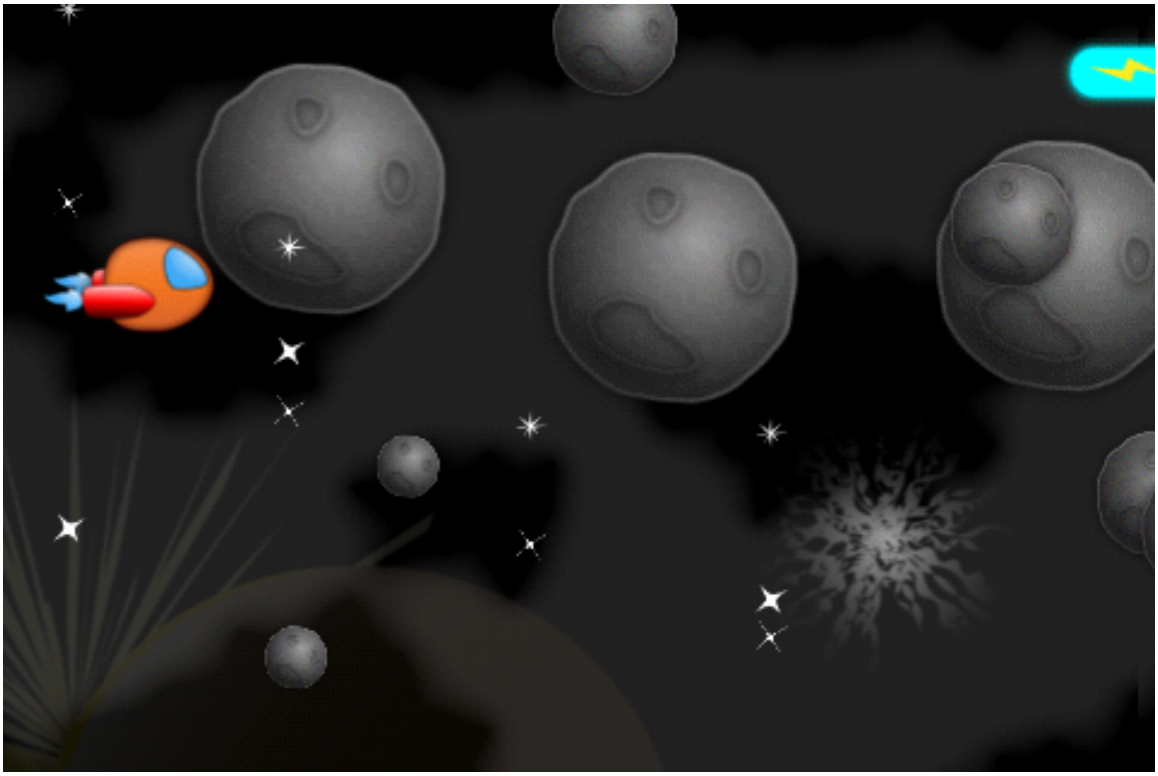
```
- (void)fadeOutDone {
    _healthBarBg.visible = FALSE;
}
```

This simply sets it to invisible. You may wonder why even bother, since the opacity is already 0 and hence it isn't visible anyway. Well marking a node as invisible gives a performance optimization—no need to draw something that isn't visible—so it's good practice.

One final step. Modify **revive** to start the health bar out as invisible:

```
_healthBarBg.visible = NO;
```

And you're done! Compile and run, and now the health bar will only show up when necessary.



But wait! You might notice a subtle problem. When the health bar fades out, it looks like the background fades out OK, but the progress portion just seems to instantly pop in and out.

That is because when you set the opacity of a node, it does not affect the opacity of its children nodes by default. We need to manually fade out the progress sprite separately!

In **GameObject.mm**'s **update** method, add this code right after running the actions on **_healthBarBg**:

```
[_healthBarProgress stopAllActions];
[_healthBarProgress runAction:
 [CCSequence actions:
  [CCFadeTo actionWithDuration:0.25 opacity:255],
  [CCDelayTime actionWithDuration:2.0],
  [CCFadeTo actionWithDuration:0.25 opacity:0],
  nil]];
```

Try it again and the entire health bar should fade out more smoothly!

Adding the Big Boss

Just when you were distracted with trivial things like a health bar—out of nowhere, our 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—in this part, he'll just roll into the scene looking foreboding, as a warning of what is to come!

The boss is going to have some custom logic, so rather than try to add it into **GameObject**, we're going to make a special subclass for the boss.

