Contents

1 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 yet. We will be discussing all the relevant concepts throughout this chapter.

1.1 Installing the software

First things first. Let's install the software used throughout this book. In general there are two ways to install Cocos2D depending on whether you want to use SpriteBuilder or not. In 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*. Note that you should always use the latest version of Mac OS X and Xcode together with SpriteBuilder (as of this writing Mac OS X 10.9 and Xcode 6.0).

Figure 1.1: Cocos2D Technology Stack

After a couple of minutes the SpriteBuilder installation should be completed. Later throughout this chapter you will learn how to set up your first project.

1.2 Why Cocos2D

Well, now you have already installed Cocos2D. I still want to spend a moment on discussing why we are using this tool. The main goal of Cocos2D is to make mobile game development *easier*. Earlier we have discussed the basic concepts and benefits of game engines. You should now know that there are many problems developers have faced while developing games, animations, physics, etc. - and most of them have been solved and put into frameworks. You should not spend your precious time trying to solve them again. So now that you know that you definitely should use a framework - **which ones are available and why should you choose Cocos2D**?

Add brief discussion on different frameworks

1.3 Introduction to Cocos2D

First let us take a look at the features of Cocos2D. That will give you a basic understanding of which tasks you will hand off to the framework, later on we will be discussing all of these features in detail:

Scene Graphs Cocos2D provides the concepts of scenes and nodes. Everything that is rendered to the screen is part of a hierarchical *scene graph*. Instead of performing custom drawing code you define what your scene looks like by providing a scene graph and Cocos2D will render it for you.

Rendering Engine When using Cocos2D you don't need to write your own rendering code. Cocos2D provides a rendering engine built on top of OpenGL ES.

Action System A sophisticated action system allows you to define movements of objects and animations instead of writing a lot of custom code.

Physics Engine The Cocos2D physics engine automatically calculates movements of objects, collisions and more.

Node Library Cocos2D provides a large set of nodes as part of the framework. Nodes can be used to represent images, UI elements, solid colors, etc.

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. Let's take a closer look at how Cocos2D works.

1.3.1 The Cocos2D technology stack

Cocos2D is built on top of OpenGL ES 2.0. If you have ever written OpenGL code before, you know that it takes a lot of code even to render the most primitive scenes. OpenGL

is a fairly low level framework that gives the graphics programmer a lot of control over how and when certain tasks are performed - more control then you need for most 2D games. Cocos2D abstracts all of these tasks for you. Many Cocos2D developers write entire games without writing any OpenGL code at all. The following diagram shows which technologies are used by Cocos2D:

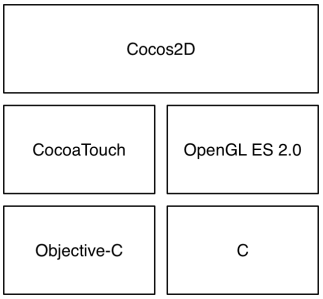


Figure 1.2: Cocos2D Technology Stack

The goal of a game engine like Cocos2D is that the game developer doesn't have to get in touch with rendering at all. Instead a developer defines which scenes exist in a game, which nodes are part of these scenes and which size, position and appearance these nodes have and Cocos2D will use OpenGL to render these scenes for him.

In order to provide this functionality Cocos2D consists of variety of classes - some important ones will be discussed in this chapter. All Cocos2D classes use the *CC* prefix (CCScene, CCNode, etc.).

When working with a 2D game engine for the first time you will be introduced to a whole set of new terminology. We have already talked about nodes and scenes but we haven't discussed what these terms mean. We will now start discussing the most important terms and get to know how the concepts behind them are implemented in Cocos2D.

How are games rendered in Cocos2D?

The most important aspect of Cocos2D is that it . . . **addmore**

1.3.2 Scenes

Scenes are the basic building blocks of all Cocos2D games, they are the highest level on which game content can be structured. Each scene in Cocos2D is a full-screen canvas. For every full-screen section of your game you will use *one* scene. **Add screenshots here** 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.

Scenes are represented by 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*.

1.3.3 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.

1.3.4 Scene Graphs

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

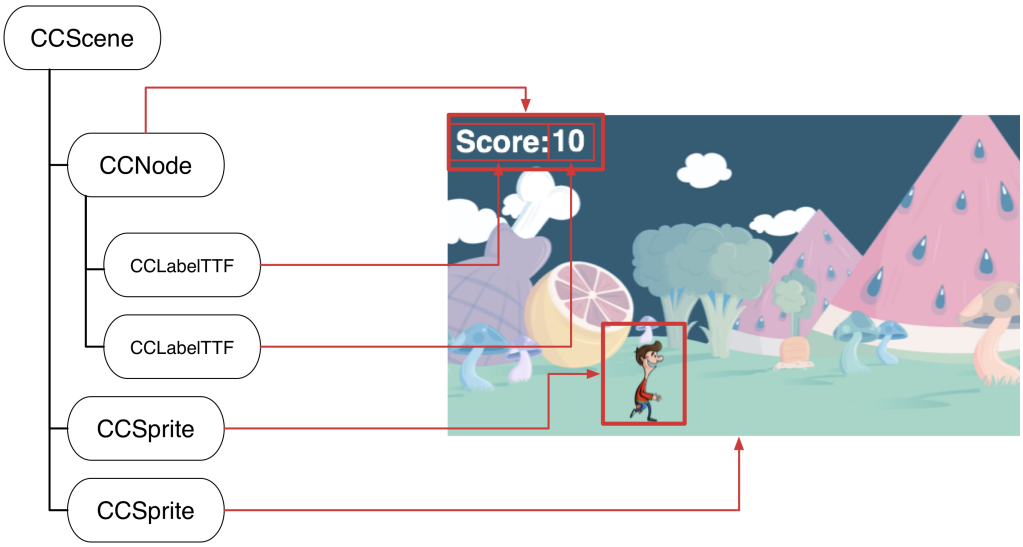


Figure 1.3: Cocos2D Scene Graph

On the left side of the image you can see the node hierarchy. On the first level you have the CCScene. As the first child we have a CCNode with two children of type CCLabelTTF. This CCNode is the first example of a grouping node, it groups the score caption label and the label displaying the actual score. 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

As you can see nodes are the most important building block of Cocos2D games - they are used to build everything that is visible in your game. Because it is so important to understand how nodes work in Cocos2D we will take a look at the most important properties and methods that CCNode provides.

1.3.5 An Introduction to CCNode

Every visible object in your game will be a subclass of CCNode. Because you use nodes to build and arrange your scenes it is important to understand how nodes are positioned and how positions of nodes can be accessed. Let's discuss the most relevant properties and methods to access and change size and position of a CCNode:

contentSizeInPoints the size of this node in points

positionInPoints the position of this node in points, expressed relative to the parent of this node

anchor point the anchor point is the center point for rotations and the reference point

for positioning this node

boundingBox the bounding box is a rectangle that encloses a node. You can only read it but not set it

The *contentSizeInPoints* and *positionInPoints* properties express the size and the position of a CCNode and should be fairly easy to understand. The *bounding box* and the *anchor point* however, are concepts related to game development and these may be new to you. The bounding box is a rectangle that encloses the entire node, you will see an example of a bounding box in the next diagram. The anchor point is relevant for positioning and rotating nodes.

Let's take a look at how anchor points influence positioning first. We know that the position of a node is expressed relative to its parent. More specifically every node position in Cocos2D is expressed from the *position reference corner* of the parent to the anchor point of the CCNode. Here's a visual example in which a bear node is placed relative to a background node:

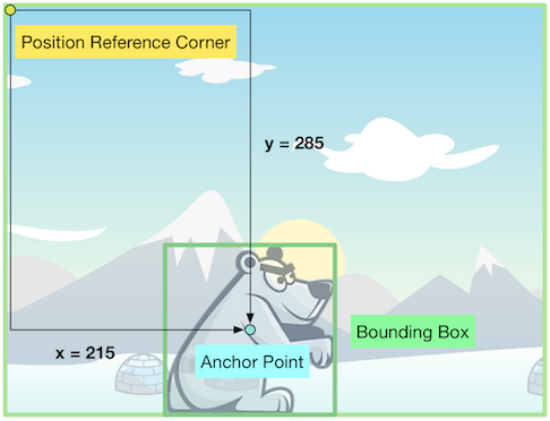


Figure 1.4: CCNode positioning example

As you can see, the *anchor point* and the *position reference corner* influence the position of a node. The anchor point can have any value between (0, 0), representing the bottom left corner of a node and (1,1), representing the top right corner of a node. In the example above, the bear has a anchor point of (0.5, 0.5) which is at the center of the bear. By choosing an anchor point of (0.5, 0.5), the *center* of the bear will be positioned at (215, 285). If we would choose an anchor point of (0,0) the *bottom left* corner of the bear would be positioned at (215, 285).

The *position reference corner* lets us define from which of the four corners of the parent node we are expressing the position of a node. In the example above the top left corner is the *position reference corner*. We will discuss how to use position reference corners when we start creating games that shall work on multiple screen sizes.

The anchor point is not only important for the positioning of a node. It has a second important function - it represents the center of rotation for a CCNode. Every CCNode rotates around its own anchor point. Here's an example of rotating the bear node with two different anchor points:

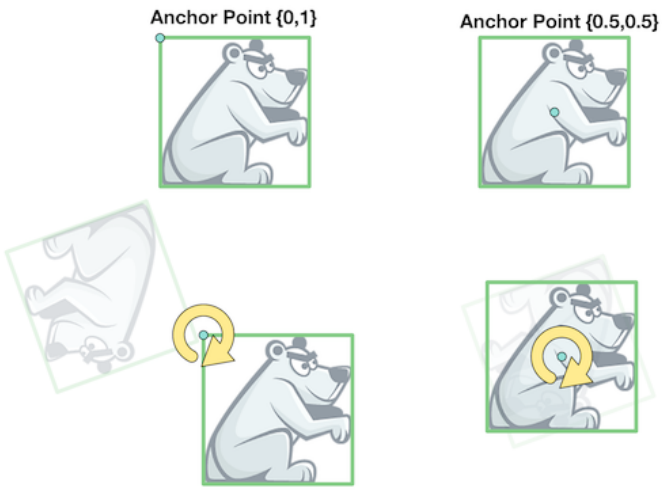


Figure 1.5: CCNode positioning example

There is a lot more to learn about CCNode, but for now our only goal is to get a basic understanding of how Cocos2D games are structured and what the most important parts of Cocos2D are.

You now know that Cocos2D game are structured into scenes. You know that everything visible in your game is a CCNode and that every CCNode can have multiple children. You also got a basic understanding of how nodes are positioned in Cocos2D.

Now that you have that basic understanding, we will take a look at a second tool which we will be using throughout this book: SpriteBuilder.

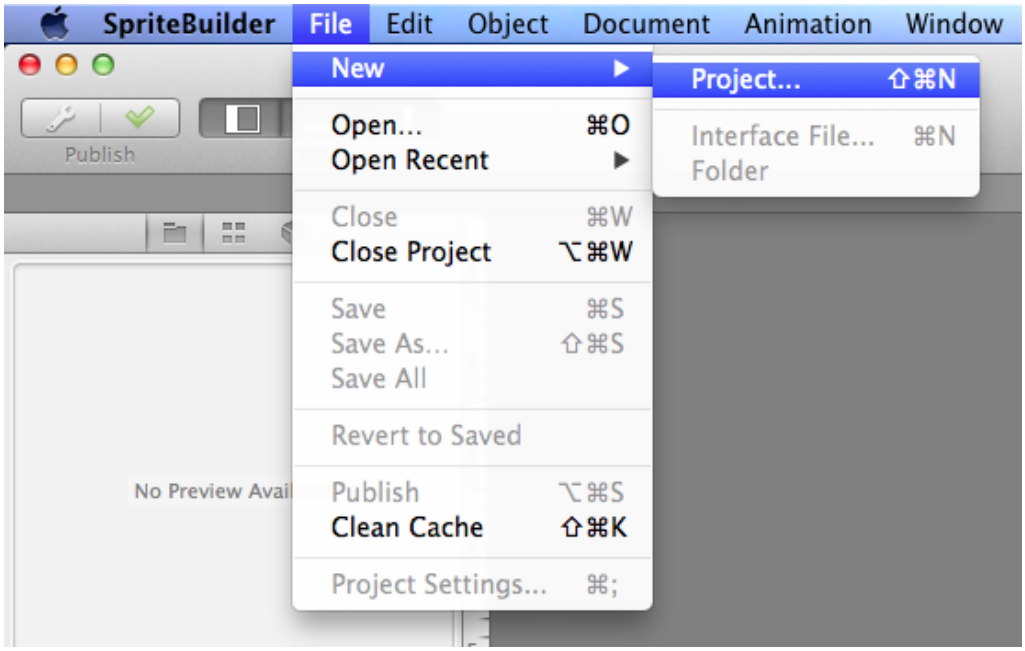
1.4 Introduction to SpriteBuilder

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 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.

If you have never used SpriteBuilder before, it is very important to understand that everything that can be implemented in SpriteBuilder can also be implemented in code. SpriteBuilder is not part of the game engine, it just allows you to configure Cocos2D scenes and nodes in an editor instead of configuring them in code.

1.4.1 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:

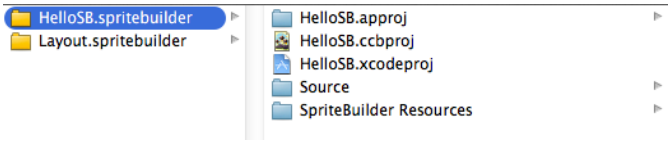


Figure 1.6: SpriteBuilder project folder structure

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

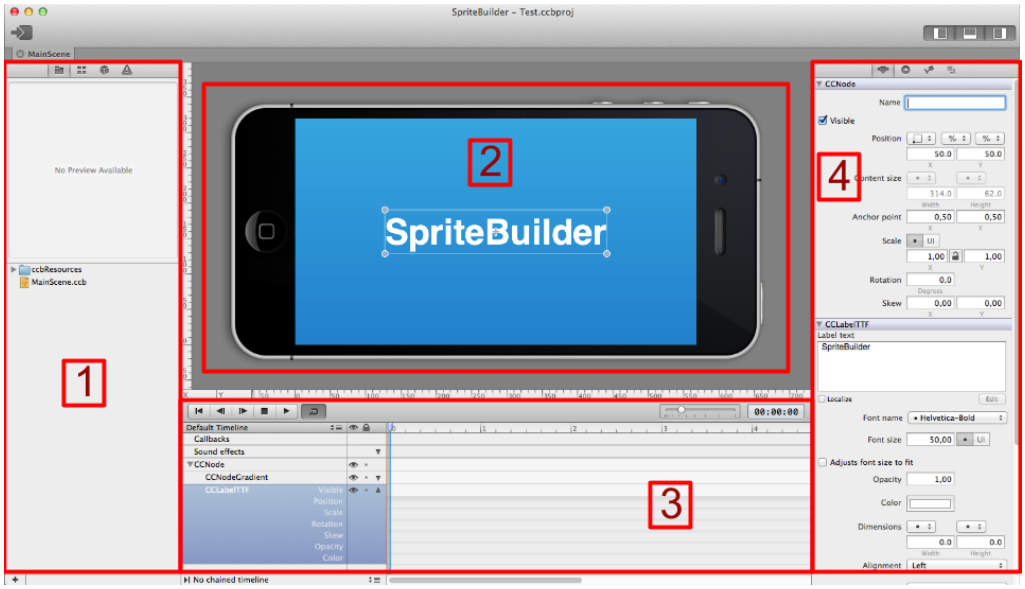
SpriteBuilder will create an Xcode project for every new project you create! The Xcode project will automatically contain the newest version of Cocos2D - very handy.

