Contents

1 Preample

Welcome to **the** most compact yet detailed guide to iOS game programming. You will be guided through absolutely everything you need to know about cocos2d and SpriteBuilder and 2d game programming in general.

While we will cover the very basics of game programming, such as scene graphs, animations and game loops - Objective-C, the language we will be using throughout the book is not in the scope of things you will learn. When starting this guide, you are expected to have a solid foundation of Objective-C knowledge.

The structure in which you will learn is the following:

- Tools: Get familiar with the very basics of cocos2d and SpriteBuilder
- Infrastructure: Understand that on a high level a game consists of scenes. Understand how to create scenes and navigation paths through these scenes with cocos2d and SpriteBuilder
- Action and Movement: Understand how objects in your game can be moved and animated. With cocos2d and SpriteBuilder
- Interaction: Understand how user interaction can be captured, including Touch interaction and Accelerometer.
- Interobject Interaction: Understand how to use the delightfully integrated Chipmunk physics engine
- Beyond the Basics; Recipes and Best Practices: Once we have the basics, we will look at a ton of recipes and exciting cocos2d classes, which you can use to create any kind of 2d game. Particle Effects, Custom Drawing, Custom Shaders, Tile Maps, Networking, Audio, cocos2d UI in depth, etc.

1.1 Structure of this book

This book shall function as a learning guide and a reference book. Therefore most examples will be small and self-contained. Instead of builind a game throughout the whole book, you will learn by implementing very small projects that are limited to the material we are currently discussing. That shall give you a better chance of understanding the concepts/code snippets and using them in your original game, instead starting of from an example game you have built in this book.

After we have discussed all the basics and you have a good understanding of the cocos2d API I will point you to resources that provide example implementations for specific game types.

There are two different ways to read this book. From the front to the beginning, gaining knowledge in logical groups. Or if you aren't a beginner and would like to use this book as an example driven extension of the API reference you can look up pages by Class names or concept names. There is a special glossar in the back of this book.

1.2 Tools used throughout this book

The two main tools we will be using are cocos2d and SpriteBuilder. Many of the problems that occur during game development can be solved by both of these tools. Wherever it makes sense I will point out both ways, one using only cocos2d and one using SpriteBuilder. This will allow you to see the advantages of each approach and finally decide which tool you want to use in certain situations for your own games.

2 First steps with cocos2d 3.0

cocos2d is a framework built upon OpenGL ES 2.0. It is designed for 2D games and abstracts all the complicated rendering work involved in OpenGL programming. It provides you with a clean and simple API.

If you have never written gaming code before, a few concepts will be new to you. We will start at the highest level most gaming engines now - the scene graph.

2.1 Scene graph in cocos2d 3.0

A scene graph is a general computer graphics concept that allows us to hierarchically organize game objects in a game world. For an example a vehicle can be contained in a game world, a person can be contained in a vehcile and so forth. Let's brake this theoretical concept down to cocos2d terminology. In cocos2d we have to key players in our scene graphs: CCScenes and CCNodes. Since a picture *can* be worth a thousand words, let's start with this diagram:

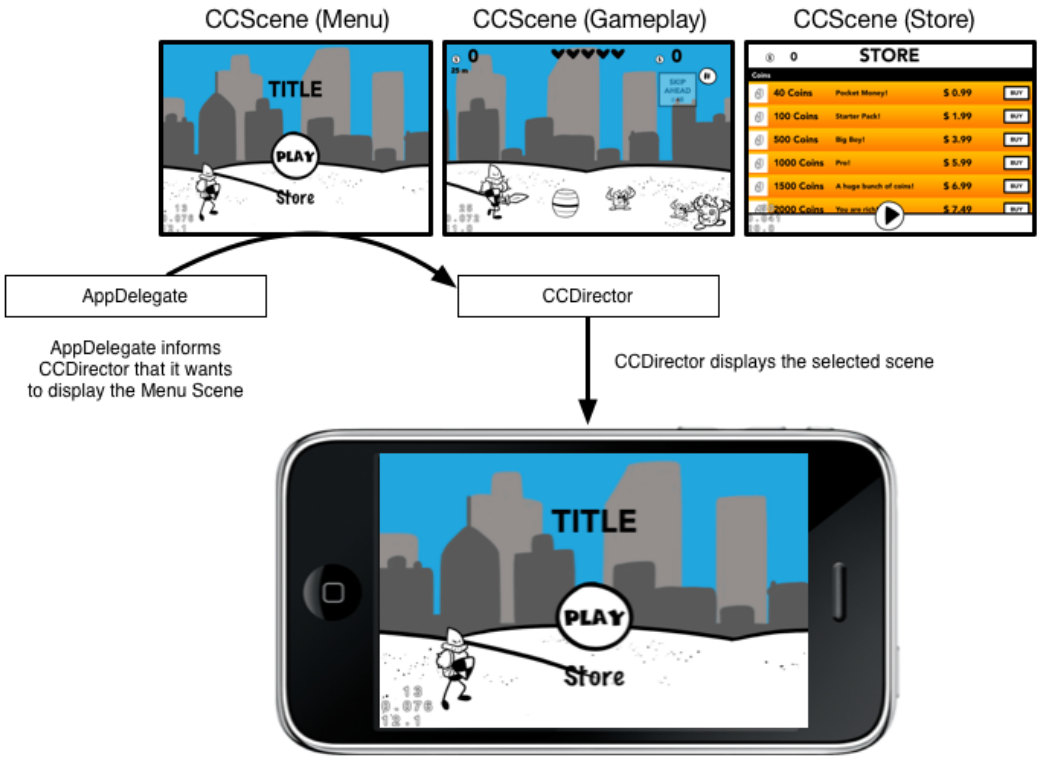


Figure 2.1: *Scene graph* in cocos2d.

