Contents

1 Introduction to SpriteBuilder and Cocos2D

Now it's time to dive into 2D Game Development! For this chapter I will assume that you haven't written a game with a game engine yet. We will be discussing all the relevant concepts throughout this chapter.

1.1 Installing the software

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

Installing SpriteBuilder is easy, simply open the *App Store* app on your Mac and search for *SpriteBuilder*. Note that you should always use the latest version of Mac OS X and Xcode together with SpriteBuilder (as of this writing Mac OS X 10.9 and Xcode 6.0).

Figure 1.1: Cocos2D Technology Stack

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

1.2 Why Cocos2D

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

Add brief discussion on different frameworks

1.3 Introduction to Cocos2D

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

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

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

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

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

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

There are many more features - but this brief outline shows the most important ones and should give you an idea why almost all game developers these days use game engines. Let's take a closer look at how Cocos2D works.

1.3.1 The Cocos2D technology stack

Cocos2D is built on top of OpenGL ES 2.0. If you have ever written OpenGL code before, you know that it takes a lot of code even to render the most primitive scenes. OpenGL is a fairly low level framework that gives the graphics programmer a lot of control over how and when certain tasks are performed - more control then you need for most 2D games. Cocos2D abstracts all of these tasks for you. Many Cocos2D developers write entire games without writing any OpenGL code at all. The following diagram shows which technologies are used by Cocos2D:

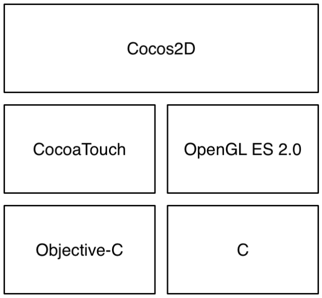


Figure 1.2: Cocos2D Technology Stack

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

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

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

How are games rendered in Cocos2D?

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

1.3.2 Scenes

Scenes are the basic building blocks of all Cocos2D games, they are the highest level on which game content can be structured. Each scene in Cocos2D is a full-screen canvas.

For every full-screen section of your game you will use *one* scene. **Add screenshots here** Here's an example from the MakeGamesWithUs game *Deep Sea Fury*: You can see that the game consists of the start scene, the gameplay scene and the game over scene.

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

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

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

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

1.3.3 Nodes

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

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

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

CCColorNode a node being displayed in one plain color.

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

CCButton a interactive node that can receive touches.