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.

1.4.2 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.

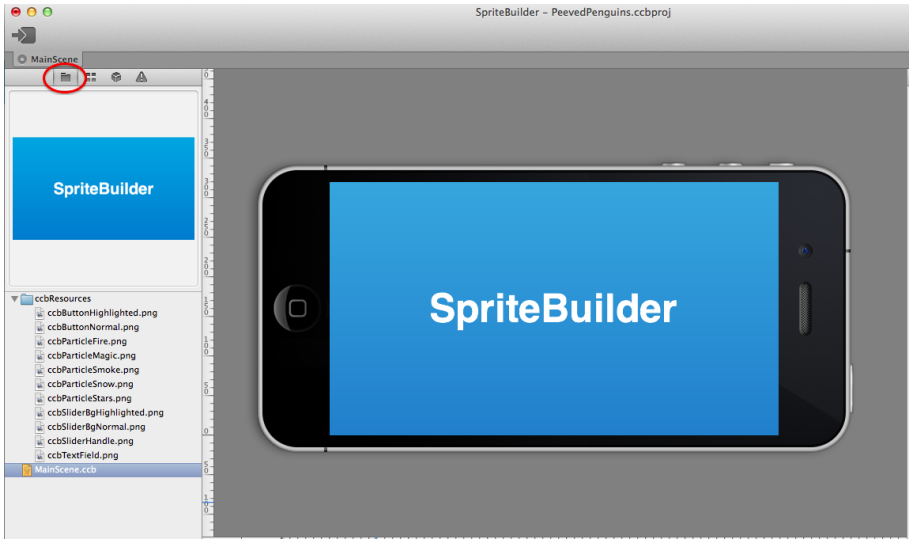
3. *Timeline:* The timeline is used to create animations within SpriteBuilder.

4. *Inspector:* 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

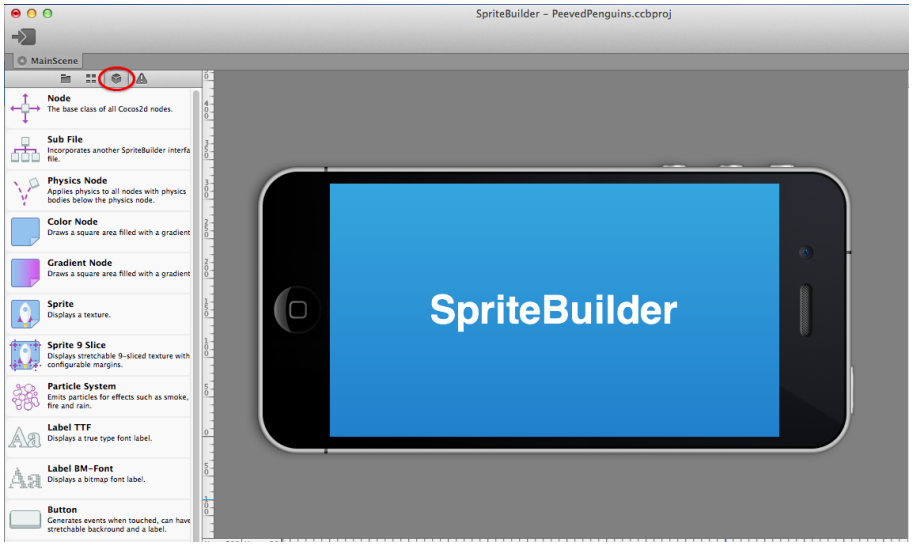
The first tab in the resource/component browser represents the *File View*. It lists all the *.ccb* files and resources that are part of the SpriteBuilder project:



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:

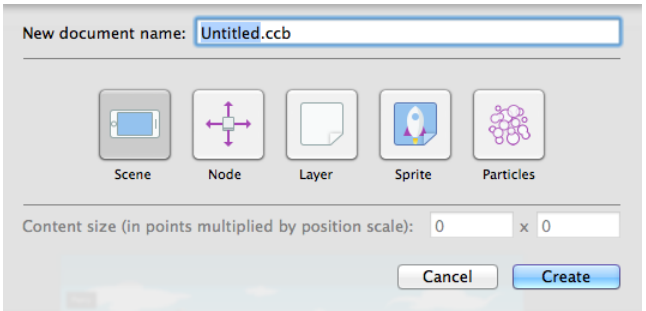
Code Connections

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.

1.4.3 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 document types depending on which type of scene graph you want to create. You get an overview of the available CCB File types when you create a new one, by selecting *New* > *File...* from the *File* menu in SpriteBuilder:



These are the different document types 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.

1.4.4 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:

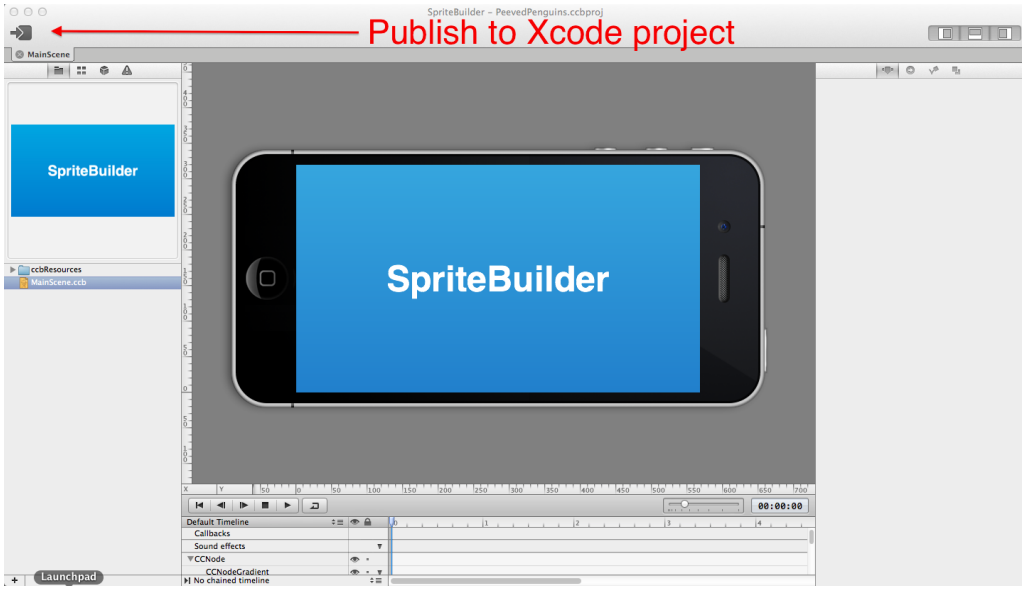


Figure 1.7: Use the publish button to update your Xcode project with the latest changes in your SpriteBuilder project.

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:

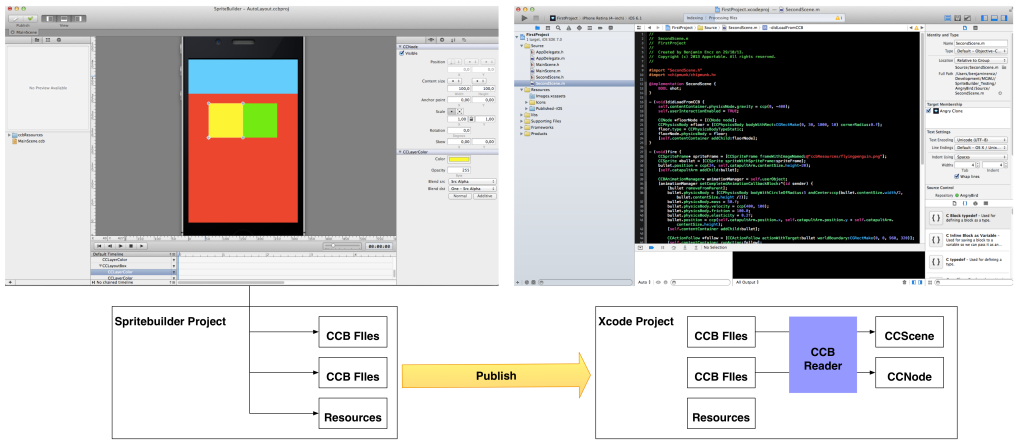


Figure 1.8: SpriteBuilder creates and organizes a Xcode project for you. Adding all the resources and scenes you have created.

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*.

1.4.5 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.

1.5 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 1.9: 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

1.5.1 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:

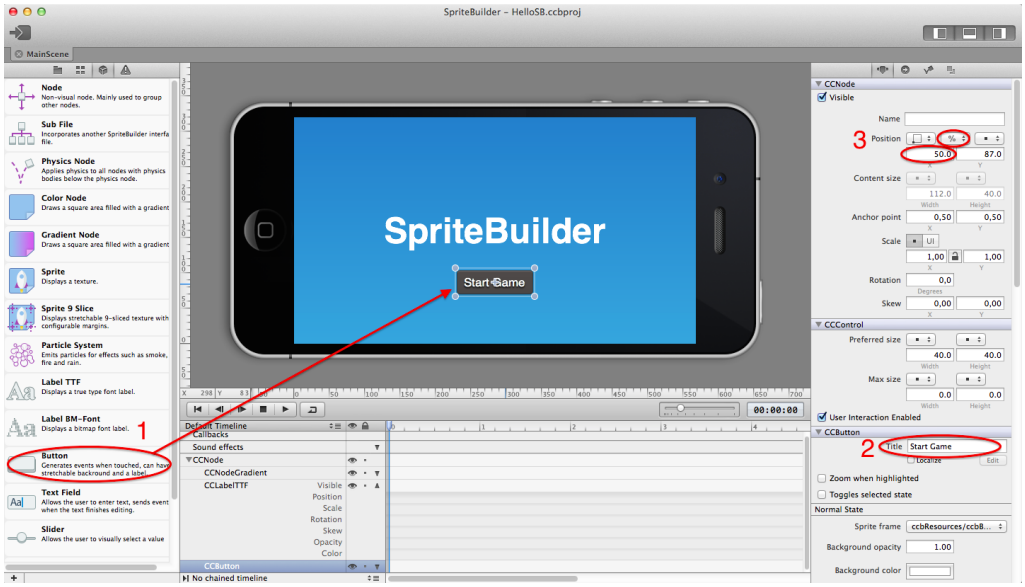


Figure 1.10: The project build throughout this chapter

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 (1.4.5). 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 1.11: Nodes that allow user interaction can use callback methods to connect to the code base

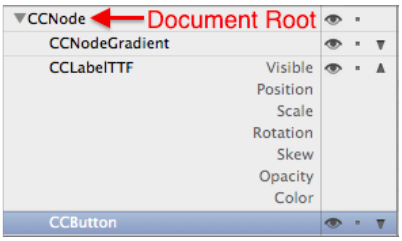
Inside the CCControll section you can see two options called *selector* and *target*. Here you can choose which method (selector) shall be called on which object (target) when this button is tapped by a user. As selector enter `startGame`. As target choose *Document Root*.

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:

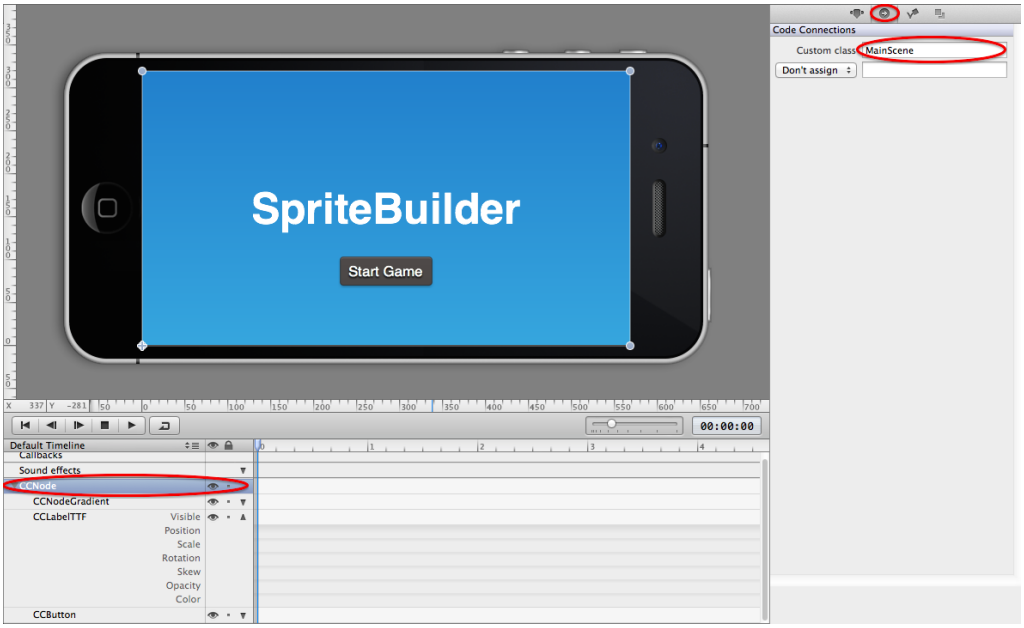
Document Root The document root is the highest node within the current CCB File. The hierarchy of the CCB File is shown in the SpriteBuilder timeline:



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 (1.4.4).

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:

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

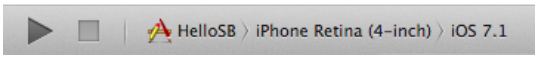
```
}
```

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.