Go to **File\New\New File**, choose **iOS\Cocoa Touch\Objective-C class**, and click **Next**. Enter **NSObject** for Subclass of, click **Next**, name the new file **BossShip.mm**, and click Save.

Replace the contents of **BossShip.h** with the following:

```
#import "GameObject.h"

@class ActionLayer;

@interface BossShip : GameObject {
    ActionLayer * _layer;
    BOOL _initialMove;
}

- (id)initWithWorld:(b2World*)world layer:(ActionLayer*)layer;
- (void)updateWithShipPosition:(CGPoint)shipPosition;

@end
```

This is a subclass of GameObject with two instance variables for now—one to keep a reference to the **ActionLayer**, and one to keep track of whether it's made it's initial move yet or not.

You may be wondering why it needs a reference to the **ActionLayer**. Well, sometimes the **BossShip** will need to tell the **ActionLayer** to do something—such as fire a bullet or laser.

It then has two methods—an initializer, and a method that the **ActionLayer** will call every frame to give the **BossShip** time to do its thing. It passes the position of the ship as a parameter, because the **BossShip** will want to know where the space ship is so it can shoot at it!

I told you he was evil.

Next switch to **BossShip.mm** and replace the contents with the following:

```
#import "BossShip.h"
#import "ActionLayer.h"

@implementation BossShip

- (id)initWithWorld:(b2World*)world layer:(ActionLayer*)layer {

    if ((self = [super initWithSpriteFrameName:@"Boss_ship.png" world:world
shapeName:@"Boss_ship" maxHp:50 healthBarType:HealthBarTypeRed])) {
        _layer = layer;
    }
    return self;
}

- (void)updateWithShipPosition:(CGPoint)shipPosition {

    CGSize winSize = [CCDirector sharedDirector].winSize;

    if (!_initialMove) {
        _initialMove = YES;
        CGPoint midScreen =
            ccp(winSize.width/2, winSize.height/2);
        [self runAction:
         [CCMoveTo actionWithDuration:4.0
            position:midScreen]];
    }

}

- (void)revive {
    [super revive];
    _initialMove = NO;
}

@end
```

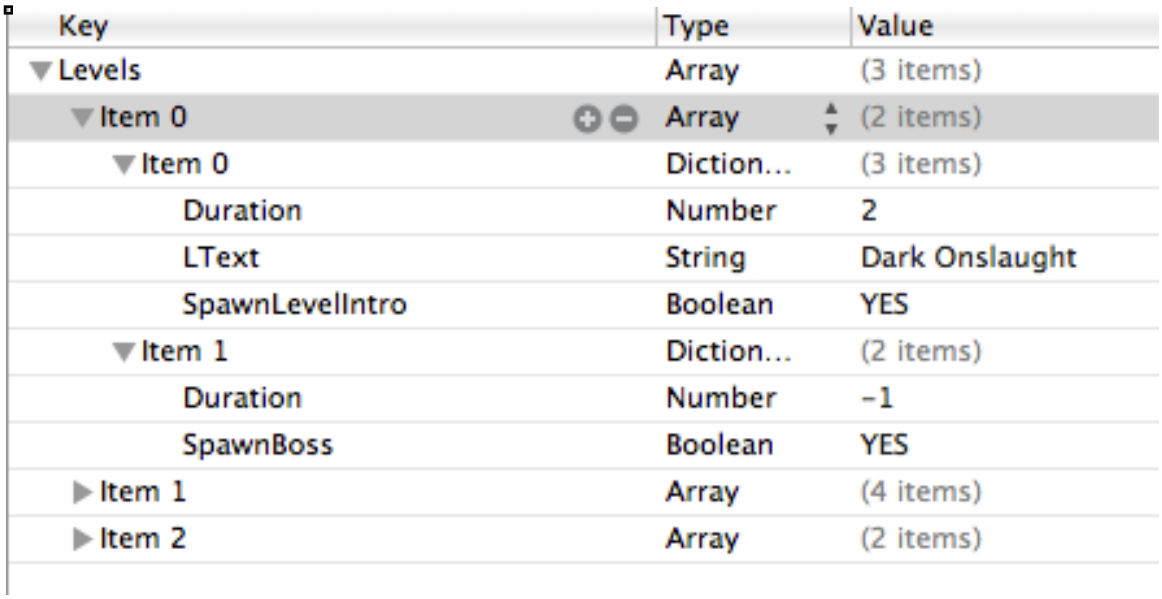
This is a very simple implementation to start. The initializer just calls the superclass's initializer (**GameObject**) passing in the appropriate parameters. Notice that the boss has 50 hp—he's not messing around!