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

1.3.4 Scene Graphs

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

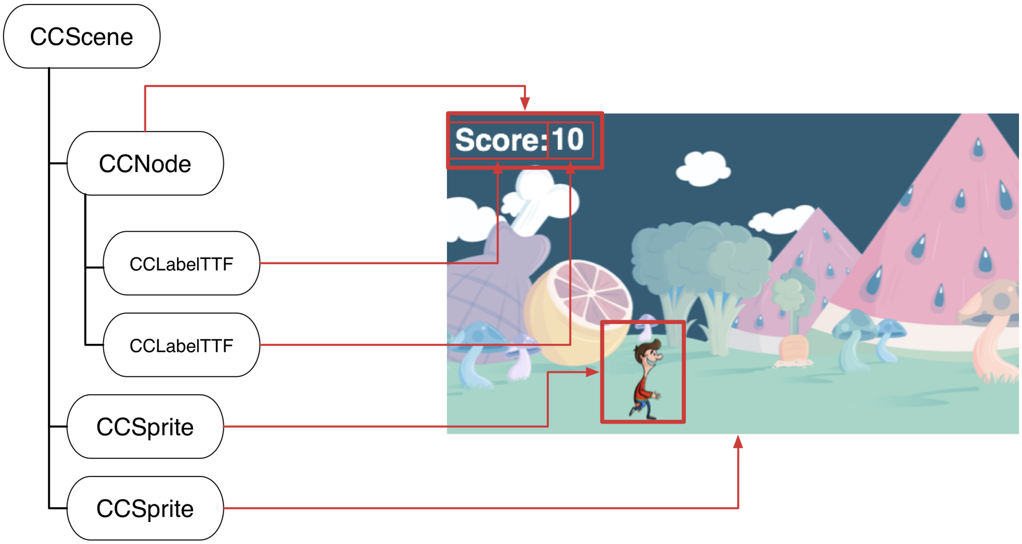


Figure 1.3: Cocos2D Scene Graph

On the left side of the image you can see the node hierarchy. On the first level you have the CCScene. As the first child we have a CCNode with two children of type CCLabelTTF. This CCNode is the first example of a grouping node, it groups the score caption label and the label displaying the actual score. Instances of CCNode don't have any appearance they are solely used to group other nodes. Throughout this book you will learn that it often makes sense to group nodes under certain parent nodes. The main reason is that all children are placed *relative* to their parents. So if we would want to move the scoreboard of the example above to the top right corner we would only have to move the parent node instead of both child nodes. As you can imagine this becomes even more relevant in games that have ten or more entries in their scoreboard.

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

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

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

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

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

1.3.5 An Introduction to CCNode

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

contentSizeInPoints the size of this node in points

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

anchor point the anchor point is the center point for rotations and the reference point for positioning this node