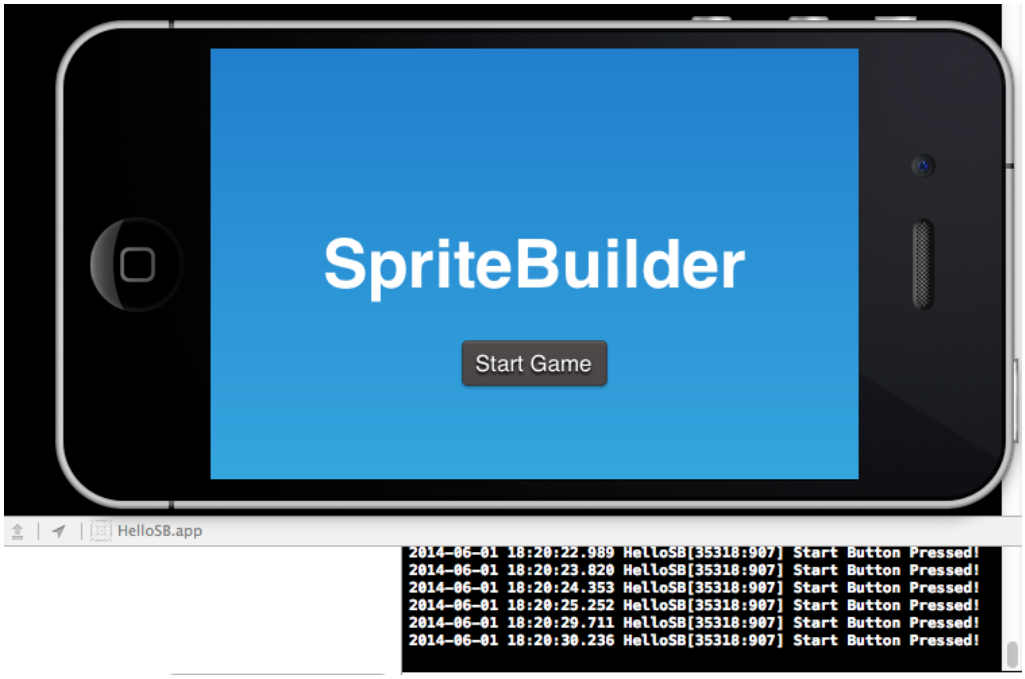
Displaying the console in Xcode

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!

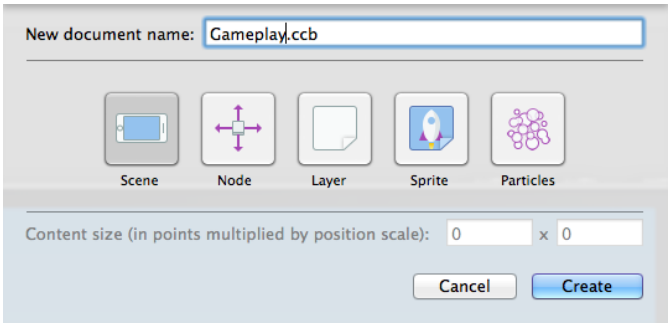
Common Error 1.1

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?

1.5.2 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.

1.5.3 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:

```
- (void)presentScene:(CCScene *)scene;
- (void)presentScene:(CCScene *)scene withTransition:(CCTransition
  ↪ *)transition;
- (void)pushScene:(CCScene*) scene;
- (void)pushScene:(CCScene *)scene withTransition:(CCTransition *)
  ↪ transition;
- (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`.

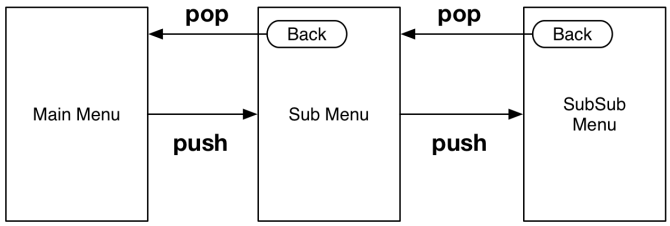
Before we look into using transition effects, let's take a look at the differences between pushing and replacing a scene.

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:

```
[[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:

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

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

```
[[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:

```
[[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:

```
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:

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

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 (1.4.4). 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

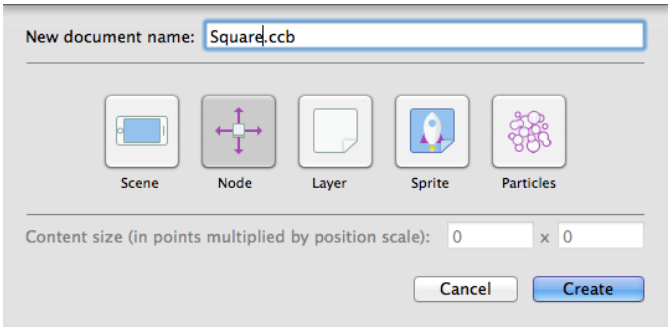
The files with the file extension *.ccb* are in XML-format and are used by SpriteBuilder to store and read information about a scene or node created in SpriteBuilder. When a SpriteBuilder project gets published, SpriteBuilder generates a binary version of each *.ccb* file. The file extension for these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

1.5.4 Implementing the Gameplay

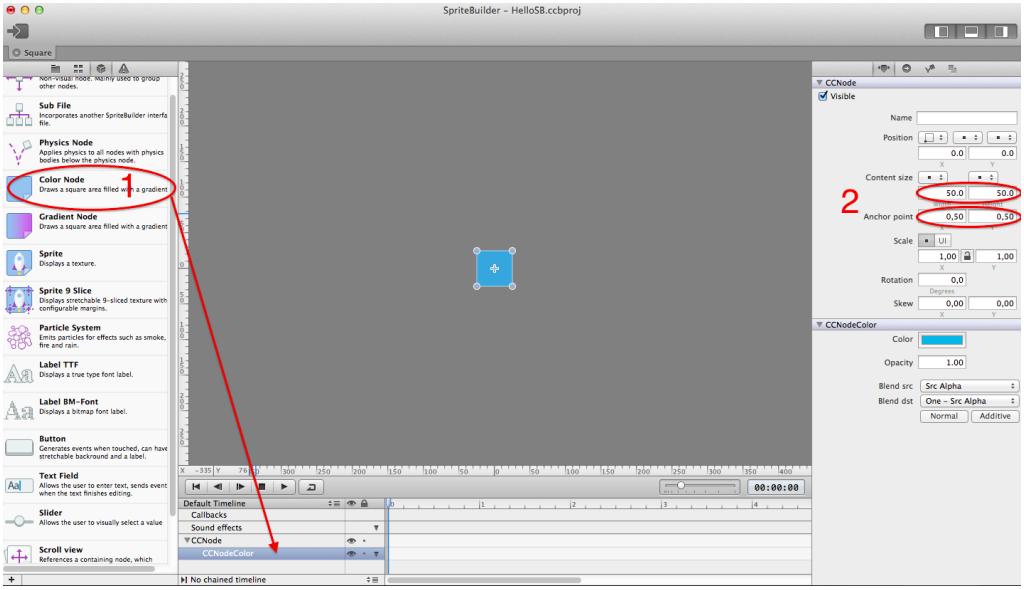
Now it's time to implement the actual gameplay. For our first project we want to keep that fairly simple. Whenever a user touches the screen, we want to add a rotating square with a random color to the gameplay scene. We position the square at the location of the touch.

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*:

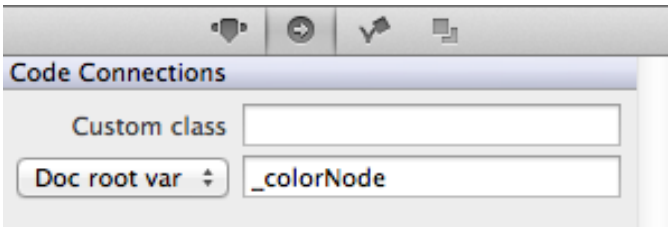


(1) Open the *Node Library* and drag a *Color Node* to the stage or the timeline in order to

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):

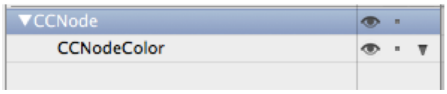


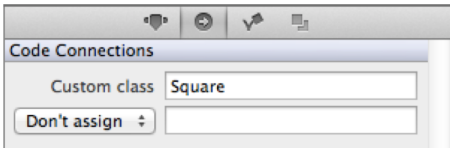
As the variable name (entered in the text field), choose *_colorNode*. As the second option you need to choose the object to which this variable will be assigned to. Just as for callbacks you can choose between the *Document Root* and the *Owner* (1.5.1). We choose the *Document Root*, which means that 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:

```
#import "Square.h"

@implementation Square {
  CCNodeColor *_colorNode;
}

@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 override `onEnter` to pick and apply a random

color for our square as soon as it gets added to the Gameplay scene. It is also important to know that you need to call the `super` implementation if you override any of the `onEnter...` or `onExit...` methods. CCNode has its own implementation of these methods and they are important for the functionality of the framework - if you do not call them this will result in unexpected behaviour throughout your game.

Overriding Cocos2D lifecycle methods

As of Cocos2D 3.1 not calling `super` when overriding one of these lifecycle methods will result in a compiler warning - this can save a lot of debugging time. You are interested in how that can be done? Cocos2D makes use of a nice compiler feature to implement this requirement. You simply need to add an according `__attribute__` to the method definition:

```
-(void) onEnter __attribute__((objc_requires_super));
```

Add this implementation of `onEnter` to *Square.m*:

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

  // arc4random_uniform(N) generates a random number between 0 and
     ↪ N-1
  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];
}
```

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:

```
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:

```
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

After we have set up all the code for the square it's now time to implement the gameplay. In SpriteBuilder we have already created the CCB File *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.

Create the new class just as you have created the `Square` class. In Xcode select *File -> New -> File...* and select *Objective-C* class. Again, this class needs to be a subclass of CCNode since the root node of *Gameplay.ccb* is a CCNode.

Adding Touch Handling to the Gameplay

Now we need to add touch handling to the Gameplay scene. This will be the first time you will add User Interaction to a Cocos2D game!

The Cocos2D touch handling system works on a *per node basis*. This means that every CCNode instance can choose to receive touches or not. You can activate touch handling on any node using the `userInteractionEnabled` property. If `userInteractionEnabled` is set to `YES`, Cocos2D will automatically check if your node is touched by the user. In Cocos2D the front most node receives touch events first.

Each touch in Cocos2D has a lifecycle. That lifecycle consists of four different states and four corresponding methods that are called on your CCNode:

touchBegan: called when a touch begins

touchMoved: called when the touch position of a touch changes

touchEnded: called when a touch ends because the user stops touching the screen

touchCancelled: called when a touch is cancelled because user moves touch outside of the touch area of a node

You can override all of these methods in any CCNode subclass in order to respond to these lifecycle events. For our simple example now, we only need to respond to the `touchBegan:` method.

The Cocos2D Touch System

We will see more complicated use cases of the Cocos2D touch system throughout other examples in this book. If you are interested in more details right away you should read:https://www.makegameswith.us/docs/#!/cocos2d/1.1/user-interaction and https://www.makegameswith.us/gamernews/366/touch-handling-in-cocos2d-30.

Now that you know the basics, let's implement touch handling for the `Gameplay` class. First, we need to enable user interaction. A great place to do this is in the `onEnterTransitionDidFinis` method. Why? If you have an animated transition that presents your gameplay scene you will likely not want to the player to interact with your game before this transition has finished entirely. Add the following method to *Gameplay.m*:

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

    self.userInteractionEnabled = YES;
}
```

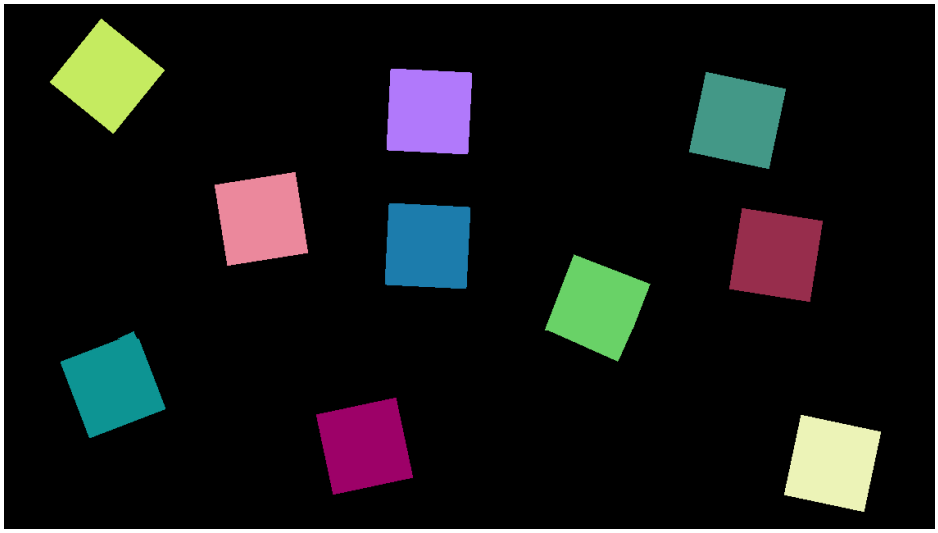
As discussed earlier you need to call the `super` implementation of the lifecycle method you are overriding. In the second step we are setting `userInteractionEnabled` to `YES`. Now Cocos2D knows that this node wants to receive touch events.

In the next step we need to decide to which touch events we want to subscribe and implement the corresponding method. For this simple game we only need to know when a touch begins, because we will add a square to the screen immediately. This means we only need to implement the `touchBegan:` method. Add the following implementation to `Gameplay.m`:

```
- (void)touchBegan:(UITouch *)touch withEvent:(UIEvent *)event {
    CGPoint touchPosition = [touch locationInNode:self];
    CCNode *square = [CCBReader load:@"Square"];
```

```
    [self addChild:square];
    square.position = touchPosition;
}
```

We have now implemented the `touchBegan:` method. It will be called every time the user taps onto the gameplay scene. As one parameter of this method we receive a `UITouch`. The `UITouch` stores all information about the touch. Cocos2D adds a method called `locationInNode:`. This method returns the touch position relative to the provided node. In the first line we call this method to receive the touch location within the gameplay scene (referred to by `self`). In the next line we load one `Square` node using the `CCBReader`. Then we add that loaded square as a child to the gameplay scene. The `addChild:` method of CCNode will add the square to the node hierarchy of the gameplay scene. As soon as node becomes part of the node hierarchy of the currently active scene it will be displayed on the screen. Finally we choose a position for the square. We provide the touch position that we determined in the first line - this way the square will be spawned exactly at the position touched by the player. Now it's time to run your project again. Once the game started, select the *Start* button to go to the gameplay scene then click onto the screen multiple times to simulate touches. Every time you simulate a touch you should see a new square spawn at the touch position:



Well done! You have come a very long way from the blank project to a first simple game that uses scene transitions, actions and the Cocos2D touch system. But this is only the very beginning. In the next chapter we will start working on a much more complex game that will teach you many more important concepts of SpriteBuilder and Cocos2D.

