Contents

1 Preample

Welcome to **the** complete 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.

1.3 What is a 2D game engine?

2 Introduction to SpriteBuilder and Cocos2D

Now it's time to dive into 2D Game Development! For this chapter I will assume that you haven't written a game with a game engine so I will explain the relevant concepts fairly detailed.

2.1 Installing the software

In general there are two ways to install Cocos2D depending on whether you want to use SpriteBuilder or not. Throughout this book we will be using SpriteBuilder to set up all of our projects, therefore we will only install SpriteBuilder which will come bundled with the latest version of Cocos2D.

Installing SpriteBuilder is easy, simply open the *App Store* app on your Mac and search for *SpriteBuilder*.

Figure 2.1: Cocos2D Technology Stack

Well done! Later throughout this chapter you will learn how to set up your first project!

2.2 Introduction to Cocos2D

To understand what Cocos2D is, it is helpful to look at the history of game development. The first video games were written in assembler (a very low level programming language) and images were drawn to the screen by manually setting colors for certain pixels. Since then a wealth of frameworks and libraries has been written to make the life of a game developer easier. Throughout this book you will learn how powerful Cocos2D is and how little code you will need for the basic building blocks of your games.

Here are some of the most important features of Cocos2D that make 2D game development a lot easier:

Structure like most game engines Cocos2D demands a specific structure of the code for your game, making most design decisions simple

Scene Graph and Sprites loading an image for a character and adding it to the Gameplay scene can happen in two lines of code

Action System a sophisticated action system allows you to define movements of objects and animations

Integrated Physics Engine automatically calculates movements of objects, collisions and more.

There are many more features - but this brief outline shows the most important ones and should give you an idea why almost all game developers these days use game engines.

Cocos2D is built on top of OpenGL ES 2.0. If you have ever written OpenGL code before you know that it takes many lines of code to draw a textured rectangle on the screen. Cocos2D abstracts all of these tasks for you, you can go a very long way without writing any OpenGL code. The following diagram shows which technologies are used by Cocos2D:

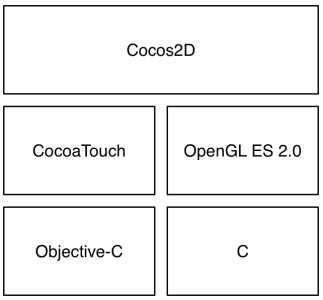


Figure 2.2: Cocos2D Technology Stack

As a Cocos2D developer you define which scenes exist in your game, which objects are part of these scenes and which size, position and appearance these objects have and Cocos2D will use OpenGL to render the scene you have described. In order to provide this functionality Cocos2D consists of many classes - some important ones will be discussed in this chapter. All Cocos2D classes use the *CC* prefix (CCScene, CCNode, etc.).

You have probably realized this already - when working with a 2D game engine for the first time you will be introduced to a whole set of new terminology. Just as a framework to write desktop applications knows the concept of windows, buttons and mouse clicks a 2D game engine comes with its own set of terms and techniques. We will spent the next sections discussing the important terms in detail.

2.2.1 Scenes

For explaining the important terms and concepts of Cocos2D I will use a top down approach. First we will learn how games that use Cocos2D are structured. On the highest level of structure we have a concept called *scenes*. By default every scene in Cocos2D is full-screen. This means for every screen in your game you will use one scene.

Here's an example from the MakeGamesWithUs game *Deep Sea Fury*: You can see that the game consists of the start scene, the Gameplay scene and the game over scene.

A Cocos2D developer creates scenes by using the CCScene class. Another important Cocos2D class for scene handling is CCDirector. The CCDirector class is responsible for deciding which scene is currently active in the game (Cocos2D only allows one active scene at a time). Whenever a developer wants to display a scene or transition between two scenes he needs to use the CCDirector class.

This means creating and displaying a new scene is a two step process:

1. Create a new instance of CCScene
2. Tell the CCDirector to display this new scene

You will learn a lot more about this down the road, but the important bottom line is: *Scenes are the highest level of structure in your game and a class called CCDirector decides which scene is currently displayed.*

2.2.2 Nodes

Everything that is visible in your Cocos2D game (and a couple of invisible objects) are *nodes*. Nodes are used to structure the content of a scene. Every node can have other nodes as its children. Cocos2D provides a huge amount of different node types. Every node type is a subclass of CCNode.

Most nodes are used to represent an object on the screen (an image, a solid color, an UI element, etc.), a few other nodes are only used to group other nodes. All nodes have a size, positions and children (and many other properties which are less important for us right now). Here are some of the popular node types of Cocos2D:

CCSprite represents an image or an animated image. Used for characters, enemies, etc.

CCColorNode a node being displayed in one plain color.

CCLabelTTF a node that can represent text in any TTF font.

CCButton a interactive node that can receive touches.

Nodes and their children form a scene graph. The concept of a scene graph isn't unique to Cocos2D it is a common concept of 2D and 3D graphics. A scene graph is a hierarchy of many different nodes.

2.2.3 Scene Graphs

Let's take a look at simple example of a scene graph:

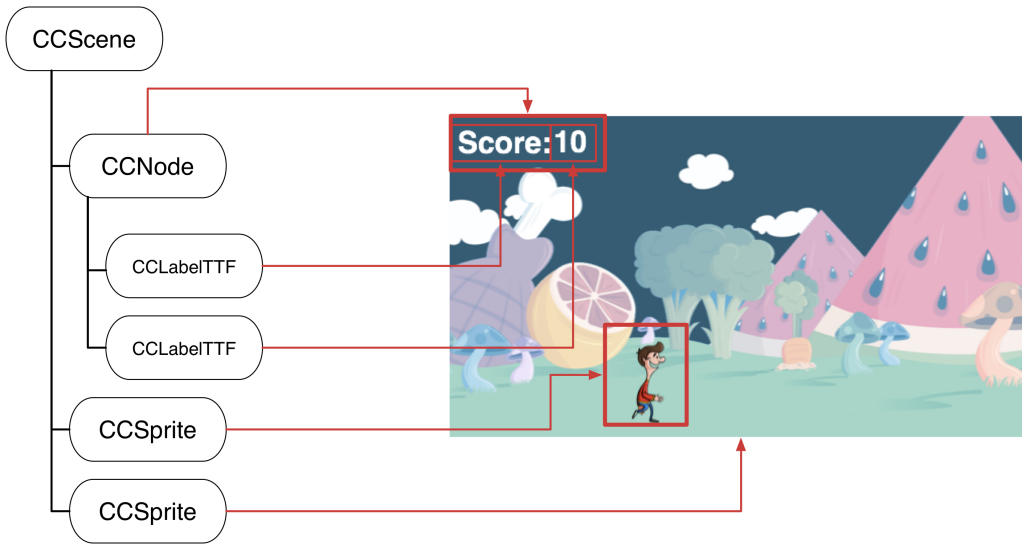


Figure 2.3: Cocos2D Scene Graph

On the left side of the image you can see the node hierarchy. On the first level you have the CCScene. Then we have a CCNode with to children of type CCLabel. This CCNode

is the first example of a grouping node. Instances of CCNode don't have any appearance they are solely used to group other nodes. Throughout this book you will learn that it often makes sense to group nodes under certain parent nodes. The main reason is that all children are placed *relative* to their parents. So if we would want to move the scoreboard of the example above to the top right corner we would only have to move the parent node instead of both child nodes. As you can imagine this becomes even more relevant in games that have ten or more entries in their scoreboard.

Structuring Nodes

Always group nodes that logically belong together under one parent node. That will save you a lot of time when you change the layout of your scene.

The other two objects in the scene graph are simpler. One represents the background image the other one the main character.

For some games scene graphs can get very complex and include hundreds of different nodes. The key takeaways for now are:

1. Every node in Cocos2D can have children
2. A hierarchy of nodes is called a scene graph
3. Children of nodes are placed relative to their parents - often it is useful to group nodes that are moved together under one parent

Now that you have a basic understanding of how games are structured in Cocos2D we will take a look at a second tool which we will be using throughout this book: SpriteBuilder.