The diagram represents three different scenes of which each contains multiple nodes (all buttons, characters, etc. you can see). There are a couple of takeaways from this diagram:

- **CCDirector** is the highest instance in cocos2d. It will control which scene is presented. You can tell CCDirector to present your main menu, your gameplay scene or your highscore scene. CCDirector will allow you to present one scene at a time.
- **CCScenes** are used to build a logical group of objects. In cocos2d this almost always means - one scene represents one screen. You will have scenes for menues, leaderboards, gameplay, etc. Scenes itself don't have a representation, their sole goal is to group CCNodes.
- **CCNodes**. Simply speaking anything that is visible in cocos2d is a subclass of

CCNode. We will get to know a diverse range of CCNode subclasses very soon, including UI Elements and Sprites[1].

The games we write in cocos2d consist of different scenes. We define the structure of our game by telling the CCDirector when which scene shall be presented (menu first, then gameplay, then leaderboard). We create the content of our scenes using CCNodes.

A CCNode can again contain other CCNodes. We are speaking of a scne graph when we refer to the strucutre of our CCNodes.

2.2 CCNodes in cocos2d 3.0

We just learned: *Every visible object in cocos2d is a subclass of CCNode.* These are the CCNode subclasses you should know for now:

CCSprite Represents an image or an animated image. Used for characters, enemies, etc. in your gameplay.

CCAnimatedSprite A convinience class that allows to create animated Sprites in just a few lines of code.

CCNodeColor A plain colored node.

CCLabelTTF A node than represent text in any TTF font.

CCButton A interactive node that allows to respond to touch input easily.

CCLayoutBox A unvisible node that layouts its children.

2.3 Let's code

The easiest way to prove, that cocos2d is an easy-to-learn, powerful framework is by starting to write some code. What you have learned so far is enough to build a simple App with multiple scenes.

Here are the code snippets for the concepts we have already discussed:

[1] Sprite basically is the game developer word for an image

Download examples/introduction.clj

```
Line 1  [CCDirector sharedDirector]
  -  basically, create a menu scene, create a gameplay scene, add a
        ↪ animated
  -  character to the gameplayscene
```

Comman Programming Error 2.1

Having problems installing cocos2d? Visit....

3 Getting started with SpriteBuilder 1.0

Spritebuilder is a tool that derived from Cocosbuilder which originally was written by Victor Lidholt at Zynga. The goal of Spritebuilder is to provide a WYSIWYG editor for cocos2d games.

In the creation of many scenes you can save a lot of time if you don't have to layout your menus in code, and instead can place them visually.

If you decide to you use SpriteBuilder, you will start your game by creating a new SpriteBuilder project instead of starting of with a Xcode project. Whenever you start a new SpriteBuilder project, it will create and maintain a Xcode project for you.

Similar to Xcode's Interface Builder you will create interface files using SpriteBuilder and you will be able to connect the interface files with classes you have created in Xcode.

Interface files in SpriteBuilder are called *CCB* files. Each CCB file maintains a own scene graph. This means a CCB file can be interpreted as a CCScene or a CCNode.

If you work on a project using SpriteBuilder you workflow will look like this:

- Create a project in SpriteBuilder
- Add images and other resources to your SpriteBuilder project.
- Create multiple CCB files for the different scenes in your project.
- **Publish** your project in SpriteBuilder. This will create or update the Xcode project related to your SpriteBuilder.
- Add code to your classes in Xcode.

Here's a diagram that shows how SpriteBuilder and Xcode work together in your SpriteBuilder projects:

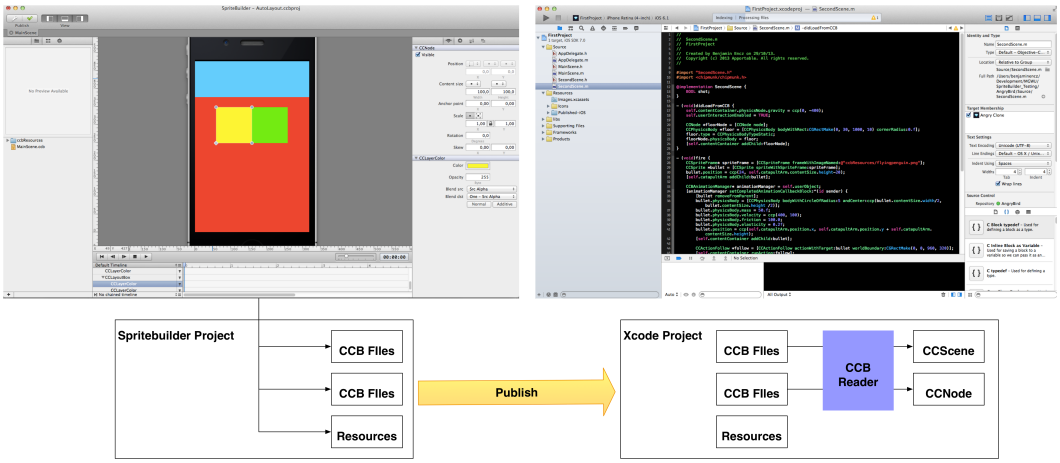


Figure 3.1: Spritebuilder creates and organizes a Xcode project for you. Adding all the resources and scenes you have created.

For those that are interested in a behind the scenes tour, I will give a short explanation of how SpriteBuilder works. In SpriteBuilder you create CCB Files, these store a scene graph; the hierarchy and positions of your nodes. When publishing in SpriteBuilderthe CCB Files and all other resources stored in your SpriteBuilder project are copied to your Xcode project. When running the project in Xcode a class called CCBReader wil parse your CCB files and create the according CCNode subclasses to reconstruct the scene graph you have designed in SpriteBuilder.

When you use SpriteBuilder, you still implement the navigation in code. The highest level you will be using CCB files at, are one scene. Ok, now let's start by creating our first project using SpriteBuilder!

3.1 The first Spritebuilder project

Open SpriteBuilder and select *File-¿New-¿Project*. Select a folder for your new project.

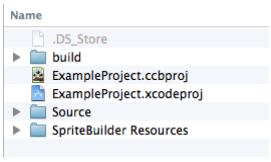


Figure 3.2: Typical folder structure after creating a Spritebuilder project. You have a SpriteBuilder project (.ccbproj) and a Xcode project (.xcodeproj)

3.2 The Spritebuilder User Interface

3.3 Layouting in Spritebuilder

3.4 Custom classes in Spritebuilder

- How custom classes for scenes?

3.5 Inheritance using Spritebuilder

4 Working with Sprites

One of the most used classes in your games will be CCSprites. A CCSprite is a CCNode Subclass that represents an image (for example a game character, a tree, etc.).

4.1 Getting started with CCSprite

The easiest way to initialize a sprite it this:

```
Line 1  -(void)update:(CCTime)delta {
        //implement any custom movement/animation here
        }
```

4.2 Working with Sprites

Sprites are a very important part of Cocos2D, they are used to render textures to the screen. This documentation will cover following aspects:

- Creating textured sprites
- Changing a sprite texture
- Sprite Frame Animations
- Using 9 Patch Sprite Images

4.2.1 Creating a textured Sprite

The easiest way to setup a textured Sprite is the following line of code:

```
CCSprite *hero = [CCSprite spriteWithImageNamed:@"hero.png"];
```

`spriteWithImageNamed:` utilizes `CCFileUtils` to automatically find the image in the right resolution, you do not have to specify the resolution manually. The line above also works for images contained in *Smart Sprite Sheets* created by SpriteBuilder. In case you are using folders in SpriteBuilder you need to include the complete path to the image, e.g.:

```
CCSprite *hero = [CCSprite spriteWithImageNamed:@"gameAssets/hero.png"];
```

The content size of the sprite will automatically be the size of the texture.

4.2.2 Changing the texture of an existing Sprite

To change the texture of an initialized Sprite the *spriteFrame* property can be used:

```
hero.spriteFrame = [CCSpriteFrame frameWithImageNamed:@"hero_dead.png"];
```

4.2.3 Sprite frame animations

Sprite frame animations are used in basically every 2D game. A sequence of different 2D images gets replaced at a defined interval forming an animation:

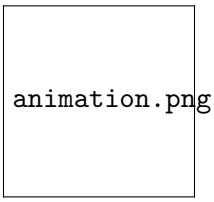


Figure 4.1: image

Sprite frame animations can be created in a plist-format which is used by plain Cocos2D games or within SpriteBuilder.

Optimizing Performance with Batch Nodes

Batch Nodes can be used to draw many Sprites witht the same textures at once.

5 Animations and Movements

We have multiple different ways to add activity to scenes in cocos2d. First, it is important to know two different forms of activity:

Movement If the position of one of our nodes changes over time, we consider this a movement.

Animation When we iterate through a set of different images, we call this an animation.

Let's take a look at movements first.

5.1 Implementing Movements

Since we are using SpriteBuilder, we have two different ways to implement this: In code or through SpriteBuilder's keyframe animations. Implementing physics for your game is another way to add movement to Nodes, but we are going to spare that one for later.

5.1.1 Implementing Movement manually in code

We will first try this the hard way - which is always the best way to learn something new. Since I assume that many readers are new to game development, we are going to take a small step back and take look at the bigger picture.

One core concept of games is the *game loop*. The game loop takes care of giving any object in the game, to implement a time based behaviour, by calling certain methods in a regular interval. In cocos2d the game loop by itself is not visible, the only aspect of it we use, is an *update* method. The update method is called every render cycle. Any CCNode can override this update method.