1.6 Exercises

Now it's time for some exercises. Note that you can't find all the knowledge for these exercises in this chapter - for some exercise you may will have to use the help of the internet (that is another exercise right there).

1.0 Add a label to the top right corner of the gameplay scene. This label shall display the amount of squares that are currently on the screen.

1.1 Make each square disappear after it has been on the screen for 2 seconds.

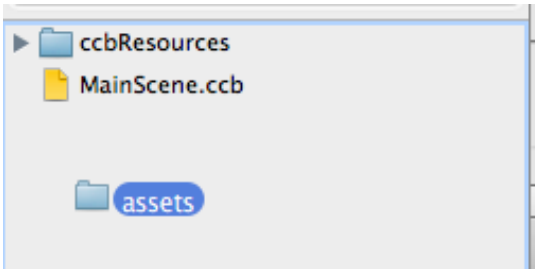
2 A Game with Assets in SpriteBuilder

Graphics and Sounds are the essence of every good game. In the first chapter you have learned the very basics of SpriteBuilder and Cocos2D by building a game that only uses plain colored shapes. In this chapter you will learn how SpriteBuilder helps you to integrate assets into your game. Learning by example is the most fun, so we will build a small game throughout this chapter that uses all aspects of asset management.

2.1 Adding Assets to a SpriteBuilder project

Start by creating a new SpriteBuilder project for this game. I have called the project *FallingObjects*.

Now you should download the assets from https://dl.dropboxusercontent.com/u/13528538/SpriteBuilderBook/assets.zip. Once the download completes you add the assets to the project by dragging the entire folder into the left *File View* in the left panel of SpriteBuilder:



Great, now we have some assets to use in our game. Now is a good time to take a close look at how SpriteBuilder and Cocos2D handle assets.

2.2 Asset Handling in SpriteBuilder and Cocos2D

One of the main goals of SpriteBuilder is to make game development for multiple device types as easy as possible. This means that games should automatically be able to run on differently sized iPhones, iPads and Android Devices. Since each of these devices has a different resolution Cocos2D and SpriteBuilder allow developers to use different assets to target them. SpriteBuilder provides four different resolution categories:

phone resolution for non-retina iPhone and Android devices

phone-hd retina resolution for iPhone and Android

tablet resolution for non-retina iPad and Android tablets

tablet-hd resolution for retina iPad and Android tablets

Luckily using SpriteBuilder, there is no need to provide four resolutions for each asset thanks to **automatic downscaling**. Per default SpriteBuilder assumes that all assets added to a project are provided in *tablet-hd* resolution, then SpriteBuilder generates downscaled images for the other resolutions. While you can provide different images for

four targets, SpriteBuilder only knows three resolution types:

1x non-retina images

2x retina images

4x double sized retina images

By default SpriteBuilder maps these resolution types to the different devices in a way that every asset has the same size (in relation to the screen size) on every device. This means games running on an iPad will look very similar to games running on an iPhone, except that the have a slightly different aspect ratio. Here is an example from on of our tutorials showing what a game looks like on different device types:

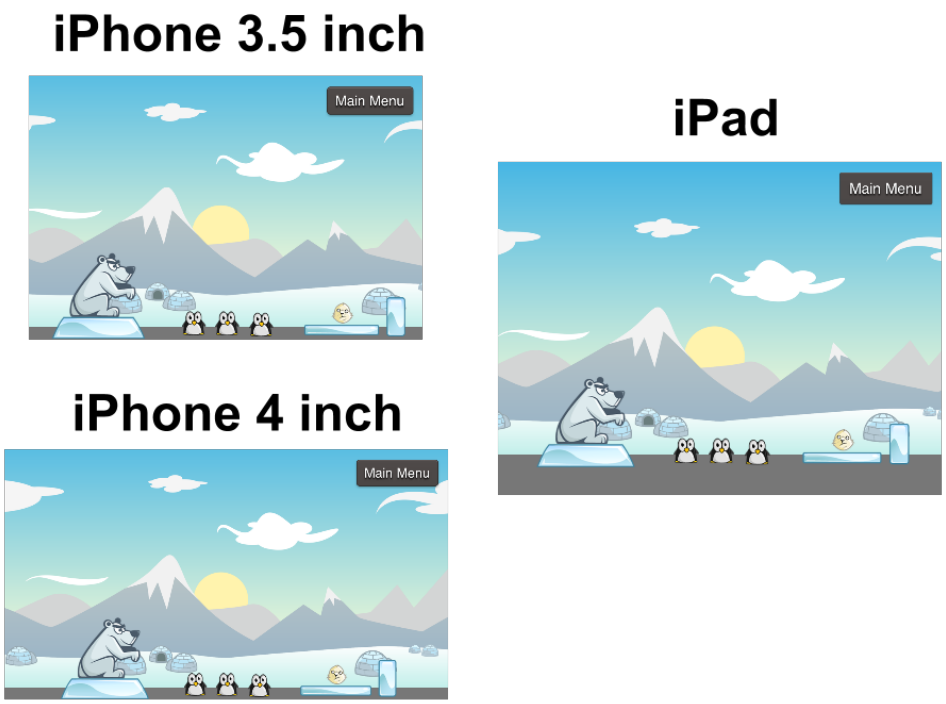
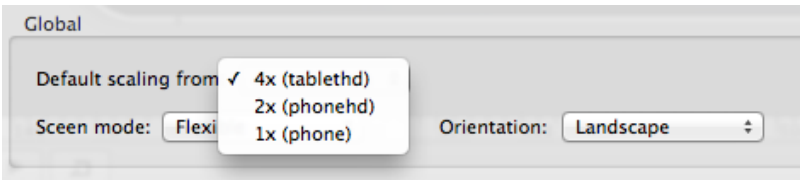


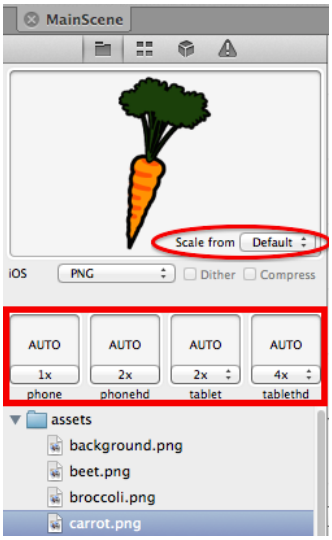
Figure 2.1: From our tutorial *Dynamic Layouts with SpriteBuilder and Cocos2D*

Let's take a look at where all the settings I mentioned are visible in the SpriteBuilder UI. When you open the project settings (*File -> Project Settings...*) you can see the available downscaling options:



This setting defines the *global* downscaling option. Individual assets can define their own behaviour, thereby overriding this global setting. To make support of multiple devices as easy as possible you should provide all of your assets in *4x* resolution and keep this default setting.

When you select an individual asset from the File View you can see different downscaling settings:



Each asset can have its own *Scale from* setting. *Default* means that the global project setting applies (in this project: downscaling from *4x*). Additionally you can see how the different resolution types are mapped to the different device types. Here you could for example choose that a certain asset should not be scaled up on retina tablets by choosing a *2x* resolution for *tablethd* - however, the default settings work best most of the time.

For future reference, this is an example that shows you which sizes your assets will have on the different devices by default:

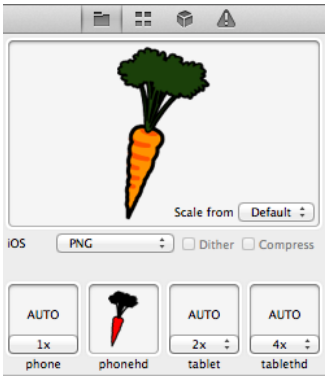
Device	Default Resolution Type	Size on Screen (points)	Size in Pixels
iPhone	1x	50x50	50x50
iPhone Retina	2x	50x50	100x100
iPad	2x	100x100	100x100
iPad Retina	4x	100x100	200x200

You can see, if you have a size in mind for a certain asset on an iPhone you should provide the asset in four times larger resolution.

A last interesting case are background images that you want to work for all 4 resolutions. A solution is discussed in the Q&A section (**??**).

Different images for different devices

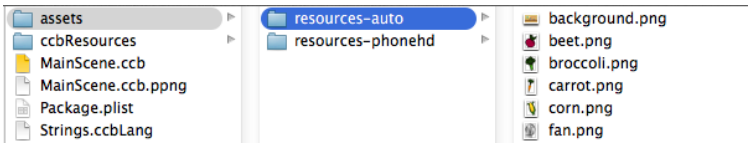
You can not only change the scaling option for an asset on different devices, you can even use an entirely different image for a certain resolution. You can do that by dragging an image **that is currently not part of the SpriteBuilder project** from Finder into one of the four boxes below the asset preview:



Note that images you add this way will be displayed in exactly the size you have added them and will not be downscaled.

Behind the scenes

If you are interested in how SpriteBuilder and Cocos2D organize assets you can take a look at the resource package (*/Packages/SpriteBuilder Resources.sbpack*) by right-clicking and selecting *Show Package Contents*:



You will see that SpriteBuilder groups images inside the assets folder into a *resources-auto* folder, all images in that folder are subject to automatic downscaling. If you explicitly add images for a certain resolution as shown with the carrot in the above example, a new folder for that resolution (e.g. *resources-phonehd*) is created.

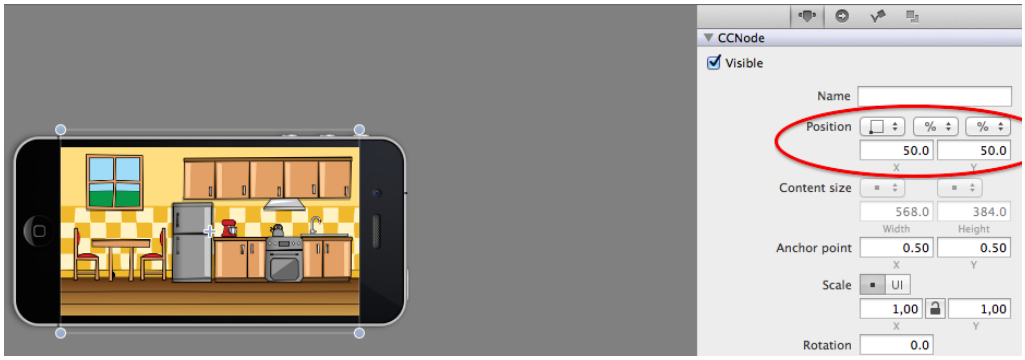
In Cocos2D a class called `CCFileUtils` is responsible for loading the correct images for the current device during runtime. SpriteBuilder uses a special configuration of `CCFileUtils` that is set up in `[CCBReader configureCCFileUtils`

2.3 Adding the background image

Now that we have a basic understanding of how asset management works, lets get started working on our game. For now our game will only consist of one scene, so we can start working in the *MainScene.ccb* that is part of the SpriteBuilder template. First, remove the existing content so that we can start with a blank scene. Now we can add the background image. To add a sprite to a scene we can simply drag the asset to the stage, SpriteBuilder will automatically create an instance of CCSprite. Add the *background.png*

image to the stage.

How should we position this background image? We already have briefly discussed the SpriteBuilder positioning system (1.5.1). Using the positioning system correctly is especially important when we create games for phones and tablets - which we always should try to do. In most cases - like in this game it is the best to center the background image. That way phones and tablets will display a very similar portion of the background image. You can center the image by choosing a *normalized* position type (*in % of parent container*) and setting the position to (50, 50).



You can preview what your game will look like on different device types directly in SpriteBuilder, without the need to compile and run the game - you should do this as often as possible! The option is available from the menu *Document -> Resolution*. You can also use the CMD+1, CMD+2 and CMD+3 shortcuts. This feature will allow you to preview the game on a 3.5-inch iPhone, a 4-inch iPhone and an iPad.

2.4 Create falling objects

Now let's dive into the implementation of the actual game. The next step should be adding falling objects. Our game will have two categories of objects, ones that should be

caught (food) and ones that shouldn't (electronic devices).

In total we have over ten different objects in our game but these just exist as visual enhancement, actually we are only differing between two types of objects. One way to implement the falling objects would be creating a CCB File for each object but that isn't actually necessary for this game. We need to create all falling objects dynamically, while the game is running, and for each object we only need to store if it should be caught or not. That can be best accomplished by a subclass of CCSprite that we create in code. This way you will also learn how to use assets you added in SpriteBuilder to create CCSprites in code. Open the Xcode project of the game to get started.

2.4.1 Create a falling object class

In general we have two ways to differentiate objects a player should catch and ones he shouldn't catch. We could:

- Create two distinct subclasses of CCSprite, each representing one type of object
- Only have one subclass and add a `type` property to it

Since our falling objects won't have any type-specific behaviour, creating two distinct subclasses is not necessary in this case. Instead, as of now, one subclass with a `type` property is the better solution.

Create a new class called `FallingObject` and make it a subclass of CCSprite. The best way to represent different types in Swift is using enumerations. Add this enum definition to *FallingObject.swift*, inside of the `class` block:

```
enum FallingObjectType: Int {
  case Good
  case Bad
}
```

Swift supports multiple types of enumerations and enumeration values. In the example above we are creating an enumeration with *Raw Values*. When we use an enumeration with raw values we need to assign a type as part of the enum definition, as shown in the first line (`enum FallingObjectType: Int`). Each enum value will be mapped to one value of this provided type. In the example shown above, the raw value for `FallingObjectType.Good` will be 0 and the value for `FallingObjectType.Bad` will be 1. Thanks to auto-increment we do not need to map entries to numbers explicitly. Associating enum values with raw values is optional, throughout this chapter you will see why this feature is useful to us.

Enumerations in Swift

You can read everything about the different ways of creating enumerations in Swift in the official documentation (https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Enumerations.html).

Additionally we add an instance variable to store the object type. Later we will add an initializer to set the type of the falling object. For now your class should look as follows:

```
import Foundation

class FallingObject: CCSprite {

  enum FallingObjectType: Int {
        case Good
        case Bad
  }

  private(set) var type:FallingObjectType

}
```

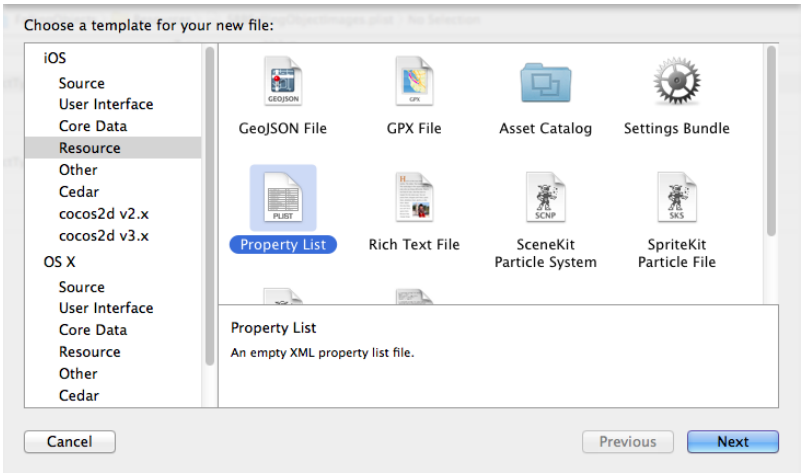
We define the instance variable to be *readonly* because we will not support changing the type of a falling object after it has been created. In Swift we can define variables as *readonly* by marking the *setter* as private.