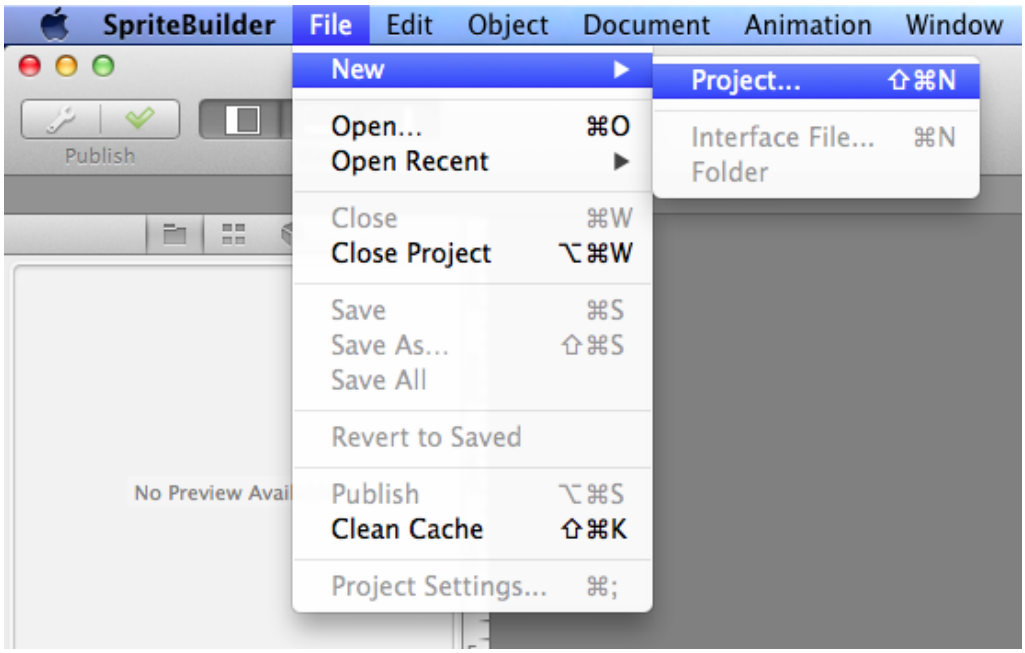
2.3 Introduction to SpriteBuilder

You have learned the fundamentals of the game engine we will use. Now we will take a look at a tool called SpriteBuilder which we will use to create the majority of our game content. The main purpose of SpriteBuilder is to provide a visual editor for the creation

of scenes, animations and more. For most games you will create some basic mechanics in code (enemy movement, score mechanism, etc.) but you will create the most of your game content in SpriteBuilder since it is a lot easier to create levels, menus and other scenes in an editor that provides you with a live preview instead of putting these scenes together in code.

2.3.1 Creating a first project

To dive into the features of SpriteBuilder we will create our first project! Create a new project by opening SpriteBuilder and selecting *File* > *New* > *Project...*:



SpriteBuilder will ask for a name and a location for the new project. Name it *HelloSB*. After you create the project the folder structure should look similar to this:

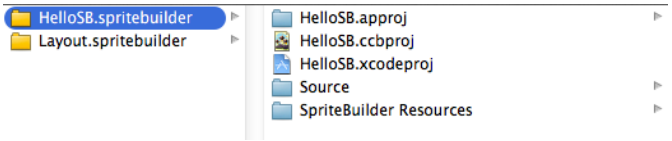


Figure 2.4: SpriteBuilder project folder structure

Every SpriteBuilder project is contained in a *.spritebuilder* folder. Within this folder all the files of the SpriteBuilder project are stored - along with an Xcode project.

SpriteBuilder and Xcode

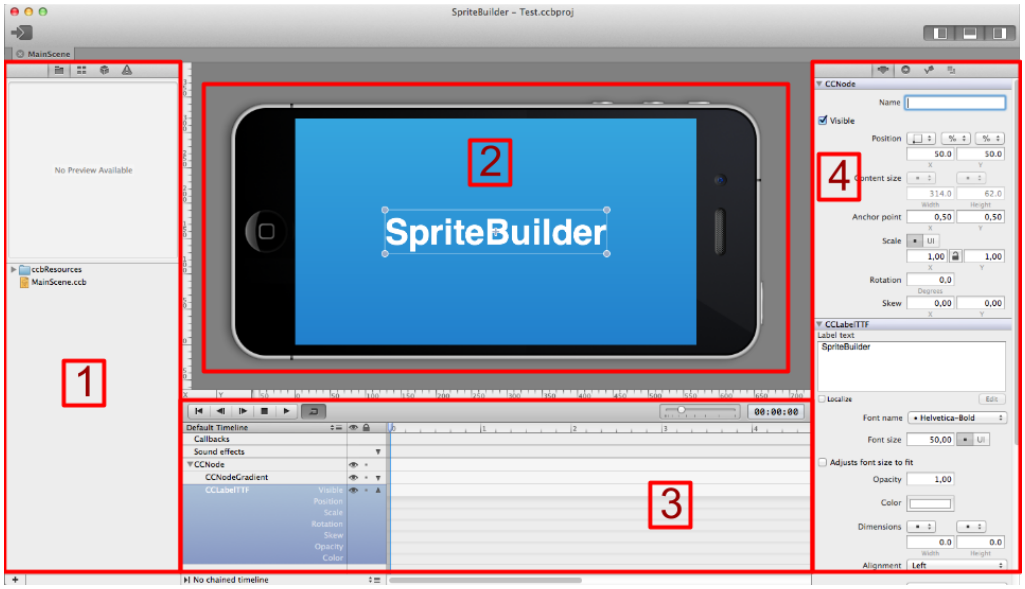
SpriteBuilder will create an Xcode project for every new project you create! The Xcode project will automatically contain the newest version of Cocos2D - very handy.

Later on you will learn more about how the SpriteBuilder project and the Xcode project work together. The general rule is that all code will be part of the Xcode project and most content creation will happen in the SpriteBuilder project.

2.3.2 The Editor

When you have created your first SpriteBuilder project you will see that the SpriteBuilder UI gets enabled. Let's take a look at the different parts of the editor to get a better understanding of SpriteBuilder.

The SpriteBuilder interface is divided into 4 main sections:



1. *Resource/Component Browser:* Here you can see the different resources and scenes you have created or added to your project. You can also select different types of Nodes and drag them into your scene.

2. *Stage:* The stage will preview your current scene. Here you can arrange all of the Nodes that belong to a scene.

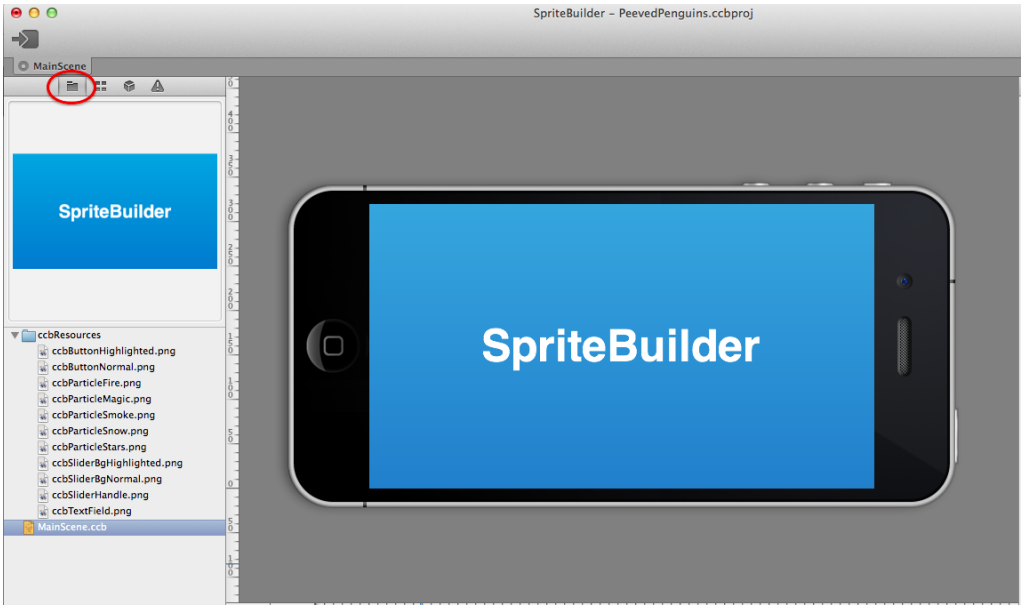
3. *Timeline:* The timeline is used to create animations within SpriteBuilder.