This update method is where we can implement any time-based actions. The method signature looks like this:

```
Line 1  -(void)update:(CCTime)delta {
-           //implement any custom movement/animation here
-   }
```

The one parameter we get passed in is the time that has passed since the update method has been called last. Whichever action we perform/trigger in the update method, we need to consider this time factor.

Let's once again implement this by example. We now want to move a simple unanimated Sprite over the screen with a constant speed:

```
Line 1  -(void)update:(CCTime)delta {
-           //TODO: implement movement code
-   }
```

5.1.2 Implementing Movement using CCActions

When developing games we are mostly confronted with very similar problem sets. cocos2d provides a lot of functionality for common use cases. One of these convinience concepts are *CCActions*. CCNodes can run CCActions. The CCActions a CCNode runs can affect different properties of the CCNode such as position, color or scale.

So for the use case we have seen above, moving a sprite across the screen, we don't need to implement custom movement code, we can use CCActions.

There are a large amount of CCAction types in cocos2d:

CCActionMove ...

We will first take a look at how we can implement the movement using a CCActionMove, then we will take a clooser look at the other CCAction types and how and when they can be used.

Actually, implementing this is amazingly easy. We create a CCActionMove and let our CCNode run this action:

```
Line 1          CCActionMove *move = [CCActionMove
                    ↪ actionWithTargetPosition:pos duration:1.f];
-           [ship runAction:move];
-   }
```

5.1.3 Implementing Movement with SpriteBuilder

5.2 Implementing Animations

This section will have to challenges. First we will need to learn a new tool, to create images in a way, that we can use them for sprite animations. Second we will learn how to implement these animations in code and in SpriteBuilder.

In cocos2d, as in many 2D game engines, animations consist of a group of frames. For example a running animation will be a set of 5 different pictures showing a character in 5 different stages of a running movement. Basically all a game has to do is to switch between these 5 different images with a certain delay that makes the animation believable. This is the same approach as used by GIF-File animations (these things that splattered the first generation of the world wide web).

5.3 Creating an animation Spritesheet

5.4 Implementing animations in code

5.4.1 Implementing an animation using CCActions

5.4.2 Implementing an animation using a convenience class

5.5 Implementing animations using SpriteBuilder

6 User Interaction

User Interaction is another important foundation for any game. This chapter concludes the minimum basic understanding you need to build an actual game - and we have a lot to discuss.

New input capabilities have revolutionized the gaming experience on mobile devices and have created completely new genres of games. In order to create great mobile games you need to be able to work with all the possible input types modern iOS devices provide:

- Touch Input
- Accelerometer Input
- Gesture Recognition

We will start looking at simple touch input first, since this is the easiest and most used control scheme in games.

6.1 Handling touch input in cocos2d

6.2 Handling Gestures in cocos2d

6.3 Handling Accelerometer input in cocos2d

7 User Interfaces in cocos2d

Similar to UIKit, cocos2d provides a set of UI components you should be familiar with:

- CCButton
- CCTextField
- CCTableView

7.1 Positioning and Layouting in cocos2d

We will first look at positioning: a Node is placed at a certain positioning. After that we will take a look at layouting: a Node's position is determined by a layouting mechanism.

7.1.1 Positioning

In order to create your first menu in cocos2d you need to understand how the layouting/positioning within CCScenes works. If you are already familiar with UIKit (Apple's Framework to build iOS Apps with the native interface components) there is one significant difference: cocos2d's root point (x=0, y=0) is in the bottom left corner.

Another aspect to positioning which isn't used by many other frameworks are anchor points. Anchor points define which position within your Node is used to place the Node. Most Node's anchor point default to 0,0 (the bottom left corner), while some CCNode subclasses, such as CCSprite override this property. CCSprite's anchor point defaults to (0.5, 0.5) the middle of itself. Changing the anchor point will affect positioning and rotation of your Nodes.

cocos2d 3.0 also introduces a new concept called *positioning types*. These positioning types influence the way, the position property of your Node is interpreted. Here an overview of the available position types:

CCNormalizedPosition Normalized means that the position is expressed relative to the parents content size. A 0.5 value for the x-Position means, that the Node will be placed at 50 percent of the parents width - thus horizontally centered.

CCAbsolutePosition Default. Interpretes the position as absolute position.