Note that the code will not compile at this point. Since `type` isn't declared as an optional value, Swift requires us to provide an initializer that sets this value. We will fix this in the next chapter.

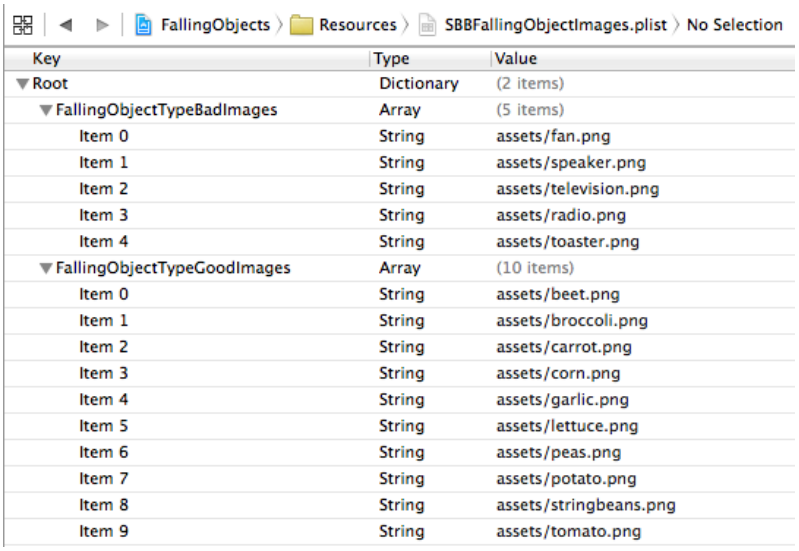
2.4.2 Choose an asset for a falling object

We want the game to spawn entirely random falling objects. As you remember we have a couple of assets for both types of objects. Whenever we spawn an object we will need to choose a random asset, based on the object type. A good place to implement this functionality is directly in the `FallingObject` class. We can provide a custom initializer that allows the class to be initalized with an object type. When this initializer gets called we choose a random asset and apply it as a texture to the `FallingObject`.

How can we create a list of images for objects that should be caught (e.g. food) and ones that shouldn't (e.g. radios)? One way of implementing this would be creating two arrays, one for each object type, and storing filenames for different assets in these arrays. As good game developers however, we try to keep game content and code as separated as possible. That makes it easier to update the list of assets later on and it keeps our codebase small and well structured. So instead of creating these arrays in code we could use some sort of resources file that stores information on available images. A very common format for storing such type of information in Cocoa Touch is a *plist* (Property List). You can create a *plist* by selecting *File -> New -> File...* from Xcode's menu. Then you need to select *Resource* from the left panel and choose *Property List* on the right:



As the name choose *FallingObjectImages*. Now fill the *plist* with two arrays that contain the filenames of the assets that we added in SpriteBuilder. When accessing assets from a SpriteBuilder project you always need to include folder names. Instead of referencing the tomato with `tomato.png` you need to use `assets/tomato.png` since the asset is in the `assets` folder in the SpriteBuilder project:



Key	Type	Value
▼ Root	Dictionary	(2 items)
▼ FallingObjectTypeBadImages	Array	(5 items)
Item 0	String	assets/fan.png
Item 1	String	assets/speaker.png
Item 2	String	assets/television.png
Item 3	String	assets/radio.png
Item 4	String	assets/toaster.png
▼ FallingObjectTypeGoodImages	Array	(10 items)
Item 0	String	assets/beet.png
Item 1	String	assets/broccoli.png
Item 2	String	assets/carrot.png
Item 3	String	assets/corn.png
Item 4	String	assets/garlic.png
Item 5	String	assets/lettuce.png
Item 6	String	assets/peas.png
Item 7	String	assets/potato.png
Item 8	String	assets/stringbeans.png
Item 9	String	assets/tomato.png

Now we have a list of all asset names grouped into the two object type categories. Time to implement the `FallingObject` class.

When a falling object is initialized we want to choose random image from the *plist* that we just created. The first step is loading the *plist* in code. Luckily *plists* consist of Dictionaries, Arrays, Strings, etc. and all of these types exist in Swift as well - there are some very convenient methods to load *plists*.

During each playing session we are going to create hundreds of falling objects. Since the images that represent these objects won't change it would be a waste of resources to load the *plist* every time we create a new instance of `FallingObject`. Instead we should only load it once and then keep a reference to it for future use. A good way of doing this is using a class constant to store the *plist* reference once it is loaded.

Class variables in Swift

While class constants and non-computed class variables are already specified as part of the Swift language they aren't implemented in the Swift compiler yet (as of Xcode version 6.1.1), therefore we need to use a workaround.

As described in the detail box above, Swift does not yet support class constants, therefore we need to use a workaround. Here is the code that loads the plist and stores the content in a class variable. You should place it within the class definition of `FallingObject` (no worries, we will discuss the code in detail right away):

```
private class var imageNames:ImageNames {
  struct ClassConstantWrapper {
    static let instance = ImageNames()
  }
  return ClassConstantWrapper.instance
}

private struct ImageNames {
  var good: [String]
  var bad: [String]

  init () {
    let path = NSBundle.mainBundle().pathForResource("
       ↪ FallingObjectImages", ofType: "plist")!
    let imageDictionary:Dictionary = NSDictionary(contentsOfFile:
       ↪  path)!
    good = imageDictionary["FallingObjectTypeGoodImages"] as [
       ↪ String]
    bad = imageDictionary["FallingObjectTypeBadImages"] as [
       ↪ String]
  }
}
```

First, we are defining a private class variable. Currently Swift only supports computed class variables, that means we need to provide a getter that will return a generated value and does not rely on a class storage variable. We provide that getter through the closure that is placed directly after the variable declaration. Whenever the class variable is accessed this closure is called. Inside of this closure we place the workaround mentioned earlier. We declare a struct called `ClassConstantWrapper` with a static constant called `instance`. This static constant is initialized with an instance of the structure `ImageNames` which we declare a few lines later. Since `instance` is a static constant, the expression will only be evaluated once and only one instance of `ImageNames` will ever be created, independently of how often the class variable is accessed.

Inside of the `ImageNames` struct, the actual work happens. First we declare two array variables that store strings. They store the filenames of the *good* object assets and the *bad* object assets. Inside of the initializer we fill these variables.

The initializer takes no parameters. The first line gets the path of the *FallingObjectImages.plist* file. Then we use a convenience initializer on `NSDictionary` called `contentsOfFile:`. That initializer creates a dictionary from a provided plist. This only works because the root element of our plist is a dictionary! Earlier we've set up our plist to contain two arrays of strings, *FallingObjectTypeGoodImages* and *FallingObjectTypeBadImages* below the root element. We can now extract these two arrays from our dictionary and assign them to the `good` variable and `bad` variable, respectively. During this assignment we need to cast the arrays into `[String]` arrays using the `as` operator. This is necessary because we are receiving an Objective-C Array (`NSArray`) that cannot store type information about the elements it contains. We however know that this array only contains strings so we want to treat it as a `[String]` array in Swift.

Due to the necessary workaround this is quite a bit of code, but things will improve as Swift matures.

Now we have successfully set up a class constant that will load the required image names

when it is accessed the fist time and store these images on a class level. That will avoid reloading the plist for every instance of `FallingObject` that we create.

Now that we have access to the image names we can implement the actual initializer of `FallingObject`. We need to:

- Pick a random image based on the object type
- Call the designated initializer of our superclass CCSprite

This is what the initializer should look like:

```
init(type: FallingObjectType) {
  self.type = type

  var imageName:String? = nil

  if (type == .Good) {
    let randomIndex = randomInteger(FallingObject.imageNames.good
       ↪ .count)
    imageName = FallingObject.imageNames.good[randomIndex]
  } else if (type == .Bad) {
    let randomIndex = randomInteger(FallingObject.imageNames.bad.
       ↪ count)
    imageName = FallingObject.imageNames.bad[randomIndex]
  }

  let spriteFrame = CCSpriteFrame.frameWithImageNamed(imageName)
     ↪ as CCSpriteFrame
  super.init(texture: spriteFrame.texture, rect: spriteFrame.rect
     ↪ , rotated: false)

  anchorPoint = ccp(0,0)
}
```

We start of by storing the type we receive in an instance variable. Then we create a local variable called `imageName` that we will use to load the correct texture for this object type. Next, we check whether we are initializing a good or a bad object. In each case we generate a random number, using our helper function `randomInteger`, that picks one image name from the set of available image names in the arrays. Next, we need to call an initializer of our superclass CCSprite. Here we run into another limitation of the current version of Swift: we can only call *designated* initializers of our superclass, but not any *convenience* initializers. This means instead of calling:

```
CCSprite(imageNamed: String!)
```

We have to call the designated initializer:

```
CCSprite(texture: CCTexture!, rect: CGRect, rotated: Bool)
```

This unfortunately means some extra code! We create a `CCSpriteFrame` with the image name that we have selected from one of our lists, then we use that sprite frame to call CCSprite's designated initializer. Finally, we set the anchor point of our object to the bottom left corner, that will make it easier to determine the spawn position later on.

That's all we need in order to create a `FallingObject`! Now we can move on and spawn some objects.

2.5 Spawn falling objects

Now it's time to implement one of the core mechanics of the game: Spawning objects and make them fall from the top of the screen to the bottom. We are going to implement this in *MainScene.swift*.

We will spawn objects after a certain time period. The spawning objects will start at the top of the screen and fall to the bottom. To not use an increasing amount of memory we

will need to take care of removing objects that have fallen below the bottom edge of the screen. A good way to do this is creating an array to store all the objects we spawn. Swift lets us define a private instance variable and initialize the array stored in it in a single line of code:

```
class MainScene: CCNode {

  private var fallingObjects = [FallingObject]()

}
```

We also need to define a falling speed and an interval at which we want to spawn objects. A good way to do this is by defining constants - we want to avoid to have these numbers all over our code. Add the two constants so that your class definition looks as following:

```
class MainScene: CCNode {

  private var fallingObjects = [FallingObject]()

  private let fallingSpeed = 100.0
  private let spawnFrequency = 0.5
}
```

We are going to spawn falling objects with the frequency that we have defined in the constant `spawnFrequency`. Through the `CCNode` class Cocos2D provides convenient methods for scheduling repeating events without the need to instantiate a timer. We schedule that timer in the `onEnterTransitionDidFinish` method:

```
  override func onEnterTransitionDidFinish() {
    super.onEnterTransitionDidFinish()

    // spawn objects with defined frequency
    schedule("spawnObject", interval: spawnFrequency)
  }
```

Remember, *onEnterTransitionDidFinish* is called as soon as the presentation transition is completed and the current scene is fully visible. This is when we want to kick of the spawning mechanism. All we need to provide is a *selector*, which simply means a method name as a string, and a frequency at which it shall be called. Now, as soon as the `MainScene` is presented on the stage, the `spawnObject` method will be called twice a second. To complete the spawning functionality we will have to implement the `spawnObject` method and additionally move the spawned objects from the top of the screen to the bottom.

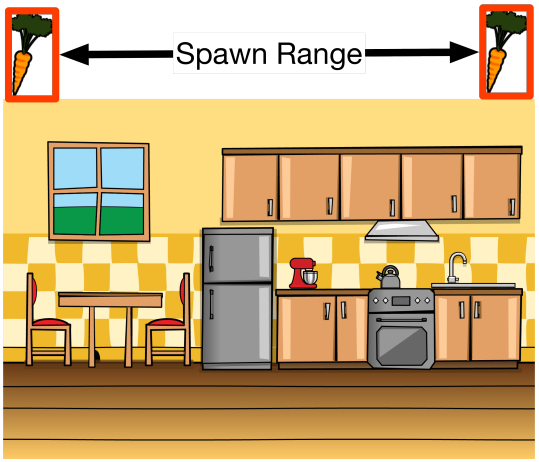
We want to randomly spawn either positive objects that should be caught or negative ones that should not, for that we will generate a random number. Based on the random number we will generate a falling object. We will place that spawning object just above the screen at a random X position. Here is how we can implement that:

```
func spawnObject() {
  let randomNumber = randomInteger(2)

  let fallingObjectType = FallingObject.FallingObjectType(
     ↪ rawValue:randomNumber)!
  let fallingObject = FallingObject(type:fallingObjectType)

  // add all spawning objects to an array
  fallingObjects.append(fallingObject)

  // spawn all objects at top of screen and at a random x
     ↪ position within scene bounds
  let xSpawnRange = Int(contentSizeInPoints.width - CGRectGetMaxX
     ↪ (fallingObject.boundingBox()))
  let spawnPosition = ccp(CGFloat(randomInteger(xSpawnRange)),
     ↪ contentSizeInPoints.height)
  fallingObject.position = spawnPosition

  addChild(fallingObject)
```

```
}
```

As you can see here, defining the `FallingObjectType` enumeration with a raw integer value allows us to create a type directly from a number. Here that comes very handy. Besides that the spawning code is not too exciting, it primarily consists of a little math and type conversions.

This is a schematic diagram of where we are spawning objects with the code shown above:



Our current version of the game spawns new objects twice a second at the top of the screen and at a random X position. However, these objects don't move yet so you won't be able to see them falling down. Let's implement the falling code to complete the entire spawning functionality!

2.6 Move falling objects

The last step for this chapter will be moving the objects we are spawning to the bottom of the screen. While building your very first SpriteBuilder game you have learned to use the Cocos2D action system to move nodes. The action system lets us describe changes over time, e.g. *move 100 points to the right over 2 seconds*. Another option to move nodes that we haven't discussed yet is using the Cocos2D *update loop*.

2.6.1 Update Loop

When we build games with Cocos2D the engine attempts to render 60 frames a second and draws theses rendered frames to the screen of the device. When we move objects between rendering frames, they will appear as moving objects to the user. Cocos2D provides a method that is called directly before a frame is rendered, the `update` method.

The `update` method is defined as part of the `CCSchedulerTarget` protocol. `CCNode` implements this protocol, that means any subclass of `CCNode` can override the method. This is the signature of the `update` method:

```
func update(delta: CCTime)
```

We receive one parameter called `delta` from the Cocos2D framework. The delta parameter contains the milliseconds since the `update` method was called last. Most of the time this value will be *0.0167* milliseconds, which is 1/60 of a second. If the performance of our game drops below 60 FPS this value will be higher, because the time between two rendered frames will increase. If we want our objects to move at the same speed, independent of the current framerate, we can use this delta parameter to calculate how far we need to move nodes between two given frames.