4. *Detail View:* Once you select a node in your scene, this detail view will display a lot of editable information about that node. You can modify positions, content (the text of a label, for example) and physics properties.

Let's take a closer look at some of the most important views.

File View

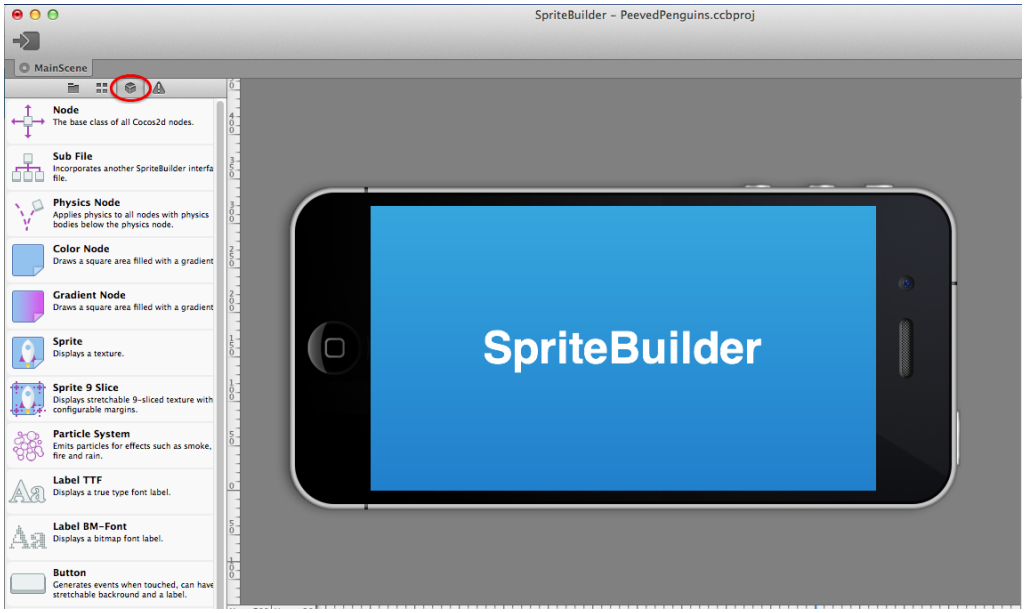
The first tab in the resource/component browser represents the *File View*. It lists all the *.ccb* files and resources that are part of the SpriteBuilder project:



In this view you can add new resources and restructure your project's folder hierarchy.

Node Library

The third tab in the left view is the Node Library:



This panel shows you all available node types you can use to construct your Gameplay scenes and menus. You will drag these nodes from this view to the stage in the center to add them to your scenes.

Inspector

The first tab of the Detail View (the right panel) is the Inspector. Once you have selected an object on your stage you can use this panel to modify many of its properties, like position and color:

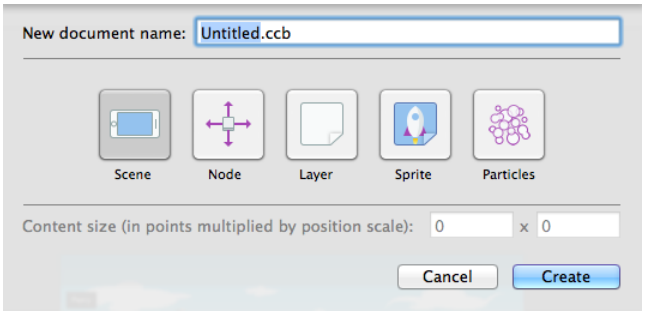
Code Connections

The second tab on the right panel let's you manage code connections for your selected node. As mentioned previously the entire code for your games will be written as part of the Xcode project. This view allows you to create connections between the Xcode project and the SpriteBuilder project. For example you can set a custom Objective-C class for a node or you can select a method in your code that shall be called once a button in your scene is tapped.

Code connections will be discussed in detail later on.

2.3.3 CCB Files

CCB Files are the basic building blocks of your SpriteBuilder project. Every scene in your game that is created with SpriteBuilder is represented by one CCB File. However CCB Files are not only used to create entire scenes - they are used to create any kind of scene graph. SpriteBuilder provides different kinds of templates depending on which type of scene graph you want to create. You get an overview of the available CCB File templates when you create a new one, by selecting *New > File...* from the *File* menu in SpriteBuilder:



These are the different templates briefly explained:

Scenes will fill the full screen size of the device.

Nodes used primarily for grouping functionality, don't have a size.

Layers are nodes with a content size. This is useful, for instance, when creating levels or contents for scroll views.

Sprites used to create (animated) characters, enemies, etc.

Particles is used to design particle effects.

You will get a good understanding when to use which type of CCB File once we get started with our example projects. The key takeaway is that CCB Files are used by SpriteBuilder to store an entire scene graph including size, positions and many other properties of all the nodes that you have added.

2.3.4 How SpriteBuilder and Xcode work together

I have mentioned how SpriteBuilder and Xcode integrate a couple of times briefly. In order to be a well versed and efficient SpriteBuilder game developer it is very important to understand the details of this cooperation.

When creating a SpriteBuilder project, SpriteBuilder will create and maintain a corresponding Xcode project. In SpriteBuilder will you create multiple CCB Files that describe the content of the scenes in your game. You will also add the resources that you want to

use in your game and set up code connections to interact with the code in your Xcode project. Xcode will be the place where you add code to your project and where you run the actual game.

Since Xcode is the tool that actually compiles and runs your game it needs to know about all the scenes and resources that are part of your SpriteBuilder project. Therefore SpriteBuilder has a **publish** functionality, provided by a button in the top left corner of the interface:

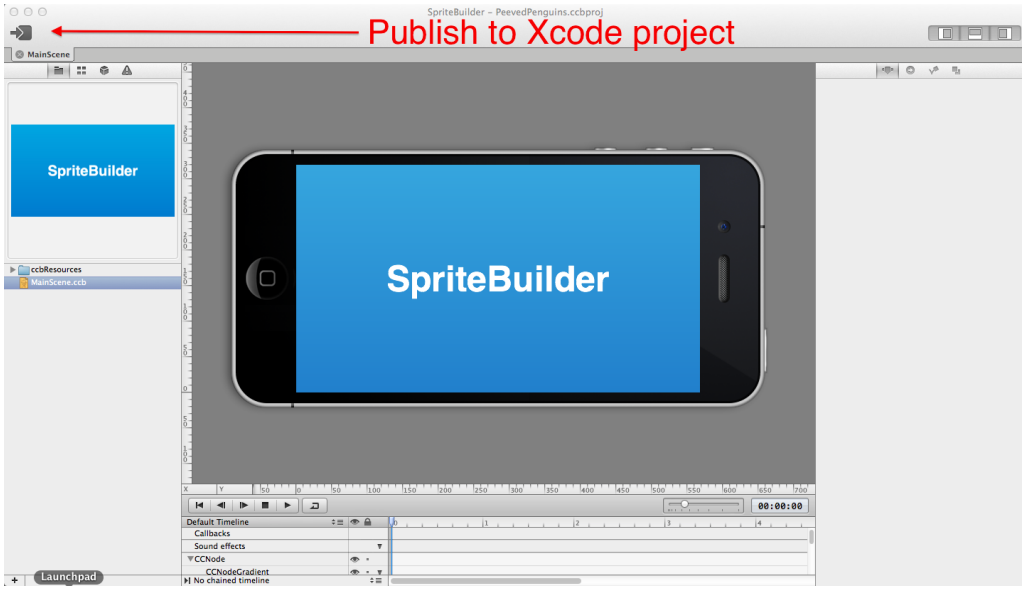


Figure 2.5: Use the publish button to update your Xcode project with the latest changes in your SpriteBuilder project.

