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 10.10 and Xcode 6.3).

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:

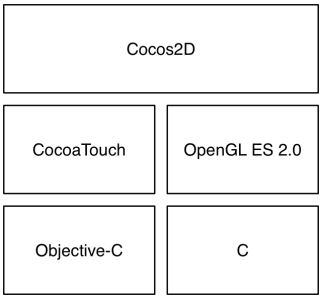


Figure 1.2: Cocos2D Technology Stack

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:

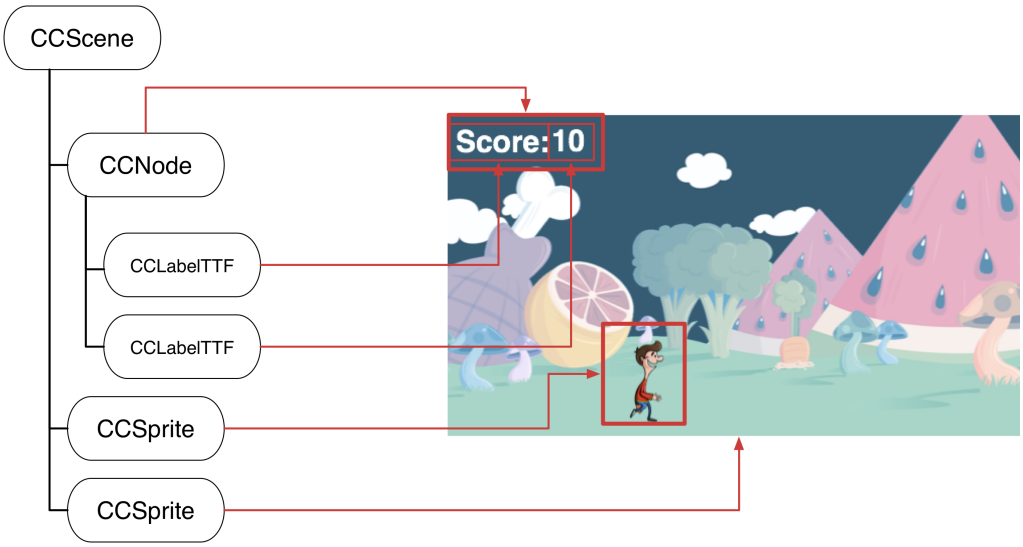


Figure 1.3: Cocos2D Scene Graph

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:

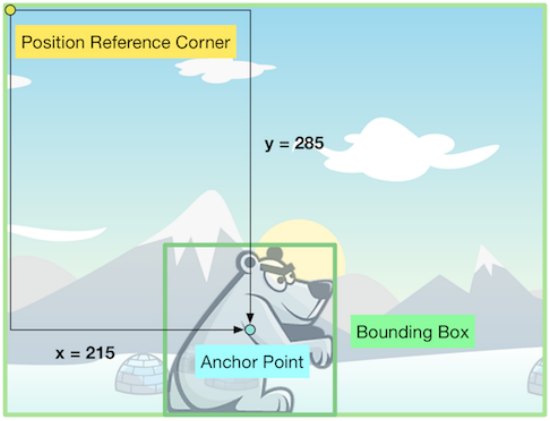


Figure 1.4: CCNode positioning example

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:

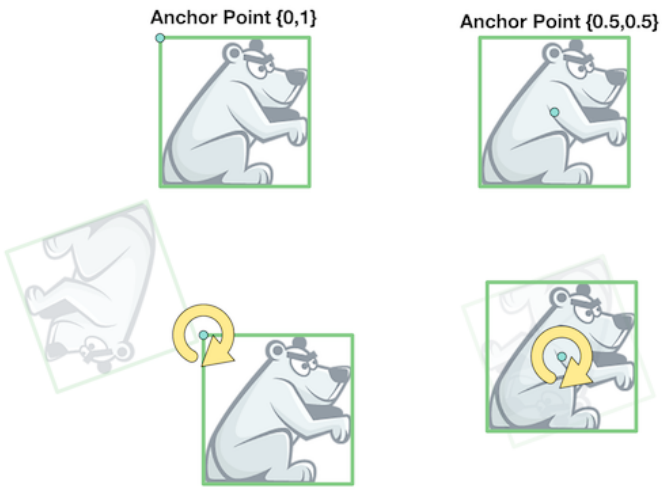


Figure 1.5: CCNode positioning example

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

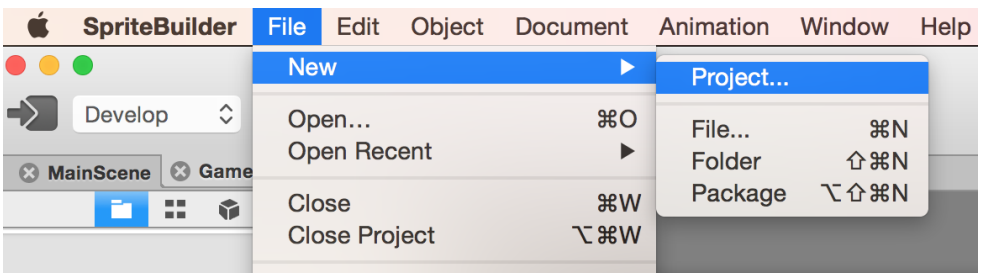
You have learned the fundamentals of the game engine we will use. Now we will take a look at a tool called SpriteBuilder which we will use to create the majority of our game content. The main purpose of SpriteBuilder is to provide a visual editor for the creation of scenes, animations and more. For most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

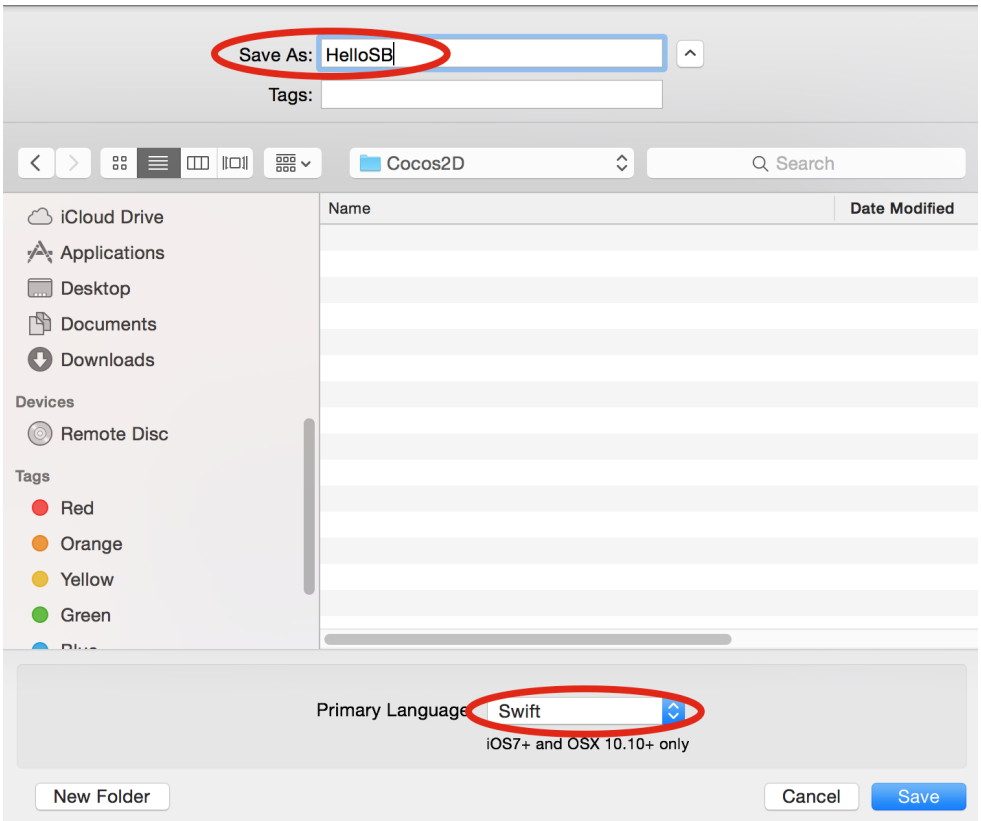
1.4.1 Creating a first project

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*. Also make sure to choose *Swift* as the primary project language:



After you create the project the folder structure should look similar to this:

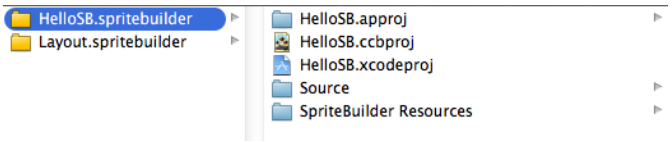


Figure 1.6: SpriteBuilder project folder structure

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

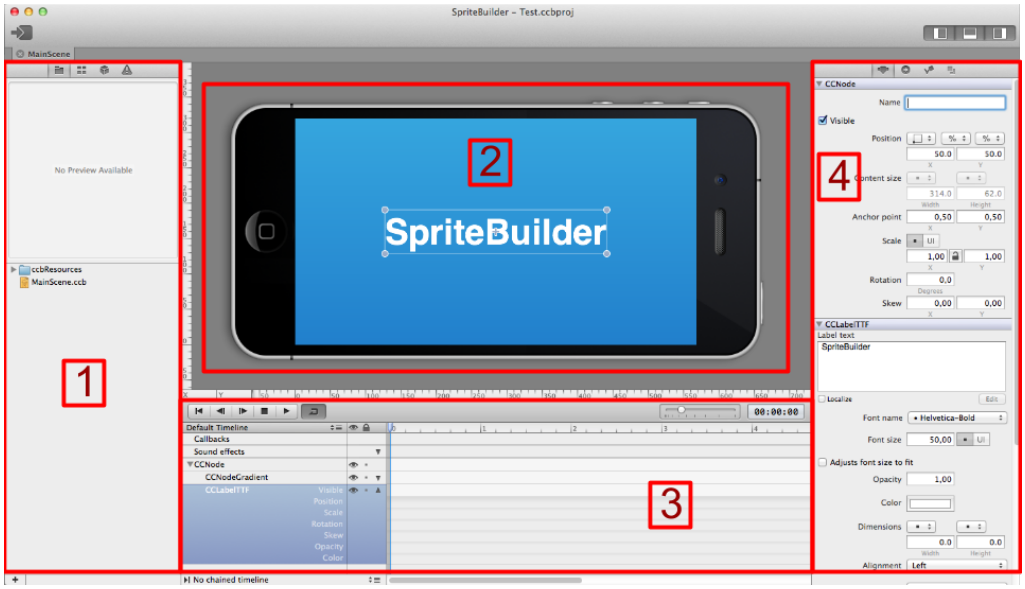
SpriteBuilder will create an Xcode project for every new project you create! The Xcode project will automatically contain the newest version of Cocos2D - very handy.

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.

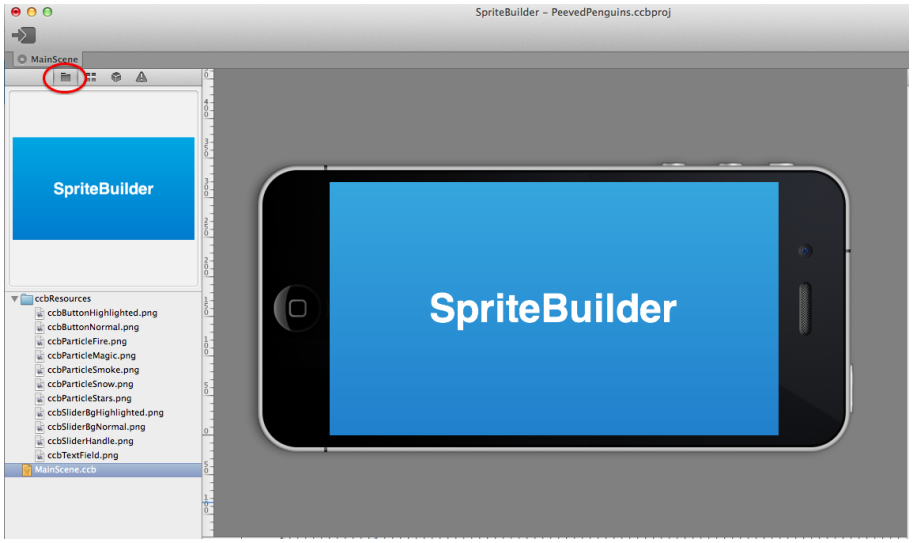
3. *Timeline:* The timeline is used to create animations within SpriteBuilder.

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

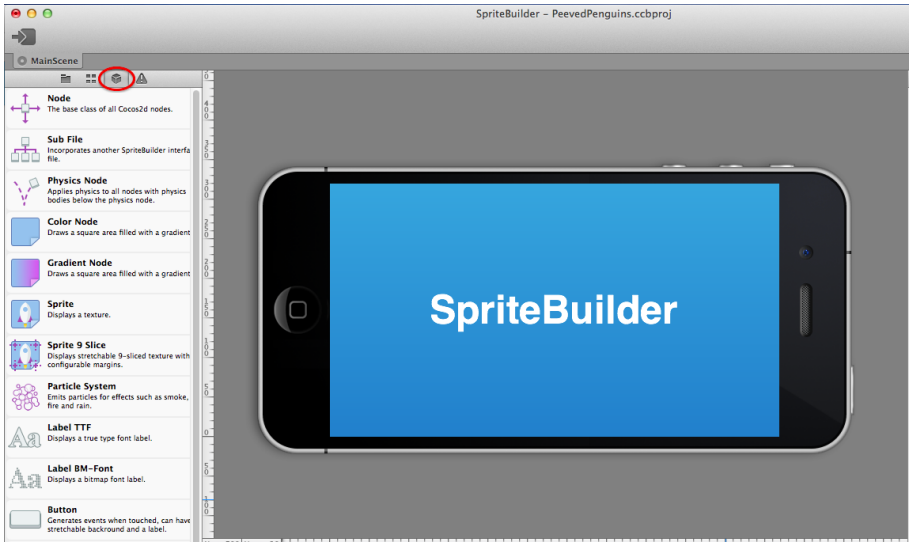
The first tab in the resource/component browser represents the *File View*. It lists all the *.ccb* files and resources that are part of the SpriteBuilder project:



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:

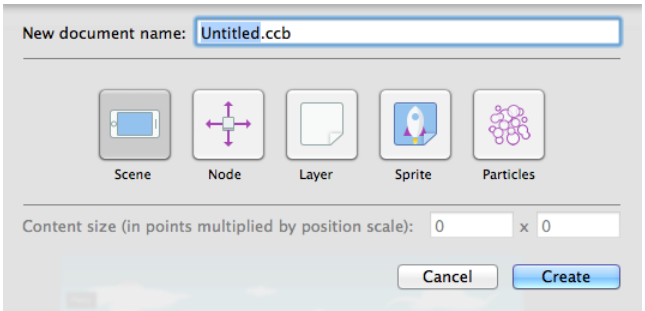
Code Connections

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:

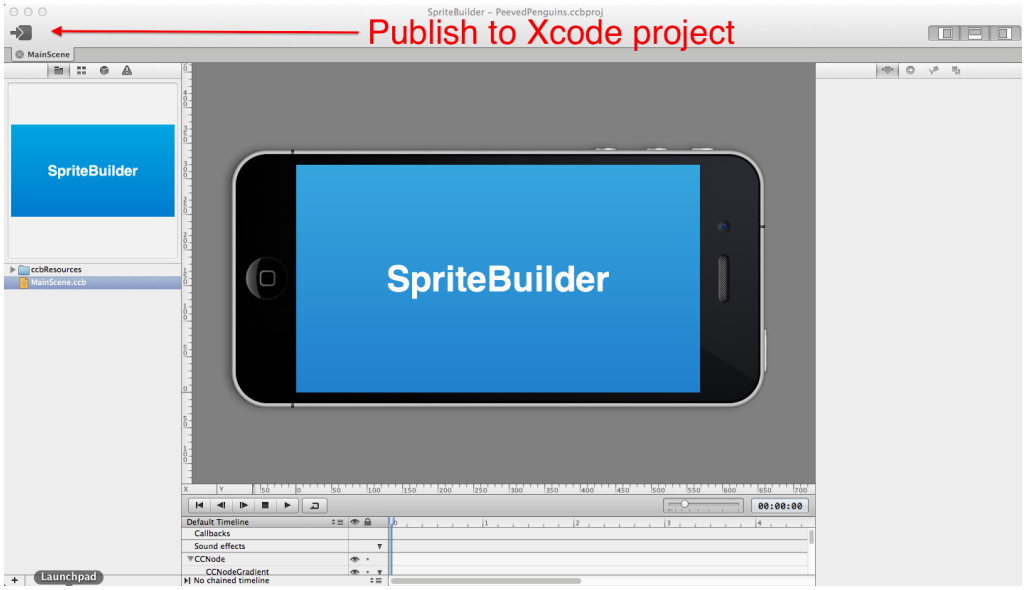


Figure 1.7: Use the publish button to update your Xcode project with the latest changes in your SpriteBuilder project.

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:

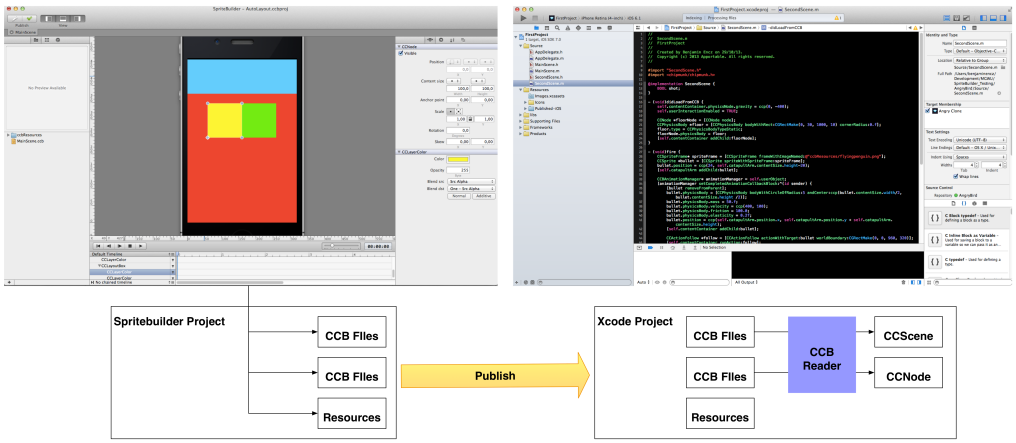


Figure 1.8: SpriteBuilder creates and organizes a Xcode project for you. Adding all the resources and scenes you have created.

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:

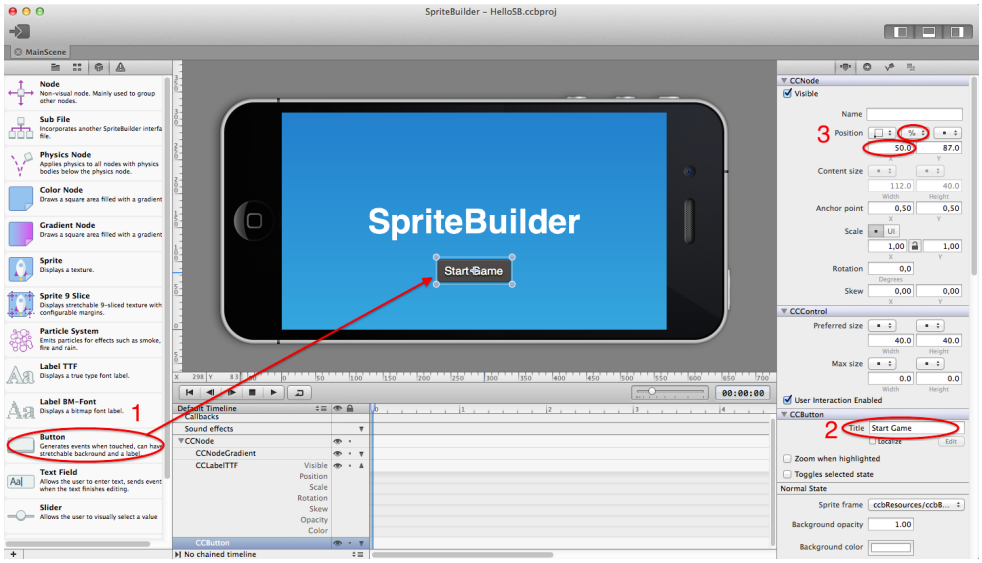


Figure 1.10: The project build throughout this chapter

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

If you want to learn more about dynamic layouts you should read our two part tutorial: https://www.makeschool.com/tutorials/dynamic-layouts-with-spritebuilder-and-cocos2d-3-x/

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

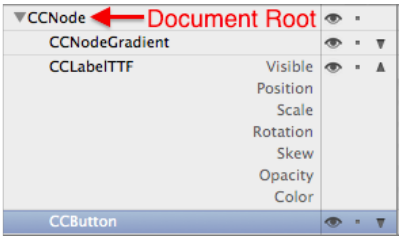
Inside the CCControll section you can see two options called *selector* and *target*. Here you can choose which method (selector) shall be called on which object (target) when this button is tapped by a user. As selector enter `startGame`. As target choose *Document Root*.

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:

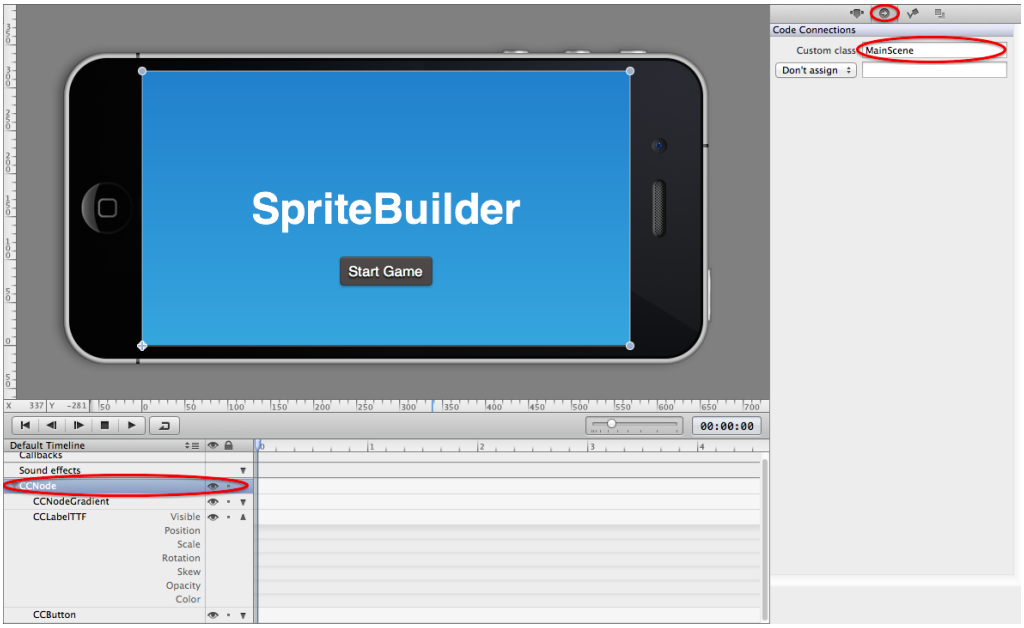
Document Root The document root is the highest node within the current CCB File. The hierarchy of the CCB File is shown in the SpriteBuilder timeline:



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 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 can also use a shortcut provided by SpriteBuilder: CMD + Shift + O

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.

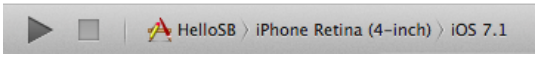
> Open the *MainScene.swift* and file and add the following method:
>
> ```
> func startGame() {
>   println("Start Game")
> }
> ```

For now we will simply use the `println` function 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.

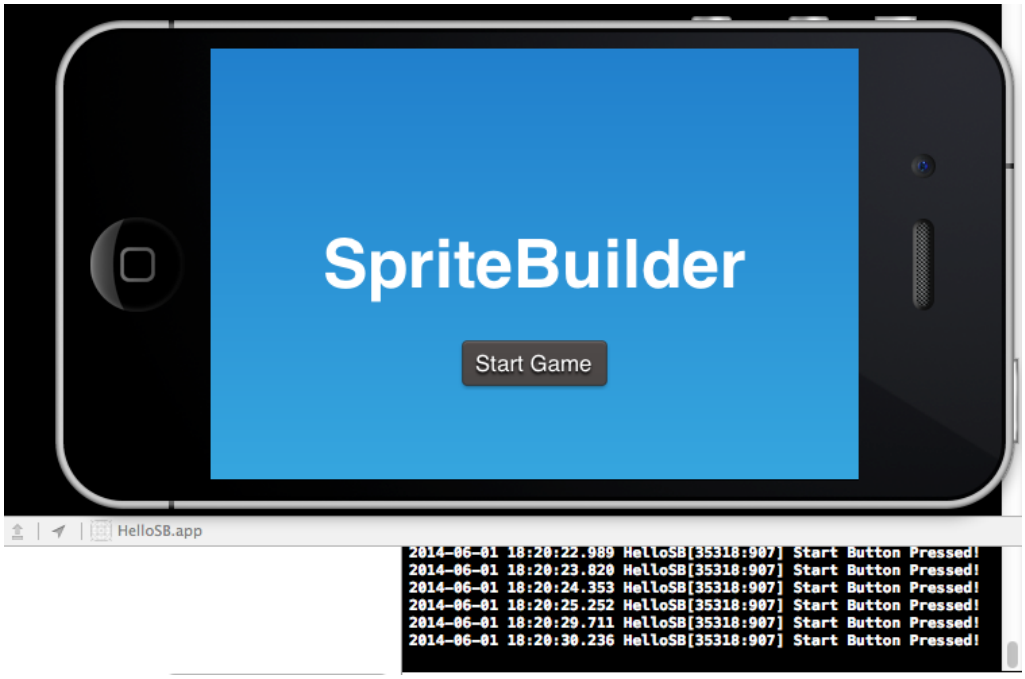
Displaying the console in Xcode

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!