Enough of the theory - let's implement our update method, that will help you understand

the details.

2.6.2 Implementing the update method

Here is what we want to do in the update method:

- Iterate over all falling objects
- For each object check if it is within the screen boundaries
- If the object is outside of the screen, remove it
- If the object is inside of the screen boundary, let it fall to the bottom

And here is how we can implement it:

```
override func update(delta: CCTime) {
  // use classic for loop so that we can remove objects while
     ↪ iterating over the array
  for (var i = 0; i < fallingObjects.count; i++) {
    let fallingObject = fallingObjects[i]

    // check if falling object is below the screen boundary
    if (CGRectGetMaxY(fallingObject.boundingBox()) <
       ↪ CGRectGetMinY(boundingBox())) {
      // if object is below screen, remove it
      fallingObject.removeFromParent()
      fallingObjects.removeAtIndex(i)
    } else {
      // else, let the object fall with a constant speed
      fallingObject.position = ccp(
        fallingObject.position.x,
        fallingObject.position.y - CGFloat(fallingSpeed * delta)
```

```
            )
        }
    }
}
```

The interesting aspect of the code snippet above is how we check if the falling object is out of bounds and how we move the falling object. Note that we are using the `CGRectGetMaxY` and `CGRectGetMinY` functions to determine the top and the bottom of the bounding boxes of the falling object and the gameplay scene. The `CGRectGetMaxY` function returns the largest Y value of the bounding box. Using these functions is preferred over accessing values directly (e.g. `fallingObject.boundingBox.origin.y`) because they also work for rectangles with negative sizes.

If we detect that the top border of the falling object is below the bottom border of the screen, we remove the falling object from the scene.

If the falling object is within the screen boundary we move it to the bottom with the constant speed that we defined earlier.

Now the falling mechanic is entirely implemented! In the next and last subchapter you will learn how to add sound assets to the game.

Update vs. Fixed Update

This chapter discusses the `update:` method of Cocos2D in detail. Cocos2D provides a second similar method called `fixedUpdate:`. Unlike the `update:` method, the `fixedUpdate:` method is **guaranteed** to be called at a specified interval (per default 1/60) and is not dependent on the framerate the game is running at. The physics engine integrated in Cocos2D uses the `fixedUpdate:` method to perform all of its calculations. For you as developer that means that you should implement code that changes physical attributes in the `fixedUpdate:` method and **not** in the `update:` method. We will discuss the physics engine of Cocos2D in later chapters in detail. A nice blog post about the `fixedUpdate` method is available here: http://kirillmuzykov.com/update-vs-fixedupdate-in-cocos2d/.

2.7 Adding sound effects

The goal of this chapter is for you to learn how to use assets with SpriteBuilder and Cocos2D. Obviously images are the most important assets in games, but sound effects also play a big role in creating games that your players enjoy. In this section you will learn how to add a sound effect that gets played whenever one of the falling objects drops out of the screen.

Start by downloading the sound file here: https://dl.dropboxusercontent.com/u/13528538/SpriteBuilderBook/drop.wav.

All sound files need to be added to your SpriteBuilder project in the *Wave* format. SpriteBuilder will then generate compressed versions of that sound in different formats for the iOS and the Android app. You can add the sound effect - just like any other asset - by dragging it from Finder to the resource pane (in the bottom left) of SpriteBuilder.

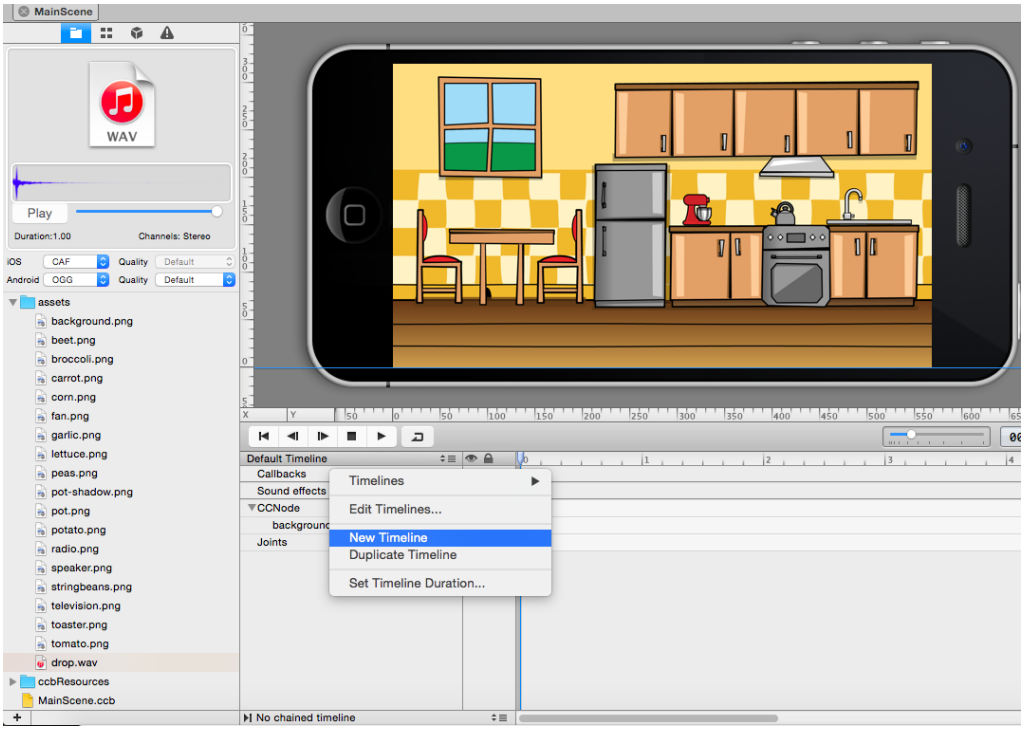


Figure 2.2: WAV files can be added by dragging them to the resource pane

There are different ways to play sound effects added to your SpriteBuilder project: you can add a sound effect to a SpriteBuilder timeline or you can play a sound effect directly from code. We will first look at the timeline approach, implementing the code approach will be an exercise at the end of this chapter. Before we set up the sound effect I want to give you a basic introduction to the timeline feature of SpriteBuilder since it is one of the most important ones!

2.7.1 SpriteBuilder's timeline feature

The SpriteBuilder timeline is a tool that allows developers to create animations and sequences of sound effects without writing code. Every CCB File has one *Default Timeline* associated with it as soon as it is created. However, a CCB File can, and often will, have multiple different timelines. Each timeline is a sequence of sound effects, callbacks and most importantly keyframes.

...

Adding the sound effect to a timeline

Start by creating a new timeline for the sound effect as show in figure 2.2. Once the timeline is created you can drag the sound effect from the asset library in the left panel to the *Sound effects* row of SpriteBuilder's timeline (the second row from the top).

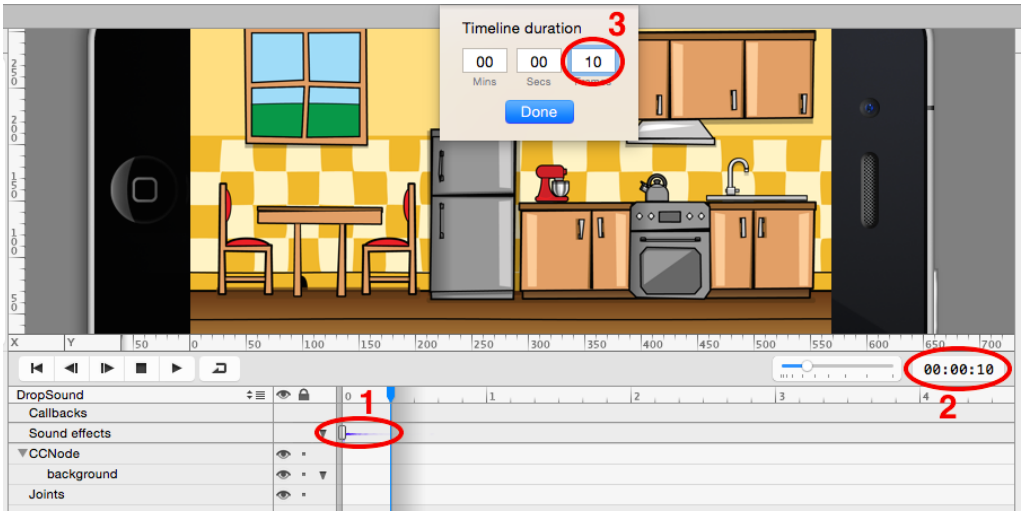


Figure 2.3: (1) Audio file added to timeline
(2) Button to change timeline duration
(3) Timeline duration dialog

When the sound is added to the timeline it will be displayed in wave form. You should adjust the duration of the timeline to match the duration of the sound effect. Figure 2.3 shows how you can change the duration. You should set it to 10 frames.

Now the sound is ready to play! The last step is assigning a unique name to this timeline which we can reference from code. You can rename a timeline by either choosing *Animation -> Edit Timelines...* or selecting the dropdown button next to the timeline name:

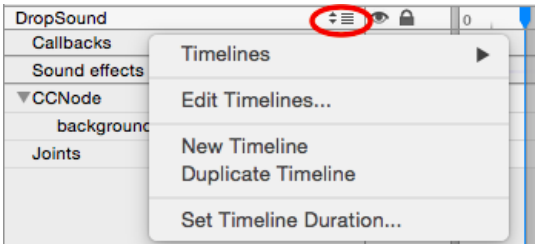


Figure 2.4: *Edit Timelines...* allows you to change the name of a timeline

I have chosen *DropSound* for the name of this timeline. Now everything is set up and you can hit the publish button in SpriteBuilder.

2.7.2 Triggering a Sound Effect

Now we have set up the sound effect in SpriteBuilder and published the project, all that is left to do is to open the Xcode project and play the sound effect as soon as an object falls below the screen boundary.

We are going to implement this in *SBBMainScene.m*. Cocos2D provides a very simple API call to run a timeline animation from code. The following line added to `SBBMainScene` will run the timeline animation and thus play the sound we added to the project:

```
animationManager.runAnimationsForSequenceNamed("DropSound")
```

The animation manager of the root node of a CCB File provides us access to the different timelines and allows us to run, pause them and to react to their completion - we will use these capabilities extensively throughout this book.

As mentioned earlier we want to play the sound effect when an object falls off the screen. We already have code that checks for that condition in our `update` method, all we need

to do this to add the line that runs the timeline. Extend the relevant part of the `update` method to look as following:

```
// check if falling object is below the screen boundary
if (CGRectGetMaxY(fallingObject.boundingBox()) < CGRectGetMinY(
    ↪ boundingBox())) {
        // if object is below screen, remove it
     fallingObject.removeFromParent()
     fallingObjects.removeAtIndex(i)
     // play sound effect
     animationManager.runAnimationsForSequenceNamed("DropSound")
} else {...}
```

Now you can compile and test the project. Every time an object falls of the screen you should hear the drop sound play!

2.8 Wrapping up

In this chapter you have learned how to work with an essential component of all video games - image and audio assets. You have learned how to design scenes with sprites in SpriteBuilder, how to load and change sprite textures in code. You got to know how SpriteBuilder handles different asset resolutions for different screen and device types and you have played your first sound effect. You have learned some of the most important essentials of game development with SpriteBuilder and Cocos2D.

The focus of the next chapter is *User Interaction*. You will learn how to implement Drag and Drop functionality, how to use the Accelerometer as an alternative control scheme and how to use gestures to make your games more enjoyable. Before you move on you should complete the exercise for this chapter.

2.9 Exercises

Now it's time for an exercise. For this exercise you will have to use internet research.

2.0 Play the 'drop' sound without using a SpriteBuilder timeline. Tips: you will have to use `OALSimpleAudio` to play the sound and `CCFileUtils` to find the location of the sound file. Note that this approach that avoids using a SpriteBuilder can often be the more convenient approach to playing sound effects in your game!

3 User Interaction and Collision Detection

In this chapter you will incorporate User Interaction into the object catching game. The first step will be implementing a drag and drop mechanism that lets the user move the pot in order to catch objects. To detect if the player has caught or missed an object we will implement basic collision detection - note that you will later learn how to use the Cocos2D physics engine that provides collision detection out of the box. Whether you want to implement your own collision detection or use the physics engine will depend a lot on the type of game you are developing and we will discuss the advantages of both approaches throughout this book.

When we've implemented the first control scheme we will add a second option for players - controlling the game with the accelerometer of the device, another common way to interact with mobile games.

As a byproduct of implementing these features we will work with translating positions and sizes between different node spaces and the world space, so we will be discussing that important concept throughout this chapter as well.

3.1 Add the pot to the game

The goal of our game will be to move a pot across the screen and try to catch all the vegetables while avoiding catching inedible objects. Before we can implement the drag and drop mechanism we need to add the pot assets to our game, we're going to do that in the SpriteBuilder project, open it now.

Typically we use individual CCB Files for each type of object in our game, however for this game we need to make an exception due to the specific way in which order Cocos2D renders our objects in the game.

3.1.1 Working with the z-order

Throughout this book we are working with a 2D engine. In a 2D engine depth can only be represented by certain objects being placed in front or behind of other objects. Cocos2D uses the following criteria to decide which nodes are rendered in front of other nodes:

1. Child nodes are rendered in front of their parent nodes
2. Siblings (nodes with the same parent) are rendered in order of their `zOrder` property; nodes with higher `zOrder` are rendered in front of nodes with a lower one
3. If two siblings have the same `zOrder` the siblings are rendered in reverse order of how they have been added (the latest added node is rendered in front of all other nodes)

As you can see from the description above the `zOrder` only affects how siblings are ordered, Cocos2D currently does not have a global `zOrder`. For our game we want to create the illusion of objects dropping into a pot, we can do that using the Cocos2D Z-order as shown in the figure below.