The **updateWithShipPosition** method checks to see if the boss has moved yet, and if not moves the ship to the middle of the screen.

Finally, **revive** is modified to reset the **_initialMove** back to false. This is important if we ever want to reuse this **BossSprite**.

Now that we have a boss we can use, we need to modify our game to use it. We need 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**), and when you open it you'll see the following:



We have an ominous new level title, then in the second stage we have a new **SpawnBoss** flag set. Also note the **Duration** is set to -1—this is a special case we'll use to indicate "the stage will have some special code to advance to the next stage itself when it's done, don't base it on time."

And the only way this stage is going to be done is when one of you is dead!

OK, let's make use of all of this. Open up **ActionLayer.h** and add this import to the top of the file:

```
#import "BossShip.h"
```

Then add the following new instance variables to the class definition:

```
BossShip * _boss;
BOOL _wantNextStage;
```

The first variable keeps track of the boss. There can be only one at a time—after all, he is the boss!

The second variable keeps track of whether we want to advance to the next stage. We'll set this to true if the boss gets defeated. [Note from the boss: "NOT LIKELY!"]

Switch to **ActionLayer.mm** and add a new method right above init to create the boss:

```
- (void)setupBoss {
    _boss = [[[BossShip alloc]
        initWithWorld:_world layer:self] autorelease];
    _boss.visible = NO;
    [_batchNode addChild:_boss];
}
```

Then add a line to call this at the bottom of **init**:

```
[self setupBoss];
```

Move to the **newStageStarted** method, and if it's the boss stage, call a method we're about to write to spawn the boss:

```
if ([_levelManager hasProp:@"SpawnBoss"]) {
    [self spawnBoss];
}
```

Write this method right above **newStageStarted**:

```
- (void)spawnBoss {

    CGSize winSize = [CCDirector sharedDirector].winSize;
    _boss.position = ccp(winSize.width*1.2,
        winSize.height*1.2);

    [_boss revive];

    [self shakeScreen:30];
    [[SimpleAudioEngine sharedEngine]
        playEffect:@"boss.caf"];

}
```

This creates the boss offscreen to the upper right, calls revive on the boss, shakes the screen a good bit, and plays a menacing sound effect.

Next add a new method right above **update**:

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

    if (_levelManager.gameState != GameStateNormal) return;
    if (![_levelManager boolForProp:@"SpawnBoss"]) return;

    if (_boss.visible) {
        [_boss updateWithShipPosition:_ship.position];
    }
}
```

This method bails if it's not in a normal stage with the spawn boss flag—but if it is (and the boss is visible) it calls the boss's update method. Recall that for now, all this does is move the boss to the center of the screen if he hasn't moved already.

Next add the line of code to **update** to call this each frame:

```
[self updateBoss:dt];
```

Almost done! Go to the **beginContact** method and find the **if ([enemyShip dead]) {}** clause, and add the following inside:

```
if (enemyShip == _boss) {
    _wantNextStage = YES;
}
```

So if an enemy ship dies—and that ship is the boss—we should advance to the next stage.

Finally add the code to check for this flag at the end of **updateLevel**:

```
if (_wantNextStage) {
    _wantNextStage = NO;
    [_levelManager nextStage];
    [self newStageStarted];
}
```

If the flag is set, it resets it to false, then advances to the next stage as requested.

Note: You may wonder why we don't just call **nextStage** directly from the collision callback. Well, when we advance to a new stage, it's quite likely we might want to create a new Box2D, such as we would do if we were to spawn a new enemy immediately. And you can't create a Box2D body while you're in a callback! So we have to schedule the level to advance later like we're doing here.

Compile and run the code—and finally our nemesis appears!

Adding the Weapons

If you look at the boss right now, you might have to force back a chuckle.

It looks like our big bad boss left his bedroom without pulling on his weapons!

So let's save our boss from further embarrassment by adding them on for him. It turns out he's well stocked—he's got two laser shooters, and a cannon turret as well!

Open up **BossShip.h** and add the instance variables for these weapons:

```
CCSprite * _shooter1;
CCSprite * _shooter2;
CCSprite * _cannon;
```

Then open up **BossShip.mm** and add the code to initialize the weapons at the end of the initializer:

```
_shooter1 = [CCSprite
     spriteWithSpriteFrameName:@"Boss_shooter.png"];
_shooter1.position = ccp(self.contentSize.width*0.65,
     self.contentSize.height*0.5);
[self addChild:_shooter1];