Using that button, you publish your changes in your SpriteBuilder project to your Xcode project. Whenever you changed your SpriteBuilder project and want to run it you should hit this button before building the Xcode project.

Here's a diagram that visualizes how SpriteBuilder and Xcode work together:

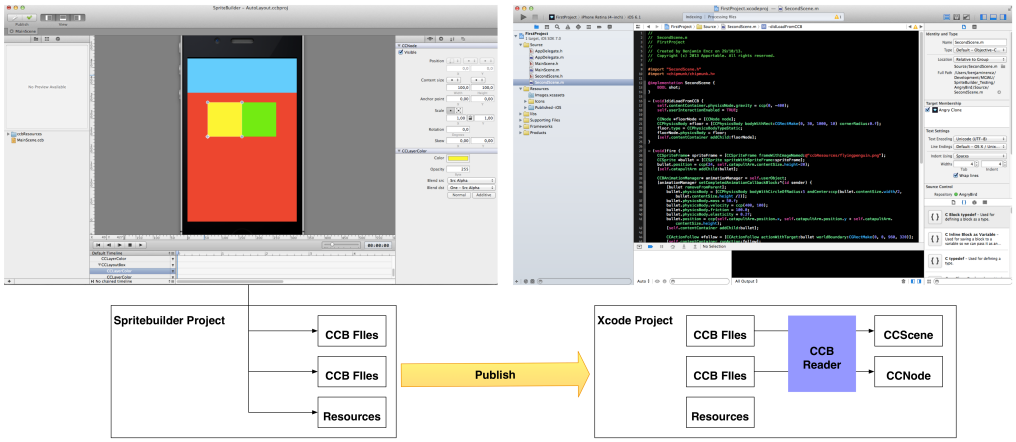


Figure 2.6: SpriteBuilder creates and organizes a Xcode project for you. Adding all the resources and scenes you have created.

CCB Files created in SpriteBuilder store a scene graph; the hierarchy and positions of your nodes. When publishing a SpriteBuilder project the CCB Files and all other project resources are copied to your Xcode project. When running the project in Xcode a class called CCBReader will parse your CCB Files and create the according CCNode subclasses to reconstruct the scene graph you have designed in SpriteBuilder.

If you would use Cocos2D without SpriteBuilder you would manually create instances of CCNode, CCSprite, etc. in code and add children to these nodes - essentially building the entire scene graph in code.

When using SpriteBuilder the CCBReader class will build this scene graph for you, based on the information stored in the CCB Files that you created in SpriteBuilder.

Another important part of information contained in CCB Files that we have not discussed in detail yet are *Code Connections*.

2.3.5 Code Connections

Code connections are used to create links between your scenes in SpriteBuilder and your code in Xcode. There are three basic types of code connections:

Custom Classes are an important information for the CCBReader. As mentioned previously the CCBReader builds the scene graph by creating different nodes based on the information in your CCB File. By default it will create an instance of CCSprite for every sprite you added in SpriteBuilder an instance of CCNode for every node you added, etc. Often however you will want to add custom behaviour to a node (for example a movement pattern for an enemy). Then you will have to use the *Custom Class* property to tell the CCBReader which class it should instantiate instead of the default one. Whichever class you enter here needs to be a subclass of the default class (e.g. a subclass of CCSprite). You will learn how to use this feature in the final project of this chapter!

Variable Assignments If you have assigned a *Custom Class* you can use variables assignments to retrieve references to different nodes in the scene. For example a character might want a reference to its right arm node (a child of the character node) in order to move it.

Callbacks are only available to UI elements like buttons and sliders. They allow you to decide which method should be called on which class once a button is pressed.

Now you should have an idea about what code connections are used for and which kinds exist. We will discuss the details of all types when we use them as part of our example projects.

2.4 A first SpriteBuilder project

You have already created the SpriteBuilder project called *HelloSB*. Now we will start adding some content to it. The project built in this chapter will consist of two scenes one start screen and one game screen. In the game screen the user will be able to spawn randomly colored squares that rotate by tapping on the screen.

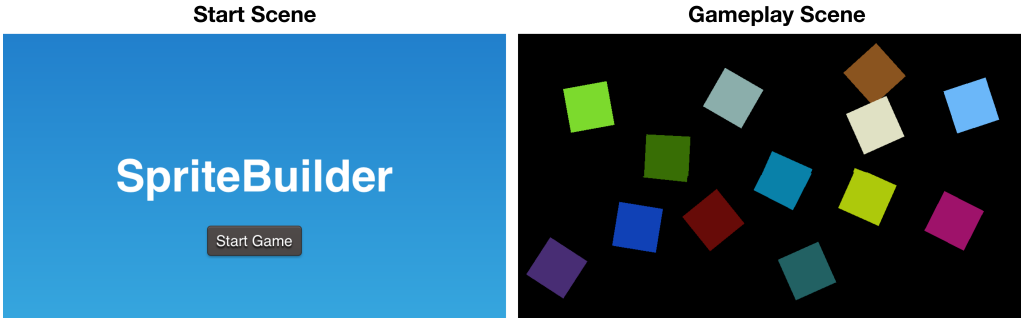


Figure 2.7: The project build throughout this chapter

By creating this project you will learn all of the following:

- Creating scenes in SpriteBuilder
- Creating code connections (callbacks, variable assignments and custom classes)
- Switching between different scenes
- Manipulate a scene graph from code (add/remove nodes, load CCB Files and add them to the scene)
- Use the Cocos2D action system to create animations
- Use the Cocos2D touch handling system to capture touches

2.4.1 Setting up the first scene

Now it is time to open the *HelloSB* SpriteBuilder project. We want to add a *Start Button* to the first scene. When this button is tapped we want to switch to the second scene.

Positioning the first button

Start by adding a button to the first scene:

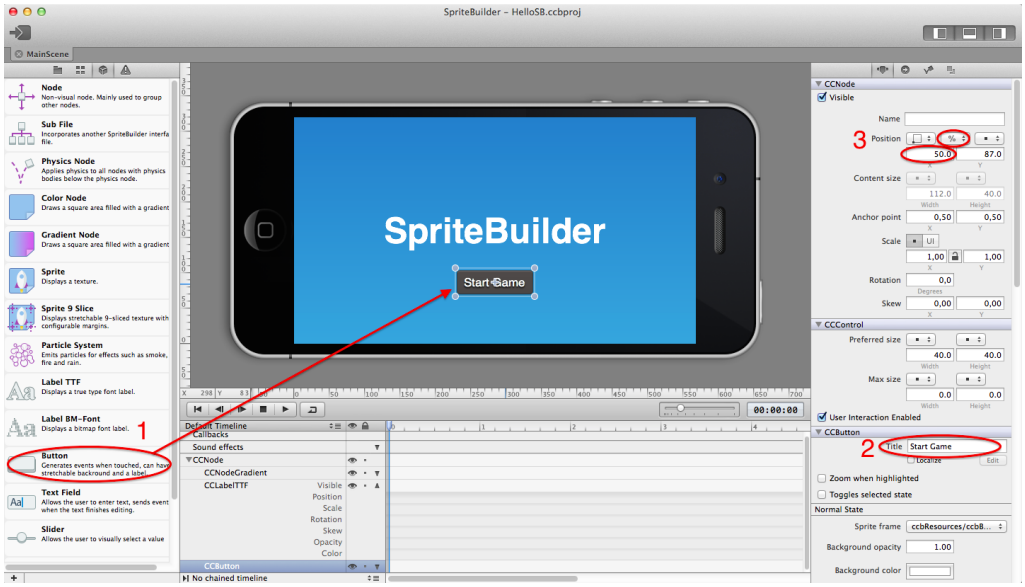


Figure 2.8: The project build throughout this chapter

One simple button, but since this is your first action in SpriteBuilder there's *lots* to explain about it. Let's look at the three steps highlighted in the image above, one by one.