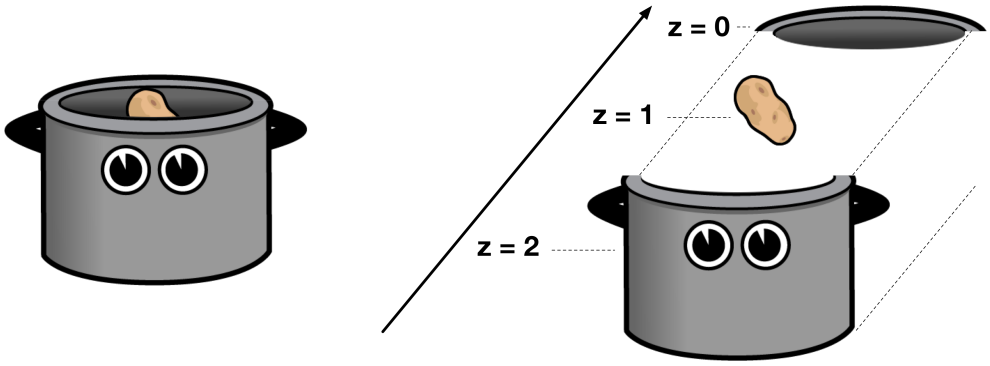


Figure 3.1: Left: Objects on different Layers, Right: How the Z-Order influences on which Layer a node is rendered

For this solution to work all the falling objects and the bottom and top part of our pot need to have the same parent node, otherwise we would not be able to use the Z-Order to place the falling objects between the two parts of the pot.

That is the reason why we are not creating a separate CCB File for the pot object and instead place it inside of `MainScene.ccb`. There would be other ways to work around this issue but adding the pot to the Main Scene is a good solution for this game.

Global Z-order in Cocos2D

While Cocos2D does not have support for global Z-order at the moment, it is being discussed as a potential feature for future releases. Many games run into issues as discussed above due to the lack of this feature. You can follow the discussion on GitHub: https://github.com/cocos2d/cocos2d-swift/issues/662.

3.1.2 Setting up the pot assets

Equipped with everything we need to know about Z-order let's add the pot assets to our Main Scene in SpriteBuilder:

Figure 3.2: Add both pot parts to the main scene

> The assets of both pot parts have exactly the same dimensions. Place both of them at 50% X position and at a Y position of 58 Create code connections for both pot parts, linked to the *Document Root*. Name them `potBottom` and `potTop` respectively. Publish the SpriteBuilder project

Next, move to the Xcode project to set up the code connection variables and implement the touch handling code.

> Open `MainScene.swift` and add instance variables for our code connections at the top of the class:
>
> ```
> weak var potTop: CCSprite!
> weak var potBottom: CCSprite!
> ```

There are three important things to remember about these code connections. Firstly, all code connections should be marked as `weak`. `MainScene` has a reference to the pot sprites but does not *own* them. Instead they are owned by their parent node. For any references that don't mark an ownership, `weak` should be used.

Secondly, we always want to declare code connections as *forcefully unwrapped optionals* as denoted by the bang (!) after the type. Swift requires that all instance variables that aren't optionals are either initialized with a default value or get set to a value in one of the initializers of the class, that way the compiler can guarantee that these variables never contain a `nil` value. Code connections however are set up after the object is initialized (they are guaranteed to be set up when `didLoadFromCCB` is called on the node), so technically these should be optional values. Adding a lot of code for `nil` checking would clutter our classes, that's why we prefer using the bang notation which basically says: *I am confident that this value will never be nil when I am trying to access it*. This is true for code connections as we now that Cocos2D guarantees to have set them up by the time `didLoadFromCCB` is called.

Lastly, be careful not to mark these variables as `private`. Otherwise Cocos2D will not have access to them and won't be able to set up the code connections.

Okay, now we have the basics set up and are ready to dive into the details of implementing a drag and drop mechanism!

3.2 Implement a Drag and Drop mechanism

For the very first project in this book we have already implemented a basic touch mechanism. You should remember that `userInteractionEnabled` is the property that activates/deactivates touch handling for a node and that Cocos2D provides four different callbacks for different state transitions in the lifecycle of a touch. Here's the recap:

touchBegan: called when a touch begins

touchMoved: called when the touch position of a touch changes

touchEnded: called when a touch ends because the user stops touching the screen

touchCancelled: called when a touch is cancelled because user moves touch outside of the touch area of a node

Knowing that, how can we implement a drag and drop control scheme for our game? Dragging and dropping includes three different steps:

1. Pick up object
2. Drag object
3. Drop object

3.2.1 Picking up an Object

In order to pick up an object we need to detect a user's touch and determine if the touch is within the boundaries of our object, if that is the case, we start dragging the object.

First of all, let's turn on user interaction for the `MainScene` class, so that we receive touch events.

Add the required line to the `onEnterTransitionDidFinish` method:

```
override func onEnterTransitionDidFinish() {
  super.onEnterTransitionDidFinish()

  self.userInteractionEnabled = true

  // spawn objects with defined frequency
  schedule("spawnObject", interval: spawnFrequency)
}
```

Next, we need to add the touch handling method. The touch handling method will need to check if the touch is within our pot. If that is the case, the method will need to set a state variable that remembers that we are currently dragging this object. If the user moves a finger across the screen and we are currently in object dragging mode, it is important that the object follows the finger of the user.

Add this implementation to `MainScene.swift`:

```
override func touchBegan(touch: CCTouch, withEvent event:
    ↪ CCTouchEvent) {
  if (CGRectContainsPoint(potBottom.boundingBox(), touch.
      ↪ locationInNode(self))) {
    isDraggingPot = true
    dragTouchOffset = ccpSub(potBottom.anchorPointInPoints,
        ↪ touch.locationInNode(potBottom))
  }
}
```

Let's discuss this implementation briefly. You already have seen the usage of `touch.locationInNode(` in the first chapter of this book, where we briefly discussed touch handling (1.5.4). This method returns the touch position within a given node. In this specific case we are receiving the touch position within `MainScene`.

Next, we are using a utility function, `CGRectContainsPoint`, to check if this touch is within the pot. Remember, that `potBottom` and `potTop` are placed at exactly the same position, so we can choose either of them for this check. `CGRectContainsPoint` takes a rectangle as its first argument and a point as its second. It returns true if the point is within the rectangle.

If the touch position is inside of the pot, we set our state variable, `isDraggingPot`, to `true`.

There is one last line that we didn't discuss upfront:

```
dragTouchOffset = ccpSub(potBottom.anchorPointInPoints, touch.
    ↪ locationInNode(potBottom))
```

In order to drag an object smoothly we need to remember where we touched that object when starting dragging. Take a look at the following diagram:

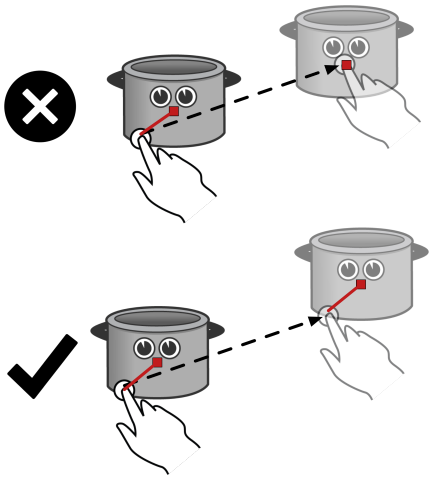


Figure 3.3: *Top Image:* incorrect implementation, object jumps to touch position. *Bottom Image:* correct implementation, touch offset is maintained while dragging the object.

As the user moves the finger, we move the object along. However, the position of the object is not exactly the touch position. Instead it is the touch position *plus* the touch offset determined when we started dragging. We determine that offset by calculating the distance between the anchor point (that's the reference point for positioning a node, typically it's in the center of the node) of the touched object and the exact touch position. No we know why it is important to store the touch offset!

> To wrap up the implementation of `touchBegan` let's add the two instance variables we have referenced: `isDraggingPot` and `dragTouchOffset`. Your list of iVars should now look like this:
>
> ```
> weak var potTop:CCSprite!
> weak var potBottom:CCSprite!
>
> private var fallingObjects = [FallingObject]()
> private let fallingSpeed = 100.0
> private let spawnFrequency = 0.5
> private var isDraggingPot = false
> private var dragTouchOffset = ccp(0,0)
> ```

3.2.2 Moving an Object

Now we'll implement the code that actually moves the pot. That code needs to run whenever a user's finger moves. That means we need to implement the `touchMoved` method.

> Add the `touchMoved` method below the `touchBegan` method:
>
> ```
> override func touchMoved(touch: CCTouch, withEvent event:
>     ↪ CCTouchEvent) {
>   if (!isDraggingPot) {
>     return
> ```

```
    }

    var newPosition = touch.locationInNode(self)
    // apply touch offset
    newPosition = ccpAdd(newPosition, dragTouchOffset);
    // ensure constant y position
    newPosition = ccp(newPosition.x, potBottom.positionInPoints.
        ↪ y);
    // apply new position to both pot parts
    potBottom.positionInPoints = newPosition;
    potTop.positionInPoints = newPosition;
}
```

In the first line we check if we are currently in dragging mode. If not, we do nothing and return immediately. This prevents the pot from jumping to the latest touch position if it has not been picked up beforehand.

If we are in dragging mode we continue. First we get the new touch position. Then we apply the offset that we discussed in figure 3.3 to that new position. The next line ensures that the y position of the pot stays constant, we want to allow horizontal movement only. Finally, we apply that new position to both pot parts. Great, we're pretty close to finishing the drag and drop functionality.

If you test the app in the current state you'll see that there's one simple yet important step missing...

3.2.3 Dropping an object

Right, the user will also want to drop the pot by releasing the finger from the screen. Otherwise we stay in dragging mode forever and the pot will keep jumping to whichever

position the user taps on the screen.

Luckily this can be easily implemented. All we need to do is to set `isDraggingPot` to false as soon as the user stops touching the screen.

> Add the `touchEnded` method below the `touchMoved` method:
>
> ```
> override func touchEnded(touch: CCTouch, withEvent event:
>     ↪ CCTouchEvent) {
>   isDraggingPot = false
> }
> ```

Awesome! Our drag and drop code is complete! Drag and drop mechanisms can be used in many types of games, so what you have learned in this chapter is very valuable.

Now we can move on to the next chapter, which is one of the hardest (thus most interesting) parts of this book. We will implement catching objects while learning more about scene graphs and node transforms.

3.2.4 Swipe Gesture

Remove all objects from screen (powerup)...

3.2.5 Exercise

Implement Accelerometer

4 Scene graphs and node transforms

From a project perspective you will learn how to implement a catching mechanism for our game. This will require you to learn more details about scene graphs, node transforms and also about modelling game mechanics in Swift.

4.1 Catching objects

Implementing drag and drop was a great warm up. In this section we are going to solve a bunch of problems that will bring our little project a large step closer to being a real game. By the end of this section the user will be able to catch and miss objects by dragging the pot below objects with the right timing.

Before we dive into coding let's think about what we actually need to implement. There are three important aspects that need to be covered through our implementation:

1. detecting if the user has caught an object
2. detecting if the user has missed an object
3. visualizing catching / missing correctly

4.1.1 Thinking in states

Our feature outline describes that objects start out as falling objects, directly after they have been spawned. At some later point in time the user can catch or miss these objects. In each of these situations we need our falling objects to behave differently. If they are falling we want them to move down the screen with a constant speed. If they are caught we need some sort of visualisation - ideally the objects move into the pot and disappear. If the user tries to catch an object too late and misses it closely we want to visualize that, too.

From the paragraph above we can extract three different states in which a falling object can be:

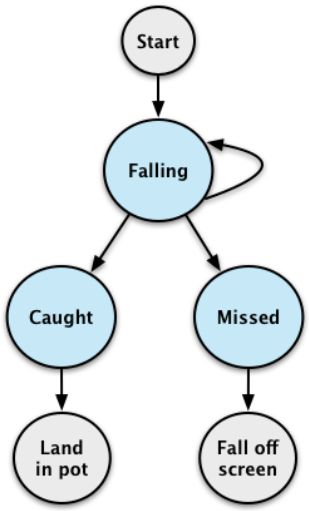


Figure 4.1: Objects start in falling state, then they end up caught or missed

As the diagram shows, a falling object can either stay a falling object or turn into a caught or missed object. It is up to us developers to decide the criteria for a state change. We

also need to decide when we want to check for state changes.

For our game I suggest that we check whether a player has caught an object or not in the `update` method. As soon as that object reaches the y position of the top of the pot we decide based on the x position whether the object has been caught or missed

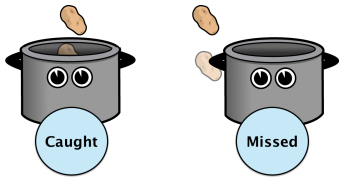


Figure 4.2: Caught objects fall into the pot, missed objects fall behind

Since we are building a 2D game we only have limited ways of expressing that a player missed a falling object - I suggest that we render missed objects behind the pot. That way players can quickly see whether they caught an object or not.

Now we have a good starting point for some coding; we need to store different states for falling objects and we need to write specific behavior code for each of these states. Additionally we need to write code that checks if we have caught or missed an object so that we can assign the correct states to falling objects.

4.1.2 Storing state

Now we'll start implementing everything described earlier. Let's start by adding a `fallingState` to `FallingObject.swift`. That state variable will remember whether an object is currently falling, has been caught or has been missed.

The best way to represent states in Swift is to use an enumeration!

> Add this enum definition to `FallingObject.swift` below the `FallingObjectType` enum:
>
> ```
> enum FallingObjectState {
>   case Falling
>   case Caught
>   case Missed
> }
> ```

As mentioned earlier, associating enum entries with a type is not mandatory. In this case our entries don't need a type (e.g. Int) since the entries will only represent a state - they are values in their own right.

> Next, add an instance variable to store the current state:
>
> ```
> var fallingState = FallingObjectState.Falling
> ```

This variable should not be private, we want to change the value as the object gets caught or missed. Our default state is `.Falling`, we assign it as part of the variable declaration.

Now we can store a `fallingState` for each falling object; next, let's implement different behaviour based on that state.

4.1.3 Implement state specific behaviour

The majority of our gameplay code is currently inside of the `update` method of `MainScene`. This is fairly common for simple games. Currently we are doing to things in the update method: moving the objects down the screen and checking whether they have left the stage entirely (in which case we delete them). Now however, we are going to add code that will only run for falling objects in certain states. That will add quite a lot of complexity. Instead of squashing everything into the update method I suggest that we create

one method for each of the three states. These methods will contain all state specific code and will be called from the update method.

Replace your existing update method with the following one:

```
override func update(delta: CCTime) {
// use classic for loop so that we can remove objects while
   ↪ iterating over the array
for (var i = 0; i < fallingObjects.count; i++) {
  let fallingObject = fallingObjects[i]

  // let the object fall with a constant speed
  fallingObject.position = ccp(
    fallingObject.position.x,
    fallingObject.position.y - CGFloat(fallingSpeed * delta)
  )

  switch fallingObject.fallingState {
  case .Falling:
    performFallingStep(fallingObject)
  case .Missed:
    performMissedStep(fallingObject)
  case .Caught:
    performCaughtStep(fallingObject)
  }
 }
}
```

Now the update method is really easy to read. We loop over all falling objects. In all cases we move the falling object towards the bottom of the screen. After that we check in which state an object is and invoke a method that contains code specific for that state. We are going to implement these methods throughout the remainder of this chapter.