_shooter2 = [CCSprite
     spriteWithSpriteFrameName:@"Boss_shooter.png"];
_shooter2.position = ccp(self.contentSize.width*0.55,
     self.contentSize.height*0.1);
[self addChild:_shooter2];

_cannon = [CCSprite
     spriteWithSpriteFrameName:@"Boss_cannon.png"];
_cannon.position = ccp(self.contentSize.width*0.5,
     self.contentSize.height * 0.95);
[self addChild:_cannon z:-1];
```

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 the positions are set as multiples of the boss sprite's size, rather than hardcoding offsets. This prevents us from having to write two different offsets—one for iPhone and one for iPad.

Compile and run the code, and now the boss comes in fully dressed! :]

Boss In Action

I'm willing to bet that one of these times while you were testing out 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 thing that made the boss an evil overlord in the first place. And now you've made him mad!

Open up **ActionLayer.h** and predeclare this method at the bottom of the file:

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

You've already written this method back in Tutorial 3, but you're predeclaring it here so that the **BossShip** class knows about it—and can use it to blow you away.

The strategy the boss is going to use is 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 **BossShip.mm**, right above **revive**:

```
- (void)randomAction {
```

```
CGSize winSize = [CCDirector sharedDirector].winSize;
int randomAction = arc4random() % 4;

CCFiniteTimeAction *action;
if (randomAction == 0 || !_initialMove) {

    _initialMove = YES;

    float randWidth = winSize.width *
        randomValueBetween(0.6, 1.0);
    float randHeight = winSize.height *
        randomValueBetween(0.1, 0.9);
    CGPoint randDest = ccp(randWidth, randHeight);

    float randVel =
        randomValueBetween(winSize.height/4,
            winSize.height/2);
    float randLength =
        ccpLength(ccpSub(self.position, randDest));
    float randDuration = randLength / randVel;
    randDuration = MAX(randDuration, 0.2);

    action = [CCMoveTo actionWithDuration:randDuration
        position:ccp(randWidth, randHeight)];

} else if (randomAction == 1) {

    action = [CCDelayTime actionWithDuration:0.2];

} else if (randomAction >= 2 && randomAction < 4) {

    [_layer shootEnemyLaserFromPosition:
         [self convertToWorldSpace:
             _shooter1.position]];
    [_layer shootEnemyLaserFromPosition:
         [self convertToWorldSpace:
             _shooter2.position]];

    action = [CCDelayTime actionWithDuration:0.2];

}

[self runAction:
 [CCSequence actions:
```

```
        action,
        [CCCallFunc actionWithTarget:self
              selector:@selector(randomAction)],
        nil]];

}
```

There's a lot of code here, but most of this should be review. I'll just point out the particularly interesting bits here:

- We pick a random number, and based on the random number, will create a different type of CCAction to run. At the end of the method, we run this action, and then call randomAction again to run another one.
- To figure out a random spot to move the boss, we figure out a random x-coordinate between 0.6-1.0 times the screen width, and 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.
- We want the boss to move at a constant rate. But when we use **CCMoveTo**, we don't pass a rate, we pass a duration. So to figure out the duration, we first figure out how far the ship will be moving by subtracting the destination from the current position. We then use a helper function to figure out the length based on this vector (**ccpLength**). Now that we have the length (in points), we can divide it by a rate (in points per second) to get seconds to move. For the rate, we get a random value between 0.25-0.5 the width of the screen per second.
- We want the lasers to shoot from the positions of the shooters. But we can't use the shooter's positions as is, because since they are children of the boss, their positions are relative to the boss. So we use the **convertToWorldSpace** method to conver the shooter's positons to the world coordinates. Note that when you use **convertToWorldSpace**, you should call it on the parent, passing in the child coordinate you want to convert. In this case the parent is the boss ship (self) and the children are the shooters.

One final step—just call this method at the end of **revive**!

```
[self randomAction];
```

Compile 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.

Open up **BossShip.mm** and add the following to the bottom of **updateWithShipPosition**:

```
CGPoint cannonHeadWorld =
    [self convertToWorldSpace:
      ccp(_cannon.position.x - _cannon.contentSize.width/2,
          _cannon.position.y)];
CGPoint shootVector =
    ccpSub(cannonHeadWorld, shipPosition);
float cannonAngle = -1 * ccpToAngle(shootVector);
_cannon.rotation = CC_RADIANS_TO_DEGREES(cannonAngle);
```

Believe it or not, this code is going to make sure that the cannon always points at the ship.

But how in the world does it work?! Well let's go over it line by line.

The first line figures out where the left side of the cannon is—you know, where the big hole is that cannon balls come out of. Remember that the position of a sprite is its center—so it subtracts half the width of the x axis to get the left hand side. It also needs to convert the coordinate to world space.

The second line figures out the vector between where the cannon is and where the ship is by simply subtracting the two.

The third line figures out the angle to point the cannon. There's a helper method to figure out the angle of a vector that we use here (**ccpToAngle**). However we need to reverse this angle to get the cannon to point toward the ship (rather than away).

The final line converts the rotation from radians to what Cocos2D needs (degrees!)

Compile and run the code, and uhoh—that big cannon is pointing at the ship!

And cannon balls are right about to follow!

Open up **ActionLayer.h** and declare an array for the cannon balls:

```
SpriteArray * _cannonBalls;
```

Also predeclare the new method we're about to write:

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

Switch to **ActionLayer.mm** and initialize the cannon balls array at the end of **setupArrays**:

```
_cannonBalls = [[SpriteArray alloc] initWithCapacity:5
spriteFrameName:@"Boss_cannon_ball.png" batchNode:_batchNode world:_world
shapeName:@"Boss_cannon_ball" maxHp:1 healthBarType:HealthBarTypeNone];
```

And add the line to release it in **dealloc**:

```
[_cannonBalls release];
```

Next write the method to actually shoot the cannon balls:

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

    CGSize winSize = [CCDirector sharedDirector].winSize;
    GameObject *cannonBall = [_cannonBalls nextSprite];

    [[SimpleAudioEngine sharedEngine] playEffect:@"cannon.caf" pitch:1.0f pan:0.0f
gain:0.25f];

    CGPoint shootVector =
        ccpNormalize(ccpSub(_ship.position, position));
    CGPoint shootTarget = ccpMult(shootVector,
        winSize.width*2);

    cannonBall.position = position;
    [cannonBall revive];
    [cannonBall runAction:
     [CCSequence actions:
      [CCMoveBy actionWithDuration:5.0 position:shootTarget],
      [CCCallFuncN actionWithTarget:self selector:@selector(invisNode:)],
      nil]];
}
```

This method takes the position where the cannon is shooting from, but this method needs to figure out where to shoot the cannon ball to—and how fast to move it.

It first figures out the vector between the ship position and the cannon's position by subtracting the two. It also normalizes it, which is a fancy way of saying it makes the vector's length 1. This is handy because then we can multiply it by a number to get a vector in the same direction, but a desired length.

And that's exactly what we do in the next line—we make the vector two times the width of the window, so we're sure the cannon ball moves far enough to be offscreen.

Since the cannon ball moves the same distance each time this method is called (just in different directions), we can have it move at a set number of seconds each time (5 seconds here) and it will always move at the same speed.

The cannon balls we create are already marked with the enemy category in sprite helper, so we don't have to add any extra collision detection code or anything.

Almost done! Switch to **BossShip.mm** and in the **randomAction** method, modify the line that creates the random action number as follows:

```
int randomAction = arc4random() % 5;
```

This just gives us 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) {

    CGPoint cannonHeadWorld =
        [self convertToWorldSpace:
            ccp(_cannon.position.x -
                    _cannon.contentSize.width/2,
                _cannon.position.y)];
    [_layer shootCannonBallAtShipFromPosition:
         cannonHeadWorld];

    action = [CCDelayTime actionWithDuration:0.2];

}
```

This figures out where the cannon's opening is, and tells the layer to shoot a cannon ball at the ship from that position.

The final step is to replace your **Levels.plist** with the final version, located in **Levels\V7** (and don't forget to **Product\Clean**).

Compile 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 this game, or take some ideas or code from it and make something completely different.

One piece of advice I'd offer though is start simple. Just like we did in these tutorials, take the simplest idea you can and get it working. Then you can keep adding features and polish step by step until you're happy with it.

Then the most important part—submit it to the App Store, so others can enjoy your creation! And when you do, please drop me a line—I'd love to see what you've come up with :]

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 works on 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 second, I would really love to hear what you thought of this Starter Kit!

Please leave a comment at www.raywenderlich.com/forums/sgsk, or if you'd rather reach me privately, don't hesitate to shoot me an email. Although sometimes it takes me a while to respond, I do read each and every email, so please don't hesitate to drop me a note!

Finally, if you haven't already, you can follow me on Twitter (@rwenderlich) and join in the conversations going on right now in our iOS Development Forums.

Please stay in touch, and best of luck with your iOS adventures!

Ray Wenderlich
ray@raywenderlich.com