Let's come up with some layouting examples to make these theoretical terms into practical code:

```
Line 1  /* position CCNode at top right; There are other solutions, but
            ↪ this is the
    -   ideal one:*/
    -   CCNode *node = [CCNode node];
    -   node.anchorPoint = (1.f, 1.f)
    5   node.positioningType = CCTypeNormalized;
    -   node.position = (1.f,1.f);
    -
    -   /* Place this in the center:*/
    -   CCNode *node = [CCNode node];
    10  node.anchorPoint = (0.5f, 0.5f)
    -   node.positioningType = CCTypeNormalized;
    -   node.position = (0.5f,0.5f);
```

Positioning types can also be combined. Assume we want to center an element horizontally, but want it's vertical position to be 100 points from the top of the screen. This means we'd like normalized position for our x positioning, but absolute positioning for our y position:

```
Line 1  /* position CCNode at top right; There are other solutions, but
            ↪ this is the
    -   ideal one:*/
    -   CCNode *node = [CCNode node];
    -   node.anchorPoint = (0.5f, 1.f)
    5   node.positioningType = CCPositionTypeMake(CCTypeNormalized,
            ↪ CCTypeAbsolute);
    -   node.position = (0.5f,parent.contentSize.height - 100);
```

7.1.2 Layouting

CCLayoutBox has already been mentioned as one of the important CCNode types in cocos2d. CCLayoutBox can take care of automatically positioning your nodes, by aligning them vertically or horizontally with a margin between the items that can be defined manually.

Here's a short example:

```
Line 1  /* TODO: layout box example code */
    -   CCLayoutBox *layoutBox = [CCLayoutBox layoutBox];
```

8 Physics

This may seem sarcastic, but we are going to start this chapter with thinking about how we can avoid using physics in our games - some games simply do not need physics calculations and can work with CCActions and simple collision detection instead - potentially making the game developers life a lot easier.

9 Particle Effects

Particle Effects are the secret sauce to the graphics of your 2d game. They can make graphics a lot more enjoyable. You now will learn how to create particle effects in code and using Spritebuilder. However, before we discuss how to create particle effects we need to understand which art is required for particle effects.

9.1 Art for particle effects

Basically a particle effect (in cocos2d) consists of a huge amount of Sprites which are colored/blurred, or otherwise graphically manipulated, and moving in patterns. This means the basic element of every particle effect is a texture for a single particle. You can download the texture for our firt particle effect here:TODO downloadlink. Two important things to note: the texture has a transparent background and its structure is colored in grayscale colors. This way the particle can be blended into any color.

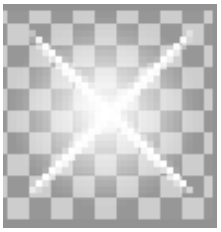


Figure 9.1: A classic texture for a particle effect.

9.2 Particle Effect Basics

cocos2d provides a convinience class that allows to create particle effects with a wide variety of parameters: CCParticleSystem. In the appendix you can find a table that

provides a brief description of each parameter. However, it is a lot easiert to start with some examples.

9.3 Particle Effects in cocos2d

In case you are not using Spritebuilder for your game - or you just want to learn the underlying basics of creating custom particle effects, this chapter is for you. We will learn how to build particle effects in code and using the popular tool *Particle Designer*.

Following this tutorial

If you want to implement this tutorial you need to start with this template project: .We will add multiple scenes for different applications of particle effects.

9.3.1 Configuring particle effects in code

Most of the time you will want to design particle effects in a graphical editor that provides a nice visual preview. However, to understand the basics we will create our first particle effect in code (this is a great example from the cocos2d tests).

The steps we need to perform:

- create a CCParticleSystem
- select a particle texture
- set a duration for the particle effect
- set a huge amount of further, optional parameters

Now let's create our first particle effect. Add a new scene called *ParticleEffectScene* to your project. You also need to add a button to the *MenuScene* that will allow us to present the new *ParticleEffectScene*. I won't provide the complete setup code (it's mostly boring but can be found here: https://gist.github.com/Ben-G/8340545), here's the relevant part:

```
Line 1      CCParticleSystem *emitter = [[CCParticleSystem alloc]
               ↪ initWithTotalParticles:50];
-       emitter.texture = [[CCTextureCache sharedTextureCache]
               ↪ addImage: @"stars-grayscale.png"];
-           emitter.duration = CCParticleSystemDurationInfinity;
-       // lots of further parameters
```

Once the particle system is created, you add it as a child to your scene:

```
Line 1      emitter.positionType = CCPositionTypeNormalized;
-       codeParticleEffect.position = ccp(0.5f, 0.5f);
-       [self addChild:emitter];
```

There is no need to start the particle effect - it is started as soon as it is added to a parent node. However you can manually start/stop a particle system by using the *stopSystem* and *resetSystem* methods. Now you can run the app and you should see a particle effect in the middle of the scene. (TODO: add screenshot)

9.3.2 Particle Effects with plists

If you took some time to read through the setup in code (https://gist.github.com/BenG/8340545) you have seen that it is very cumbersome and nothing that actually should be part of your code base. Luckily cocos2d allows us to read particle effects from plist-files. You can fill these plists manually, using the same parameters as when setting up particle systems in code - however, the value of using plists only fully unfolds when you use graphical editors to create your particle effects. Spritebuilder comes with an integrated particle effect designer. If you don't use Spritebuilder, *Particle Designer*(http://particledesigner.71squared.com/) is the preferred tool to create particle effects.