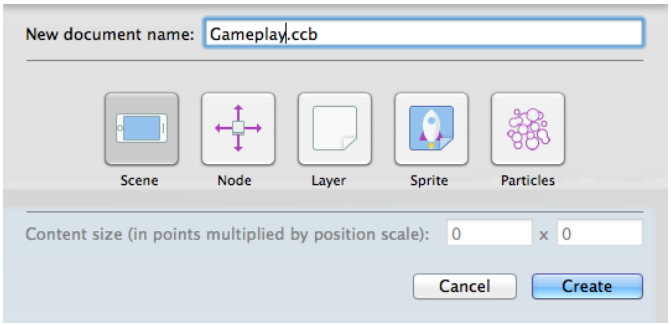
Common Error 1.1

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.

> Open SpriteBuilder to create a new 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`.

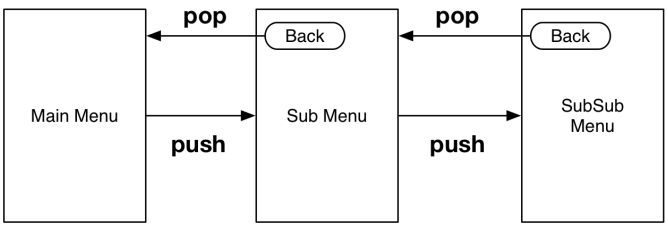
Before we look into using transition effects, let's take a look at the differences between pushing and replacing a scene.

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:

```
let transition = CCTransition(fadeWithDuration: 1.0)
```

```
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 we are going to implement a transition.

> Replace the current implementation of `startGame` with this one:
>
> ```
> func startGame() {
>   let gameplayScene = CCBReader.loadAsScene("Gameplay")
>   let transition = CCTransition(fadeWithDuration: 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 in order to present is as a full-screen 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.

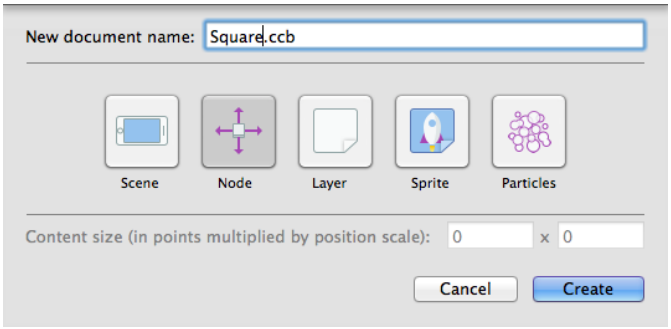
1.5.4 Implementing the Gameplay

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

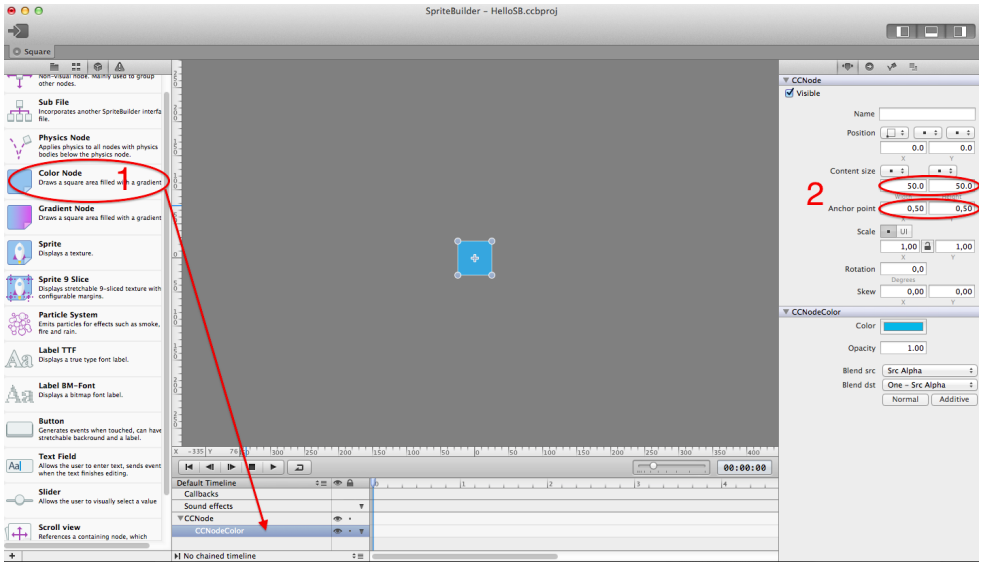
Let's start by creating the square we want to spawn during the game in SpriteBuilder.

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*.

Add a *Color Node* by performing the following steps:

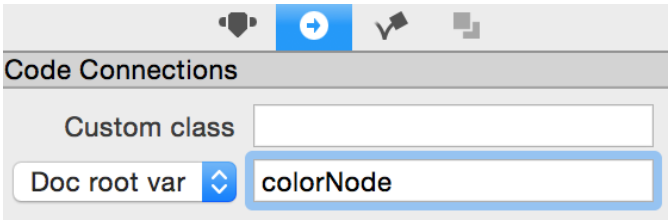


(1) Open the *Node Library* and drag a *Color Node* to the stage or the timeline in order to add it to the root node of of *Square.ccb*.

(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. Let's add a code connection to make this possible.

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):



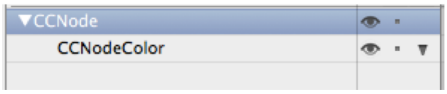
As the variable name (entered in the text field), choose *colorNode*. As the second option you need to choose the object to which this variable will be assigned to. Just as for callbacks you can choose between the *Document Root* and the *Owner* (1.5.1). We choose the *Document Root*, which means that SpriteBuilder will attempt to store a reference to the *Color Node* in a property called *colorNode* on the root node object of this CCB File.

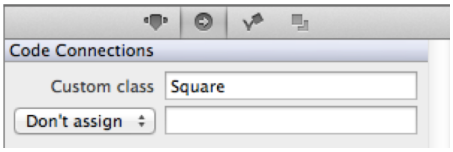
We now face the same 'problem' as earlier when we set up a *Callback*. The root node of *Square.ccb* is a plain CCNode and a plain CCNode does not have an instance variable called *colorNode*! We once again need to define a custom class for the root node of this CCB File.

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 a property 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*).

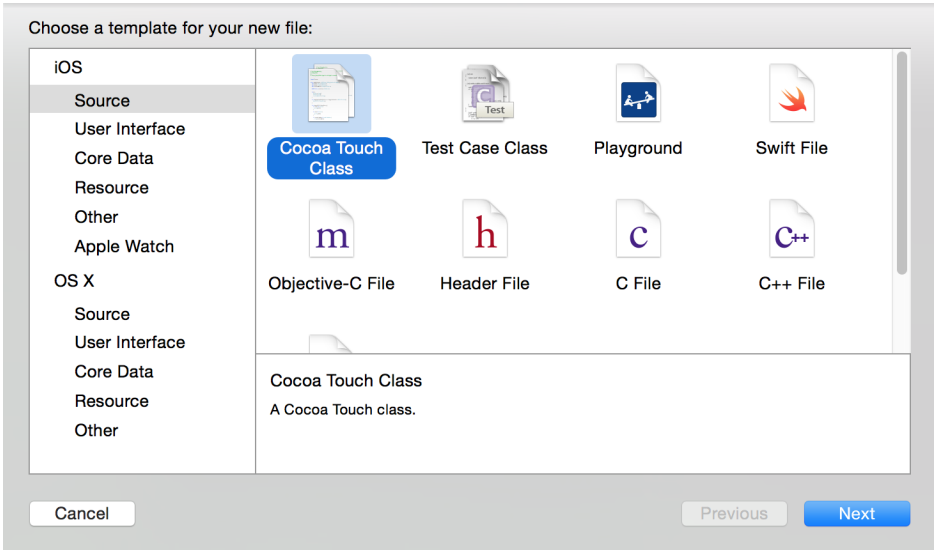
1. Open *Gameplay.ccb*
2. Select the root CCNode from the timeline
3. Open the code connections tab (the second tab on the right pane)
4. Define the *Custom Class* to be *Gameplay*

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.

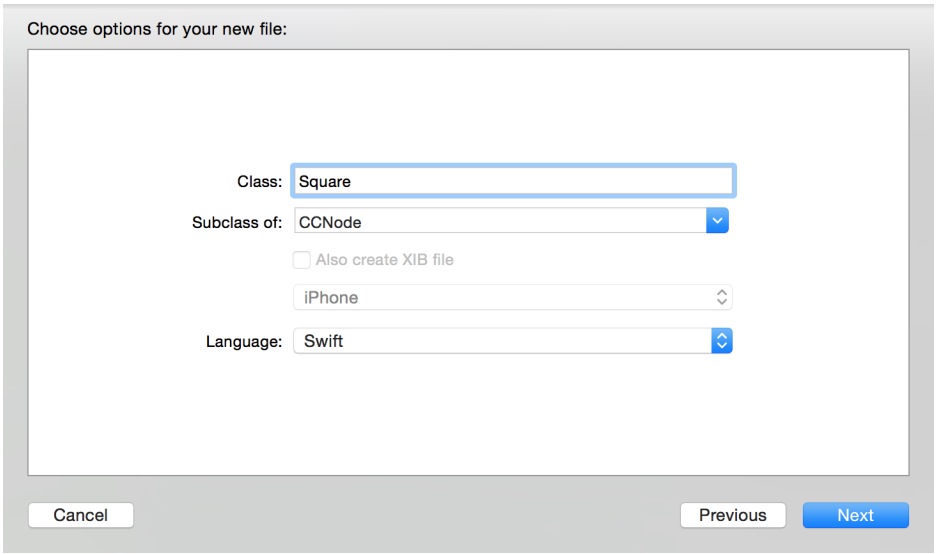
Creating the Square class

Now let's add the code to our project.

Open Xcode and create a new Swift class by selecting *File -> New File...* and choosing *Cocoa Touch Class*:



Make sure you have selected **Swift** as the language! As class name choose *Square* and define it to be a subclass of CCNode:



Then hit the *Next* button to create the file.

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.swift* and add the property 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:

```
class Square : CCNode {

  weak var colorNode : CCNodeColor!

}
```

After adding the property, the code for *Square.swift* should look as shown above. Note

that we are using a `weak` variable of forcefully unwrapped type (indicated by the trailing *!*). This will be default for all code connections throughout this book.

The property is declared as `weak` because it doesn't *own* the referenced child node. We only want to use this property as a reference - we know that another part of our game is responsible for keeping the referenced object around. In this case the `colorNode` will stay in memory as long as it is a child node of a scene that is rendered on the screen.

Since the code connection we set up will not have a value assigned as soon as `Square` is initialized we need to declare the property as either *optional* or *forcefully unwrapped*. Swift has pretty strict initialization rules! We choose *forcefully unwrapped* because we know that Cocos2D guarantees that this property will have a value as soon as the CCB File is loaded and `didLoadFromCCB` is called. Given this knowledge it's preferable to choose *forcefully unwrapped* over *optional* because it avoids a lot of optional-unwrapping code.

Swift initialization and optionals

You can read more about Swift initialization and optionals in our following tutorial: https://www.makeschool.com/tutorials/learn-swift-by-example-part-3-classes-and-initialization

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*:

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

  let red = Float(arc4random_uniform(256)) / 255.0
  let green = Float(arc4random_uniform(256)) / 255.0
  let blue = Float(arc4random_uniform(256)) / 255.0

  colorNode.color = CCColor(red: 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. Since `CCColor` wants to be initialized with `Float` values we need to explicitly create `Float` numbers from the result of calling the `arc4random_uniform` function which returns a `UInt32`.

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:

```
let move = CCActionMoveTo(duration:2.0, 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.makeschool.com/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:
>
> ```
> let rotate = CCActionRotateBy(duration: 2.0, angle: 360.0)
> let repeatRotation = CCActionRepeatForever(action: rotate)
>
> 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!

Now 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 *Cocoa Touch 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 `true`, 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.makeschool.com/docs/#!/cocos2d/1.1/user-interaction and https://www.makeschool.com/tutorials/touch-handling-in-cocos2d.

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 `onEnterTransitionDidFinish` 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.swift*:

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

  self.userInteractionEnabled = true
}
```

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 `true`. Now Cocos2D knows that this node wants to receive touch events.

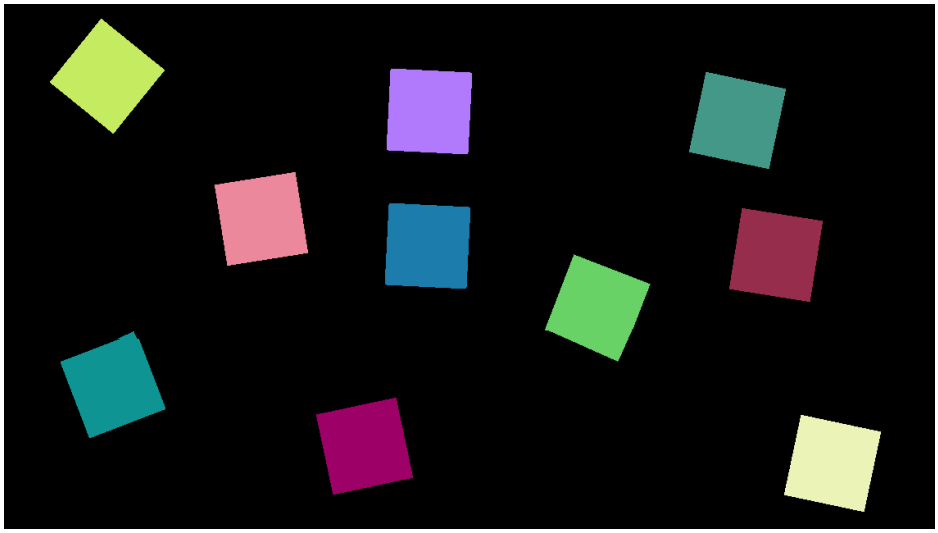
In the next step we need to decide to which touch events we want to subscribe and implement the corresponding method. For this simple game we only need to know when a touch begins, because we will add a square to the screen immediately. This means we only need to implement the `touchBegan` method.

Add the following implementation to `Gameplay.swift`:

```
override func touchBegan(touch: CCTouch!, withEvent event:
    CCTouchEvent!) {
  let touchPosition = touch.locationInNode(self)
  let square = CCBReader.load("Square")
  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.

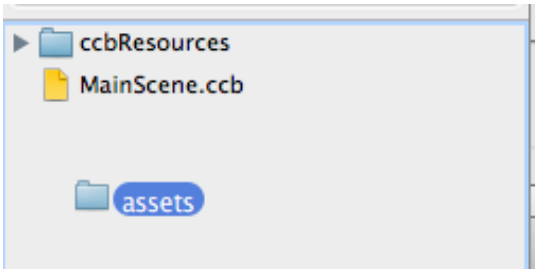
2 A Game with Assets in SpriteBuilder

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:

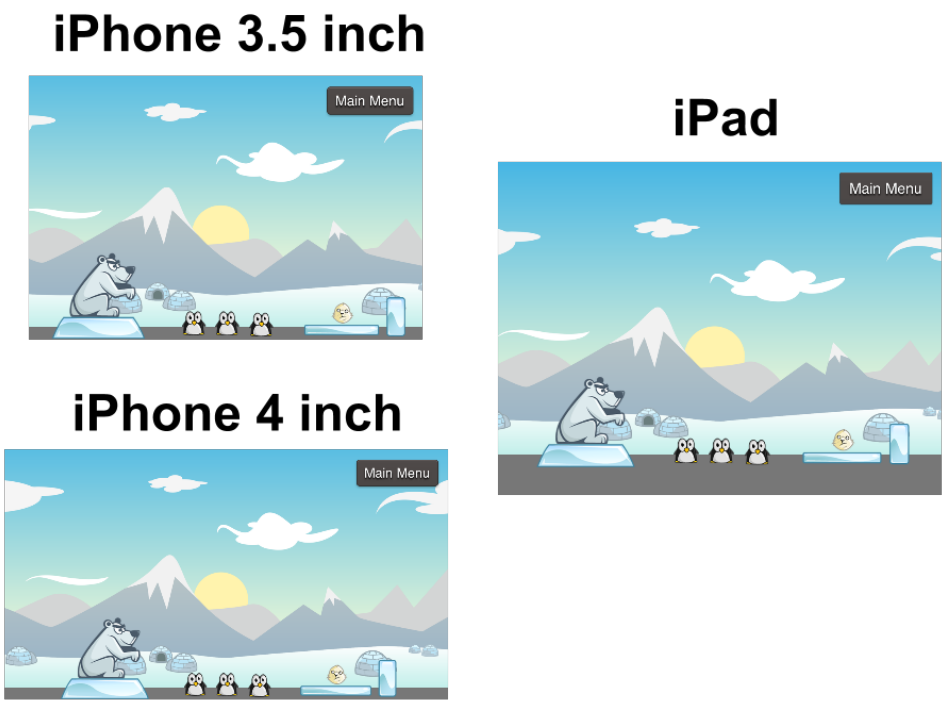
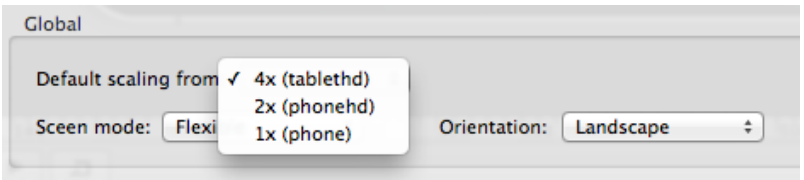


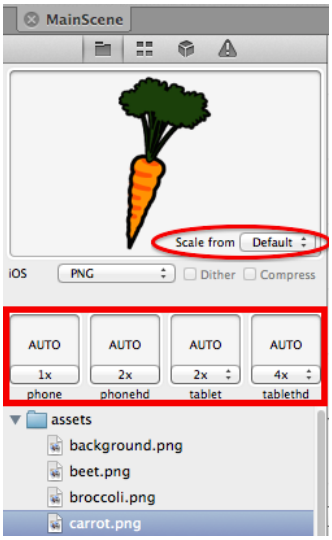
Figure 2.1: From our tutorial *Dynamic Layouts with SpriteBuilder and Cocos2D*

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:

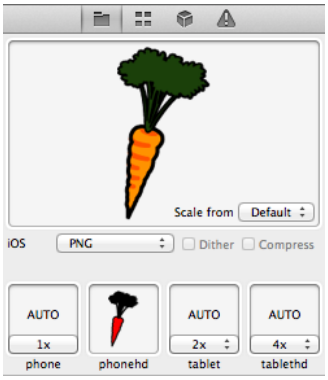
Device	Default Resolution Type	Size on Screen (points)	Size in Pixels
iPhone	1x	50x50	50x50
iPhone Retina	2x	50x50	100x100
iPad	2x	100x100	100x100
iPad Retina	4x	100x100	200x200

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

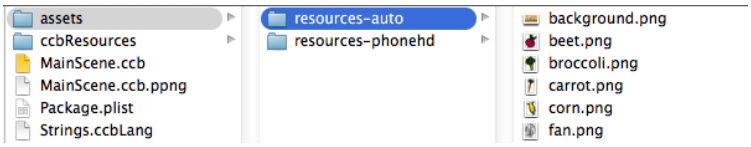
You can not only change the scaling option for an asset on different devices, you can even use an entirely different image for a certain resolution. You can do that by dragging an image **that is currently not part of the SpriteBuilder project** from Finder into one of the four boxes below the asset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

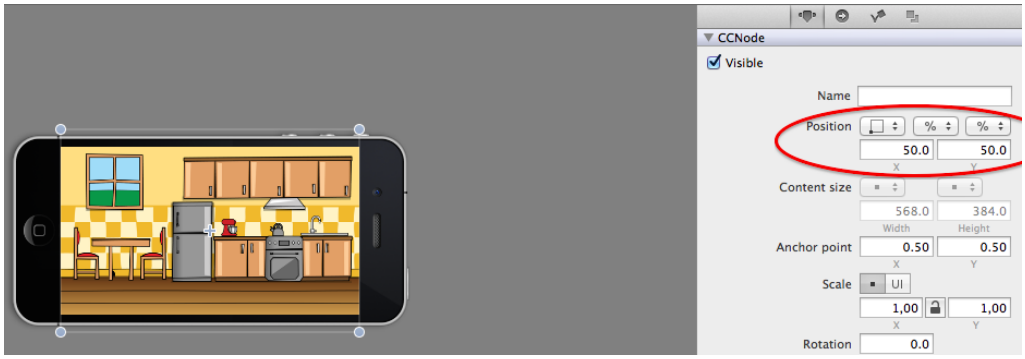
In Cocos2D a class called `CCFileUtils` is responsible for loading the correct images for the current device during runtime. SpriteBuilder uses a special configuration of `CCFileUtils` that is set up in `[CCBReader configureCC-FileUtils]`.

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.

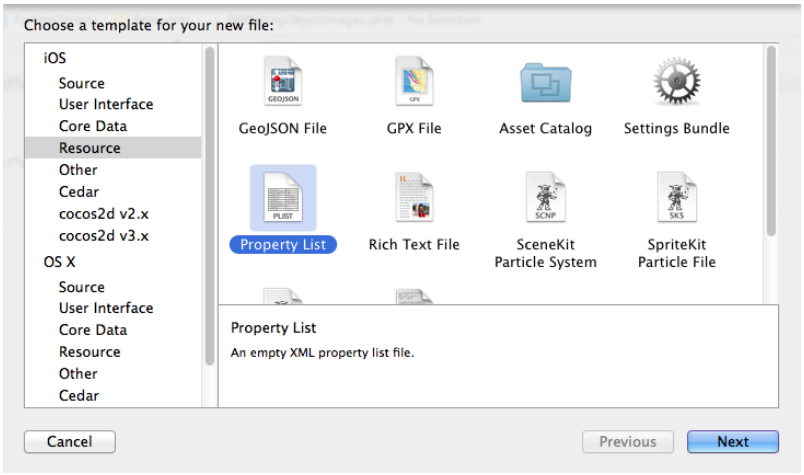
Note that the code will not compile at this point. Since `type` isn't declared as an optional value, Swift requires us to provide an initializer that sets this value. We will fix this in the next chapter.