(1) Open *MainScene.ccb* by double clicking it in the left resource pane. Then open the third tab in the left pane, the *Node Library*. Remember, this section shows you all the different node types supported by SpriteBuilder. Select the *Button* and drag it over to the stage, dropping it below the existing label. Dropping it on the stage will add this node to your scene. Another way of adding a node to a scene is dropping it to the timeline at the bottom of the screen - we will look at this later.

(2) Make sure the button is selected, because we want to change some properties of it. Whenever you have selected a node the right pane will display all the properties you can edit. Navigate to the *Title* textfield in the property pane and change the title of the button to *Start Game*.

(3) So far - so simple. Step number three will expose you to a very interesting feature of SpriteBuilder: the positioning system. It will allow you to not only use absolute positions but also positions that are relative to the size of the parent node. We want to center the button horizontally so we choose the position type for the X component to be *in percent of parent container* by selecting that option from the dropdown menu. Now we assign *50* as value, because that expresses the horizontal center of the parent container. Whichever screen this button will be displayed on, it will always be vertically centered (yes, even on an iPad)!

Positioning System in Cocos2D and SpriteBuilder

The positioning system in Cocos2D is designed from the ground up to make it easy to design scenes and user interfaces for different screen sizes and resolutions. The comfortable days where the 3.5-inch iPhone was the only available iOS device and defining layouts with absolute positions was acceptable are finally over. Today app and game developers face a variety of different devices and customers justifiably expect your software to work great on all of them. Cocos2D offers the following properties on CCNodes to allow developers to design their interfaces with great flexibility:

- Anchor Point
- Reference Corner
- Position Type
- Size Type

Check the extra chapter on dynamic layouts in SpriteBuilder.

Now the button is placed correctly. Next, we want to assign an action to it. When the button is tapped we want to transition to our second scene.

Setting up a code connection

Earlier you learned that SpriteBuilder has three types of code connections (2.3.5). Now we will use one of them in our project - *Callbacks*. Callbacks are only available to nodes that allow for some sort of user interaction (this means they need to be subclasses of `CCControl`). Buttons, next to Sliders and Text Fields are one of these types of nodes. Select the button we have added to the scene earlier and select the third tab of the right pane:

Figure 2.9: Nodes that allow user interaction can use callback methods to connect to the code base

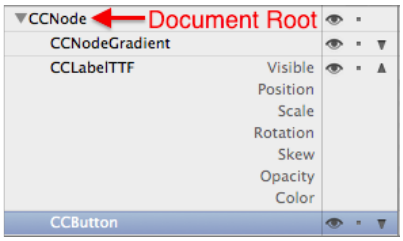
Inside the CCControll section you can see two options called *selector* and *target*. Here you can choose which method (selector) shall be called on which object (target) when this button is tapped by a user. As selector enter `startGame`. As target choose *Document Root*.

Targets and Selectors

The concept of targets and selectors is part of design pattern widely used throughout the Cocoa framework (Target/Action pattern). A *selector* is a method name and a *target* is the object that shall receive this method. Further reading: https://developer.apple.com/library/ios/documentation/general/conceptual/Devpedia-CocoaApp/TargetAction.html

As you can see you cannot choose an arbitrary object to be the target of this callback, you can only choose between two different ones:

Document Root The document root is the highest node within the current CCB File. The hierarchy of the CCB File is shown in the SpriteBuilder timeline:

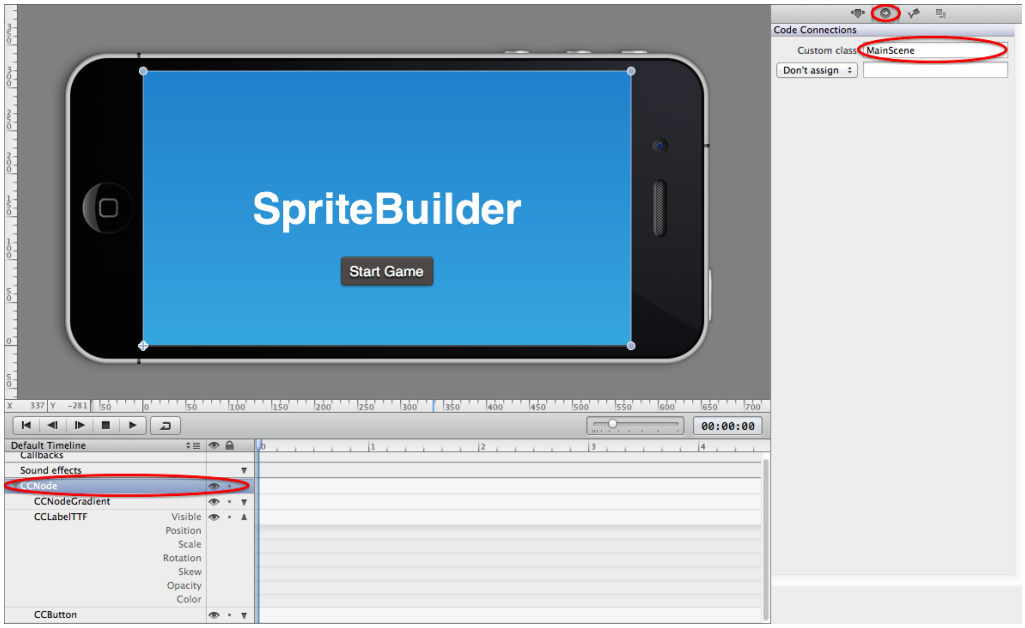


If you select the document root as target, the `startGame` method will be called on the top level CCNode.

Owner if you want the callback to call an object that is not part of your CCB File you can use the *owner* option. Later in this book you will learn how to set up an owner object for a CCB File.

For our button we have decided that the `startGame` method should be called on the document root when the button is tapped. Next, we will have to implement this `startGame` method within our document root. But to which *class* could we add this method? In order to find that out we need to understand the concept of *Custom Classes* . Think about it - by default our document root is an instance of a plain CCNode class. Now we want to call

a method called `startGame` on this object. Our problem: the CCNode class does not have a `startGame` method! This is where custom classes come to rescue us, they allow us to tell SpriteBuilder that our document root node should **not** be a plain CCNode but should be an instance of a class that we have created and that knows about our `startGame` method. To define a custom class for the document root you need to select the document root (the top-level CCNode) from the timeline and open the third tab in the right pane:



In the *Custom class* textfield a developer can enter a class name. The class entered here needs to be part of the Xcode project related to this SpriteBuilder project. As you can see every new SpriteBuilder project already comes with a custom class set up for the root node of *MainScene.ccb*. When the CCBReader loads this CCB File it will create an instance of `MainScene` instead of an instance of `CCNode`. Now our document root object is a `MainScene` object! That also means that we have saved the puzzle of where to add the code for the `startGame` method - it needs to be part of the `MainScene` class.

Requirements for Custom Classes

Every custom class has to be a subclass of the default class for a given node. For example, the default class for the *Sprite* node in SpriteBuilder is CCSprite. If a developer wants to set a custom class for a Sprite node, that class has to be a subclass of CCSprite. **Why?** SpriteBuilder expects custom classes to only **add** behaviour to a default class. All the functionality of the default class should remain available. If your custom class for a Sprite node doesn't allow SpriteBuilder to set an image, because it is a subclass of CCNode the CCBReader and finally also you will run into big problems!

Adding Code to a SpriteBuilder project

When creating games with SpriteBuilder we are always working with two tools. SpriteBuilder to create interfaces and scenes (our game content) and Xcode to add code (game mechanics, etc.). Now we will add our first few lines of code to the `MainScene` class. Now it's time to publish the changes in our SpriteBuilder project, so that they are available in our Xcode project. Use the publish button in the top left corner of the SpriteBuilder interface (2.3.4).

Now open the Xcode project (it's called *HelloSB.xcodeproj* and is located inside the *HelloSB.spritebuilder* folder). You will see that project contains two classes, `AppDelegate` and `MainScene`. As part of the template for new SpriteBuilder projects the `MainScene` class has already been created for you. For any subsequent custom classes you link in your SpriteBuilder project you will need to create the according class in Xcode on your own.