9.3.3 Particle Effects in Particle Designer

9.3.4 Including particle effects in your gameplay

Now that we have the basics it's the right time to add some best practices. Mostly you will want to add particle effects dynamically to your scenes, based on certain events such as collisions or explosions. Let's create a new class called *ParticleGamePlayScene*, again,

don't forget to add a entry to *MenuScene* that let's you push the new scene. We're going to build following scene:

- two spaceships, one at the left and one at the right edge of the sceen
- when a ship get's touched it fires a bullet at the other ship
- when a bullet hits another ship we play an explosion particle effect

To create this setting, we will add a lot of code that is not directly related to particle effects. Let's discuss that code briefly, all of this takes place in *ParticleGamePlayScene.m*:

Download examples/ParticleGamePlayScene.m

```
Line 1
  @implementation ParticleGamePlayScene {
      CCSprite *leftShip;
      CCSprite *rightShip;
5     NSMutableArray *bullets;
  }

  - (id)init
  {
10    self = [super init];

      if (self) {
          leftShip = [CCSprite spriteWithImageNamed:@"ship.png"];
          leftShip.anchorPoint = ccp(0.f, 0.5f);
15        leftShip.position = ccp(0, self.contentSize.height/2);
          [self addChild:leftShip];

          rightShip = [CCSprite spriteWithImageNamed:@"ship.png"];
          rightShip.anchorPoint = ccp(1.f, 0.5f);
20        rightShip.position = ccp(self.contentSize.width-rightShip.
              ↪ contentSize.width, self.contentSize.height/2);
          rightShip.scaleX = -1.f;
          [self addChild:rightShip];

```

```
        // enable touches
        self.userInteractionEnabled = TRUE;

        // initialize bullets array
        bullets = [NSMutableArray array];
    }

    return self;
}
```

Nothing special going on here. We create two ships, enable user interaction and initialize a array that we use to keep track of active bullets. Next, we need to react to touch input:

Download examples/ParticleGamePlayScene.m

```
#pragma mark - Touch Handling

- (void)touchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
    CGPoint touchPoint = [touch locationInNode:self];

    if (CGRectContainsPoint(leftShip.boundingBox, touchPoint)) {
        // fire left ship
        [self fireLeftShip];
    } else if (CGRectContainsPoint(rightShip.boundingBox,
        ↪ touchPoint)) {
        // fire right ship
        [self fireRightShip];
    }
}
```

We detect if the left or right ship has been touched by checking if the bounding box of either ship contains the touch position. Depending on which ship has been touched, we call the according firing method.

Code Structure

Please note that we usually would create a subclass for the ship to handle touches and/or bullet spawning, however for the sake of readibility we reduce the amount of involved classes and handle everything in the scene itself.

Here are the firing methods: (TODO: improve positioning when anchor points are fixed)

Download examples/ParticleGamePlayScene.m

```
Line 1  #pragma mark - Firing methods
-       - (void)fireLeftShip
-       {
-           CCSprite *bullet = [CCSprite spriteWithImageNamed:@"bullet.png
                ↪ "];
5           bullet.anchorPoint = ccp(0.f, 0.5f);
-           bullet.position = ccp(leftShip.boundingBox.origin.x+leftShip.
                ↪ boundingBox.size.width, leftShip.position.y);
-           [self addChild:bullet];
-           CCAction *bulletMove = [CCActionMoveTo actionWithDuration:3.f
                ↪ position:ccp(self.contentSize.width, rightShip.position.
                ↪ y)];
-           [bullet runAction:bulletMove];
10          [bullets addObject:bullet];
-       }
-
-       - (void)fireRightShip
-       {
15          CCSprite *bullet = [CCSprite spriteWithImageNamed:@"bullet.png
                ↪ "];
-           bullet.scaleX = -1.f;
-           bullet.anchorPoint = ccp(0.f, 0.5f);
-           bullet.position = ccp(rightShip.boundingBox.origin.x-bullet.
                ↪ contentSize.width, rightShip.position.y);
-           [self addChild:bullet];
```

```
    CCAction *bulletMove = [CCActionMoveTo actionWithDuration:3.f
        ↪ position:leftShip.position];
    [bullet runAction:bulletMove];
    [bullets addObject:bullet];
}
```

The firing methods take care of spawning bullets. We use *CCActionMoveTo* to shoot in the direction of the other ship.

Now that we went through all of the boilerplate code we get to the actually interesting part of our small project: detecting collisions and launching particle effects. For collision detection we use the *update* method, that is called every frame. To create particle effects, we use the code from 9.3 with some slight modifications:

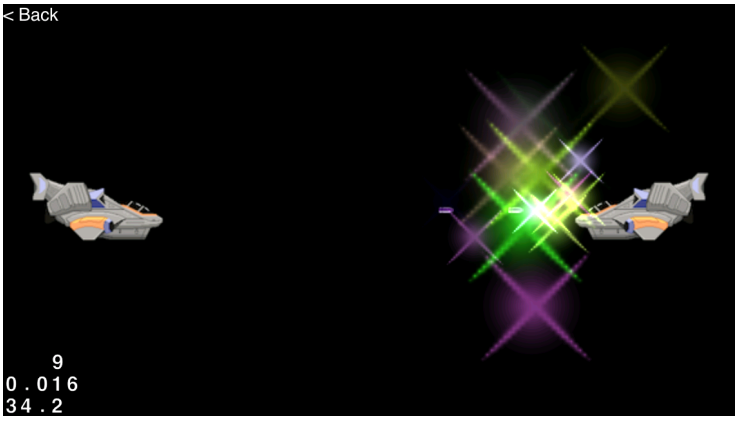
```
#pragma mark - Collision Detection

- (void)update:(CCTime)delta
{
    for (int i = 0; i < [bullets count]; i++) {
        CCSprite *bullet = bullets[i];

        if (CGRectIntersectsRect(bullet.boundingBox, leftShip.
            ↪ boundingBox) || CGRectIntersectsRect(bullet.
            ↪ boundingBox, rightShip.boundingBox)) {
            [bullet removeFromParent];
            [bullets removeObjectAtIndex:i];
            CCParticleSystem *particleEffect = [self
                ↪ createParticleEffectInCode];
            particleEffect.position = bullet.position;
            [self addChild:particleEffect];
        }
    }
}

#pragma mark - Provide particle effect

- (CCParticleSystem *)createParticleEffectInCode
```

```
{
    CCParticleSystem *emitter = [[CCParticleSystem alloc]
        ↪ initWithTotalParticles:50];
        emitter.texture = [[CCTextureCache sharedTextureCache]
            ↪ addImage: @"stars-grayscale.png"];
        // duration
        emitter.duration = 1.f;
    emitter.autoRemoveOnFinish = TRUE;
        [...]
        return emitter;
}
```

Collision Detection. We iterate through the bullets, determining if any bullet intersects one of the two ships. Once we detect a collision we remove the bullet from the scene and the bullets array and create a particle effect at the same position.

Creating particle effect. I stripped away all the visual setup. The two important changes to the particle effect, compared to 9.3 are setting the duration to 1 second and setting *autoRemoveOnFinish* to *TRUE*. This means that once the particle effect is added in the collision detection method it will run for 1 second and after it completed it will automatically be removed from the scene. This means we don't have to take care of cleaning the partical effect up manually.

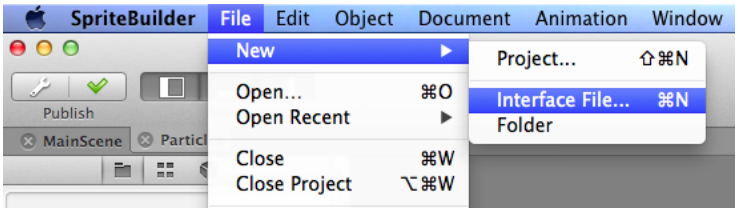
If you run the project now, you should be able to fire bullets and you should see particle effects every time a bullet collides with a ship.



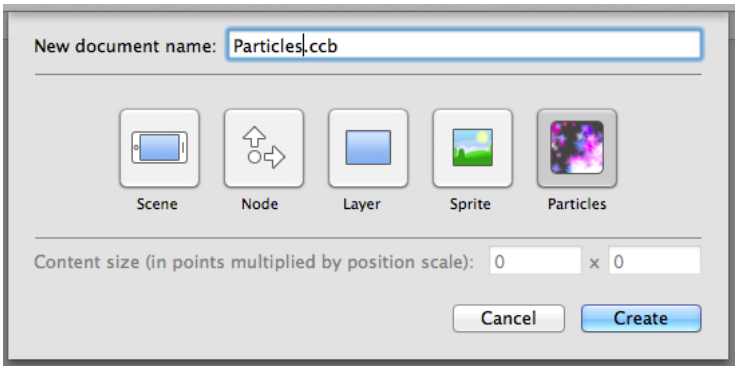
Congratulations! Now you know how to dynamically add particle effects to your game in cocos2d. In the next chapter you will learn how to accomplish this in Spritebuilder.

9.4 Particle Effects in Spritebuilder

Now that we understand the nature of particle effects and have implemented some basics, we will take a look at how Spritebuilder works with particle effects. Similar to the preceeding chapter we will start by statically adding particle effects to a scene first and then will integrate particle effects to our gameplay. **First, let's get started by creating a new Spritebuilder project**. First open the menu to add a new interface file:



Next, select *Particles* as document type:



Now you should see the newly created particle effect on your stage. To select the node you can use the timeline and select the *CCParticleSystemQuad* (highlighted blue in the screenshot):