2.4.2 Choose an asset for a falling object

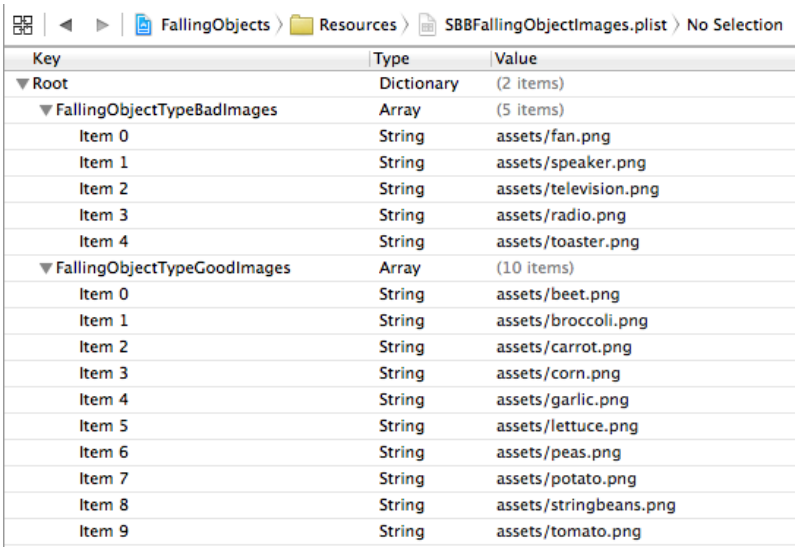
We want the game to spawn entirely random falling objects. As you remember we have a couple of assets for both types of objects. Whenever we spawn an object we will need to choose a random asset, based on the object type. A good place to implement this functionality is directly in the `FallingObject` 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:



FallingObjects › Resources › SBBFallingObjectImages.plist › No Selection

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)
  }
```

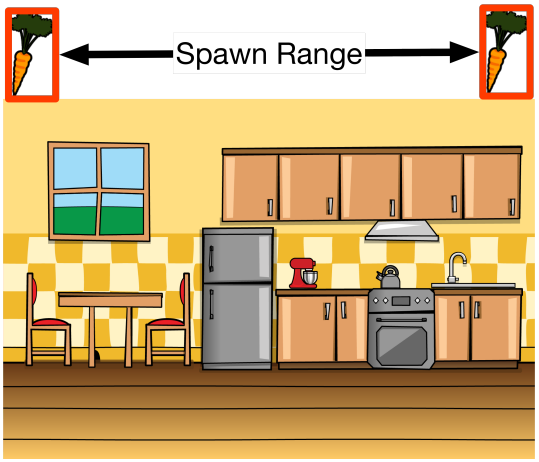
Remember, *onEnterTransitionDidFinish* is called as soon as the presentation transition is completed and the current scene is fully visible. This is when we want to kick of the spawning mechanism. All we need to provide is a *selector*, which simply means a method name as a string, and a frequency at 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.

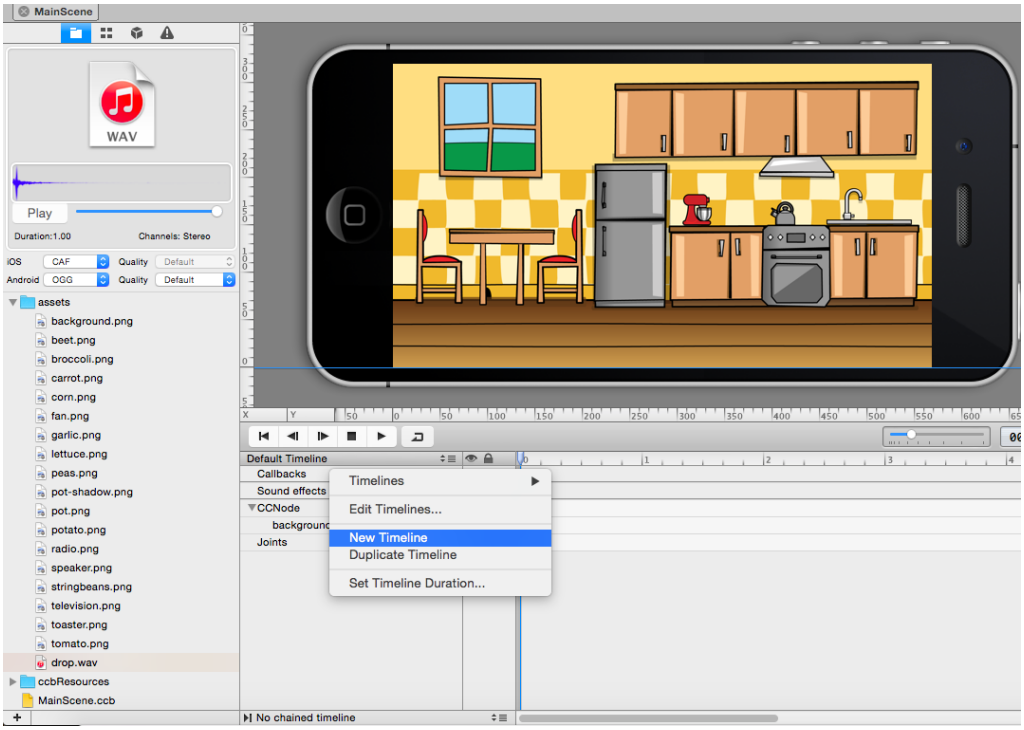


Figure 2.2: WAV files can be added by dragging them to the resource pane

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).

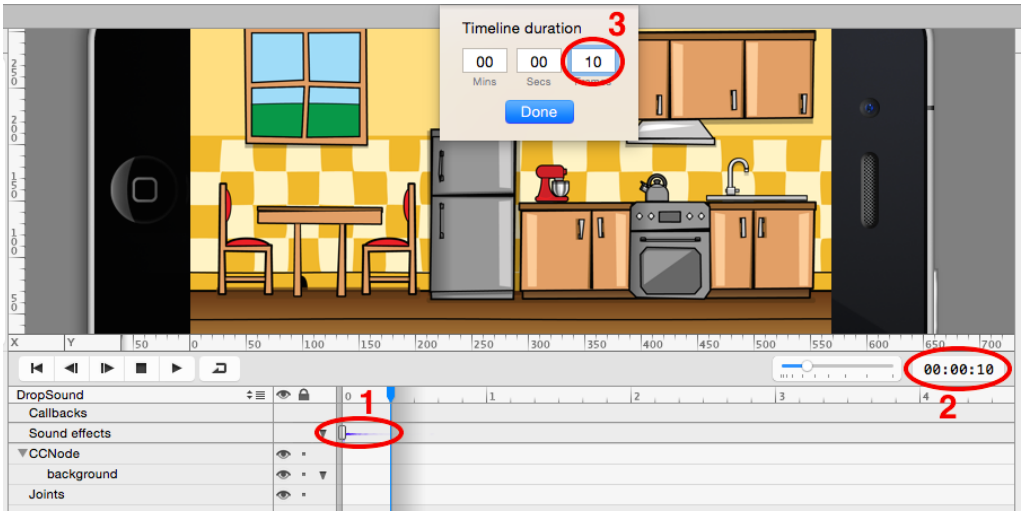


Figure 2.3: (1) Audio file added to timeline
(2) Button to change timeline duration
(3) Timeline duration dialog

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:

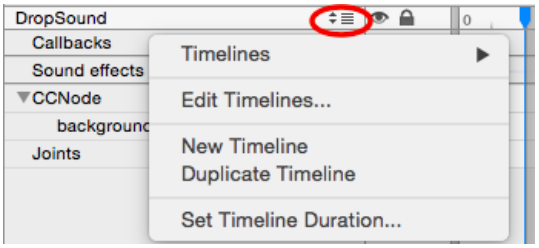


Figure 2.4: *Edit Timelines...* allows you to change the name of a timeline

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.

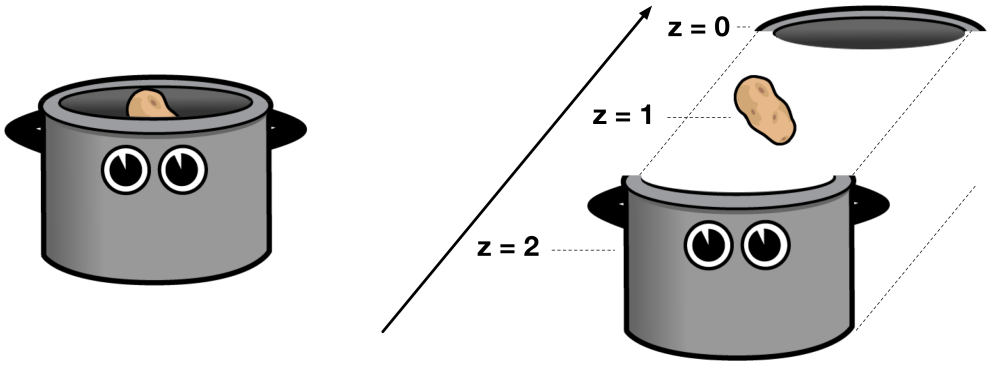


Figure 3.1: Left: Objects on different Layers, Right: How the Z-Order influences on which Layer a node is rendered

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(s` 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:

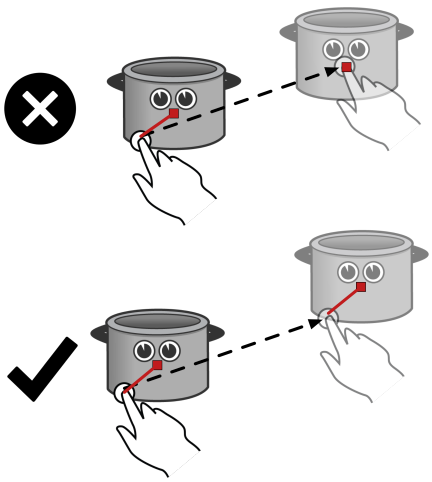


Figure 3.3: *Top Image:* incorrect implementation, object jumps to touch position. *Bottom Image:* correct implementation, touch offset is maintained while dragging the object.

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:

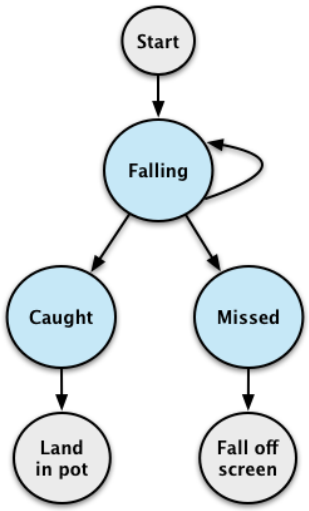


Figure 4.1: Objects start in falling state, then they end up caught or missed

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

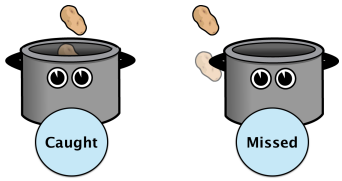


Figure 4.2: Caught objects fall into the pot, missed objects fall behind

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:

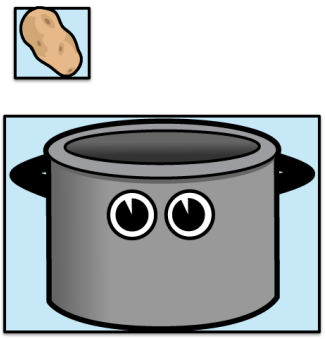


Figure 4.3: The pot frame is too large to use it for collision detection

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*.

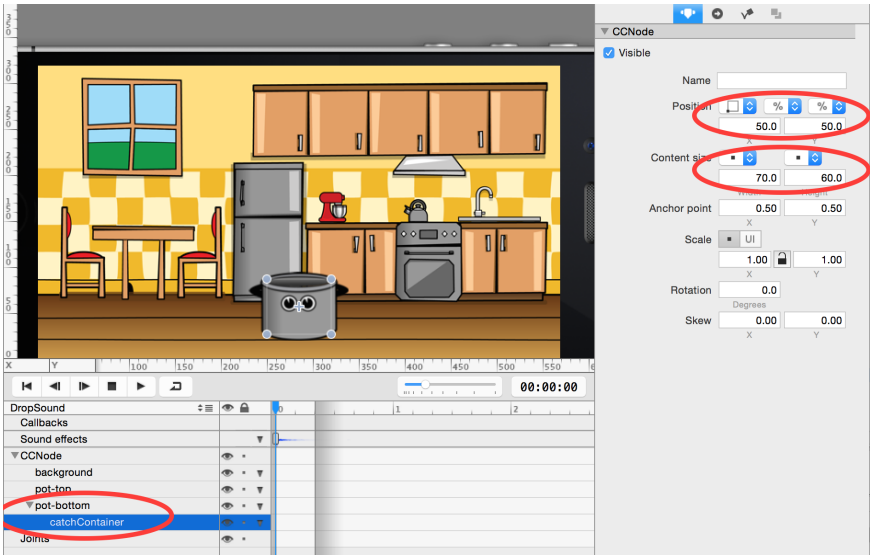


Figure 4.4: The size and position of this container will determine when objects are caught / missed

Add a plain *Node* from the node library and add it as a child to *pot-bottom* node, as highlighted in the screenshot above. A short reminder: the easiest way to do this, is dragging the node from the node library into the timeline and dropping it on top of *pot-bottom* node.

Because we want to reference this catch container in code you will need to set a code connection, too. Set the target to *Doc root var* and call the variable `catchContainer`.

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:

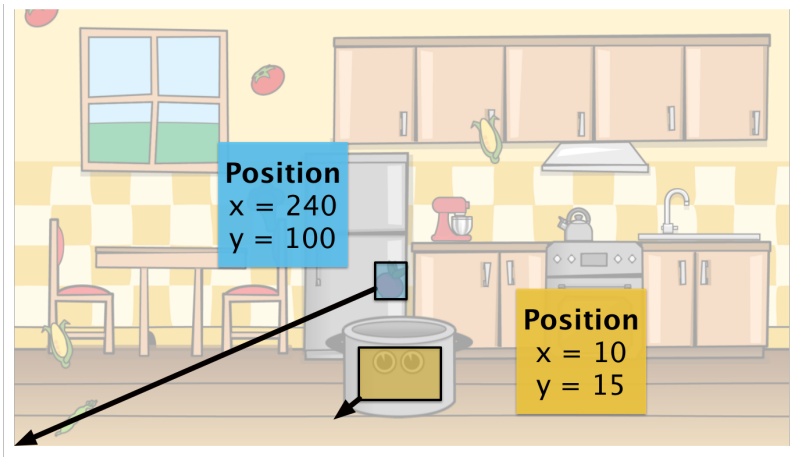


Figure 4.5: Even though the two nodes illustrated above are close to each other, their position values are entirely different, since they are placed relative to different parents

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*

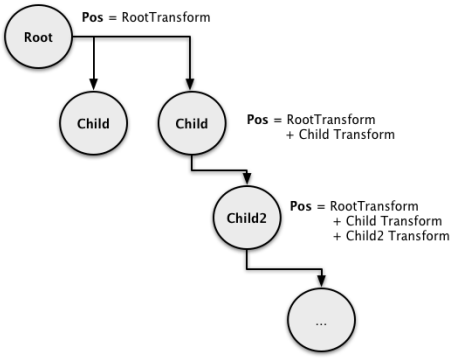
node's (local) space coordinates into the parent's space coordinates.

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()
```

```
        let fallingObjectIndex = find(fallingObjects,
            fallingObject)!
        fallingObjects.removeAtIndex(fallingObjectIndex)
      }
    }
```