Now it's finally time to implement the `startGame` method. Open the *MainScene.m* and file and add the following method:

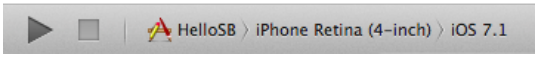
```
Line 1  - (void)startGame {
    -     CCLOG(@"Start Button Pressed!");
    - }
```

For now we will simply use the `CCLOG` macro to log a text to the console once the button is pressed, this is an easy way to check if our code connection is set up correctly.

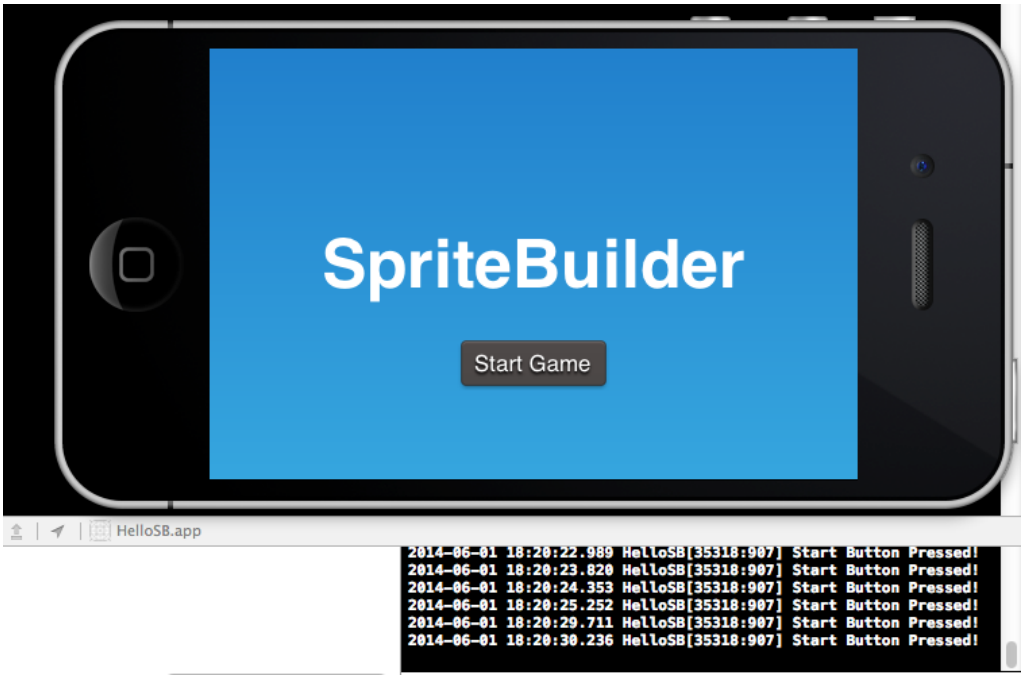
Displaying the console in Xcode

To display the console in Xcode select *View -> Debug Area -> Activate Console*.

Now, run the Xcode project by hitting the play button in the top left corner. You should check that you have selected *HelloSB* as target and are set up to run the app on a simulator (indicated by a device description instead of a device name):



Hitting the run button will compile your app and launch it on an iOS simulator. Once your app is launched, click on the start button and check the console for the log message. You should see something similar to this:



You have successfully set up your first SpriteBuilder scene and have created a working code connection! Later on this button shall trigger a transition to the second scene in the game. Before we can implement that we need to create the second scene in our SpriteBuilder project!

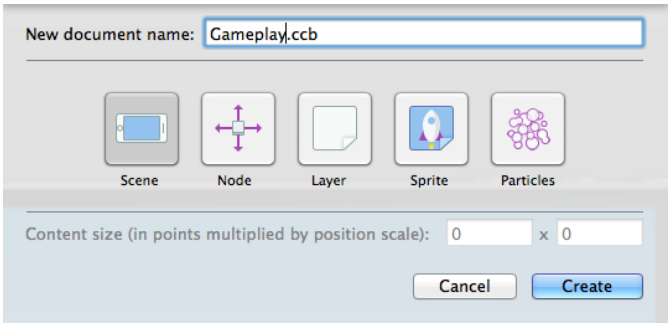
Common Error 2.1

If you are not getting the expected result, check for all of these common errors:

- Have you published your SpriteBuilder before running in Xcode?
- Is the custom class of the root node of *MainScene.ccb* set to `MainScene`
- Does the button in *MainScene.ccb* have the correct target and selector?

2.4.2 Creating the Gameplay Scene

Now it's time to create your first scene using SpriteBuilder from scratch. The scene we are going to create is the Gameplay scene. To create a new scene (or any other CCB File) select: *File -> New -> File...* from the SpriteBuilder menu. Then you will see the following dialog appear:



The dialog will ask you for a name for the CCB File and a template type. For now we are going to use the name *Gameplay.ccb* and the type *Scene*. Once you hit the create button you will see the new, blank scene appear.

Our Gameplay scene will remain empty. As you have seen in the outline of the project, we want to dynamically add colored objects to the game, whenever the user taps into our Gameplay scene - initially however, the scene will be blank. Now that we have created the Gameplay scene, we can add the transition from the Main scene to the Gameplay scene.

2.4.3 Adding a Scene Transition

Transitions are essential for any game. We use them whenever we want to switch from one scene to another. Transitions cannot be configured in SpriteBuilder, they always need to be implemented in code. To implement this step, you need to open your Xcode project again.

Cocos2D has one central class that is responsible for displaying the active scene and generating transitions between different scenes: `CCDirector`. CCDirector is implemented

as a singleton - thus there's only one CCDirector per Cocos2D game. The instance can be accessed through the class method `[CCDirector sharedInstance]`.

CCDirector is versatile!

CCDirector is responsible for a lot more than only handling active scenes and scene transitions. It is basically a collection of different global Cocos2D settings. The scene handling methods however are the most frequently used CCDirector methods.

CCDirector provides a large collection of methods to present scenes with and without transitions, here are the most important ones:

```
Line 1  - (void)presentScene:(CCScene *)scene;
    -   - (void)presentScene:(CCScene *)scene withTransition:(CCTransition
            ↪ *)transition;
    -   - (void)pushScene:(CCScene*) scene;
    -   - (void)pushScene:(CCScene *)scene withTransition:(CCTransition *)
            ↪ transition;
    5   - (void)popScene;
    -   - (void)popSceneWithTransition:(CCTransition *)transition;
    -   - (void)popToRootScene;
    -   - (void)popToRootSceneWithTransition:(CCTransition *)transition;
```

Cocos2D has two different approaches for displaying a new scene. **Replacing** the current scene with a new one, using the `presentScene:` methods, or **Pushing** the new scene on top of the currently active one using the `pushScene:` methods. Whichever type you choose, you always have the option to provide a transition effect for presenting a scene, or not to provide a transition effect and display the new scene instantaneously. If you want to provide an effect you need to create an instance of `CCTransition`.

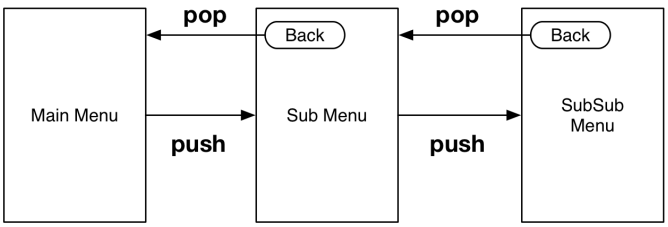
Before we look into using transition effects, let's take a look at the differences between pushing and replacing a scene.

Replacing scenes vs. pushing scenes

When you simply want to replace the current scene with a new one you should use the `presentScene:` method. Here's an example:

```
Line 1  [[CCDirector sharedDirector] presentScene:myNewScene];
```

Very simple! So why would one use the `pushScene:` method? Let's assume the following scenario where we want to implement a menu with multiple submenus. Whenever a player hits the back button, he wants to return to the previous menu:



This is a case where it is a lot easier to use `pushScene:` and `popScene:` instead of simply replacing the currently running scene. Whenever a player selects a button that opens a sub-menu, we call:

```
Line 1  [[CCDirector sharedDirector] pushScene:submenu];
```

And whenever a player hits the *back* button in one of the sub-menus, we simply call:

```
Line 1  [[CCDirector sharedDirector] popScene];
```

This works, because CCDirector will remember the scene that we pushed before the current one and can easily return to it. This concept is called a *Navigation Stack*.