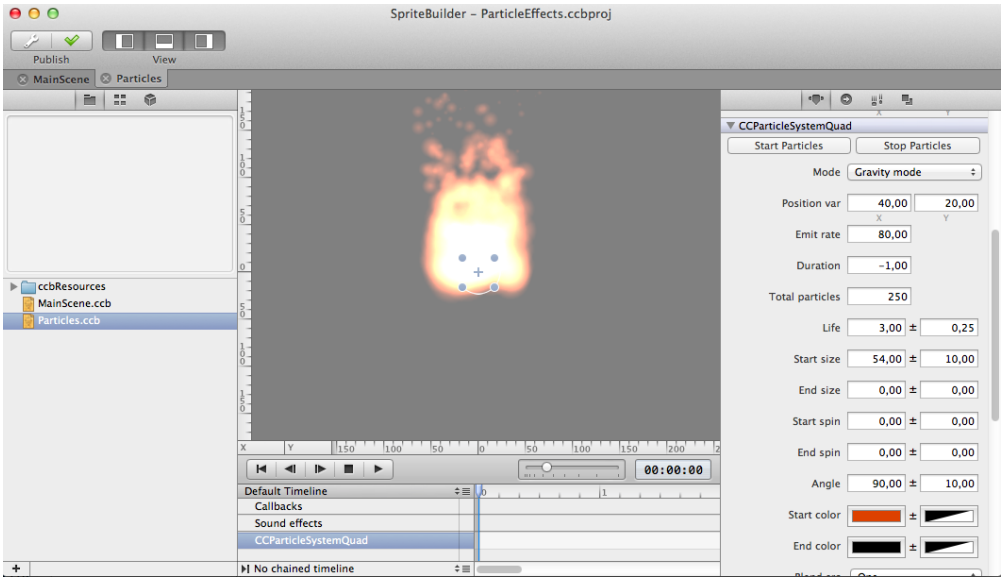


Figure 9.2: Stage with a default particle effect

In the inspector on the right you can see all (check if this is true) properties that we have set in code previously. You can update them and get a live preview of how they influence your particle effect.

Spritebuilder comes with a nice library of template particle effects. You can view them in the rightmost tab of the inspector:

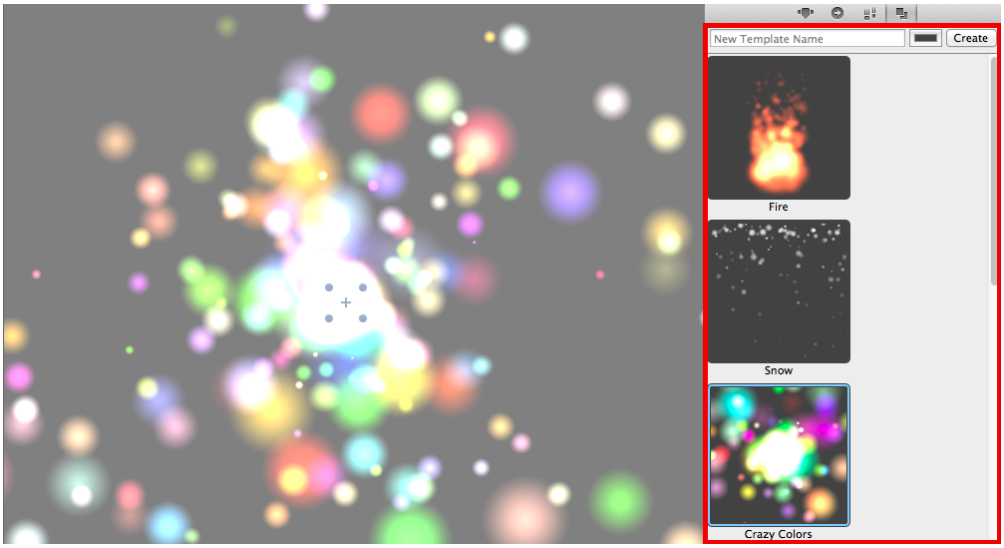


Figure 9.3: Spritebuilder provides many particle effect templates

to select one of the templates, simply double-click it.

9.4.1 Adding the particle effect to your scene

Spritebuilder creates a new CCB File for each of your particle effects. If you want to add a particle effect to one of your gameplay scenes you need to add the CCB File of the particle effect as a child. You can do this by dropping a *Interface File* child node on your target scene:

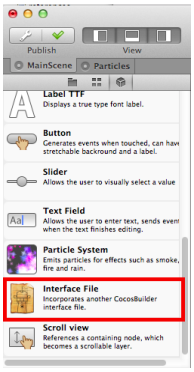


Figure 9.4: Interface Files let you include CCB Files in other CCB Files

Once the *Interface File* node is added, you can choose which CCB File shall be included. The inspector provides a dropdown with all available CCB Files, here you can choose the CCB of your particle effect.

Figure 9.5: In the dropdown of the inspector you can choose which CCB File should be included

Now you have succesfully added the particle effect to your scene. However, mostly when you work with particle effects you will not add to your scene statically, as we just did, but you will want them to appear when certain events occur, e.g. collisions or explosions. In the next chapter we will take a look how we can add particle effects to our gameplay - dynamically.

9.4.2 Including particle effects in your gameplay

9.5 Q and A: Let's solve some problems

Once again, this section provides some practicle solutions to common problems:

9.5.1 Q: I want to add multiple particle effects of the same type to my scene

10 Audio

11 Tilemaps

12 Appendix A: Writing your own SpriteBuilder Plugins

13 Appendix B: Particle Parameters

Mode	Parameter	Description
Gravity Mode (Mode A)		
	Gravity	a
	Direction	a
	Speed	a
	Tangential acceleration	a
	Radial acceleration	
Radius Mode (Mode B)		
	Start Radius	a
	End Radius	a
	Rotate	a
Properties for all Modes		
	Life	a
	Start spin	a
	End spin	a
	Start size	a
	End size	a
	Start color	a
	End color	a
	Life	a
	Blending function	a