4.1.4 Implementing the falling state

Let's start implementing the default state: falling. In this state we will need check whether an object has been caught, has been missed or simply remains falling.

Later in this book you will learn how to use the Cocos2D physics engine and its built in collision detection - for this game however we are not using the physics engine and will implement catch detection code ourselves.

In figure 4.2 we have illustrated what we consider a caught/missed object. So how can we implement this? Basically all we need to do is compare the frame of the falling object to the frame of the pot. However, there is one small issue. The frame of CCSprite is always a rectangle that contains the entire texture, here's what the dimensions of the frames of our pot and a falling object looks like:

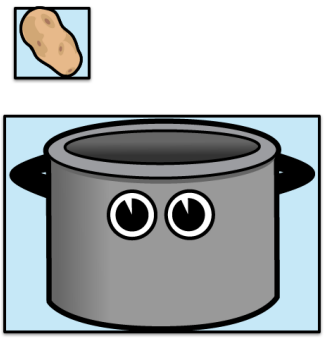


Figure 4.3: The pot frame is too large to use it for collision detection

From the illustration above you can see that the frame of the pot is too large to use it for collision detection. It could easily happen that an object landing on the handle of the pot would still be considered a catch.

Instead of using the pot dimensions we will need to add a separate, smaller, node in

SpriteBuilder that marks the catch area.

> Open the SpriteBuilder project and open *MainScene.ccb*. ...Now publish the SpriteBuilder project and switch back to the Xcode project.

Next, we need to add a variable for the code connection we just set up.

> Add the catch container variable to the other code connection variables:
>
> ```
> weak var catchContainer: CCSprite!
> ```

Now we have a reference container set up that will allow us to test if objects have been caught. We can start implementing falling step function.

> Add the method stub for the falling step to *MainScene.swift*:
>
> ```
> func performFallingStep(fallingObject:FallingObject) {
>
> }
> ```

Before we can dive into collision detection we will have to take a little detour and talk about node transformations. As part of the introduction to Cocos2D we have discussed that nodes are always positioned relative to their parent node (chapter: 1.3.5). The catch container that we just added in SpriteBuilder is a child of the `potBottom` node. We chose that setup so that the catch container always moves together with the pot.

For our collision detection algorithm we want to compare the position of a falling object to the position of our catch container. **Here the problem arises: falling objects and the catch container have different parent nodes, that means we cannot compare their positions and frames directly.** Since the position is relative to the parent node, comparing nodes with different parents would resolve in unexpected behavior. Take the following illustration as an example:

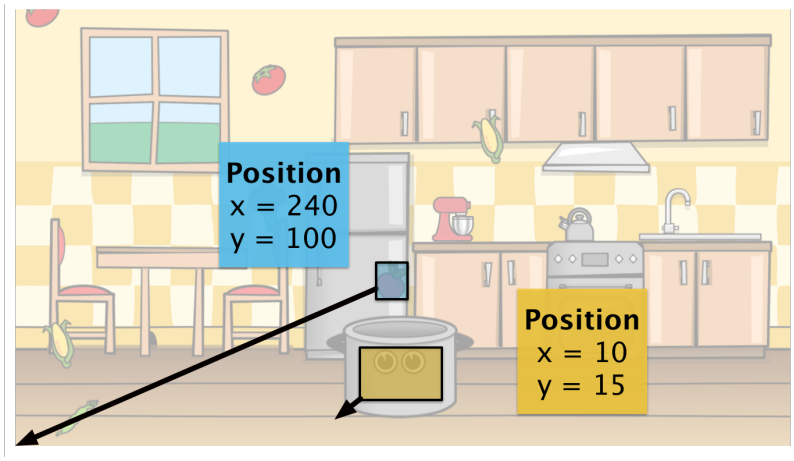


Figure 4.4: Even though the two nodes illustrated above are close to each other, their position values are entirely different, since they are placed relative to different parents

How can we work around this? Luckily Cocos2D exposes a couple of variables and methods that allows us to transform positions and frames between different *node spaces*. Each node lives in the *node space* of its parent. In our example the catch container is in the node space of the pot and the pot is in the node space of the main scene.

If we want to know the position and size of the catching container in the main scene node space, we need to apply the following transform:

```
let containerWorldBoundingBox = CGRectApplyAffineTransform(
  catchContainer.boundingBox(), catchContainer.parent.
    ↪ nodeToParentTransform()
);
```

We are transforming the bounding box of the catch container using the `nodeToParentTransform` of the catch container's parent node (the pot). The Cocos2D documentation describes the `nodeToParentTransform` as following: *Returns the matrix that transform the node's (local)*

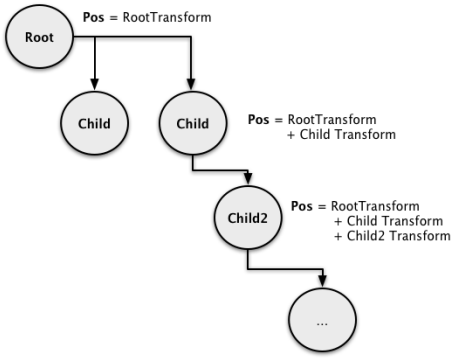
space coordinates into the parent's space coordinates.

This means after applying the transform we know the position of the catch container in the main scene space.

If you are new to graphics programming this concept will likely seem a little confusing; frankly you won't need it too often when working with Cocos2D. If you aren't getting a hold of transforms yet, don't worry too much!

The role of transforms in graphics programming

Transforms are an essential part of all graphics engines - also of Cocos2D. When determining the positions for all nodes in a scene, Cocos2D starts with the root node. After the root node is laid out, the engine moves to the children of the root node, calculates their position and places them *relative to the root node*. This is repeated all the way down the node hierarchy:



Now that we have a solution for the transformation issue, the rest of the code that we need for the falling step is not too complicated.

Here's the entire function that implements the stub that you added earlier:

```
func performFallingStep(fallingObject:FallingObject) {
  let containerWorldBoundingBox = CGRectApplyAffineTransform(
    catchContainer.boundingBox(), catchContainer.parent.
       ↪ nodeToParentTransform()
  );

  let yPositionInCatchContainer = CGRectGetMinY(fallingObject.
     ↪ boundingBox()) < CGRectGetMaxY(
     ↪ containerWorldBoundingBox)
  let xPositionLargerThanLeftEdge = CGRectGetMinX(
     ↪ fallingObject.boundingBox()) > CGRectGetMinX(
     ↪ containerWorldBoundingBox)
  let xPositionSmallerThanRightEdge = CGRectGetMaxX(
     ↪ fallingObject.boundingBox()) < CGRectGetMaxX(
     ↪ containerWorldBoundingBox)

  // check if falling object is inside catching pot, trigger
     ↪ this when object reaches top of pot
  if (yPositionInCatchContainer) {
    if (xPositionLargerThanLeftEdge &&
       ↪ xPositionSmallerThanRightEdge) {
      // caught the object
      let fallingObjectWorldPosition = fallingObject.parent.
         ↪ convertToWorldSpace(fallingObject.
         ↪ positionInPoints)
      fallingObject.removeFromParent()
      fallingObject.positionInPoints = potTop.
         ↪ convertToNodeSpace(fallingObjectWorldPosition)
      potTop.addChild(fallingObject)
      fallingObject.fallingState = .Caught
    } else {
      fallingObject.fallingState = .Missed
    }
```

```
        }
    }
```

We have already discussed the first statement extensively, we transform the bounding box of our catch container. That allows us to compare its position to the position of falling objects.

The next three lines are each used to determine if the falling object is within the relevant boundaries of our transformed catch container. The `CGRectGetMin...` utility functions are used to get the lowest/highest value on a certain axis from the bounding box. These three statements check for the conditions outlined in figure 4.2. If all three are true the player has caught the object.

Next, we have an if statement that combines the three boolean variables. The first if statement checks if the falling object is in the *critical area* using the `yPositionInCatchContainer` constant. Here the y position of the falling object is the only relevant metric. If we aren't in the critical area we do nothing at all - the object is still too far above the pot for us to decide whether the player caught it or not.

If the object is in the critical area we now need to determine if it has been caught or missed. This is where we need the two x position variables. If the object is outside of the bounds we set the `fallingState` to `.Missed`.

If the object is inside of the bounds we set the `fallingState` to `.Caught`. Additionally we need to ensure that once the object is caught it stays within the pot. Without additional code the caught objects are not attached to the pot. The player could move the pot left or right and the objects would fall out to the side of the pot. As soon as an object is caught we need to turn it into a child node of the pot, that way they will stick together.

Here we once again need a transform. We want to turn the falling object into a child of the pot instead of being a child of main scene. That means we are moving the object to

a different node space. We don't want the player to see this move happen; visually the object should stay at exactly the same position.

In such situations we need to use a two step transform. First, we need to find the *world space* position of the node that we are moving to a different node space. The position in the world space is expressed relative to the world root (in most cases the bottom left corner of the screen) and not relative to the parent node. You can think of the position in world space as a global or absolute position. We can use the world position to find the corresponding relative position in any node space.

Let's take a look at our specific code. First we call:

```
let fallingObjectWorldPosition = fallingObject.parent.
    ↪ convertToWorldSpace(fallingObject.positionInPoints)
```

This line asks: *What is the global position, independent of the parent node, of this falling object?* The node that receives this question needs to be the parent node of `fallingObject`, because that is the node responsible for placing the `fallingObject` node by applying its transform.

Now that we have saved the position, we remove the node from its parent. Next we perform the second step of the transformation:

```
fallingObject.positionInPoints = potTop.convertToNodeSpace(
    ↪ fallingObjectWorldPosition)
```

This line asks: *Dear pot, I have a global position for this falling object, could you tell me what the relative position to you would need to be? I want the falling object to remain at the same global position after adding it to you as a child.* After we have determined the right position we finally add the falling object to the pot. The object will now switch to a different node space and become a child of the pot without that the player will realize it, awesome!

This was a pretty intense implementation so here's recap what we did to implement the

code that runs while our object is in the *falling state*:

1. We added a catch container do define the area in which a player can catch objects. We did this because the frame of the entire pot is too large to serve as catch area

2. We transformed this catch container from the pot space into the main scene space. We did that because we need the falling object and the catch container to be in the same space in order to compare their positions

3. When determine that an object has been missed we set the state of the falling object to `.Missed`

4. When we determine that an object has been caught we set the state of the falling object to `.Caught`. Additionally we add the caught object as a child to the pot, to ensure that the object stays within the pot after it has been caught. Before we add the object as a child to the pot we use a two way transform to figure out the position the object needs to have as a child of the pot node

This concludes almost all the features we need for the *falling* step. Later we will come back for some visual tweaks but for now can move on to the missed state. This is also a great time for a break and your favorite hot beverage!

4.1.5 Implementing the missed state

Good news: the remain two steps are a lot simpler. The missed step only consists of code that we have already written.

> Add the method for the *missed* step:

```
func performMissedStep(fallingObject:FallingObject) {
  // check if falling object is below the screen boundary
  if (CGRectGetMaxY(fallingObject.boundingBox()) <
     ↪ CGRectGetMinY(boundingBox())) {
```

```
        // if object is below screen, remove it

        fallingObject.removeFromParent()
        let fallingObjectIndex = find(fallingObjects,
            ↪ fallingObject)!
        fallingObjects.removeAtIndex(fallingObjectIndex)
        // play sound effect
        animationManager.runAnimationsForSequenceNamed("DropSound"
            ↪ )
      }
    }
```

All of this code was part of the `update` method earlier. All we do here is move it into a separate method. As soon as an object is in the missed state we now that it has fallen below the pot opening and can be no longer caught. Now all we need to do is to wait until the object falls below the screen boundary, then we play our sound and remove it.

4.1.6 Implementing the caught state

The last state is the simplest of all. When we have caught an object we want to create the illusion of the object disappearing into the pot. The first step is adding the object as a child to the pot, we've already done that in the *falling* step.

All we need to do in the *caught* state is wait until the object disappears into the pot entirely; then we can remove it.

Add the method for the *caught* step:

```
func performCaughtStep(fallingObject:FallingObject) {
  // if the object was caught, remove it as soon as soon as it
      ↪  is entirely contained in the pot
```

```
    if (CGRectContainsRect(catchContainer.boundingBox(),
        ↪ fallingObject.boundingBox())) {
      fallingObject.removeFromParent()
    }
  }
```

As soon as the catch container bounding box fully encloses the caught object we can remove it. For the player it will seem that the object disappeared into the inner darkness of our bottomless pot.

4.1.7 Time to test

Now we're finally back to a state where we can run and test the game. You should now be able to catch objects in the game, the illusion of them disappearing in the pot should be perfect. There's one last thing I'd like to see tweaked before we move on: missed objects should be rendered behind the pot instead of in front of it.

4.2 A rendering tweak

We have discussed the basics of influencing the z-order in chapter 3.1.1. Now we will use the `zOrder` property to render the falling object behind the pot in case the user missed the object (remember that the z-order currently only applies to the ordering of siblings; children are always rendered in front of their parents).

There's a neat trick for managing the rendering order in a scene. We can use an enum in which each entry represents a different *layer* in the scene.

> Add this enum definition to the `MainScene` class:
>
> ```
> enum DrawingOrder: Int {
>   case BehindPot
>   case PotTop
>   case FallingObject
>   case PotBottom
> }
> ```

Here we are defining four different layers. Each of them haven an associated integer value that we can directly apply to the `zOrder` variable of our nodes. This enum describes that the `FallingObject` layer will be rendered in front of the `PotTop` layer. By using this enum technique we can easily change the rendering order in scenes without modifying a lot of code.

Next, we need to assign the `zOrder` values to their corresponding nodes.

> Add this implementation of `didLoadFromCCB` that initializes the pot's `zOrder` to `MainScene`:
>
> ```
> func didLoadFromCCB() {
>   potTop.zOrder = DrawingOrder.PotTop.rawValue
>   potBottom.zOrder = DrawingOrder.PotBottom.rawValue
> }
> ```

Now we need to take care of the falling object. When it is spawned it should be rendered on the `DrawingOrder.FallingObject` layer which is between the front and back part of the pot.

> Add the according line to the `spawnObject` method:
>
> ```
>     ...
>     fallingObject.zOrder = DrawingOrder.FallingObject.rawValue}
> ```

```
    addChild(fallingObject)
    ...
```

When the object is missed, we want to render it behind the pot.

> Add the relevant line to the else case of the `performFallingStep` method:
>
> ```
> ...
>   } else {
>     fallingObject.fallingState = .Missed
>     fallingObject.zOrder = DrawingOrder.BehindPot.rawValue
>   }
> ...
> ```

Great! With this tweak we have completed a significant portion of the core gameplay.

Index