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
  }
...
```

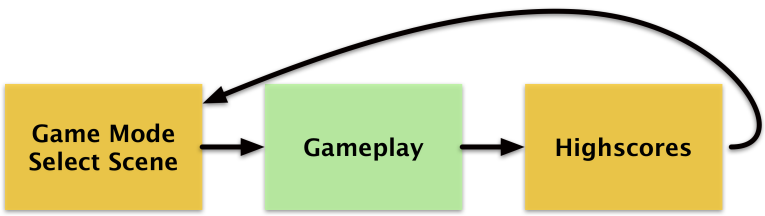
Great! With this tweak we have completed a significant portion of the core gameplay. We have completed what we call the *core mechanic*. A player can drag the catching pot across the screen and collect items. To turn this mechanic into a game we will need to add some rules and game modes on top. In the next chapter we will make this step and finish our first game. When we're done it will include two game modes and highscore system (using Apple's GameCenter).

5 User Interfaces

So far we have worked hard on the core mechanic of our game. Another important aspect are the screens and components that wrap this mechanic. In this chapter you will learn how to implement menus, popups and other user interface elements in Cocos2D.

Cocos2D provides its own set of basic UI components, such as buttons and labels. For most games these components are sufficient. However, in this chapter we also learn how to integrate Apple's UI framework *UIKit*. Knowing that is important for integrating many core Apple API's. As a specific example, we will add integrate the Game Center framework in this chapter.

By the end of this chapter we will have a fully functional game! Here's the basic screen flow our game will have:



Throughout this chapter you will not only learn how to build user interfaces with Cocos2D and SpriteBuilder, you will also learn how to structure this game to support two different gameplay modes. We will implement and *endless* and a *timed* gameplay mode, each with

a different set of rules and behaviors.

Let's start out by adding the game mode selection scene!

5.1 Adding a game mode selection scene

We will now change the screen flow of our existing game. Instead of diving into the gameplay directly the user will see a game mode selection scene when starting the game.

The game mode selection scene will allow the user to swipe to switch between the endless and timed game mode. Luckily Cocos2D provides a component called `CCScrollView` that implements most of the functionality that we need for that scene.

5.1.1 Setting the up the Start Scene

> Open the SpriteBuilder project and create a new File (File -> New -> File...). Name the new file *StartScene* and select Scene as the type.

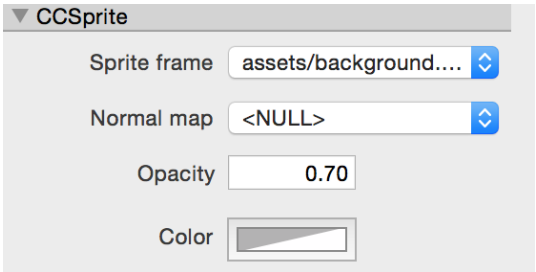
We will create a game mode select scene that smoothly transitions into the gameplay. To accomplish that we'll use the same background image for this scene as for the actual gameplay.

> Drag the image *backround.png* image onto the stage; it becomes the first child of the root node of our new *StartScene*.

The background image should have exactly the same settings as in *MainScene* so that it fills the entire scene.

> Select the background sprite in the timeline and apply the following steps:
>
> 1. Set the position type for X to *percentage of parent container*
> 2. Set the X position to *50*
> 3. Set the position type for Y to *percentage of parent container*
> 4. Set the Y position to *50*

Now the background image should fill the entire background. We will be presenting some information in front of that background. To make that information stand out more we will dim the background a little bit by turning down its opacity. Since the fill color behind the background image is black a lower opacity will result in a darker image.

> Select the background sprite in the timeline. Set the opacity in the property inspector to *0.7*:
>
> 
>

Next, we are going to add a label with an instruction for the player. A label is a simple UI component that can display text. When building games with Cocos2D we want to place the most UI components relative to screen edges. Using this approach the UI will still look good when the game runs on a device with a different screen size. Here's a little illustration:

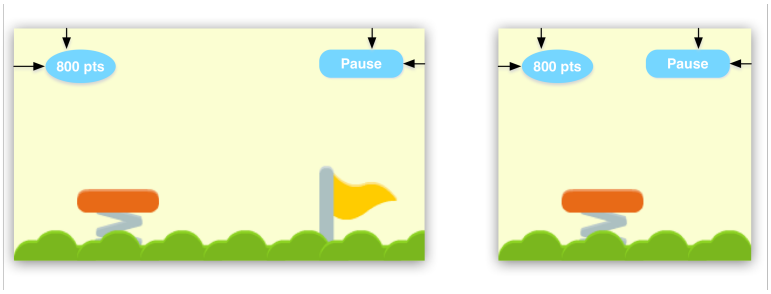


Figure 5.1: UI elements should be placed relative to screen edges to preserve their position on different screen sizes

Throughout this chapter we will use Cocos2D's reference corner feature to accomplish resizable user interfaces.

> Drag a *CCLabelTTF* from the node library *onto* the background sprite, so that it becomes a child of it. Set the position up as following:
>
> 1. Set the position to be relative from the *Top-left*.
> 2. Set the position type for X to *percentage of parent container*
> 3. Set the X position to *50*
> 4. Set the Y position to *80*
>
> Set the label text to: *Choose your game mode:*. We also want to change the font and appearance of this label a little:
>
> 1. As font name choose: *Optima-Bold*
> 2. As font size choose: *40*
> 3. Set the draw color to *black*

4. Set the outline color to *white*
5. Set the outline width to *6*

After setting the label up your start scene should look as following:

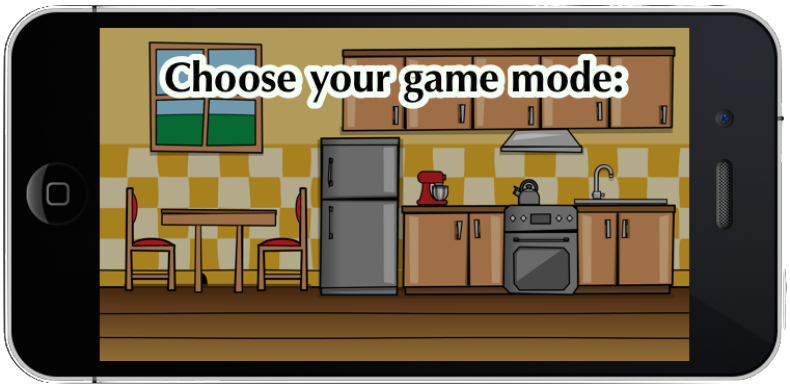


Figure 5.2: UI elements should be placed relative to screen edges to preserve their position on different screen sizes

There's a lot more work left to do! As mentioned earlier we will create a scroll view that let's the user swipe between two different game modes. Every scroll view has a content node. That content node is larger than the size of the scroll view and the scroll view can be used to view different parts of this content node. Here's an illustration:

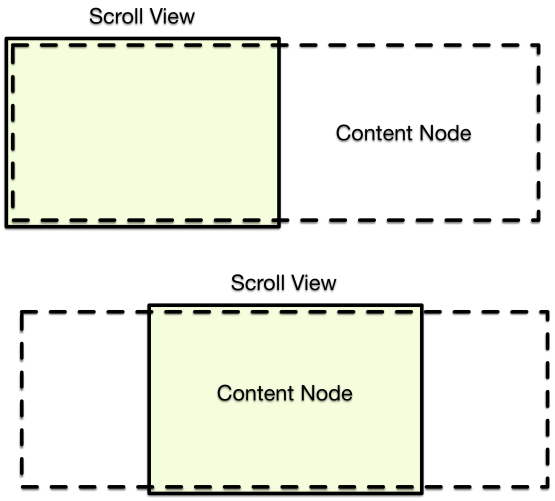


Figure 5.3: The scroll view can present different portions of its larger content node. The user can change the displayed portion by swiping.

Our next step will be creating this content node. In general scroll views allow users to scroll to any arbitrary position within the scroll view's content node. In our specific example we would like to change this behavior. We only want the user to select between two different game modes, each of these game modes will be represented by a full screen node. It wouldn't make sense to allow the user to scroll half way between the endless and timed game mode. For this specific case the scroll view provides an option called *Paging enabled*. If you want to use a scroll view with paging your content node needs to have a size that is a multiple of the scroll view's size. In our particular example the content node for our scroll view will be twice as wide as the scroll view itself, resulting in a scroll view with two pages. When paging is enabled, the scroll view will always snap to one of the two pages as soon as a user stops scrolling. If this sounds a little bit too abstract for you, it should become clearer as we implement and use the scroll view.

5.1.2 Creating the content node for the scroll view

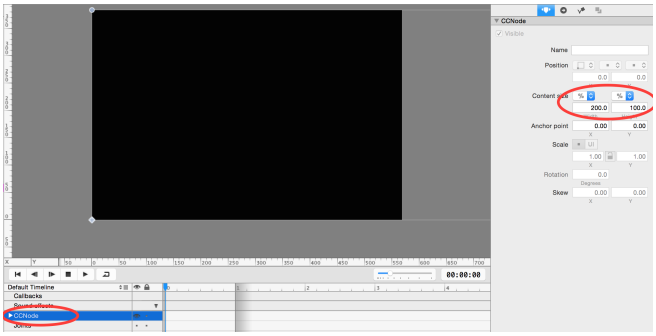
First we need to create the content view. We will set it up in a separate CCB File file.

Create a new CCB File called *GameModeSelectLayer* and choose its type to be a *Layer*.

Why are we using a Layer to create the scroll view content?

A short refresher on the different CCB File types: Scenes are used to represent full screen content. Sprites are used for simple Sprite Images. Nodes are used for node compositions that don't have a specific content size. Layers are used to create content within a stage that has a fixed size. Layers are typically used for popups or scroll view content nodes because we want to layout the content based on a container size. We can still use relative positioning within layers. When the root node of the layer CCB File gets added to another node its size will be determined and all relative layouts will be applied accordingly. We'll use relative positioning for the content node we are about to create.

Select the root node of *GameModeSelectLayer.ccb* and choose the width and height to be in percentage of the parent container. Choose 100% for the height and 200% for the width of the node:



We want the scroll view to contain two pages, that means the content node needs to be exactly double as wide as the scroll view. Because we are setting up the size of the root node of this CCB File in percentage of the parent container, its actual size will only be determined when it is added to a scroll view. This is a great example of dynamic layouts; our scroll view could have any arbitrary size and this content node would always be exactly twice as wide.

Inside of this root node we are going to place the content for our two different pages. To provide a clean structure we will create one container node for each page.

> Add a CCNode as a child to the root node. Name this child *endless-mode* by selecting the node in the timeline and hitting the return key. Set the content type of the node to be in percentage of the parent container. Set the width to 50% and the height to 100%. The container for the left page is set up.

> Add a second CCNode as a child to the root node. Name this child *timed-mode*. Set the content type of the node to be in percentage of the parent container. Set the width to 50% and the height to 100%. Additionaly, set the x position to be in percentage of the parent container and set the value to 50%.

Now we have containers for each page set up. We are going to fill them with labels that describe the game mode represented on each page and an arrow indicating that there is another game mode available by swiping across the screen. This is what the completed content node will look like:

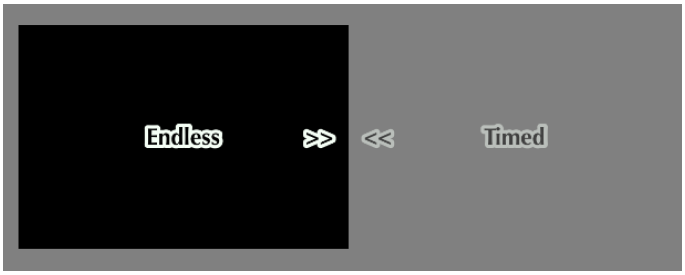
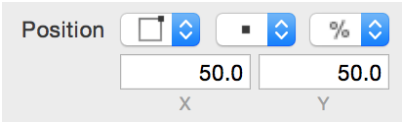


Figure 5.4: The completed content node by the end of this section.

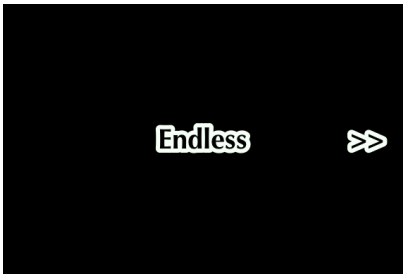
First, let's add the labels for the endless mode.

> Drag a CCLabelTTF from the node library onto the *endless-mode* node. Center the label within its parent node by choosing a position type of percentage of parent container. Then set x and y position to 50%. This label should look the same as the *Choose your game mode* label on the start scene. Apply the following settings:
>
> 1. Set the label text to *Endless*
> 2. As font name choose: *Optima-Bold*
> 3. As font size choose: *40*
> 4. Set the draw color to *black*
> 5. Set the outline color to *white*
> 6. Set the outline width to *6*

Next, let's add the arrow on the right side that will indicate that the player can switch to the timed game mode.

> Drag a CCLabelTTF from the node library onto the *endless-mode* node. Apply the following settings:
>
> 1. Set the label text to »
> 2. As font name choose: *Optima-Bold*
> 3. As font size choose: *40*
> 4. Set the draw color to *black*
> 5. Set the outline color to *white*
> 6. Set the outline width to *6*
> 7. Set the position up as following:
>
> 
>

Now your stage should look like this:



We're going to add a little visual detail to this game mode selection layer. The arrows indicating the other available shall blink. This can be easily accomplished using SpriteBuilder's timeline feature.

Set the timeline duration to 1 second:

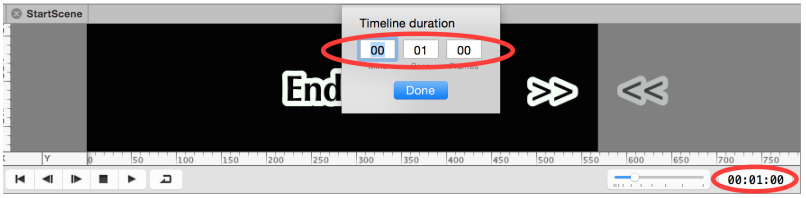


Figure 5.5: Changing the timeline duration

Now we are going to use three *opacity* keyframes to create the blinking animation.

Select the label with the arrows in the timeline. Then create three keyframes by hitting the *O* (like in opacity) key on your keyboard. Alternatively you can create keyframes through the top bar menu: *Animation -> Insert Keyframe... -> Opacity*. Place the first keyframe at 0 seconds the second one at 0.5 seconds and the third one at 1 second. You can choose the exact position of a keyframe by dragging the timeline ruler to the according position:

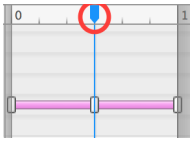


Figure 5.6: Drag the timeline ruler to select a frame at which you want to create the keyframe

Now we can set different opacity values for each of these keyframes and SpriteBuilder will create smooth animations between them. There are two ways to set a specific value for a keyframe. You can select the keyframe and change the relevant property in the property inspector in the right panel of SpriteBuilder. The easier way however is to double-click onto a keyframe. That will bring up a small popup in which you can modify the relevant values:

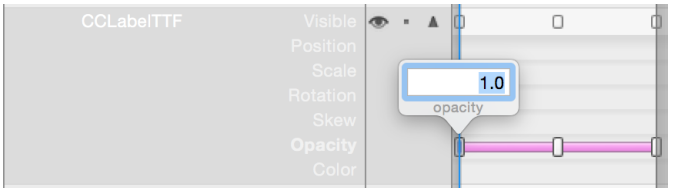


Figure 5.7: Double-click onto keyframes to modify their values

> Set the opacity in the first keyframe to 0. Set the opacity to 1 in the second keyframe. For the third keyframe set the opacity to 1 again.

Now the arrow will appear for half a second and then disappear for another half a second. We don't want this animation to be over after 1 second. Instead we want to loop it forever. We can do so by *chaining* the timeline to itself. In SpriteBuilder timelines can be chained to each other. That means that you can define that another timeline should run after the current timeline is completed. If you use this feature to chain a timeline to itself you have an endlessly running timeline animation!

- Chain the default timeline to itself as shown below:

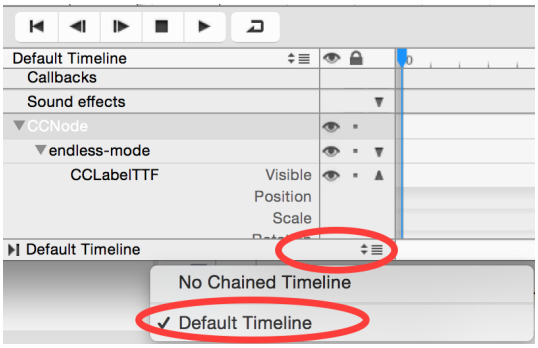


Figure 5.8: Double-click onto keyframes to modify their values

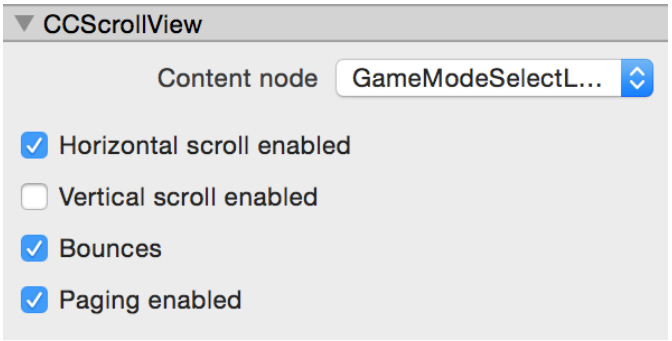
Looping animations in SpriteBuilder

Animations that are set up with a chained timeline will loop endlessly when your game is running on a simulator or phone. In SpriteBuilder itself the animation will only run once. If you want to preview what your animation will look like when it is looped, you need to use the following control in the timeline playback panel:

Note that this control will only affect your previewed animation in SpriteBuilder. Not the actual animation running in your game.

Now we are finished setting up one of the two game modes. Setting up the node for the timed game mode involves exactly the same steps as you have seen just now. The only difference is the arrow label. The arrows should be pointing to the left and the arrow should be positioned from the left edge of the *timed-mode* node. I will leave this as an exercise to you. Remember that you can always check the solution on GitHub if you get

stuck. Once you have set up the node for the second game mode come back and we'll integrate this game mode selection layer into the start scene.

> Switch to the *StartScene.ccb* file and drag a *CCScrollView* from the node library onto the *background* node. Set the position to 0,0. The scroll view shall cover the entire screen, so set the size type to *percentage of parent container* and choose 100% for the width and the height of the scroll view. Set up the scroll view specific settings in the property inspector as follows:
>
> 
>

Let's discuss the scroll view settings briefly. The most important property is the *Content node* property. Here you can choose a CCB File that will be displayed inside of the scroll view. We choose the `GameModeSelectLayer.ccb` that we just created. We check *Horizontal scroll enabled* because the user shall only be able to scroll left and right, not up or down. We discussed the option *paging enabled* briefly at the beginning of this section. With this setting activated the scroll view will always snap to one of the game modes and will not allow the user to stop scrolling in the middle of two game modes.

Great! At this point the set up of our start scene is almost complete.

5.1.3 Finishing up the game mode selection scene

As a last step we will implement the actual selection of one of the two game modes. So far we have a scroll view that will allow users to switch between the game modes but we don't have a mechanism to select one of the two and start a game.

We will add a *start* button to *StartScene.ccb* that will allow users to confirm a selected game mode and start the game. Once we have the button set up we will add an animated transition from this game mode selection screen to the gameplay scene. That transition will be triggered as soon as the player taps the start button.

Let's start by adding a plain button. It is going to be positioned towards the bottom of the screen, below the label that shows the selected game mode.

> Open *StartScene.ccb* and drag a *Button* from the node library to the stage, make sure it is a child of the *background* node. Set the button up as following:
>
> 1. Set the *Position Reference Corner* to *Bottom-left*
> 2. Set the x position to be *In percent of parent container*
> 3. Set the x position to *50* to center the node
> 4. Set the y position to *80*
> 5. Set the *Preferred size* to *100.0, 40.0*
> 6. Set the *title* to *Start!*
> 7. Set the *Font name* to *Optima-Bold*
> 8. Set the *Font size* to *17.00*

Let's discuss some of these properties we just set up. As a short reminder - in almost

all cases we want to position UI elements relative to the screen corner which they are closest to. This will result in the best behavior when the game runs on different screen sizes. Therefore we choose the *Bottom-Left* corner as reference corner (technically we could also choose the *Bottom-Right*, since we are centering the button horizontally this wouldn't make any difference).

Another interesting property is the *Preferred Size*. Buttons automatically resize to be large enough surround their content (in this case the button text). If we want a button to appear larger than necessary we can set the *Preferred Size* property. In this example we make the button a little bit larger than required to fit the text *Start!*.

We also need to set up some code connections. When the user taps the button we want to start the transition. And there's another little feature that's important. We only want to activate the *Start!* button when the user has selected one of the two game modes. If the user is currently scrolling between two screen modes it shouldn't be possible to start the game. Otherwise it could be pretty unclear to the user which game mode has been selected. We're going to solve this issue by deactivating the button when a user is scrolling between game modes.

> Select the *Start!* button and open the *Code Connections* tab. Set up a code connection with the *Doc root var* and call it *playButton*.

We are going to use this code connection to active and deactivate the start button. Towards the beginning of this book we have discussed how we can connect method calls to button taps (1.5.1). Now we are going to use this functionality for the start button.

> Set the *selector* (method name) to *playButtonPressed* and choose the *target* to be *Document root*.

As soon as the button is clicked the `playButtonPressed` will be called on the class of the root node. Right now the root node has no class set up. Let's change that.

> Select the root node (top most node in the timeline) of *StartScene.ccb*, then open the code connections tab. Inside of the code connections tab set the *Custom class* to *StartScene*.

We'll need one more code connection for this scene - a connection for the scroll view. Later, when we add some code to this scene you will see that we need to connect to the scroll view to be informed when the scroll view starts/stops scrolling. Whenever that happens we need to deactivate and activate the start button accordingly.

> Select the scroll view from the timeline and open the code connection tab. Set up a code connection to *Doc root var* and name that connection *scrollView*.

Now we have all the code connections set up. Before we add some code to make this scene work we will work through one last step in SpriteBuilder - creating a nicely animated transition from the selection scene to the gameplay scene.

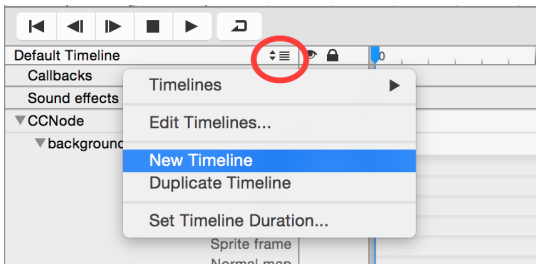
5.1.4 Adding a fancy transition animation

This is what our transition shall look like:

Figure 5.9: By moving UI elements out of the screen and brightening the scene up we transition from the start scene to the gameplay scene

This transition is a nice example of how menus can be integrated into games seamlessly. Luckily creating this animation is pretty straightforward, using SpriteBuilder's timeline.

There are a bunch of steps involved, but none of them are complicated. First, we'll need to create a new timeline. The default timeline runs automatically as soon as the scene becomes visible - the animation we are about to build, in contrast, shall only run when the user has selected a game mode. Therefore we need to create new timeline which we can run when triggered in code.

> Create a new timeline for our transition animation:
>
> 
>
>
> Next, rename the timeline. You can access the timeline editor through SpriteBuilder's menu (*Animation -> Edit Timelines...*). Change the timeline name to *StartGameplay*.

We want the transition to be pretty fast, so let's set the timeline duration to 1 second.

> Set timeline duration to 1s. If you forgot how to change the timeline duration you can skim back a few pages to figure 5.5.

Next, we are going to set up keyframes for all of the UI elements that we want to move off the screen. Additionally we will fade in the background sprite. Make sure to follow the instructions exactly!

> 1. Select the *background* sprite
> 2. Create an *Opacity Keyframe* at 0 seconds and set the opacity value to *0.7*

3. Create a second *Opacity Keyframe* at 1 second and set the opacity to *1.0*
4. Select the *CCButton*
5. Create a *Position Keyframe* at 0 seconds, leave the position unchanged
6. Create a second *Position Keyframe* at 1 second and set the position to *(50%, -400 points)*
7. Select the *CCScrollView*
8. Create a *Position Keyframe* at 0 seconds, leave the position unchanged
9. Create a second *Position Keyframe* at 1 second and set the position to *(0, -400)*
10. Select the *CCLabelTTF*
11. Create a *Position Keyframe* at 0 seconds, leave the position unchanged
12. Create a second *Position Keyframe* at 1 second and set the position to *(50%, -50 points)*

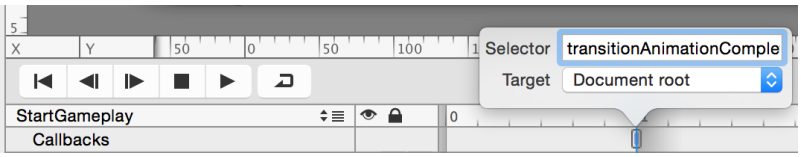
Great! If you want you can test the animation with the SpriteBuilder timeline playback feature.

There's one last step left in SpriteBuilder before we move to Xcode and implement the actual transition between start scene and gameplay. As soon as the animation completes, that means all UI elements have moved off the screen and the background is brightened up completely, we want to switch to the gameplay scene. Switching scenes has to be implemented in code. This means that we need a callback in code that gets triggered as soon as this animation completes.

SpriteBuilder and Cocos2D provide three different ways to implement this:

- Provide a completion block to the animation manager of a scene. This completion block will be called whenever a timeline animation completes (`setCompletedAnimationCallback`)
- Implement a delegate method that gets called by the animation manager when a timeline animation completes (`completedAnimationSequenceNamed:`)
- Set up a callback method within a SpriteBuilder timeline animation

For this section I want to go with the last option. The ability to call methods as part of timeline animations can be useful in many situations, so I want to use it as early as possible!

> To add a callback method to a timeline animation you need to *Option-Key + Click* into the *Callbacks* line of the timeline editor:
>
> 
>
>
> Place that callback at 1 second. You can also choose the *target* and *selector* for this callback. Select *Document root* as target and *transitionAnimationComplete* as selector.

Great! This was quite a lot of work, but now we have a game mode selection scene (including a scroll view) and a great transition into the gameplay scene. Now it's time to switch back to code and implement the scene transition.

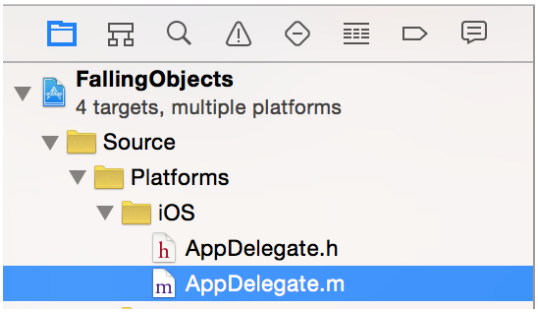
5.1.5 Implementing the game mode selection

Publish the SpriteBuilder project and switch to the Xcode project.

The first change we need to make to the Xcode project, is setting up which scene gets presented when our game starts. By default it's the *MainScene*. However, we have created a new *StartScene* and want that one to be the first scene of the game.

We can change this setting in the *AppDelegate* of the project.

Open *AppDelegate.m* in *Source/Platforms/iOS/*:



The `AppDelegate` of Cocos2D is written in Objective-C. If you don't know Objective-C, don't worry, the change we need to make is trivial.

When a Cocos2D game starts, the `startScene` method of the `AppDelegate` gets called. That method is responsible for loading and returning the start scene of the game. Let's modify it to load `StartScene` instead of `MainScene`.

Modify the `startScene` method of *AppDelegate.m* to look as following:

```
- (CCScene*) startScene
{
    return [CCBReader loadAsScene:@"StartScene"];
}
```

Great! Now our new start scene will be presented when our game starts. Note that the game won't run at this point. We need to create the `StartScene` class. We're already referencing this class from the SpriteBuilder project, but we haven't created it yet.

> Create a new Swift file. Call it `StartScene`. Set up the basic `StartScene` class like this:
>
> ```
> class StartScene: CCNode {
>
>   weak var scrollView: CCScrollView!
>   weak var playButton: CCButton!
>   var selectedGameMode: MainScene.GameMode = .Endless
>
> }
> ```

The first two variables are necessary for the code connections that we set up in our SpriteBuilder project. The third variable stores which game mode the player has currently selected. Since the game mode is a choice between multiple options, an enum is a great way to model this. The enum referenced here (`MainScene.GameMode`) does not exist yet. We need to add it to the `MainScene` class.

> Open *MainScene.swift* and add the following lines:
>
> ```
> var gameMode:GameMode?
>
> enum GameMode: Int {
>   case Endless
>   case Timed
>  }
> ```

The first line is a property that stores the game mode of the current game. Later we will use that property to apply different rules to the gameplay and display different score information. Once the user has selected a game mode, the `StartScene` will set this property on the `MainScene`.

We've also added an enum definition. The `GameMode` enum has two different states, one for each game mode. For now we will only implement an endless and a timed game mode. As we've done earlier, we are associating *raw values* with this enum by adding the `Int` type after the enum name. This means that the `Endless` value corresponds to 0 and the `Timed` value corresponds to 1.

Now we can switch back to working on our new start scene. There are three features we need to implement in *StartScene.swift*. We need to keep track of the movement of the scroll view. When the user scrolls to one of the two pages of the scroll view, we need to remember which game mode has been chosen. Further, as discussed earlier, we need to deactivate the *Start!* button while the user scrolls. Finally, we need to trigger a transition to the gameplay scene (`MainScene`) as soon as a user taps the start button. As part of that transition we need to inform `MainScene` which game mode was selected. Let's start with the scroll view related code.

Implementing a scroll view delegate

If you have written code for the iOS platform before, you are very familiar with the principle of delegation. Since this book does not necessarily require previous iOS app development knowledge, I will give you a brief introduction.

Delegation is often used in iOS frameworks when an object A wants to provide an object B with the ability to modify object A's behavior or to listen to changes of object A. However, object A does not need to know the exact type of object B. Instead, all of the methods that object B can implement are described in a protocol. Any object can implement this protocol and be the *delegate* of object A.

Delegate pattern

Delegation is used so frequently as part of iOS development that Apple has a short chapter about it in it's documentation: https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html

The `CCScrollView` in Cocos2D also uses the delegate pattern. This is what the (Objective-C) protocol looks like:

```
@protocol CCScrollViewDelegate <NSObject>

@optional
- (void)scrollViewDidScroll:(CCScrollView *)scrollView;
- (void)scrollViewWillBeginDragging:(CCScrollView *)scrollView;
- (void)scrollViewDidEndDragging:(CCScrollView * )scrollView
   willDecelerate:(BOOL)decelerate;
- (void)scrollViewWillBeginDecelerating:(CCScrollView *)scrollView;
- (void)scrollViewDidEndDecelerating:(CCScrollView *)scrollView;

@end
```

There are a total of five methods that we can implement. The optional keyword marks a section of the protocol in which all listed methods are not required to be implemented. In this specific protocol *all* methods are optional. We want to be informed when the scroll view starts and ends scrolling. There are two methods that get called in these cases: `scrollViewWillBeginDragging` and `scrollViewDidEndDecelerating`. Let's implement these methods and become the delegate of our scroll view.

The first step, is setting ourselves up as the delegate of the scroll view. We can set ourselves as the delegate as soon as the scene is entirely loaded (when `didLoadFromCCB` is called).

Add the following implementation of `didLoadFromCCB` to `MainScene`:

```
func didLoadFromCCB() {
  scrollView.delegate = self
}
```

Now the scroll view knows about us and will call all the methods of the `CCScrollViewDelegate` protocol that we implement.

Now we need to add the protocol implementation. In Swift, it is common to implement protocols in class extensions. Class extensions allow us to add functionality to a class outside of its original definition. Moving all protocol implementations into separate class extensions is a nice way of keeping our code organized.

The implementation of our two protocol methods is pretty simple. When the user starts scrolling we deactivate the start button. When the user ends scrolling, we activate the start button and remember the selected game mode.

Add the following protocol implementation to *StartScene.swift*. It is **important** that the class extension is **not** part of the class, but placed after the closing curly brackets of the class definition:

```
class StartScene: CCNode {
  ...
}

extension StartScene: CCScrollViewDelegate {

  func scrollViewWillBeginDragging(scrollView: CCScrollView) {
    playButton.enabled = false
  }

  func scrollViewDidEndDecelerating(scrollView: CCScrollView) {
    playButton.enabled = true
```

```
        selectedGameMode = MainScene.GameMode(rawValue: Int(
            scrollView.horizontalPage))!
    }

}
```

Activating and deactivating the button is very simple, `CCButton` provides a property for it. Choosing the game mode based on the selected scroll view page is similarly straightforward. `CCScrollView` provides a `horizontalPage` property that allows use to read which page the user has currently scrolled to. We use that value to select the according game mode. In order to create an enum value from a number we need to use the `rawValue:` initializer (remember that 0 corresponds to the endless game mode and 1 to the timed game mode).

Our work with the scroll view is completed. Next, let's implement the callback method for the button interaction. As soon as the play button is tapped we want to play the transition animation that we created in SpriteBuilder. Additionally, we should deactivate the scroll view. That way users cannot modify the selected game mode while the scene is in transition.

> Implement the `playButtonPressed` method that we've set up in SpriteBuilder inside of `StartScene`. This should be implemented as part of the core class, not of the extension:
>
> ```
> func playButtonPressed() {
>     scrollView.userInteractionEnabled = false
>     animationManager.runAnimationsForSequenceNamed("
>         StartGameplay")
> }
> ```

First, we deactivate user interaction on the scroll view. Whichever game mode has been

selected by the user before hitting the play button will stay selected throughout the animation. Then we run the actual animation. Remember that all timelines created in SpriteBuilder can be referenced in code by using their name.

Now there's a last method left to be implemented. We have added a callback to the timeline animation that gets called when the transition animation is completed. In the implementation of that method we should perform the actual scene transition. The animation that we are running only moves the UI elements off the screen. But as you might remember transition between scenes can only be implemented in code.

Transitions between scenes can also be animated. For the transition from the start scene to the gameplay scene we will use a cross fade transition. Add the end of our timeline animation the start scene is entirely empty. Then we use a cross fade animation to transition to the gameplay scene where the pot will be displayed. The cross fade animation will make the pot appear slowly before the actual game starts.

> Add the following implementation for the timeline callback that we set up in SpriteBuilder to `MainScene`:
>
> ```
> func transitionAnimationComplete() {
>   let scene = CCBReader.loadAsScene("MainScene")
>   let gameplay = scene.children[0] as MainScene
>   gameplay.gameMode = selectedGameMode
>   let transition = CCTransition(crossFadeWithDuration: 0.7)
>   CCDirector.sharedDirector().replaceScene(scene, withTransition
>       : transition)
> }
> ```

Great! Now our game mode selection screen is complete; including an awesome transition to the gameplay scene. You can run the project and try it out.

Next, we are going to implement our two different game modes. We will start by creating two different UIs for the two game modes. In the endless game mode the player will loose

health for dropping items or collecting incorrect items and will gain health for collecting correct items. For the endless mode we need to display a health bar. For the timed mode we need to display a counter that shows the remaining time and a score label.

5.2 Implementing multiple game modes

Lets implement the two different game modes. We're going to start with setting up the UI in for each game mode in SpriteBuilder. Then we will come up with a way to implement the two game modes without duplicating code - this is a great use case for exploring some potential design patterns for game development.

5.2.1 Adding UIs for different game modes

Open your SpriteBuilder project. We're going to create two separate CCB Files, one for the UI elements of each game mode.

UI for the timed game mode

> Create a new CCB File of type *Layer* and name it *TimedModeUI*. Select the root node of the new CCB File and change the size type to be *percentage of parent container* for both, width and height. Set the values for width and height to *100%*.

We will place all UI elements relative to the edges of the screen. That way the UI will look good on any given screen size. That's why we want the root node to take 100% the size of its parent container. If we instead would use a fixed size for this layer our UI would not respond to multiple screen sizes.

The UI for the timed game mode is not too complicated, it consists of two separate labels. We want the style of these labels to be consistent with the labels that we used on the start scene. The easiest way to do accomplish this is to actually copy the label in SpriteBuilder.

> Open *StartScene.ccb* and select the *CCLabelTTF* from the timeline. Now select *Edit -> Copy* from the SpriteBuilder menu (or use the shortkey: *CMD+C*).
>
> Next, open *TimedModeUI.ccb* and select *Edit -> Paste* from the SpriteBuilder menu (or use: *CMD+V*).

Now you should have an exact copy of the start scene label in our new CCB File. Now we can make a small tweak to this label and lay it out correctly.

> Apply the following changes to the label:
>
> 1. Select the *Position Reference Corner* to be the top left corner
> 2. Change the position to *(20,20)*
> 3. Change the anchor point to *(0.0, 1.0)*
> 4. Change the label text to *Time: 0*
> 5. Change the font size to *30*
> 6. Set up a code connection to the **Owner var** (not the *Doc root var*, this is very important!) and name it *timeLabel*

Now the label should be nicely positioned in the top left corner! We will not be creating a custom class for this UI layer, since it doesn't have any behavior associated with it. Instead we will later assign the label code connections to the classes that contain the actual gameplay logic. That's why it's important to set up the code connection with the *Owner var* instead of the *Doc root var*.

Next, let's create the points label. Since it will look almost identical to the time label we should save some time and just copy and modify the label again instead of starting with a new label from scratch. Note however, that you should be concentrated when using this approach. When you copy a node all of it's properties are copied (including code connections). If you aren't careful you might end up with hard to debug issues (e.g. because two nodes attempt to use the same code connection variable).

> Copy the time label that you just created it and paste it. Change the pasted label as following:
>
> 1. Select the *Position Reference Corner* to be the top right corner
> 2. Change the anchor point to *(1.0, 1.0)*
> 3. Change the label text to *Points: 0*
> 4. Set up a code connection to the **Owner var** and name it *pointsLabel*

Great! The UI for the timed game mode is completed. It should look as following:

UI for the endless game mode

Now lets create the second UI. We'll start of by creating a new CCB File.

> Create a new CCB File of type *Layer* and name it *EndlessModeUI*. Select the root node of the new CCB File and change the size type to be *percentage of parent container* for both, width and height. Set the values for width and height to *100%*.

Next, let's add a label that displays the amount of seconds a player has survived. Once again we can copy the existing label from the *TimeModeUI.ccb* file since we want both labels to look identical.

> Open *TimeModeUI.ccb* and copy the label in the **top left corner**. Then open *EndlessModeUI.ccb* and paste the label. Select the pasted label and modify it as following:
>
> 1. Change the label text to *Survived: 0*
> 2. Set up a code connection to the **Owner var** and name it *survivedLabel*

Besides this label we will also need to display a health bar in the endless game mode. The easiest way to implement a health bar in Cocos2D is taking a plain `CCColorNode` and scaling it depending on the current health level of the player.

> Drag a *Color Node* from the node library to the stage. Set it up as following:
>
> 1. Select the *Position Reference Corner* to be the top right corner
> 2. Change the position to *(20,20)*
> 3. Change the anchor point to *(1.0, 1.0)*
> 4. Change the node color to *green* (pick any color you enjoy)

5. Change the font size to *30*

6. Set up a code connection to the **Owner var** (not the *Doc root var*, this is very important!) and name it *healthBar*

Now the UI for the endless mode should look like this:

We're done with the UI setup. To make these score displays actually show up as part of the game, we'll now move to implementing the two game modes in code.

5.2.2 Implement game logic for different modes

We need to change multiple parts of our game to support two different game modes.Depending on which game mode a player selects, we want to display a different score board and also implement a different set of rules.

Before we dive into coding, let's try to figure out what we'll need to implement:

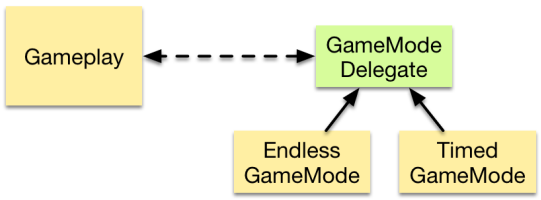
- When we start a game, the `MainScene` class needs to know which game mode a user has selected

- `MainScene` shouldn't know anything about the rules of the selected game mode. That way we could add game modes in the future without increasing the complexity of `MainScene`
- Every game mode should know about its rules (when does a player earn / loose points, when does a game end)
- Every game mode should know which score board entries it has and it should be responsible for updating them based on the current state of the game

We should try to come up with a solution that lets us add more game modes as easily as possible. One modular way of implementing this is setting up different game modes as individual classes. These classes can be referenced by the `MainScene` class. In this scenario, the `MainScene` class remains responsible for the core of the game, it spawns falling objects (though, even this could be moved into the gameplay classes eventually) and lets them fall to the ground. It handles collision detection and determines if an object was caught or dropped.

The consequences of catching or dropping objects however, are not implemented in the `Gameplay` class itself. Instead, they are implemented by the game modes. This means that the `Gameplay` class needs to inform the game mode when a significant event occurs. The game mode itself can then decide how this event impacts the gameplay.

Here's a diagram that illustrates our solution:



Defining a protocol for game modes

Let's start implementing this design. The first step is defining a protocol that can be implemented by all game modes. In the diagram above I've highlighted the protocol in green.

> Create a new Swift file in Xcode and call it *GameModeDelegate.swift*. Then add the following protocol definition to it:

```
typealias GameOver = Bool

protocol GameModeDelegate: class {
  var userInterface: CCNode! { get }

  func gameplay(mainScene:MainScene, droppedFallingObject:
     FallingObject)
  func gameplay(mainScene:MainScene, caughtFallingObject:
     FallingObject)
  func gameplayStep(mainScene:MainScene, delta: CCTime) ->
     GameOver
}
```

Note that I have omitted the source code documentation of this protocol for the sake of brevity. The protocol we've just defined will be implemented by multiple classes, it's important to document its methods and properties for developers that want to add game modes in future. You can take a look at the source code for this chapter on GitHub() to read the documentation.

Source code documentation in Swift

Many details about source code documentation in Swift are still up in the air. NSHipster provides a great article that describes the currently supported documentation style: http://nshipster.com/swift-documentation/.

Let's discuss this protocol in detail. The first interesting aspect is a `typealias` statement before the protocol declaration. Using typealiases is simple in Swift, and it's a nice way of better capturing the semantics of a type. Returning a type of `GameOver` is easier to understand the returning a `Bool`.

The second thing to note is that we have defined the protocol as a *class-only* protocol. We can do that by adding the `class` keyword to the protocol's inheritance list. Later you'll see that any type that implements `GameModeDelegate` is required to be a reference type, because we need to pass references to it to Cocos2D.

We start our protocol by requiring a property: `userInterface`. This property shall provide access to the game mode's user interface. The `MainScene` class can use this property to access the UI for the currently active game mode. We only expose as a getter as part of the protocol, the UI cannot be changed from outside of the game mode class. Each game mode knows the CCB File of its UI and loads it as soon as the game mode is created.

Next, we define two methods that get called when objects get dropped or caught. Both methods receive a reference to the `MainScene`. This is pretty common when using the delegate pattern. Theoretically a game mode instance could be the delegate of multiple `MainScene` instances. By passing the the `MainScene` in which the event occurred to the delegate the delegate can perform different actions based on which `MainScene` called the method. The second parameter sent to both methods is the object which has been caught or dropped. The class implementing the `GameModeDelegate` can use this information to determine whether points should be added or subtracted.

The last method(`gameplayStep:`) fulfills two purposes. Firstly, it allows game modes to hook into the update method of `MainScene`. That is necessary for any time based actions, e.g. capturing the total time that has passed since the beginning of the game. The second purpose is fulfilled by the `GameOver` return value. The return value allows the game mode to tell the `MainScene` that the game is over. All game modes will use this method to implement the game over condition.

Now that our protocol is defined we can implement the two game modes.

Implementing the timed game mode

Let's start by implementing the timed game mode. Since we thought about the software design upfront and even defined a protocol for game modes, the implementation itself is not too complicated.

> Create a new Swift file and name it *TimedGameMode*. Add the following class definition to the new file:
>
> ```
> @objc(TimedGameMode)
> class TimedGameMode: GameModeDelegate {
>   var timeLabel: CCLabelTTF!
>   var pointsLabel: CCLabelTTF!
> }
> ```

There's a lot going on in these few lines. Firstly, we need the `@objc` annotation to make this class visible to Cocos2D. Cocos2D is written in Objective-C and can only see classes that are subclasses of `NSObject` or that have the `@objc` annotation. So far all of our classes have been subclasses of Cocos2D classes, and all of them in turn are derived from `NSObject`, therefore this annotation wasn't necessary. The `TimedGameMode` class is our first class that is not a subclass of an Objective-C class.

In the second line we declare that our class conforms to the `GameModeDelegate` protocol. We also declare to properties for the code connections that we set up in the CCB File for the timed game mode. We have access to two different labels, one displays the player's the other one displays the time that's left.

We'll need a whole set of additional variables. We need a `userInterface` variable to conform to the `GameModeDelegate`, we also need variables to store the amount of points

the player has scored and the time that is left in the current game.

> Add the following properties and property observers to the `TimedGameMode` class:

```
let minPoints = 0
let minTime = 0.0

private(set) var userInterface: CCNode!

private var time: CCTime = 10 {
  didSet {
    updateTimeDisplay(time)
  }
}

private var points: Int = 0 {
  didSet {
    updatePointsDisplay(points)
  }
}
```

The first two constants are used to define the bottom line for time and points in this game mode. In almost all cases using constants should be preferred over using numbers directly in code. By defining constants our intentions are obvious to other developers.

Further, we add the `userInterface` variable. This variable will store the loaded CCB File that belongs to the timed game mode. It is also required to conform with the `GameModeDelegate` protocol.

Next, we declare and define the `time` property that stores the time that is left during the current game. For testing purposes our games only last 10 seconds. We also add a property observer. When the time value changes we call the (yet to be implemented) `updateTimeDisplay:` method, that method updates the label that displays the leftover

time.

We essentially do the same for the `points` property.

Next, let's implement the two helper methods that update the time and point labels.

> Add the following two methods to `TimedGameMode`:
>
> ```
> func updatePointsDisplay(points: Int) {
>   pointsLabel.string = "Points: \(points)"
> }
>
> func updateTimeDisplay(time: CCTime) {
>   timeLabel.string = "Time: \(Int(time))"
> }
> ```

These methods are very simple. We only put this code into separate methods because we need the functionality in multiple places, as you'll see shortly.

Next, before we implement the methods defined in our protocol, let's tackle the initializer of this class. The main task the the initializer needs to perform is loading its UI from a CCB File.

> Add the following initializer:
>
> ```
> init() {
>   userInterface = CCBReader.load("TimedModeUI", owner:self)
>   updatePointsDisplay(points)
>   updateTimeDisplay(time)
> }
> ```

In the first line we load the *TimedModeUI* CCB File. We store the loaded node hierarchy in the `userInterface` property, that way it can be accessed by `MainScene`. We also call our

to label helper methods, so that the labels display the correct initial values for time and points.

Now we can move on to the core of this class. The methods that are required by the `GameModeDelegate` protocol.

Let's start with the `gameplay(mainScene:, droppedFallingObject:)` method. Now it's time to make some decisions on the rules of the timed game mode. I suggest that we subtract 1 point when the player drops an object that should be caught. Because we don't want to be too mean we set the lowest possible score to 0.

> Add the following method to `TimedGameMode`:

```
func gameplay(mainScene:MainScene, droppedFallingObject:
    FallingObject) {
  if (droppedFallingObject.type == .Good) {
    points = max(points - 1, minPoints)
  }
}
```

If the object that has been dropped is a `.Good` object, we subtract one point. Using Swift's `max` function we make sure that the total score never drops below `minPoints`, which we have defined as 0.

Next, we'll implement the method that gets called when objects are caught. Once again time to decide on some rules. I propose to add a point when a good object is caught and subtract a point when a bad object is caught. You can obviously feel free to use different values!

> Add the method for caught objects to `TimedGameMode`:

```
func gameplay(mainScene:MainScene, caughtFallingObject:
    FallingObject) {
  switch (caughtFallingObject.type) {
```

```
    case .Bad:
      points = max(points - 1, minPoints)
    case .Good:
      points += 1
    }
  }
```

Since we are checking for multiple possible values of an enum using a switch statement results in nicely readable code. Besides the switch statement the implementation is pretty straightforward.

Now there's only one last method to implement: `gameplayStep(mainScene:, delta:)` We need to do two things in the implementation of this method:

1. Subtract passed time from time that is left over
2. Check if time reached minimum time (0.0) and return `true` if that's the case

Add the following implementation of the step method to `TimedGameMode`:

```
func gameplayStep(mainScene: MainScene, delta: CCTime) ->
    GameOver {
  time -= delta
  return !(time > minTime)
}
```

First, we subtract `delta` from the remaining game time. Next, we check if the total time is over or not and return a `GameOver` value based on that.

Congratulations! We have fully implemented our first game mode. Now, let's implement the endless game mode.

Implementing the endless game mode

The procedure for implementing the endless game mode will be very similar to the timed one: Setting up code connections, implementing game mode rules and updating the scoreboard.

Let's start with setting up the basic class and code connections.

> Create a new Swift file in Xcode and name it *EndlessGameMode*. Then add the following class definition and properties:
>
> ```
> @objc(EndlessGameMode)
> class EndlessGameMode: GameModeDelegate {
>   var healthBar: CCNode!
>   var survivedLabel: CCLabelTTF!
> }
> ```

As I said, this is pretty similar to the timed game mode. We need the `@objc` annotation because this class does not inherit from any Objective-C object. We declare two properties for the two UI elements in the *EndlessModeUI* CCB File.

In the endless game mode we will have two game parameters that we want to keep track off: the current *health* of the player and the *survivalTime* of the current game. In endless game mode the player has the goal of surviving as long as possible. Instead of gaining points for catching objects the player regains some health. Whenever the player catches an incorrect object or drops a good object she looses points. We'll need properties to keep track of these values.

Additionally we'll need a property that stores the UI node to conform with the `GameModeDelegate` protocol.

Add the following properties and property observers to `EndlessGameMode`:

```
private(set) var userInterface: CCNode!
private let minHealth = 0
private let maxHealth = 10

private var health:Int = 10 {
  didSet {
    let newScale = Float(health) / Float(maxHealth)
    let scaleAction = CCActionScaleTo.actionWithDuration(0.2,
       scaleX: newScale, scaleY: 1.0) as CCAction

    healthBar.stopAllActions()
    healthBar.runAction(scaleAction)
  }
}

private var survivalTime: CCTime = 0.0 {
  didSet {
    survivedLabel.string = "Survived: \(Int(survivalTime))"
  }
}
```

We start of with a variable that stores the user interface for this game mode. Then we define two constants for the upper and lower bounds of the player's health.

Next, we define the actual `health` variable and initialize it to *10*. We also add a property observer to `health`. Whenever the value changes we need to rescale the green health bar to reflect the new value. We calculate the new scale by dividing the current health by the maximum health. Instead of simply assigning new scale we apply the change as an animation. It's these small details that make games look more polished, and in Cocos2D they are easy to implement. Cocos2D provides a scale action (`CCActionScaleTo`), we define it to run in 0.2 seconds. Before we start the action we ensure that all other actions

are stopped. Since the action takes 0.2 seconds to complete it is likely that we start a new scale action while an old one is still in progress. Two actions fighting against each other would result in serious glitches, therefore it's important to call `stopAllActions()`.

Finally, we add a property that stores the survival time. We once again add a property observer, in this case we update the time label whenever the survival time changes.

Next, we need to add the initializer that will load the user interface.

> Add the following init method to `EndlessGameMode`:
>
> ```
> init() {
>   userInterface = CCBReader.load("EndlessModeUI", owner:self)
> }
> ```

Now all that is left is implementing the game rules as part of the three methods defined in the `GameModeDelegate` protocol.

When the player drops a `.Good` object, we subtract a point.

> Add the following method to `EndlessGameMode`:
>
> ```
> func gameplay(mainScene:MainScene, droppedFallingObject:
>    FallingObject) {
>   if (droppedFallingObject.type == .Good) {
>     health = max(health - 1, minHealth)
>   }
> }
> ```

We once again use the `max` function to ensure that the health does not drop below our defined minimum.

Next, let's implement the caught method. When the user catches a `.Good` object he regains a point, when he catches a `.Bad` object he looses one.

Add the caught method:

```
func gameplay(mainScene:MainScene, caughtFallingObject:
    FallingObject) {
  switch (caughtFallingObject.type) {
  case .Bad:
    health = max(health - 1, minHealth)
  case .Good:
    health = min(health + 1, maxHealth)
  }
}
```

The last method we need to implement is the `gameplayStep(mainScene:, delta:)` method. We need to keep track of the time that the user has survived, we can do that by adding up all the delta time frames that we receive in this method. Additionally we need to determine the game over situation. In this game mode the player loses as soon as his health drops to the minimum health.

Add the `gameplayStep(mainScene:, delta:)` to `EndlessGameMode`:

```
func gameplayStep(mainScene: MainScene, delta: CCTime) ->
    GameOver {
  survivalTime += delta
  return (health <= minHealth)
}
```

And this completes our second game mode. As you can see there weren't too many surprises after implementing the `TimedGameMode`.

Before we can test these two game modes in our app we need to integrate them into the `MainScene` class.

5.2.3 Connecting the game modes to the main scene

The last important step in implementing the game modes is connecting them to `MainScene`. Currently the `StartScene` passes the selected game mode to the `MainScene`. The next step for `MainScene` will be creating an instance of a game mode based on this selection.

Additionally we need to modify `MainScene` to call the methods defined in `GameModeDelegate` whenever objects are caught, dropped and when the `update` method is called.

Let's start with instantiating one of our two game modes. In order to do that we need a variable that can store the created game mode and a property observer for the `gameMode` property. Observing the `gameMode` property will allow us to instantiate a game mode object as soon as a game mode has been selected:

> Add a property to store the instantiated game mode:
>
> ```
> var gameModeDelegate:GameModeDelegate?
> ```
>
> Next, add this property definition and observer to `gameMode`:
>
> ```
> var gameMode:GameMode = .Endless {
>   didSet {
>     switch (gameMode) {
>     case .Endless:
>       gameModeDelegate = EndlessGameMode()
>     case .Timed:
>       gameModeDelegate = TimedGameMode()
>     }
>     self.addChild(gameModeDelegate?.userInterface)
>     gameModeDelegate?.userInterface.zOrder = DrawingOrder.
>         ScoreBoard.rawValue
>   }
> }
> ```

Swift requires all non Optional properties to have a value after the initializer of the class is called. Since `gameMode` is is not initialized from the `init` method, but is instead set by an outside caller after initialization we would have to mark it as an Optional type. An alternative to that is assigning an initial value. In this case it makes sense to work with such a default value. If the outside caller does not select a specific game mode, we run the `.Endless` game mode and all of the code in `MainScene` works as expected.

Inside of the property observer we first switch over the potential game modes. Depending on the game mode we instantiate a different game mode class. Next, we add the user interface of the created game mode to main scene. If you are fairly new to Swift you might be surprised by the question mark in `gameModeDelegate?.userInterface`. This pattern is called *optional chaining*. In this specific case the `userInterface` variable is only being accessed if `gameModeDelegate` actually contains a value. More general optional chaining can be used to call methods and access properties on values that might be `nil`, without causing an runtime error.

In the last line of the property observer we set the `zOrder` of the game mode UI to `DrawingOrder.ScoreBoard.rawValue`. You probably remember that we have used an enum to keep track of our drawing order. The UI of the current game mode should be rendered on top of all gameplay elements, so we need a new entry in that enum.

> Extend the drawing order enum to include the case `DrawingOrder.ScoreBoard` that we've just used:
>
> ```
> enum DrawingOrder: Int {
>   case BehindPot
>   case PotTop
>   case FallingObject
>   case PotBottom
>   case ScoreBoard
> }
> ```

Having the scoreboard as the last case (with the highest integer value), means that the game mode UI will be rendered above all other elements in this scene.

Now we have instantiated a game mode - we can move on to the code that will communicate with it. Let's start by calling the `gameplayStep(mainScene:, delta:)` method. This method needs to be called from within the `update:` method of `MainScene`. Additionally to calling the method we need to capture the return value to see wether the current game is over or not.

> Add the following statements to the end of the `update:` method of `MainScene`:
>
> ```
> let gameOver = gameModeDelegate?.gameplayStep(self, delta: delta
>     )
> if let gameOver = gameOver {
>   if (gameOver) {
>     self.gameOver()
>   }
> }
> ```

We call the step method of our delegate, passing a reference to `self` and the `delta` value. We store the boolean result and use it to check whether the game over condition occurred or not. If the game is over we call the `gameOver` method.

We should add the `gameOver` method next. For now all it will do is displaying the start scene of this game. Later on we will display a popup that summarizes the player's results.

> Add the `gameOver()` method to `MainScene`:
>
> ```
> func gameOver() {
>   let startScene = CCBReader.loadAsScene("StartScene")
>   let transition = CCTransition(crossFadeWithDuration: 0.7)
>   CCDirector.sharedDirector().replaceScene(startScene,
>       withTransition: transition)
> }
> ```

All we do is loading the *StartScene* and presenting it with a cross fade transition.

Now all that is left is calling the `GameModeDelegate` when objects have been dropped or caught. Let's begin with dropped objects.

> Extend the `performMissedStep:` method to call the game mode delegate:

```
func performMissedStep(fallingObject:FallingObject) {
  // check if falling object is below the screen boundary
  if (CGRectGetMaxY(fallingObject.boundingBox()) < CGRectGetMinY
     (boundingBox())) {
    gameModeDelegate?.gameplay(self, droppedFallingObject:
       fallingObject)
    ...
  }
}
```

We again use optional chaining to only call the `gameplay(mainScene:, droppedFallingObject:)` method if `gameModeDelegate` actually contains a value.

As a last step implement the same functionality for caught objects.

> Extend `performCaughtStep:` with a call to the game mode delegate:

```
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())) {
    gameModeDelegate?.gameplay(self, caughtFallingObject:
       fallingObject)
    ...
  }
}
```

Now the game modes and all the communication between the selected game mode and `MainScene` should be set up correctly. Finally we can test the game mode selection.

Run the app and select the endless game mode, you should see something similar to the following screen once the game starts:

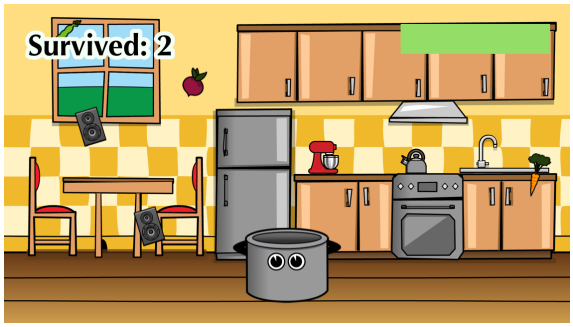


Figure 5.10: The endless game mode features a survival time label and a health bar

You should see the health bar grow and shrink according to the game mode rules that we implemented - finally all the parts have come together.

Currently the automatic transition from the main scene back to the start scene is a little unexpected. It's typical for games to present a summary after each session the player has completed. In the next section we'll add a game over popup to our game.

5.3 Adding a game over popup

Whenever a session ends, either because of a game over situation or because the time has run out, we want to present a popup that shows the results of this session. The popup should also provide easy ways to play the same game mode again and to switch back to the start scene.

By the end of this section we will have built a popup that looks like this:

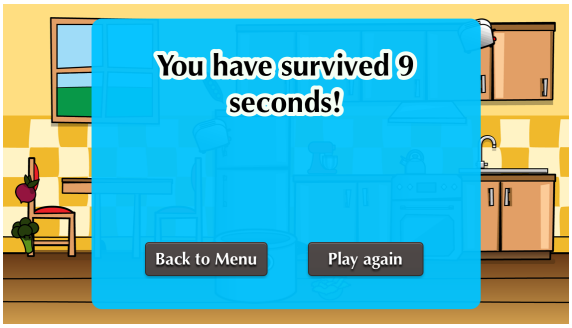


Figure 5.11: A popup at the end of each playing session summarizes the player's results

We will get started by creating this new UI element in SpriteBuilder, then we will work on integrating it in code.

5.3.1 Setting up the popup in SpriteBuilder

We'll start by creating a new CCB File for our popup.

> Create a new CCB File in SpriteBuilder. Select *Layer* as the type. Choose the size to be *(400, 300)* and name the file *GameOverPopup*.

We will center the popup when we present it. Centering is a lot easier when the anchor point of the popup is at (0.5, 0.5).

> Select the root node and change the *Anchor point* to (0.5, 0.5).

Next, we'll add the background for the popup. As you might have seen in figure 5.11 our popup has rounded corners. The easiest way to accomplish rounded corners in Cocos2D

is using a stretchable image. The specific type of image we want to use is called a *9 slice* image. 9 slices images are divided into stretchable and non-stretchable parts. Using 9 slices images we can use small images and scale them to arbitrary dimensions without distorting the non-scalable parts of the image.

Here's an illustration of how a 9 slice image works:

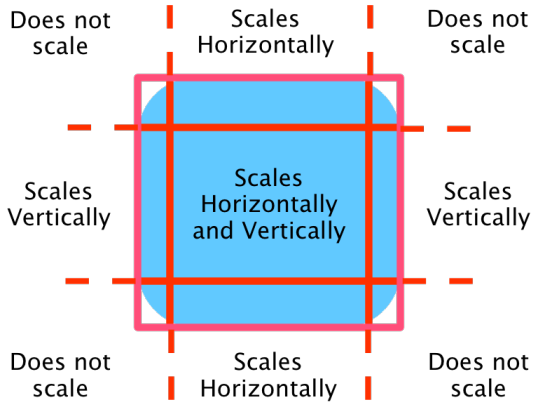


Figure 5.12: A 9 slice image is divided into stretchable and non-stretchable regions

In the example above we have a blue image with rounded corner. If we want to scale that image, it's important that the corners don't get scaled, otherwise they would be distorted. SpriteBuilder allows us to define the scalable area of 9 slice images, that way we can choose a scalable area that is appropriate for each asset. The assets you have downloaded include an image with rounded corners. We'll use that for the background of our popup.

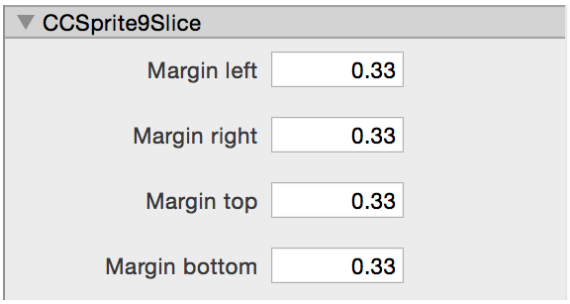
> Drag a *Sprite 9 Slice* from the node library and add it to the root node of *GameOverPopup.ccb*.
>
> Set up the sprite 9 slice as following:

1. Set the position type to *percentage of parent container* for *x* and *y* position
2. Set the position to *(50%, 50%)*
3. Set the content size type to *percentage of parent container* for *width* and *height*
4. Set the width to *(100%, 100%)*
5. Set the anchor point to *(0.5, 0.5)*
6. Set the sprite frame to *assets/popup-background.png*
7. Set the opacity to *0.95*
8. Select a *light blue* color

At this point your stage should look like this:

Note, that the asset *popup-background.png* has a white background. That allows us to create a sprite with rounded corners of any color. You might also have realized that the sprite 9 slice scales correctly without that we hat to define a scalable and non-scalable region.

That's because SpriteBuilder provides default values that work for many sprites. When you've selected a sprite 9 slice you'll see the following entry in the *Item properties* tab:



These values are expressed in percentage of the image size. This means the non-scalable are is defined by 33% of the image size from each edge. These settings work well for many 9 slice images, if you ever run into problems with a 9 slice image you now know where to change these settings.

Now we'll add the label that displays the highscore when the game is over. We'll add this label and the two buttons as a child to the sprite 9 slice because we'll add a little scale up animation to the popup later on. We want the entire popup content to scale with the sprite 9 slice.

> Drag a *Label TTF* to the stage. Drop it onto the *CCSprite9Slice* in the timeline to make the label a child of the sprite.
>
> Set the label up as following:
>
> 1. Set the position reference corner to *Top-left*
> 2. Set the *X* position type to *percentage of parent container*
> 3. Set the position to *(50, 30)*
> 4. Set the label text to *You have survived 10 seconds!*, this is a placeholder text. We'll

set the actual text in dymanically in code

5. Set the font name to *Optima-Bold*
6. Set the font size to *30*
7. Set the size type of the width of the *Dimensions* to *percentage of parent container*. Labels resize dynamically, the *Dimensions* size determines the maximum size for the label
8. Set the the Dimensions to *(95, 0)*
9. Set the horizontal alignment to *Center*
10. Set the draw color to *black*
11. Set the outline color to *white*
12. Set the outline width to *4*
13. Set up a code connection to *Owner var* and name it *gameOverPopUpHighscoreLabel*

By providing the *owner* of the popup with access to the label we avoid providing a custom class for the popup. Instead, whichever class displays this popup can set itself as the owner and modify the popup appropriately. For UI components that don't have any custom behavior I prefer this approach over providing custom classes.

Now we're going to create two buttons, one to play another round of the current game mode, a second one to return to the main menu.

Drag a *Button* to the stage. Drop it onto the *CCSprite9Slice* in the timeline to make the label a child of the sprite.

Configure the button like this:

1. Set the *X* position type to *percentage of parent container*
2. Set the position to *(30, 55)*
3. Set the preferred size to *(120, 40)*. Buttons resize automatically based on their content, the preferred size determines the minimum size of the button
4. Set the title to *Back to Menu*
5. Set the font name to *Optima-Bold*
6. Set up a selector (in the code connections tab). Name it *backToMenu* and set the target to *Owner*.

Just as we let the owner change the text of the label, we will also invoke the button callbacks on the owner - we want to avoid creating a custom class for this simple popup.

You can now copy this button to create the second one.

Copy the button that we just created. Apply the following changes:

1. Change the position reference corner to *Bottom-right*
2. Change the selector to *playAgain*

Now the popup should look exactly as depicted in figure 5.11 at the beginning of this chapter.

We're almost done with designing the popup in SpriteBuilder. However, so far we haven't considered how this popup is going to be presented. It would be weird if it would suddenly appear at the end of the game, without any visual transition. Especially iOS users are used to smooth transitions and UI elements that are presented and dismissed with animations - our game should live up to these standards! SpriteBuilder's timeline

editor makes implementing this a matter of seconds!

To present the popup smoothly we will use a scale animation (very similiar to the animation that iOS uses for its alert views). Here's what it will look like:

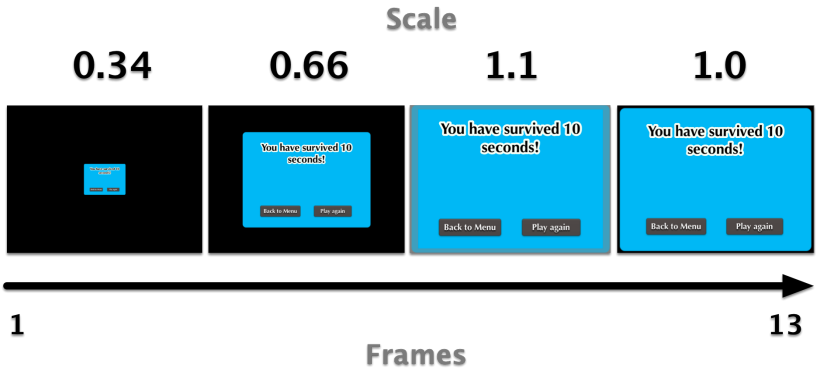


Figure 5.13: We make the popup appear over 13 frames by animating its scale property

The animation will have a length of 13 frames. We will scale the popup from 0 to 1.1 and finally to 1. This approach creates a nice little bounce animation when the popup appears.

Instead of creating a new timeline, we will use the *Default Timeline* that is provided by SpriteBuilder for every CCB File. The default timeline has *autoplay* activated, which means that it runs as soon as the root node of the CCB File is added to an active scene. By coincidence this is exactly the behavior we want: as soon as the popup is added to a scene we want this animation to play once.

Note, that SpriteBuilder does not allow us to add keyframe based animations to the root node, instead we will animate the *CCSprite9Slice* which has all other elements of the popup as its children.

> Select the *CCSprite9Slice* in the timeline. Since this animation is very short, you should use the timeline's maximum zoom level to place the keyframes accurately. Drag the zoom lever in the top right corner all the way to the right:
>
> 
>
>
> Now, create three keyframes by hitting the *S* key, one at *00:00:00*, one at *00:00:09* and a third one at *00:00:13*. Next, set the scale values by double-clicking onto each of the keyframes. As show in figure 5.13 we want the values to be *0*, *1* and *1.1*.

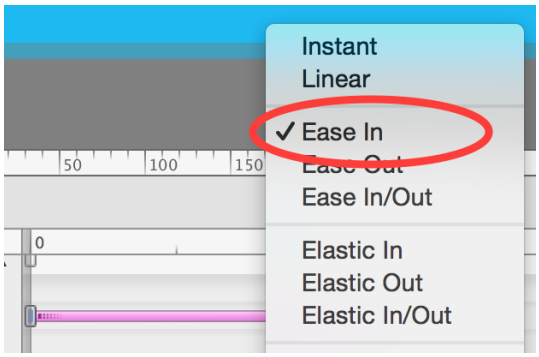
You can now run the animation and you should see a nice bounce effect. If you have ever paid close attention to the animations of system elements on iOS you will realize that something with this animation still doesn't feel exactly right. Why?

As a default setting SpriteBuilder uses linear interpolation between keyframes. This results in animations that move smoothly at a constant speed. However, real life objects seldom move at a constant speed. Instead they accelerate an decelerate. Believe it or not, the difference between a linear animation and an accelerated one is noticeable, even if the entire animation only lasts 13 frames.

Since the developers of Cocos2D are aware of this effect, they have provided us with a whole bunch of different interpolations. Once you have created two keyframes and a pink bar appears between them you can select between all of the variations by right-clicking onto the pink bar.

Many developers have done a good job of visualizing the different interpolation types…

For this animation we are going to use the *Ease In* interpolation.

> Right-click onto the segment between the first and the second keyframe and select *Ease In* as the interpolation function:
>
> 
>

Now you can run the animation again. You should notice that it is looking much better now! These are the small details that catch the player's attention.

Now we are ready to present this popup in code!

5.4 Presenting the popup in code

As always, let's get started by setting up properties for the code connections we've added in SpriteBuilder. The code to present the popup will be part of `MainScene`. In SpriteBuilder we've set up all code connections for the game over popup to be connected to the *Owner*. Since `MainScene` will be the owner of `GameOverPopup` we need to add code connection variables and callback methods there.

Let's first add the property for the label on the game over popup.

Add the following property to *MainScene.swift*:

```
weak var gameOverPopUpHighscoreLabel: CCLabelTTF!
```

This property will allow us to change the presented text on the game over popup from within `MainScene`. Later we'll use it to present the final score at the end of each game.

Our game over popup should be presented above all other content of `MainScene`, so we should add a new entry to the drawing order enum, that places the popup at the highest z-order.

Extend the drawing order enum by adding a case for our new popup:

```
enum DrawingOrder: Int {
  case BehindPot
  case PotTop
  case FallingObject
  case PotBottom
  case ScoreBoard
  case GameOverPopup
}
```

Now we're set up to work on the code that adds the popup to `MainScene` as soons as the game ends. We'll modify the existing `gameOver` function in this step. So far `gameOver` transitions back to the `StartScene` when the game ends, we can remove all of that code for now.

Change the `gameOver` method, so that it no longer switches back to the `StartScene`:

```
func gameOver() {
  userInteractionEnabled = false
  isDraggingPot = false
  presentGameOverPopup()
}
```

In the new implementation of `gameOver`, we're turning of user interaction on `MainScene` by setting the `userInteractionEnabled` property to `false`. This means that `MainScene` will no longer receive touch events. Without this line it would be possible for the user to drag the pot across the screen, even though the game has ended. Typically as soon as a game ends you want to disable all interactions between the player and any game elements - everything else tends to look awkward.

Additionally to disabling user interaction, we also need to set `isDraggingPot` to `false`. Ongoing touch events are not cancelled when user interaction gets disable, so if a player would be dragging the pot when the game over condition is met, they could go one forever. With `isDraggingPot` disabled, the code in `touchMoved` will not be performed and the dragging is interrupted immediately.

Additionally we're calling the `presentGameOverPopup` method that we haven't implemented yet. As you'll see shortly, presenting the popup involves quite a few lines of boilerplate code, so it makes sense to bundle this functionality into a separate method.

Let's implement presentGameOverPopup now. There aren't a lot of new concepts involved in this method, so I'll provide the entire implementation and will discuss the details afterwards.

> Add the following lines to the end of the `MainScene` class, directly before the closing curly braces of the class definition:

```
// MARK: Game over popup
func presentGameOverPopup() {
  let gameOverPopup = CCBReader.load("GameOverPopup", owner:self
      )

  // workaround because CCPositionTypeNormalized cannot be used
      at the moment
  // https://github.com/spritebuilder/SpriteBuilder/issues/1346
```

```
    gameOverPopup.positionType = CCPositionType(
      xUnit: .Normalized,
      yUnit: .Normalized,
      corner: .BottomLeft
    )

    gameOverPopup.position = ccp(0.5, 0.5)
    gameOverPopup.zOrder = DrawingOrder.GameOverPopup.rawValue

    gameOverPopUpHighscoreLabel.string = gameModeDelegate?.
        highscoreMessage()

    addChild(gameOverPopup)
  }
```

In the first line we are loading the *GameOverPopup.ccb* using the `CCBReader`, we are setting `MainScene` as the owner.

Next, we're setting up the position type of the popup. We want the popup to be presented centered on the screen. The easiest way to do that is to use the `Normalized` position type (which is called *Percent of parent container* in SpriteBuilder). We need to configure the position type in code, because SpriteBuilder does not support setting the position type of the root node of a CCB File. If you select the root node of *GameOverPopup.ccb*, you'll see that the control is disabled:

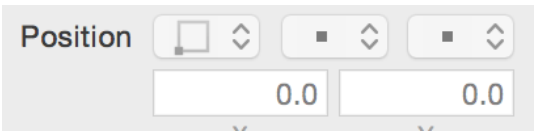


Figure 5.14: It isn't possible to change the position type or position of the root node in SpriteBuilder

When setting the position type in code, we run into a small Swift <-> Objective-C related issue. The way that many Cocos2D constants are defined is incompatible with Swift, these constants are not available to Swift classes. Usually we could simply set the position type to `CCPositionTypeNormalized`, but because of this issue we need to construct the position type manually (if you are interested in the details you can read https://github.com/spritebuilder/SpriteBuilder/issues/1346).

After we've set up the position type, we center the node by setting the position to *(0.5, 0.5)* and we set the `zOrder` to render this popup on top of all other game content.

Then we configure which text is displayed on the popup. We haven't implemented this on the game mode classes yet, but we're adding a `highscoreMessage` function to each game mode that will return a suitable highscore message for the current game state. This way each game mode can present a different message at the end of the game. This design makes it easy to add more game modes in future. We'll discuss the `highScoreMessage` function in detail as soon as we implement it.

In the next line we add the game over popup to the gameplay scene. Adding the popup will trigger it's timeline to start playing. That will scale the popup from 0% to 100% size, with the animation that we have defined in SpriteBuilder.

With this method in place we're getting close to presenting the popup! We'll need to satisfy two more code connections, the callback methods for both buttons on the popup, and we'll need to add the `highScoreMessage` method to our game modes.

Let's start with the button and implement the callback for the *Back to menu* button. All this button does is bring us back to the main scene, so the implementation is very simple.

> Add the `backToMenu` method below the `presentGameOverPopup` method in *MainScene.swift*:
>
> ```
> func backToMenu() {
>   let startScene = CCBReader.loadAsScene("StartScene")
> ```

```
    let transition = CCTransition(crossFadeWithDuration: 0.7)
    CCDirector.sharedDirector().replaceScene(startScene,
        withTransition: transition)
}
```

Nothing special here, this is the code that used to live in the `gameOver` method - when the player hits the *Back to menu* button, we transition back to `StartScene`.

The *Play again* button is a little bit more exciting. This is a feature that you'll want to add to many types of games and the implementation isn't very complicated. The easiest way to restart the game is to create a new instance of the `MainScene` class and replace the currently active scene with that new one. I've often seen beginners in game development write code to reset different state variables in their gameplay scene, in order to start a new match or a new round of the same game. Creating a new instance of the core gameplay class is the simplest and most reliable way of resetting the entire state of the game.

For this specific game there's on small gotcha that we need to avoid. The `MainScene` needs to know which gameplay mode has been selected. When a player hits the *Play again* button, we want to start the *same* game mode that the player has been playing up until then. That means as we create a new instance of `MainScene`, we need to take care of preserving the selected game mode.

With all of this in mind, let's add the `backToMenu` method.

Add the `playAgain` method below the `backToMenu` method:

```
func playAgain() {
    let mainSceneContainer = CCBReader.loadAsScene("MainScene")
    let mainScene = mainSceneContainer.children[0] as MainScene
    mainScene.gameMode = gameMode
    let transition = CCTransition(crossFadeWithDuration: 0.7)
    CCDirector.sharedDirector().replaceScene(mainSceneContainer,
        withTransition: transition)
```

```
}
```

Remember, when we use the `CCBReader.loadAsScene()` method, the entire scene graph from the loaded CCB File is wrapped into a CCScene instance. That is why we need to access the first child of the loaded scene to get a reference to the new `MainScene` instance. With that reference at hand, we set the `gameMode` of the new `MainScene` to be the same as the currently selected game mode. The rest is business as usual, replacing the scene with an animated transition.

And that's all there is; now you know how to add a *play again* feature to your game easily. Next, we'll extend the game modes to provide individual highscore messages.

5.5 Providing a highscore for each game mode

Whenever we want to extend the functionality of our game modes we should start by adding the new features to the `GameModeDelegate` protocol.

Add the following method the `GameModeDelegate` in *GameModeDelegate.swift*:

```
/**
Provides a highscore message for the current game

:returns: Highscore message
*/
func highscoreMessage() -> String
```

Now every game mode will be required to implement a `highscoreMessage` method. Let's go ahead and add them, so we can finally see the popup in action.

Let's start with the `EndlessGameMode`. When the `EndlessGameMode` ends, we want to display how many seconds a player has survived.

> Add the following implementation of `highscoreMessage` to `EndlessGameMode`:
>
> ```
> func highscoreMessage() -> String {
>   let secondsText = Int(survivalTime) == 1 ? "second" : "seconds
>       "
>   return "You have survived \(Int(survivalTime)) \(secondsText)!
>       "
> }
> ```

The most exciting part of this implementation is the ternary operator that we use to determine whether or not we need to pluralize the term `second`.

The implementation of this method inside of `TimedGameMode` looks almost the same:

> Implement the `highscoreMessage` method in `TimedGameMode` as following:
>
> ```
> func highscoreMessage() -> String {
>   let pointsText = points == 1 ? "point" : "points"
>   return "You have scored \(Int(points)) \(pointsText)!"
> }
> ```

Awesome! Now we have all the parts together to actually test our new popup. Run the game on the simulator or on a phone, lose one of the two game modes, and see what happens.

You should see something very similar to this:

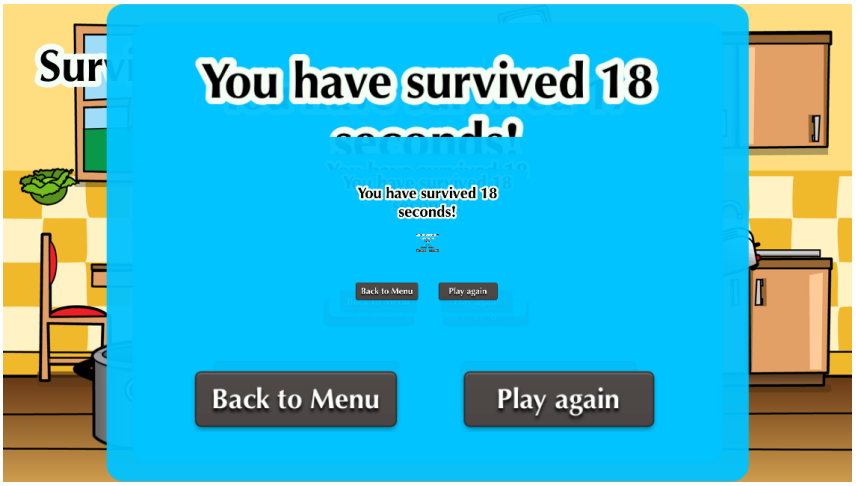


Figure 5.15: An endless amount of popups is filling up the screen (and our main memory)

As soon as the game ends our popup gets presented. But not only once - new popups are added endlessly. We'll look into fixing this in the next section.

5.6 Tweaking the game over popup

The issue we are experiencing is very common among developers that are implementing their first game over popup. What is going wrong?

We are presenting the popup from within the `update:` method, and that method gets called 60 times a second. We are currently not checking whether or not we've already presented the popup. Instead, every single frame the game over condition is met and we present a new game over popup. If you have the patience to wait long enough you will experience a crash due to excessive use of memory.

The fix for this problem is pretty straightforward. We'll add a boolean flag to `MainScene`

that will indicate whether we are in *Game over* state or not. If we are in game over state, we will skip the entire `update:` method. That way we will no longer check if the game over condition is met which means we'll no longer present an endless amount of popups. This approach has another advantage: since we've implemented all object movement inside of the `update:` method, all objects will freeze as soon as the game ends. That's the behavior that most players expect.

Let's add a new property to reflect the state of the game.

> Add the following property to `MainScene`:
>
> ```
> private var isGameOver = false
> ```

By default, `isGameOver` is `false`, as soon as the game ends we'll set it to `true`. The most convenient place to set the flag is inside of the `gameOver` method, since this method is called as soon as the game ends.

> Modify the `gameOver` method of `MainScene` to look as following:
>
> ```
> func gameOver() {
>   isGameOver = true
>   userInteractionEnabled = false
>   isDraggingPot = false
>   presentGameOverPopup()
> }
> ```

No we'll also need to modify the `update:` method to check for this flag.

> Add the following statements to the beginning of the `update:` method:
>
> ```
> override func update(delta: CCTime) {
>   if (isGameOver) {
>     return
> ```

```
    }
...
}
```

If the `isGameOver` flag is set we immediately return from the `update:` method, without performing any game logic.

Now you can run the game again, and you should notice that our issue is fixed. However, there's one thing left that is throwing me off. When the game ends and the popup appears, we have multiple score displays on screen. The popup informs us about the final score, and in the background behind the popup we can see the scoreboard that is displayed during gameplay.

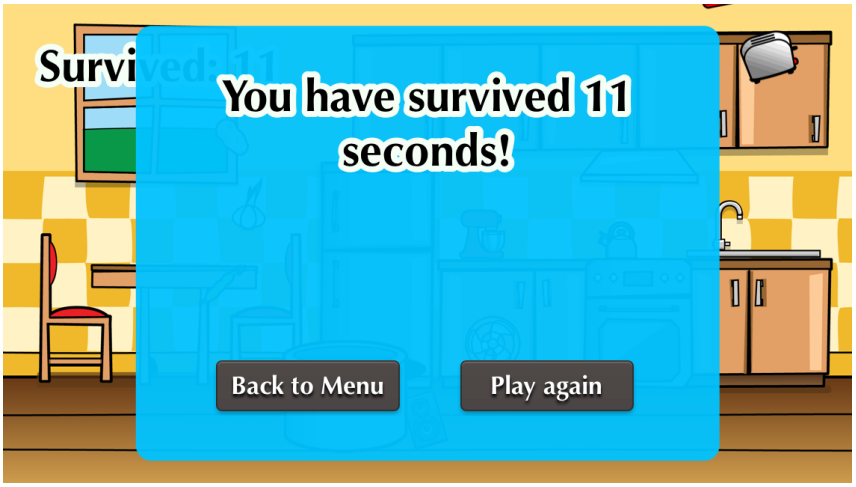


Figure 5.16: The player is confronted with score information in two places. The popup and the scoreboard contain the same information.

To fix this, we'll hide the scoreboard UI as soon as the game over popup is presented. We'll extend `presentGameOverPopup` to implement this.

Add the following three lines to the end of `presentGameOverPopup`:

```
func presentGameOverPopup() {
  ...
  let fadeOutAction = CCActionFadeOut.actionWithDuration(0.3) as
      CCAction
  gameModeDelegate?.userInterface.cascadeOpacityEnabled = true
  gameModeDelegate?.userInterface.runAction(fadeOutAction)
}
```

Animated appearances and disappearances always look a lot better than unanimated ones. With Cocos2D we can accomplish visual effects with only a few lines of code, so we fade out the scoreboard UI instead of simply making it invisible.

With this code in place you can test the game once again. Now you should see a nicely presented popup with no duplicate information displayed behind it:

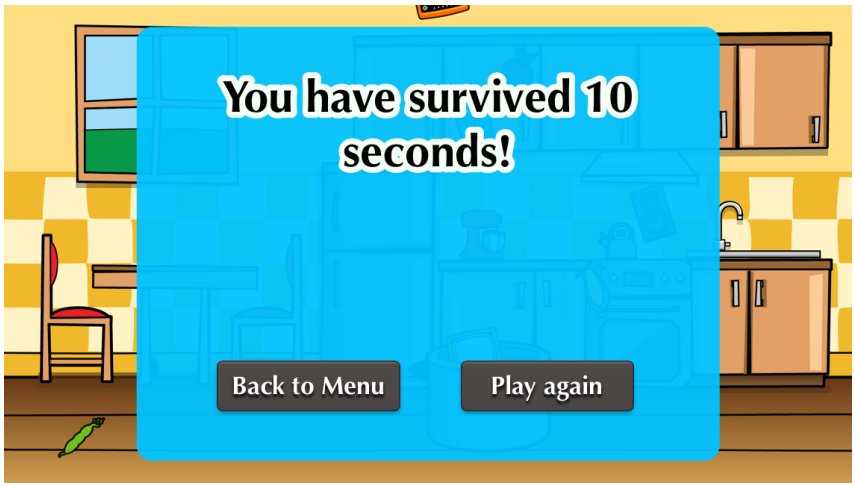


Figure 5.17: Without the score information in `MainScene` this popup looks more polished

Well done! In this chapter you have learned a lot about user interfaces in SpriteBuilder and Cocos2D. You know how to use scroll views, how to present popups and connect them to code efficiently and you've also learned how to implement a restart mechanism. Additionally we've spoken a lot about code design in this chapter. We have focused on creating this game in a way that decouples game modes from the actual gameplay. With this design it is very easy to add more game modes in future, and the lessons learned from this approach should apply to many of your original games as well.

In the next chapter we will look at how we can store highscores for this game. It is crucial to make your games competitive and to implement ways for players to track their progress as they play your game over longer periods of time. It's one of the best ways of keeping players engaged with your products.

5.7 Pausing the Game (Optional)

6 Persisting Highscores

The game we have built so far is clearly fun to play, however, if we want players to come back regularly we will need a highscore feature. Highscores will motivate players to improve their skills by playing the game frequently.

For this game we will keep the mechanism very simple. For each game mode we will store the highest score the player has achieved. These scores will only be stored locally on the device. Then we'll extend the game over popup to show the current highscore and to inform the player in case she has beaten her old one.

6.1 Extending the GameModeDelegate

We'll start implementing this feature by extending the definition of the `GameModeDelegate`. We'll add a required method called `saveHighscore`. That method will be called from within `MainScene` as soon a game ends. All game modes will implement this method and perform the highscore storing code within it.

> Open *GameModeDelegate.swift* and add the following method to the end of the protocol definition:
>
> ```
> /**
> Should be invoked when the receiving Game Mode should store the
>     latest Highscore
> ```

```
*/
func saveHighscore()
```

Now that we've extended the definition of the protocol, let's add a method call to `MainScene` that uses the new `saveHighscore` method. Whenever a game ends, we want to store the latest highscore of the current game mode.

Extend the `gameOver` method in *MainScene.swift* to look as following:

```
func gameOver() {
  isGameOver = true
  userInteractionEnabled = false
  isDraggingPot = false
  gameModeDelegate?.saveHighscore()
  presentGameOverPopup()
}
```

This change was simple. Now let's implement the highscore saving mechanism for both game modes.

6.2 Storing highscores for the endless game mode

iOS provides us a variety of options to persist application data. Core Data offers a feature-rich object persistence API that allows for advanced features such as search and migration between different versions of a data model. Through `NSKeyedArchiver` we are able to serialize objects and store them in files. Another option, preferred for simple tasks, is using the `NSUserDefaults` class to persist information.

`NSUserDefaults` is a persistent key-value store with a very simple API. We can store an integer with the following call:

```
NSUserDefaults.standardUserDefaults().setInteger(20, forKey: "
    highscore")
```

And retrieving the information is just as straightforward:

```
let oldHigschore =
NSUserDefaults.standardUserDefaults().integerForKey(highscoreKey)
```

Since we only want to score one integer per game mode, this simple API is ideal for our purposes.

Let's start with the implementation. First we'll define a new variable and a new constant. We'll use a variable called `newHighscore` to store whether or not the latest achieved score was a highscore. Based on this variable we will display a slightly different message to the user later on.

We'll also define a constant for the *key* that we use to store and retrieve the highscore from `NSUserDefaults`.

> Add the following two member definitions to *EndlessGameMode.swift*:
>
> ```
> private let highscoreKey = "EndlessGameMode.Highschore"
> private var newHighscore = false
> ```

When defining a key for working with `NSUserDefaults` it's good practice to prefix it with the current class name. That avoids conflicts between different parts of your app that might store and access information in the user defaults.

Now we can implement the `saveHighscore` method. We'll check if the latest score is higher than the current highscore, if that's the case we will persist the latest score and set the `newHighscore` variable to `true`. Otherwise we'll simply set `newHighscore` to `false`.

Add the following method to *EndlessGameMode.swift*:

```
func saveHighscore() {
  let oldHigschore = NSUserDefaults.standardUserDefaults().
      integerForKey(highscoreKey)

  if (Int(survivalTime) > oldHigschore) {
    // if this score is larger than the old highscore, store it
    NSUserDefaults.standardUserDefaults().setInteger(Int(
        survivalTime), forKey: highscoreKey)
    NSUserDefaults.standardUserDefaults().synchronize()
    newHighscore = true
  } else {
    newHighscore = false
  }
}
```

Now we are conforming to the new `GameModeDelegate` protocol and are successfully storing new highscores! One interesting line that we did not discuss yet is the following:

```
NSUserDefaults.standardUserDefaults().synchronize()
```

This line forces `NSUserDefaults` to write the latest changes to disk immediately. This method is called periodically by default. If we however store more or less sensitive information, such as the latest highscore a player just achieved, we call the method explicitly. That way the changes are persisted right away, eliminating the risk of losing data if the app crashes or is quit by the user.

Now there's a last step left. We should change the highscore message that we are displaying at the end of the game to include the player's highscore. Further, if the player just beat her own highscore we want to display a special message to congratulate the player.

Replace the existing `highscoreMessage` method with the following one:

```
func highscoreMessage() -> String {
  let secondsText = "second".pluralize(survivalTime)

  if (!newHighscore) {
    let oldHighscore = NSUserDefaults.standardUserDefaults().
        integerForKey(highscoreKey)
    let oldHighscoreText = "second".pluralize(oldHighscore)

    return "You have survived \(Int(survivalTime)) \(secondsText
        )! Your highscore is \(Int(oldHighscore)) \(
        oldHighscoreText)."
  } else {
    return "You have reached a new highscore of \(Int(
        survivalTime)) \(secondsText)!"
  }
}
```

One of the first things you might notice is that we've introduced a `pluralize` method on `String`. This method is part of the helpers that we've included right at the beginning of this project. Since we now have multiple occasions in which we need to use the pluralized form of a word it makes sense to factor this functionality out and avoid code duplication. This `pluralize` method is very primitive, it will append and *s* to a word in case the integer passed to the method is larger than one. That is obviously not the correct way to pluralize all English words, but for our game in which we use *points* and *seconds* it works just fine.

In the first line of this method we determine whether we need to use the word *second* or *second**s*** for this highscore message. Since we need this part of the message in any case, we keep it outside of the `if` statement.

Next, we check if the player has achieved a new highscore. If not, we display the latest score along with the current highscore. Else, we let the player know that he just reached a

new highscore.

And this is all it takes to build a simple highscore system - `NSUserDefaults` can go a pretty far way when storing this kind of simple information.

All that is left for this chapter is adding the same highscore functionality to the timed game mode.

6.3 Storing highscores for the timed game mode

The implementation for the timed game mode is very similar to the one we've implemented just now. In fact they are so similar that I briefly thought about factoring the implementation out, so that it can be used by both game modes without duplicating code. However, I've decided not to follow through on that idea. I think the amount of duplicate code in this case isn't large enough to require a more abstract but more complex solution. If you were to add a few more game modes this might change, for now we are going to accept some code duplication.

Since the implementation is so similar to what we've just seen, we won't discuss it in the usual detail.

> Add the `newHighscore` variable and the `highscoreKey` constant to *TimedGameMode.swift*:
>
> ```
> private let highscoreKey = "TimedGameMode.Highschore"
> private var newHighscore = false
> ```

Next, add the highscore saving method.

Add `saveHighscore` to *TimedGameMode.swift*:

```
func saveHighscore() {
  let oldHigschore = NSUserDefaults.standardUserDefaults().
      integerForKey(highscoreKey)

  if (points > oldHigschore) {
    // if this score is larger than the old highscore, store it
    NSUserDefaults.standardUserDefaults().setInteger(points,
        forKey: highscoreKey)
    NSUserDefaults.standardUserDefaults().synchronize()
    newHighscore = true
  } else {
    newHighscore = false
  }
```

And finally update the method that displays the highscore message.

Replace the existing `highscoreMessage` method within *TimedGameMode.swift* with the following one:

```
func highscoreMessage() -> String {
  let pointsText = "point".pluralize(points)

  if (!newHighscore) {
    let oldHighscore = NSUserDefaults.standardUserDefaults().
        integerForKey(highscoreKey)
    let oldHighscoreText = "point".pluralize(oldHighscore)

    return "You have scored \(points) \(pointsText)! Your
        highscore is \(Int(oldHighscore)) \(oldHighscoreText)."
  } else {

    return "You have reached a new highscore of \(points) \(
```

```
            pointsText)!"
        }
    }
```

Overall this implementation is almost identical to the Endless Game Mode's one.

6.4 Wrapping up

Now we've implemented the entire highscore functionality for both game modes! I'll admit that it is a very simple functionality, but it will definitely increase the motivation of players to come back to our game. Here's what the popup message should look like once you've finished a game:

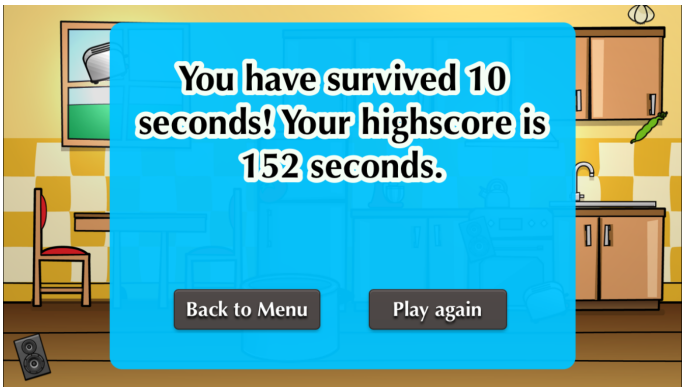


Figure 6.1: When the game ends the user retrieves information about their highscore

Advanced Highscores

If you are interested in implementing highscores and leaderboards that work across devices you should take a look at Apple's *Game Center API* (https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/GameKit_Guide/Introduction/Introduction.html). This book is mainly focused on game development so we won't discuss the API, however Apple's documentation should be very helpful in adopting Game Center.

We're almost done with implementing this game and thus you are getting close to the end of this book. In the last chapter we will discuss something that makes the difference between a good game and a fantastic game: *Effects and Animations*! Get ready for the Grand Finale.

7 Effects and Animations

Up until this chapter we have built a fully functional game that could be shipped to the App Store! This last chapter will deal with polishing the game and making it more delightful. SpriteBuilder and Cocos2D provide powerful, yet simple to use tools to create visual effects and animations. In this chapter we will add light effects and we'll use the SpriteBuilder timeline to bring some more motion to our gameplay.

7.1 Lighting with CCEffects

Let's start by adding some light effects to our game. Lighting effects can make a game feel a lot more polished. Here's a comparison of what our game looks like right now, and how it will look once we're completed this section:

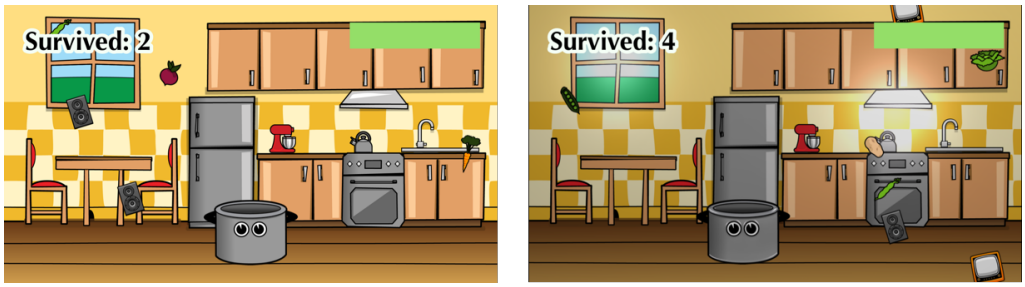


Figure 7.1: Unlighted scene on the left, lighted scene on the right

Most game engines require developers to write *shader programs* to use visual effects such as lighting. Cocos2D provides an API called `CCEffects` that implements many common visual effects, such as lighting, refraction, blur, etc. Using CCEffects we can enhance our games without needing to learn how to write shader programs.

The CCEffects can even be configured with SpriteBuilder! We can set up all of the lighting for this game with only a handful lines of code.

The first step of adding lighting to our game is understanding some of the theory that goes into lighting in 2D games.

7.1.1 Lighting in 2D games

The simplest way to light a 2D scene is to use brighter and darker colors, depending on the distance to a given light source, I'll refer to this technique as *flat lighting*. Figure 7.1 shows flat lighting on the background image of our game. Some areas of the background are lighter than others. The background is the lightest around the window and the kitchen light, the two light sources for our game.

Flat lighting comes entirely out of the box using Cocos2D. However, using flat lighting alone doesn't create great visual effects. In 2D games lighting can be used to give objects a 3D feel. We are going to use that technique for the pot and the falling objects.

We will create *normal maps* for each of our game objects. A normal map is a special kind of texture that describes the 3D surface of an object. That way the game engine can calculate how strong certain areas of a texture should be light up by a light source. Here's what a normal map looks like:

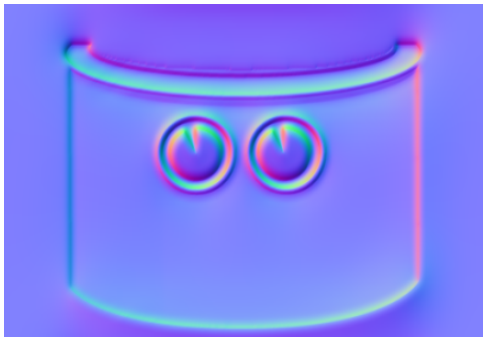


Figure 7.2: The normal map for the bottom part of the pot. Each color encodes information about the object's surface

And here's a comparison that shows how lighting and normal maps together give the pot a 3D feel:

Figure 7.3: No lighting on the left, lighting with normal map on the right

The comparison above shows that certain parts of the pot appear brighter and shinier. Cocos2D calculates the brightness for each part of the texture based on the normal map that we provide and the position of the light.

Lighting and normal maps for 2D games are topics that are worth dedicating entire books

to (and there are a ton out there!) so for now we will stick with this overview of how lighting works. Even this basic knowledge will allows us to make the game look quite a bit better. You might wonder where the normal maps for our textures come from. The answer is: we need to create them ourselves.

7.1.2 Creating normal maps

There are a bunch of tools available that turn creating simple normal maps into an easy task. For this book I've provided all of the normal maps for you, as part of the asset pack that you downloaded at the beginning of this book.

What if you want to create normal maps for your own game? The normal maps for this book have been created with the tool *CrazyBump* (http://www.crazybump.com/). CrazyBump uses shape detection to guess normal maps and the results have been great for me!

If you have complicated textures, or you need to polish the lighting effects in your game in great detail, you might want to resort to a tool that allows you to create a normal map manually. *SpriteIlluminator* (https://www.codeandweb.com/spriteilluminator) is a great tool for manually creating normal maps.

Since we already have all the normal maps we need, let's dive right into SpriteBuilder and set up some light sources!

7.1.3 Setting up lighting effects in SpriteBuilder

Now that we have a basic understanding of 2D lighting it's time to dive into the CCEffects API. There are three important components that we will be using:

CCEffectNode is a container for all visual effects. If you want to apply effects to CC-

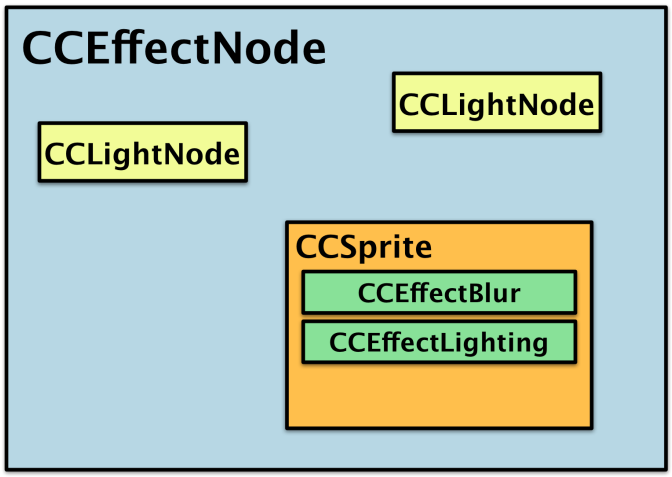
Sprites, all of them need to be the child of a `CCEffectNode`. In practice this means that you'll mostly have a `CCEffectNode` as a container for your entire gameplay scene.

CCEffect represents one of the different available visual effects, e.g. lighting, refraction, blur, etc. Currently effects can only be applied to CCSprites. You add an effect to a sprite by instantiating a `CCEffect` and assigning it to a CCSprite.

CCLightNode is used in combination with the effect `CCEffectLighting` to define light sources.

To summarize what we've discussed so far: Effects can only be applied to CCSprites. All affected sprites need to be a child of a `CCEffectNode`. Light nodes are used together with the lighting effect and define light sources. Direction and brightness of these light nodes is used by Cocos2D to render lighting effects.

Here's a diagram of how the components play together:



We will only discuss a subset of the `CCEffects` API, but the good news is that it is very easy to use and therefore can be explored without much instruction!

There a many simple effects that can be added with a single line of code, such as *blur* or *saturation*. You can add all effects in code or in SpriteBuilder. You can change the relevant properties of a `CCEffect` in code, that alows you to animate effects. You can for example create a *saturation* effect that starts with full saturation and over time reduces the saturation until the sprite turns into a grayscale image.

Finally, `CCEffectStack` allows you to combine multiple effects.

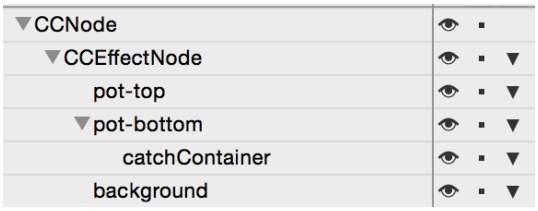
More on effects

If you want to learn a little bit more about effects in Cocos2D you should read our tutorial that introduces the `CCEffects` API (https://www.makeschool.com/tutorials/cocos2d-3-2-with-cceffects-is-coming).

This introduction gives us enough understanding to move to SpriteBuilder and set up our lighting. Open the SpriteBuilder project and select the *MainScene.ccb* file.

1. Drag an *Effect Node* from the node library to the timeline of *MainScene.ccb*
2. Use the Shift key to select all of the nodes that are currently added to Main Scene:

3. Drag the selected nodes onto the *Effect Node* to turn all of the nodes into children of the effect node. Your node hierarchy should look like this:



4. Set the content size of the effect node to *percentage of parent container* and set the size to (100%, 100%)

Now we have the `CCEffectNode` set up, which allows us to add visual effects!

Next, let's look into adding the two light sources. We want the light sources to be place on the window and above the stove. As you might remember, we are designing this game to work on 3.5 inch iPhones, 4 inch iPhones and on iPads.

In order to position the lights correctly for all of these device types we need to use a little trick, simply using relative positioning with percentage values does not work. Here's what the scene looks like on different device types using relative positioning:

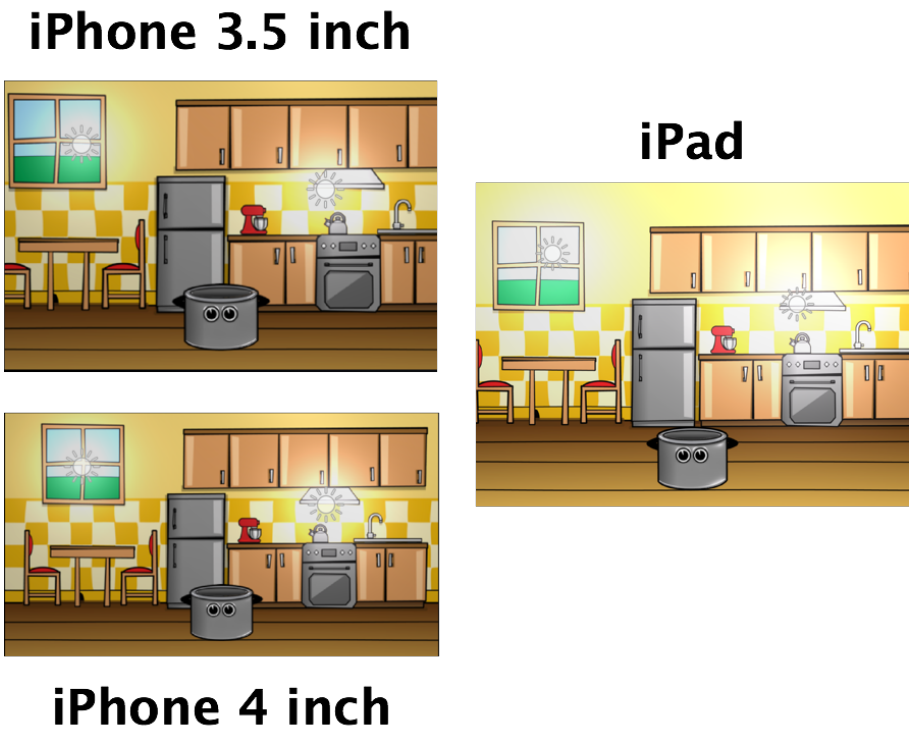


Figure 7.4: Relative positioning results in slightly different light positions for each device type

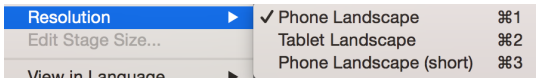
We have set up the background image to be centered on all device types. Therefore the distance of the window and the stove from the screen edges varies. This means we cannot use the screen edges as reference points for positioning the lights.

The center of the background image is always at exactly the same position. The trick in this situation is to position the light sources relative to that center position. We can do so by adding an empty CCNode to the center of the scene and adding our two light sources

as children.

Previewing a scene on different device types

We have discussed this briefly in 2.3, but just as a short reminder: SpriteBuilder allows you to preview your scenes on different device types. To switch between different device types, go to the *Document -> Resolution* menu:



You can also use the shortcuts displayed next to the device types!

Now we can start adding the center node and the two light sources.

First, let's add the center node.

1. Drag a plain *Node* from the node library and add it as a child of the *CCEffectNode*
2. Set its position type to be *percentage of parent container*
3. Set the position to *(50%, 50%)*
4. Rename the node to *light-container* by selecting the node in the timeline and hitting the return key

Now we can add the light source for the window. We will discuss the interesting properties of our light sources in detail as soon as they are set up:

1. Drag a *Light Node* from the node library and add it as a child of the *CCNode* that we created just now
2. Set the position of the light to *(-180, 90)*

3. Set the *Diffuse Intensity* to *0.25*
4. Set the *Specular Intensity* to *0.60*
5. Set the *Ambient Intensity* to *0.73*
6. Set the *Cutoff Radius* to *500.00*
7. Set the *Half Radius* to *0.70*
8. Set the *Depth* to *180*

And finally, add the second light source:

1. Drag a *Light Node* from the node library and add it as a child of the *CCNode* that we created just now
2. Set the position of the light to *(140, 40)*
3. Set the *Diffuse Intensity* to *0.20*
4. Set the *Specular Intensity* to *0.60*
5. Set the *Ambient Intensity* to *0.00*

Now you should be able to switch between the different screen sizes and see that the lights are positioned correctly all the time! Let's discuss some of the light properties that we just set up.

Diffuse Intensity the intensity of the diffused portion of the light. Diffused light brightens nearby objects, without resulting in a *shininess* effect. Diffused light is soft, directional light.

Specular Intensity the intensity of the specular portion of the light. Specular light is a

sharp, directional light. It causes reflections on surfaces of objects that are lit up.

Ambient Intensity the intensity of the ambient portion of the light. Ambient light has no direction and the distance to objects in the scene is not relevant for this kind of light. All objects in the scene are lit up by ambient light in exactly the same way. This factor is used to set a base brightness.

Cutoff Radius the reach of the light in points. Only nodes within the radius of the light will be lit.

Half Radius the portion of the radius at which the intensity of the light has fallen to half of its maximum value. You can choose a value between 0 and 1. A smaller value will create a sharper light, a higher value will create a smoother light.

Depth defines the distance of the light from the pane it is lighting. A small value means that objects are lit from the side. A large value means that the object is far away from the pane, therefore the objects are lit from the front.

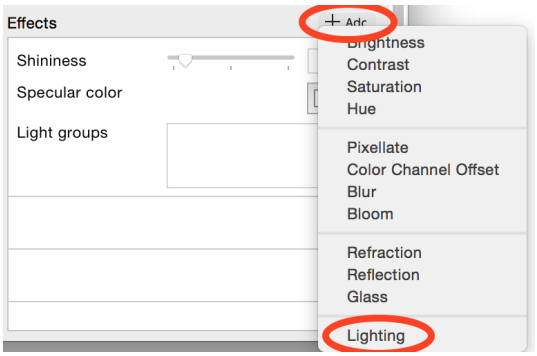
The last thing to note is that Cocos2D provides two different light types: *Point lights* and *Directional lights*. For this game we are only using point lights. You can change the type in the *Light type* dropdown in the property inspector. A point light only lights up objects that are within the cutoff radius, and objects that are further away from the light source are lit up less than closer objects. Directional lights light up all objects in the same way. They are similar to ambient lighting, except that the light comes from a certain direction.

More details on lighting

The creator of *Sprite Illuminator*, one of the normal map tools mentioned earlier, has written a nice article on lighting in SpriteBuilder and Cocos2D that covers many light properties in detail: `https://www.codeandweb.com/blog/2015/03/17/cocos2d-dynamic-lighting-tutorial`. The source code annotations in *CCLightNode.h* of the Cocos2D also reveal many details about the different adjustable properties.

At this point we have the lighting set up, but we don't see any change in our scene. Setting up lighting is a two step process in Cocos2D. We need to define the light sources, which we just did. Then we need to add the *Lighting effect* to sprites that shall be effected by these light sources. In this scene we want the background image and the bottom and top part of the pot to be affected by our light sources.

1. Select the *background* sprite from the timeline
2. Add a lighting effect in the property inspector as following:



3. Select *pot-top* and add a lighting effect in the same way you added it to the background

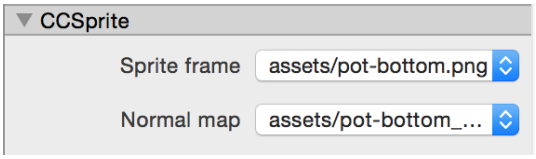
4. Adjust the *shininess* of this lighting effect to *0.3*
5. Select *pot-bottom* and add a lighting effect
6. Adjust the *shininess* to *0.3*

Now you should see the lighting taking effect! The last thing we need to do is add our normal maps.

7.1.4 Assigning normal maps in SpriteBuilder

Now that our light sources and lighting effects are set up, let's assign the normal maps to our sprites so that we get more detailed lighting. For the *pot-top* and *pot-bottom* sprite, we can set up the normal map in SpriteBuilder. The falling objects are spawned dynamically in code, we will look at how to set up their normal maps in the next section.

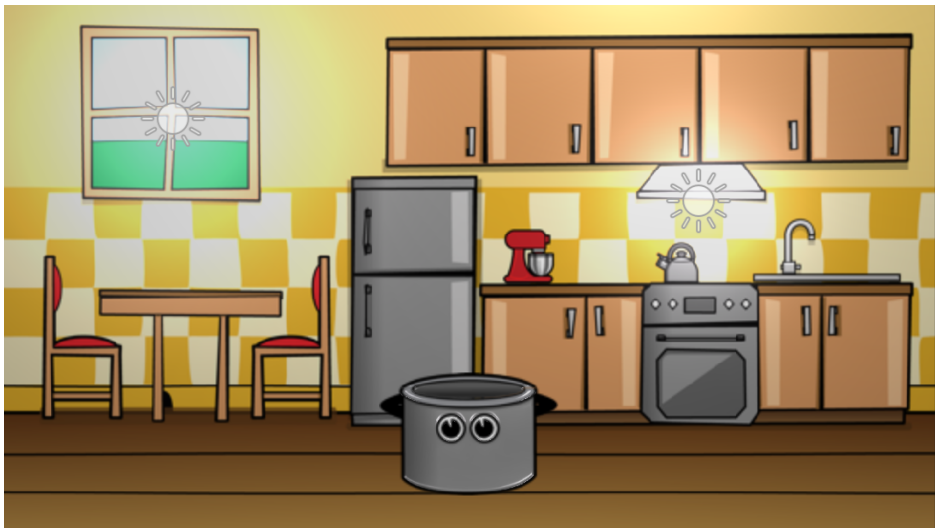
Adding normal maps to sprites is very simple in SpriteBuilder. In the property insepctor of a CCSprite there's a separate dropdown below the *Sprite frame* dropdown that allows us to assign a normal map:



1. Select *pot-top* in the timeline
2. Set the normal map to *assetspot-top_NRM.png*
3. Select *pot-bottom* in the timeline

4. Set the normal map to *assetspot-bottom_NRM.png*

Great! Now you should see a nicely lit scene, with subtle reflections on the pot:



Note that the preview of the lighting effects in SpriteBuilder currently don't match the lighting effects on the simulator or device exactly. To get a good understanding of how your lighting effects are working you'll need to run your game from Xcode.

Now all that's left in terms of lighting is assigning normal maps to our falling objects as well!

Testing lighting effects on the simulator

In case you've been curious and have tried running the current version of the project, you might have realized that the performance in the simulator is extremely poor. The reason is that the iOS simulator doesn't use your computers GPU for all OpenGL features, but instead simulates many of them in software, which in many cases can be extremely slow. When working with CCEffects, I highly recommend testing your game on an actual device, even though that means that you're required to purchase an Apple Developer license.

7.1.5 Assigning normal maps in code

As mentioned earlier, the falling objects are spawned dynamically in code, therefore we cannot assign the normal maps in SpriteBuilder. However, adding normal maps in code is pretty simple. Additionally we will also need to look into adding the lighting effect to all falling objects in code, so that they are affected by our two light sources.

As you might remember, we had a clever way of managing the different assets for our falling objects. We are using a *plist* file that stores the image names for all *positive* and *negative* objects. The goal of this attempt is to keep information about game content outside of our codebase. In software design this principle is called *configuration over code*.

Ideally we want to use the same approach for the normal maps we are about to assign. We could simply add the image names of all the normal maps to the plist to accomplish that. However, there's another interesting principle in software design: *convention over configuration*. When creating the normal maps for the sprites in our game, I have followed a *convention*. Each normal map file name is built as following: *{ImageName}_NRM.png*

This means that we know the file name of the normal map as soon as we now the file

name of the sprite's texture. Using this knowledge, the implementation of setting up normal maps in code becomes pretty straightforward.

> Open *FallingObject.swift* and extend the `init(type:)` initializer by adding the following lines **to the end of the initializer**:
>
> ```
> effect = CCEffectLighting()
>
> let imageNameSplit = split(imageName!) { $0 == "." }
> let imageNameFirstPart = imageNameSplit[0]
> let normalMapName = "\(imageNameFirstPart)_NRM.png"
>
> normalMapSpriteFrame = CCSpriteFrame.frameWithImageNamed(
>    normalMapName) as! CCSpriteFrame
> ```

By adding the code above to the initializer of `FallingObject`, we make sure that the lighting effect and the normal map are set up as soon as the object is created.

In the first line we set up the `CCEffectLighting` and assign it to the `effect` property of our `FallingObject`. The `effect` property is inherited from `CCSprite`. Now our falling object will be affected by light sources.

The remaining lines are used to assign the correct normal map to this object. We split the image name to get the part before the file extension. Then we append *_NRM.png* to that first part to retrieve the filename of the normal map. Within the closure passed to the `split` function we access a variable called `$0`. This is a cool feature in Swift, within closures we can access parameters either by name or by using the `$` symbol in combination with the index of the argument. Using these shorthand argument names is pretty common when working with standard library functions such as `split`, `map`, etc.

Finally we use that filename to load a `CCSpriteFrame` that we can assign to the `normalMapSpriteFrame`. That property is also inherited from `CCSprite`.

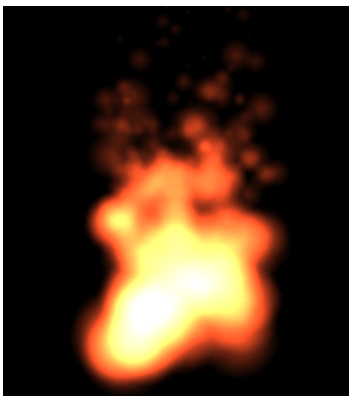
Now our falling objects are good to go! If you run this version of the game you should see detailed lighting effects and subtle reflections on all of the spawned objects.

This concludes our coverage of the `CCEffects` API for this book. You now know the basic set up that effects require. You also know how to configure effects in SpriteBuilder and in code. The lighting effect is one of the more complicated ones provided by the effects API - you should be able to pick up simpler effects such as blur and saturation pretty easily.

If you find some time, I highly recommend playing around with this API. With little time investment you will be able to create stunning visual effects.

7.2 Adding Particle Effects

Another great way to improve the look and feel of your app is to add particle effects. Particle effects are used to animate huge amounts of very small sprites, which can be very useful when animating fire, smoke or explosions. Here's an example of what a particle effect in Cocos2D can look like:



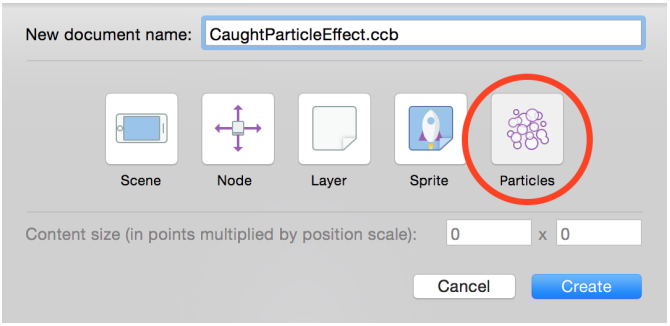
SpriteBuilder comes with built in support for particle effects which makes it easy to create

them and iterate on them while watching a live visual preview. For our game we will add a particle effect that takes place as soon as the player catches one of the good objects.

We are going to create the particle effect in SpriteBuilder, then load it and add it to the scene dynamically in code.

7.2.1 Creating a Particle Effect in SpriteBuilder

Let's open our SpriteBuilder project to get started!

> Create a new CCB File of type *Particle* and call it *CaughtParticleEffect.ccb*:
>
> 
>

Now you should see a new file with the default fire particle effect. We want to use a different effect for our game.

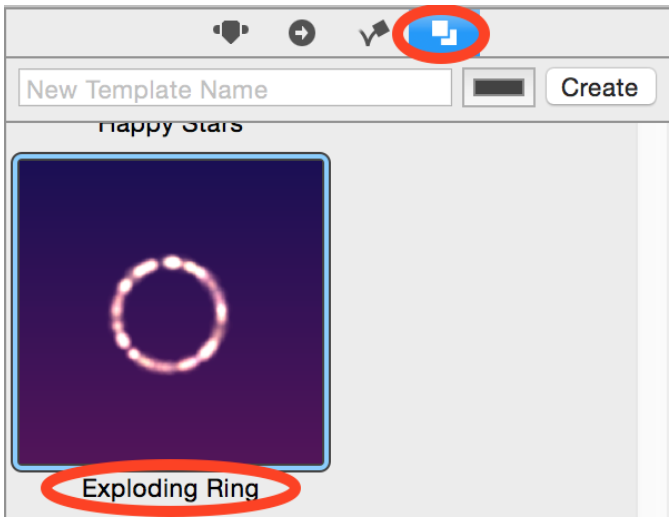
Now how do we go about building our explosion effect? There are two different aspects that make up a particle effect. The first is the texture for the individual particles, the second are all the settings that define how the particles of the system move, change over time, how long the entire particle systems lasts, etc. There are well over 20 settings that you can use to build your particle effects - discussing all of them goes well beyond the scope of this book.

There are two essentially different ways to start out building your own effect. You can start entirely from scratch, choosing your own particle texture and experimenting with the vast amount of available configuration options. Alternatively you can select a particle effect from SpriteBuilder templates and modify it to build your custom effect.

In most cases this second approach makes more sense. You will likely be able to find an existing particle effect that is a good starting point for your custom one. For this game we will use that second approach and will use a very slightly modified library effect.

The best way to dive deeper into particle effects is spending some time playing around with the different options the editor in SpriteBuilder provides.

Let's turn our fire effect into an explosion effect.

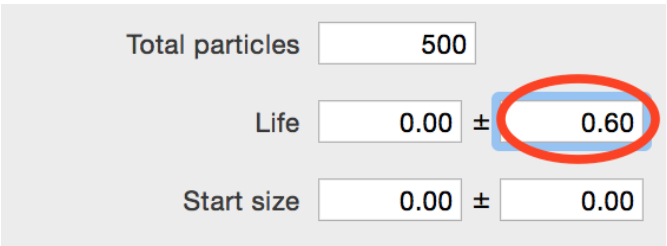
> Select the *CCParticleSystem* in the timeline. Then open the last tab in the inspector panel on the right. Scroll down to the *Exploded Ring* effect and **double-click** onto it to select it:
>
> 
>

Now you should see the fire being replace by an exploding ring. You will notice that this particle effect will only run once, then you will see a blank stage. That's because this template creates an effect in which has a limited lifetime. The fire effect had a *duration* of *-1.0* seconds which indicates an endlessly looping effect. This effect has a *duration* of *0.01* which means that particles are only emitted for a very short time. If you want to see what the particle effect looks like again, you can click the *Start Particles* in the inspector panel.

The *duration* property defines the lifetime of the particle system, which defines how long particles will be created.

Each individual particle has a lifetime as well. In the exploded ring template the lifetime is set to *1.0* seconds. The larger this value, the longer the particles stay visible as the move from the center to the edge of the stage. This makes the ring appear larger.

We actually want the ring to be a little bit smaller. Later we will add it to the inside of the pot as soon as an object is caught. We want it to look like all the particles are contained inside of the pot. After playing around with a few values I settled with *0.6* for the lifetime of each particle.

> Change the *Life* property of this effect in the inspector panel to *0.6*:
>
> 
>

When you start this particle effect again you will see that the explosion circle now appears smaller. This completes our particle effect setup in SpriteBuilder. Feel free to spend some time exploring the other properties of particle effects before moving on.

Next, we'll load this particle effect in code and add it to our game scene dynamically whenever the user catches a good object!

7.2.2 Loading particle effects in code

Now it's time to pull up the Xcode project again. We want to run the particle effect when the player catches a good object. This means we need to extend the `performCaughtStep` method that we implemented earlier.

We'll first load the CCB File that stores the particle effect, then we position it and add it to the top part of the pot, this will make the effect appear to take place inside of the body of the pot. The particle effect starts automatically, as soon as it is added to a visible scene.

Let's implement this feature and then discuss some more details!

> Extend the implementation of `performCaughtStep(fallingObject:)` to load and add the particle effect:

```
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())) {
    gameModeDelegate?.gameplay(self, caughtFallingObject:
       fallingObject)
    fallingObject.removeFromParent()
    let fallingObjectIndex = find(fallingObjects, fallingObject)
       !
    fallingObjects.removeAtIndex(fallingObjectIndex)

    if (fallingObject.type == .Good) {
      let particleEffect = CCBReader.load("CaughtParticleEffect"
         ) as! CCParticleSystem
```

```
        particleEffect.autoRemoveOnFinish = true
        particleEffect.positionType = CCPositionType(
          xUnit: .Normalized,
          yUnit: .Points,
          corner: .TopLeft
        )
        particleEffect.position = ccp(0.5, 20)
        potTop.addChild(particleEffect)
      }

    }
  }
```

Let's look at the code we added, step by step. First, we check the object type. We want to display the particle effect only if the user caught a `.Good` object. If that's the case, we use the `CCBReader` to load the CCB File that contains the particle effect. We parse the value returned by the `CCBReader`.

Next, we configure an important setting on the particle effect. We set `autoRemoveOnFinish` to `true`. This means that Cocos2D will take care of removing this particle effect from its parent object as soon as the particle effect completed. A particle effect completes once all emitted particles have reached the end of their lifetime. Without this line of code you would need keep track of all added particle effects manually and make sure to remove them - an easy way to introduce memory issues.

With the particle loaded and set up, we take care of positioning it. This particle effect should be rendered inside of the pot's body. I've decided to center the effect horizontally. We set up a position type that makes the centering easy and that allows us to position the particle effect from the top edge of the top instead of from the bottom.

Then we set the position: horizontally centered, vertically slightly below the top edge of

the pot.

In the last step we add the `particleEffect` to `potTop`.

An alternative positioning approach

Instead of setting the position type and the position of the particle system in code, you could also add an empty node in SpriteBuilder at the position where you want to render the particle effect. Then you could add the particle effect to that node instead of to `potTop`. I tend to use this approach for positions that are harder to express in code.

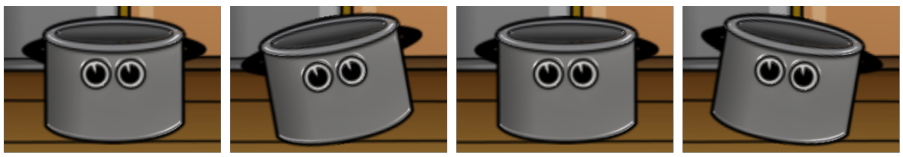
Now you can run the game. You'll see the particle effect run as soon as you catch a good object. This game is already looking a lot more polished than at the beginning of this chapter.

In the next and last section we will look into using the SpriteBuilder timeline to add some more animations to our core gameplay.

7.3 Delightful animations with SpriteBuilder's timeline

We will now add some animations to the game to wrap this chapter up. So far we've used the SpriteBuilder timeline to animate user interfaces, now we want to animate parts of our gameplay as well.

Whenever the player catches a good object we'll let the pot play a little wobble animation:

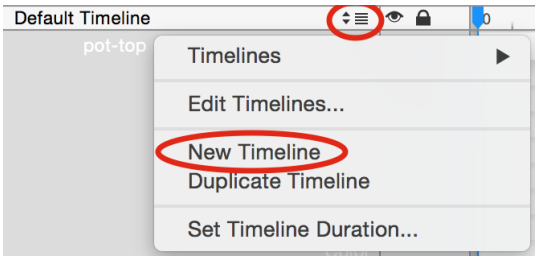


Later on we will create a second animation based on this one where we add a coloring effect that turns the pot red. We will use that second animation whenever the user catches a bad object.

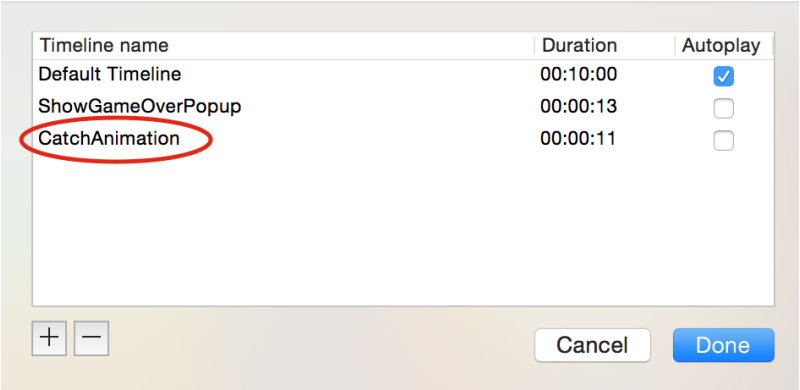
Let's get started on creating these animations! You might remember that the top and the bottom part of the pot need to have the same parent node as the falling object due to a limitation in Cocos2D. This means we cannot group them under a container node. We discussed this in detail in chapter 4.2 (*A rendering tweak*). This unfortunately means that we need to copy the keyframes for the animation for both parts of the pot. Luckily SpriteBuilder makes this pretty easy!

Open up the SpriteBuilder project and select the *MainScene.ccb* file.

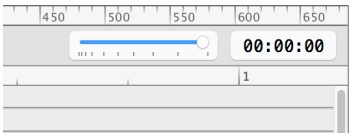
1. Create a new timeline :



2. Name that timeline *CatchAnimation*:



3. Zoom in all the way in the timeline since the animation we're building will be very short:



4. Select the **pot-bottom** node from the timeline

5. Create rotation keyframes at the following frames and rotations (remember you can see the current frame in the last segment of the time displayed next to the zoom indicator, as shown in the image above):

    a) frame:0, rotation: 0

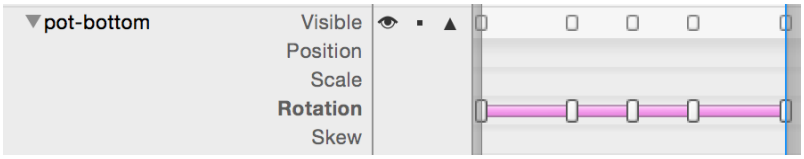
    b) frame: 3, rotation: 10

    c) frame: 5, rotation: 10

    d) frame: 7, rotation: -10

e) frame: 10, rotation: 0

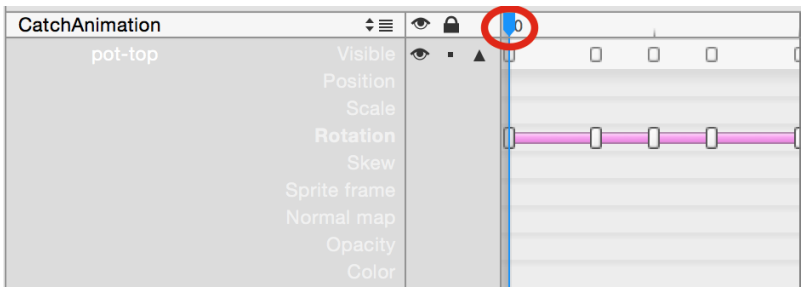
The result should look as following:



Great, you should be able to play this animation and see the bottom part of the pot wobbling. Once you are up to speed with SpriteBuilder's timeline, building animations becomes extremely fast. Next, we need to copy this animation for the top part of the pot.

Let's copy the keyframes we just created and add them to **pot-top**:

1. Hold down the Shift key and click onto each of the 5 keyframes
2. Hit the *CMD + C* keys to copy the keyframes
3. Select the **pot-top** node from the timeline. Drag the timeline ruler all the way to the left, to frame 0 (the copied keyframes will be added at the position of the ruler). Then hit the *CMD + V* keys to paste the keyframes:



Now we're done with the catch animation. You can use the timeline playback tool to

preview the animation.

Before we switch to Xcode to write code that will trigger this timeline to play, let's add the animation for catching negative objects as well.

Essentially all we need to do is copy the entire animation that we've created so far and add an animation to fade the color of the pot to red.

> Let's create the second timeline with the animation for catching negative objects.
>
> 1. Create a new timeline
> 2. Name the timeline *CatchNegativeAnimation*
> 3. Switch to the *CatchAnimation* timeline
> 4. Copy the 5 keyframes from the wobble animation from the *CatchAnimation* timeline by selecting all keyframes with the *Shift* key and then hitting the *CMD* + *C* keys
> 5. Switch back to the *CatchNegativeAnimation* timeline
> 6. Select the **pot-top** node from the timeline
> 7. Make sure the timeline ruler is at frame 0
> 8. Use the *CMD* + *V* keys to paste the animation
> 9. Select the **pot-bottom** node from the timeline
> 10. Use the *CMD* + *V* keys to paste the animation

Now it's time to add the color fade animation! This animation will not be a linear one. In chapter 5.3.1 we discussed that Cocos2D provides different types of interpolations for

our animations. I've built the color animation using the **Ease Out** / *Ease In* interpolations. That makes the color snap to red pretty fast, and then fade out to white a little bit slower. Choosing different interpolations every once in a while, instead of always using the default linear interpolation, will make your animations look more interesting.

> Make sure you have the *CatchNegativeAnimation* timeline selected before starting!
>
> 1. Select the **pot-bottom** node from the timeline
> 2. Add the following *Color* keyframes:
>   a) frame:0, color: white
>   b) frame: 5, color: red
>   c) frame: 10, color: white
> 3. *Right-Click* onto the pink bar between the first and the second keyframe
> 4. Select *Ease Out* from the context menu
> 5. *Right-Click* onto the pink bar between the second and the third keyframe
> 6. Select *Ease In* from the context menu

And now, as a very last step, we need to copy the animation from **pot-bottom** to **pot-top**.

> Make sure you have the *CatchNegativeAnimation* timeline selected.
>
> 1. Select the **pot-bottom** node from the timeline
> 2. Copy the three keyframes from the color animation
> 3. Set the timeline ruler to frame 0

4. Select the **pot-top** node from the timeline

5. Paste the three keyframes from the color animation

And now we are all set! You should be a little bit more experienced at using the timeline at this point, and we have to new animations that will make our game look better.

Now the final step for this chapter is adding code that will trigger the playback of our new timelines.

7.3.1 Playing the timelines

Now it's time to switch back to our Xcode project. Since both animations will be playing once an object is caught, we once again need to extend the `performCaughtStep(fallingObject:)` method.

We simply need to add two lines that trigger the timeline playback:

Extend the `performCaughtStep` method to look as following:

```
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())) {
    gameModeDelegate?.gameplay(self, caughtFallingObject:
       fallingObject)
    fallingObject.removeFromParent()
    let fallingObjectIndex = find(fallingObjects, fallingObject)
       !
    fallingObjects.removeAtIndex(fallingObjectIndex)
```

```
      if (fallingObject.type == .Good) {
        animationManager.runAnimationsForSequenceNamed("
            CatchAnimation", tweenDuration: 0.1)

        let particleEffect = CCBReader.load("CaughtParticleEffect"
            ) as! CCParticleSystem
        particleEffect.autoRemoveOnFinish = true
        particleEffect.positionType = CCPositionType(
          xUnit: .Normalized,
          yUnit: .Points,
          corner: .TopLeft
        )
        particleEffect.position = ccp(0.5, 20)
        potTop.addChild(particleEffect)

      } else if (fallingObject.type == .Bad) {
        animationManager.runAnimationsForSequenceNamed("
            CatchNegativeAnimation", tweenDuration: 0.1)
      }
    }
  }
```

If the player catches a good object, we play *CatchAnimation*, if the player catches a bad object we play *CatchNegativeAnimation*.

You might have noticed the new `tweenDuration` argument that we are using as part of calling `runAnimationsForSequenceNamed`. This duration defines how much time is used to animate a transition between two different timelines.

Since multiple objects can be caught in a very small timeframe, it could happen that the player catches a good object and then catches a bad object immediately afterwards. This might mean that the *CatchNegativeAnimation* would be started before the *CatchAnimation*

can complete. If we wouldn't use a `tweenDuration`, then the transition would happen instantly, which can result in noticeable jumps of the rotation and the blend color. Using a `tweenDuration`, Cocos2D generates a smooth transition between the timelines by animating the properties that need to be changed, instead of changing them instantly.

There's one issue with timelines that we haven't discussed yet. **Only one timeline per CCB File can be played back at a time**. As soon as we play a timeline, any other timeline that might have been playing will be stopped. This causes a problem with code that we have set up very early in this book: the sound that gets played whenever a falling object is missed and drops to the ground.

Playing this sound will now interrupt our catch animation timelines. Since players will drop a lot of objects, this issue will occur frequently.

What can we do? One solution would be moving the timeline to a separate sub-CCB File within *MainScene.ccb*. However, there is luckily a simpler solution for this use case. We can trigger the sound to be played in code, without using a timeline for it.

7.4 Playing sound effects from code

Playing back sound in Cocos2D is also just a few lines of code.

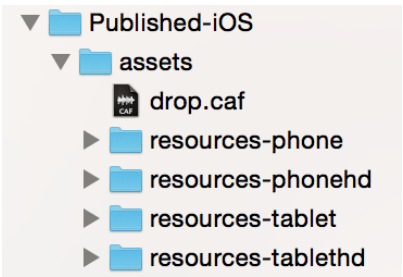
> Modify the `performMissedStep` method to look as following:

```
func performMissedStep(fallingObject:FallingObject) {
  // check if falling object is below the screen boundary
  if (CGRectGetMaxY(fallingObject.boundingBox()) < CGRectGetMinY
     (boundingBox())) {
    gameModeDelegate?.gameplay(self, droppedFallingObject:
       fallingObject)
    // if object is below screen, remove it
    fallingObject.removeFromParent()
```

```
        let fallingObjectIndex = find(fallingObjects, fallingObject)
            !
        fallingObjects.removeAtIndex(fallingObjectIndex)
        // access audio object
        let audio = OALSimpleAudio.sharedInstance()
        // play sound effect
        let filePath = CCFileUtils.sharedFileUtils().
            fullPathForFilenameIgnoringResolutions("assets/drop.caf")
        audio.playEffect(filePath)
    }
}
```

As a first step we grab a reference to the audio engine, using `OALSimpleAudio.sharedInstance()`. Then we find the file path to the audio file we want to play. Note that we need to use the `fullPathForFilenameIgnoringResolutions` method. Typically `CCFileUtils` will look for our file inside of a path that is specific to the current device resolution. This makes a lot of sense for images, because they are scaled for different devices and then stored in specific subdirectories. For sound files however, we don't want to use this device specific file search. Sound files are used without any modification across all devices and are not stored in the resolution specific directories.

You can see it in the folder structure of the *Published-iOS* folder, that is stored under *Resources/Published-iOS*:



Once we have the file path, we simply call the `playEffect` method. Now you know how to play back sound effects in code!

There's even a small advantage we can gain using this approach. We can *preload* sound effects. Typically a sound effect is loaded by `OALAudio` as soon as it is played the first time. If you have larger sound files this can lead to a small freeze in your gameplay whenever you play a sound effect for the first time.

We can avoid this issue by explicitly telling `OALAudio` to load a sound effect. Typically you want to do this as soon your gameplay scene is loaded, e.g. in the `didLoadFromCCB` method.

Let's preload our audio effect so that you know how to do it in case you build a game that is more audio heavy.

> Extend `didLoadFromCCB` to preload the sound effect we are using:
>
> ```
> func didLoadFromCCB() {
>   potTop.zOrder = DrawingOrder.PotTop.rawValue
>   potBottom.zOrder = DrawingOrder.PotBottom.rawValue
>
>   let audio = OALSimpleAudio.sharedInstance()
>   let filePath = CCFileUtils.sharedFileUtils().
>       fullPathForFilenameIgnoringResolutions("assets/drop.caf")
>   audio.preloadEffect(filePath)
> }
> ```

Instead of calling `playEffect`, we call `preloadEffect` to load the sound file and prepare it for playback.

Now we have added enough polish to this game!

Index