If you would try to implement the menu hierarchy using `presentScene:` you would have to explicitly define which scene each back button will present. The code for the back button of *SubMenu* would look like this:

```
Line 1  [[CCDirector sharedDirector] presentScene:mainMenu];
```

If you would ever change the menu hierarchy in your game, you would have to change the code for each back button.

Scene transitions - the right way

For **one time transitions** for example from a splash screen to the gameplay of a game, use `presentScene:`. Whenever a user can navigate between your scenes, e.g. by using a back button to return to the previous scene, make use of the navigation stack by using the `pushScene:` and `popScene:` methods.

Adding transition effects

For every scene replacement method there's one variation that takes an instance of `CCTransition`. The CCTransition instance provides an animation for transitions between different scenes. CCTransition provides multiple class methods to easily create them. Here's an example of how to provide an animated transition:

```
Line 1  CCTransition *transition = [CCTransition
           ↪ transitionCrossFadeWithDuration:1.f];
    -  [[CCDirector sharedDirector] presentScene:gameplayScene
    -  withTransition:transition];
```

Implementing a scene transition for our game

Now that you know the most important details about scene transitions, let's add the transition from our start scene to our Gameplay scene. Open *MainScene.m* in Xcode. Earlier we have already implemented a test version of the `startGame` method, where we printed a log message to the console. Now replace the current implementation of `startGame` with this one:

```
Line 1  - (void)startGame {
    -     CCScene *gameplayScene = [CCBReader loadAsScene:@"Gameplay"];
    -     CCTransition *transition = [CCTransition
              ↪ transitionFadeWithDuration:1.0];
    -     [[CCDirector sharedDirector] presentScene:gameplayScene
              ↪ withTransition:transition];
```

```
5 }
```

Now that you are familiar with scene transitions, the only interesting line should be the one where we use the `CCBReader` to load a CCB File. The CCBReader class was briefly introduced at the beginning of this chapter (2.3.4). It is capable of reading SpriteBuilders *.ccbi* files and creating the according Cocos2D classes from the information stored in them. Whenever we want to load a scene or any other type of node that we created in SpriteBuilder into code we use the CCBReader class. In the lines shown above, we load the content of our *Gameplay.ccb* into a variable called `gameplayScene`. The `loadAsScene:` method wraps whatever scene graph you load into an instance of `CCScene`, use it whenever you want to load a CCB File as a scene. Then we create a simple fade transition and store that object in the `transition` variable. Finally we use the `CCDirector` to present our loaded scene with the transition we just created.

You are now ready to run this version of the game from Xcode! When you tap the *Start* button on the first scene, you should see a transition to our black Gameplay scene that lasts for one second.

Well done! You have learned how to create a new scene in SpriteBuilder and how to implement transition between different scenes in a game. Now let's implement the actual gameplay of our first example game!

.ccb and .ccbi

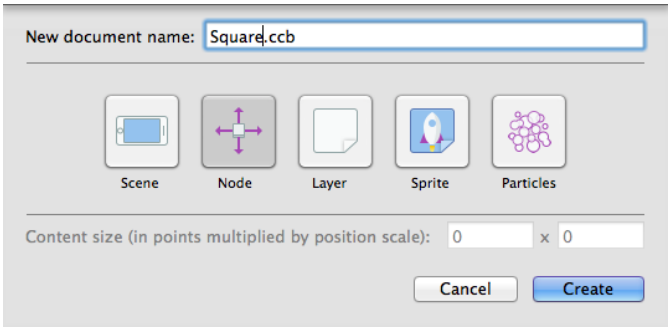
The files with the file extension *.ccb* are in XML-format and are used by SpriteBuilder to store and read information about a scene or node created in SpriteBuilder. When a SpriteBuilder project gets published, SpriteBuilder generates a binary version of each *.ccb* file. The file extension for these binary files is *.ccbi* and they are a lot smaller than their corresponding *.ccb* files. The CCBReader reads these smaller binary files.

2.4.4 Implementing the Gameplay

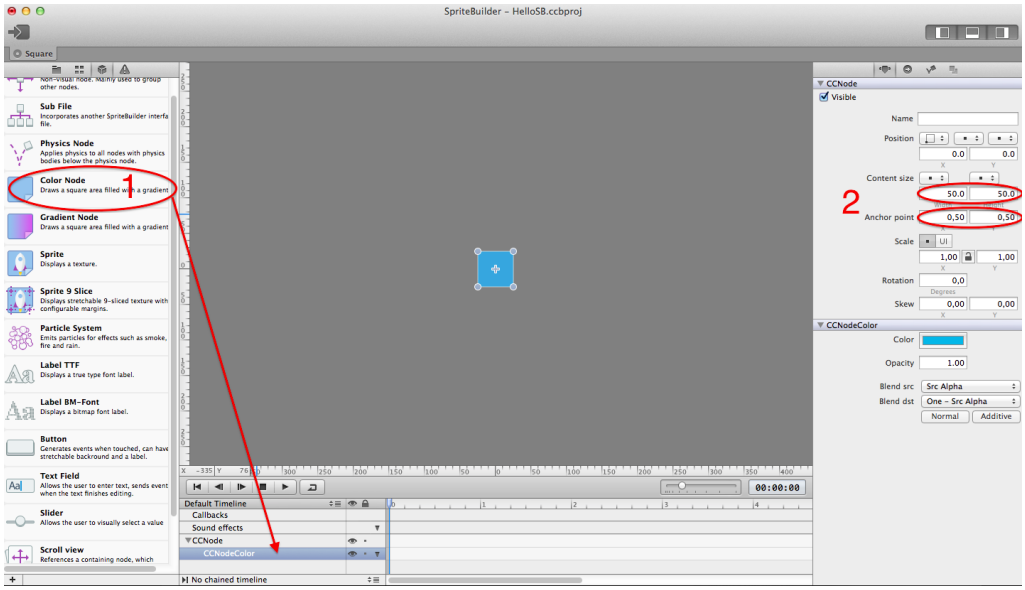
Now it's time to implement the actual gameplay. For our first project we want to keep that fairly simple. Whenever a user touches the screen, we want to add a rotating square with a random color to the gameplay scene. We position the square at the location of the touch.

Creating the Square CCB File

Let's start by creating the square we want to spawn during the game in SpriteBuilder. Create a new CCB File of type *Node*:



The squares we generate in the game shall have a color. A default CCNode cannot display a color. In order to display a color we need to use a `CCNodeColor`. The SpriteBuilder node for a CCNodeColor is called *Color Node*. The root node of every CCB File is a plain CCNode, that cannot be changed. This means we need to add the *Color Node* as a child of the root node of *Square.ccb*:

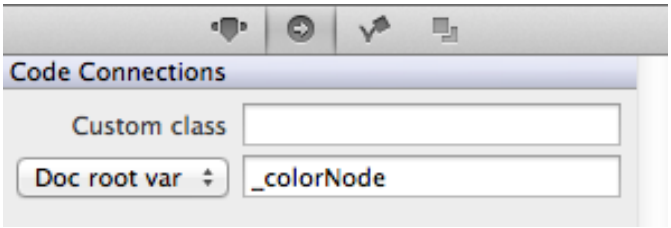


(1) Open the *Node Library* and drag a *Color Node* to the stage or the timeline in order to add it to the root node of of *Square.ccb*.

(2) Center the new *Color Node* on the root node by selecting an *Anchor Point* of (0.5, 0.5). Change the *Content Size* of the node to (50, 50).

Now the basic square is set up. Next, we need to set up a code connection. Earlier you have seen the use of *Custom Classes* and *Callbacks*, now we will use the third type of code connections supported by SpriteBuilder a *Variable Assignment.*. Variable assignments are generally used when we want to access a part of our scene graph in code. In our game, whenever a new square is created we want to set a random color for this square. Generating a random color is something we need to do in code and cannot do in SpriteBuilder. This also means that we need a way to *apply* the random color we generate in code to our square that we have set up in SpriteBuilder. The displayed color is defined in the *Color Node* that we just added. We will need a reference to this *Color Node* to change the color of our square from code. **Select CCNodeColor from the timeline** (and make sure

that you have selected the Color Node and not the Root Node!) and open the connection tab (the second tab on the right pane):

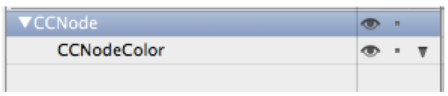


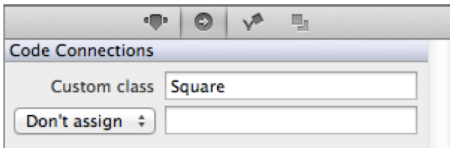
As the variable name (entered in the text field), choose *_colorNode*. As the second option you need to choose the object to which this variable will be assigned to. Just as for callbacks you can choose between the *Document Root* and the *Owner* (2.4.1). We choose the *Document Root*, which means that SpriteBuilder will attempt to store a reference to the *Color Node* in an instance variable called *_colorNode* on the root node object of this CCB File. We now face the same 'problem' as earlier when we set up a *Callback*. The root node of *Square.ccb* is a plain CCNode and a plain CCNode does not have an instance variable called *_colorNode*! We once again need to define a custom class for the root node of this CCB File.

Variable Assignments, Callbacks and Custom Classes

Always remember that you practically cannot set up a *Variable assignment* or a *Callback* for the *Document Root* without also setting a custom class for the root node of the corresponding CCB File.

Select the root CCNode node from the timeline and set the custom class for this node to *Square*:





When the *CCBReader* reads this CCB File it will create instance of the class *Square* as the root node and it will assign a reference to the *Color Node* to an instance variable of *Square* called *_colorNode*. This way we will be able to access the *Color Node* and change the color of our square programmatically!

Setting up a custom class for the Gameplay

In our *Gameplay* scene we want to respond to touches and spawn squares. All of that functionality needs to be implemented in code. Therefore we need to define a custom class for the root node of our *Gameplay.ccb* (if you struggle with the following instructions you can double check how we set up a custom class for *Square.ccb*).

1. Open *Gameplay.ccb*
2. Select the root CCNode from the timeline
3. Open the code connections tab (the second tab on the right pane)
4. Define the *Custom Class* to be *Gameplay*

We've set up multiple code connections throughout this chapter. In order for all of them to work, we need to **publish the SpriteBuilder project** and switch to the Xcode project and create the classes and instance variables that we are referencing in the SpriteBuilder project.

Creating the Square class

Open Xcode and create a new Objective-C class by selecting *File -> New File...* and choosing *Objective-C class*. As class name choose *Square* and define it to be a subclass of CCNode. Remember, a custom class always has to be a subclass of the node type you have selected in SpriteBuilder. The node type of the root node of *Square.ccb* is a CCNode therefore `Square` needs to be a subclass of CCNode.

Now open *Square.m* and add the instance variable `_colorNode` to the `Square` class. This variable is the one that we defined in SpriteBuilder to store the reference to the `CCNodeColor` that displays the color of our square:

```
Line 1  #import "Square.h"
-
-   @implementation Square {
-     CCNodeColor *_colorNode;
5   }
-
-   @end
```

After adding the instance variable the code for *Square.m* should look as shown above. Now that we have a reference to the `CCNodeColor` we need a position in code where we can set a random color for that node.

The requirements for this projects state that we need to choose a random color for our Square as soon as it is added to the Gameplay scene. **How can we be informed about the square being added to the Gameplay scene?** Therefore we need to take a closer look at what we call the **Node Lifecycle**.

We have five important methods that inform us about certain lifecycle events on CCNode subclasses. All of the methods below are called on all nodes that are part of the scene that is being loaded/presented/hidden:

didLoadFromCCB this method is called when the `CCBReader` has created the complete node graph from a CCB file and all code connections are set up. You implement this method to access and manipulate the content of a node. You cannot access child nodes of the node or code connection variables before this method is called. Note that this method is only called on nodes that are loaded from CCB Files.

onEnter/onEnterTransitionDidFinish are called as soon as a node enters the stage. If you are presenting a scene with an animated transition, `onEnter` will be called on that scene as soon as the transition starts and `onEnterTransitionDidFinish` will be called when the transition completes. If a scene or node is being presented/added without an animated transition both methods are called directly after each other.

onExitTransitionDidStart/onExit are called as soon as a node leaves the stage. If you are hiding a scene with an animated transition, `onExitTransitionDidStart` will be called on that scene as soon as the transition starts and `onExit` will be called when the transition completes. If a scene or node is being hidden/removed without an animated transition both methods are called directly after each other.

You will get to see lots of examples of how to use the lifecycle methods throughout this book, for now we know that we need to implement `onEnter` to pick and apply a random color for our square as soon as it gets added to the Gameplay scene. Add this implementation of `onEnter` to *Square.m*:

```
Line 1  - (void)onEnter {
    -     [super onEnter];
    -
    -     // arc4random_uniform(N) generates a random number between 0 and
              ↪ N-1
    5     float red =   arc4random_uniform(256) / 255.f;
    -     float green = arc4random_uniform(256) / 255.f;
    -     float blue =  arc4random_uniform(256) / 255.f;
    -
    -     _colorNode.color = [CCColor colorWithRed:red green:green blue:
              ↪ blue];
   10 }
```

The lines above generate three random numbers, one for each color component with a value between 0.0 and 1.0. These three numbers are used to create an instance of `CCColor` and set it as the color of our node.

Now the square will appear in a random color as soon as we add it to a scene. The second requirement for our square is that it shall rotate while on the screen. One of the ways to move and/or animate a node in Cocos2D is using the Cocos2D Action System. The Action System provides a simple and expressive way for developers to implement animated changes like: *Move the main character to the top left corner in 2 seconds.*

The Action System consists of dozens of subclasses of `CCAction` - a majority of these actions represent some type of animated movement or transformation. `CCActionMoveTo`

for example moves a node to a target position within a provided time interval. This is how to use it:

```
Line 1  CCAction *move = [CCActionMoveTo actionWithDuration:2.f position:
            ↪ ccp(20, 100)];
    -   [aSimpleNode runAction:move];
```

All actions can be run by calling the `runAction` method and providing the action as an argument.

More about the Cocos2D Action System

The Cocos2D Action System is one of the most important building blocks for most games and we will discuss it in detail throughout this book. If you want to learn more about the Cocos2D action system right away, you can check the according chapter in the Cocos2D documentation: https://www.makegameswith.us/docs/#!/cocos2d/1.0/animations-movements

The Action System also provides several actions that take other actions as arguments. One example is `CCActionReverse` that reverses the action it is initialized with - for example moving a node backwards instead of forwards. Another example is `CCActionRepeatForever` that takes another action and - exactly, repeats it forever!

Add the following lines to the `onEnter` method of *Square.m* to make the square rotate endlessly:

```
Line 1  CCActionRotateBy *rotate = [CCActionRotateBy actionWithDuration:2.f
            ↪  angle:360.f];
    -   CCActionRepeatForever *repeatRotation = [CCActionRepeatForever
            ↪ actionWithAction:rotate];
    -   [self runAction:repeatRotation];
```

One of the nicest aspects of the Action System is that it produces very readable code, just as the one shown above. We rotate our square by 360 degrees in 2 seconds and repeat that forever!

Finally our implementation of `Square` is complete. Along the way you have learned about code connections, generating random numbers and using the action system. Now let's move on to implement the `Gameplay` class so that we can see our delightfully colored and rotating squares in action.

Creating the Gameplay class

In SpriteBuilder we have already created *Gameplay.ccb* and set up the custom class for the root node to be `Gameplay`. Now we need to add the `Gameplay` class in Xcode and implement touch handling code that creates a square and adds it to the Gameplay scene as soon as a player touches the screen.

Index