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

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

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

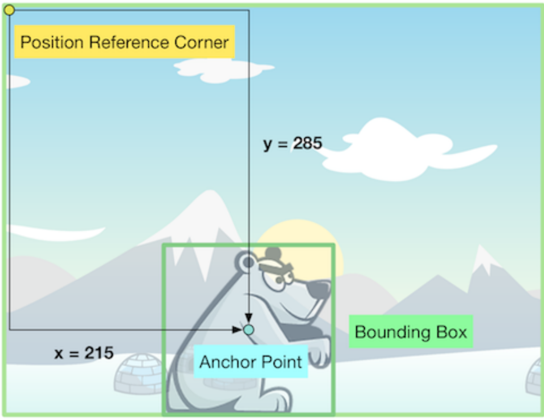


Figure 1.4: CCNode positioning example

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

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

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

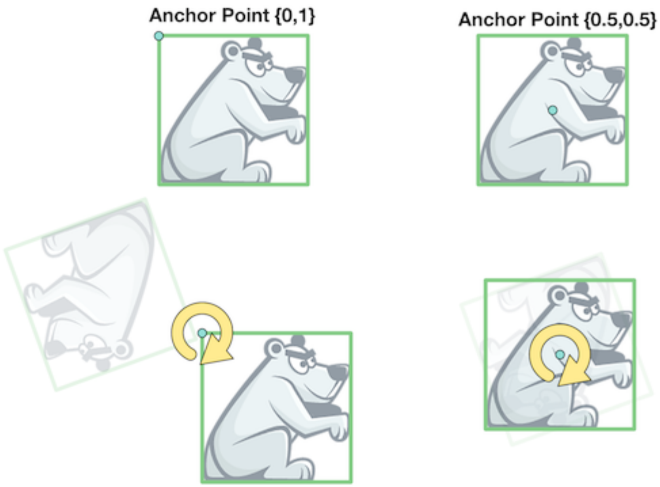


Figure 1.5: CCNode positioning example

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

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

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

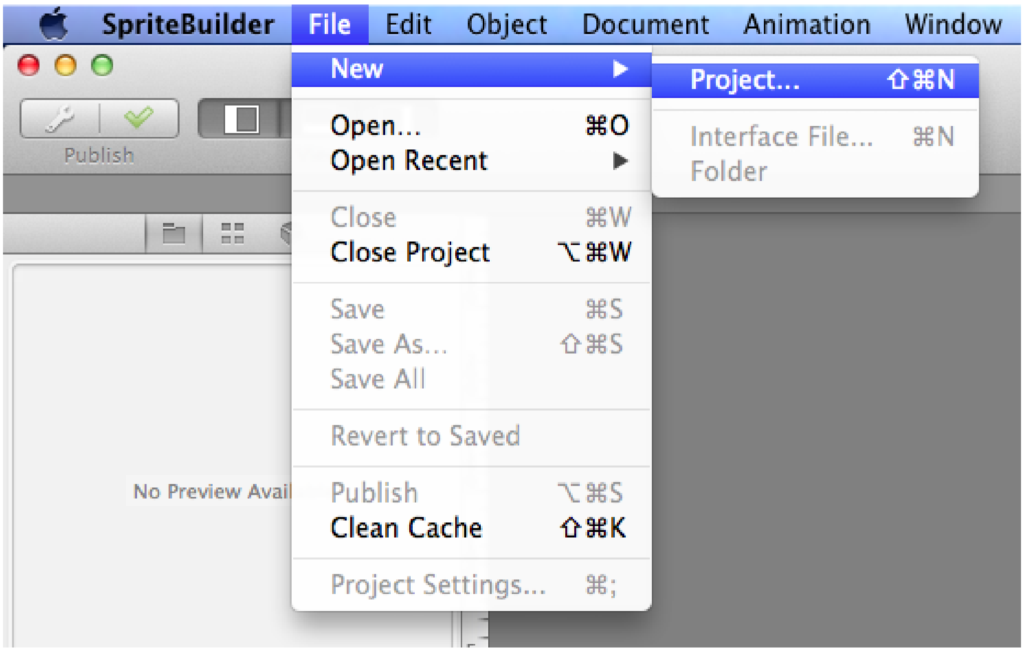
1.4 Introduction to SpriteBuilder

You have learned the fundamentals of the game engine we will use. Now we will take a look at a tool called SpriteBuilder which we will use to create the majority of our game content. The main purpose of SpriteBuilder is to provide a visual editor for the creation of scenes, animations and more. For most games you will create some basic mechanics in code (enemy movement, score mechanism, etc.) but you will create most of your game content in SpriteBuilder since it is a lot easier to create levels, menus and other scenes in an editor that provides you with a live preview instead of putting these scenes together in code.

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

1.4.1 Creating a first project

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



SpriteBuilder will ask for a name and a location for the new project. Name it *HelloSB*.

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

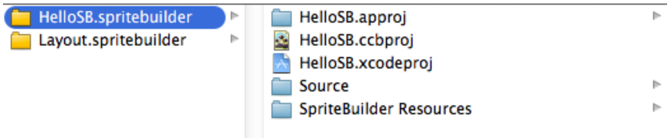


Figure 1.6: SpriteBuilder project folder structure

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

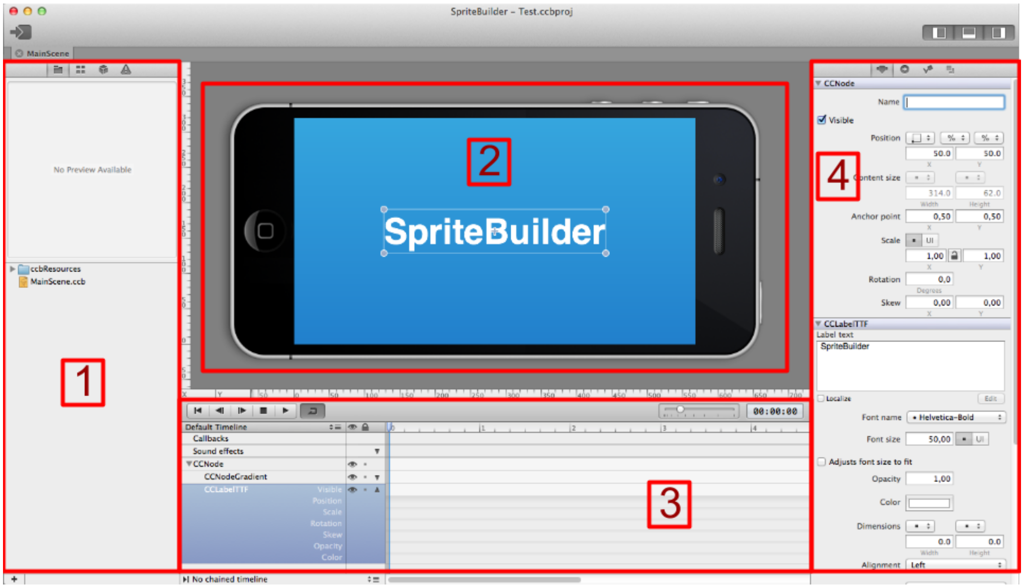
> **SpriteBuilder and Xcode**
>
> ☞ SpriteBuilder will create an Xcode project for every new project you create! The Xcode project will automatically contain the newest version of Cocos2D - very handy.

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

1.4.2 The Editor

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

The SpriteBuilder interface is divided into 4 main sections:

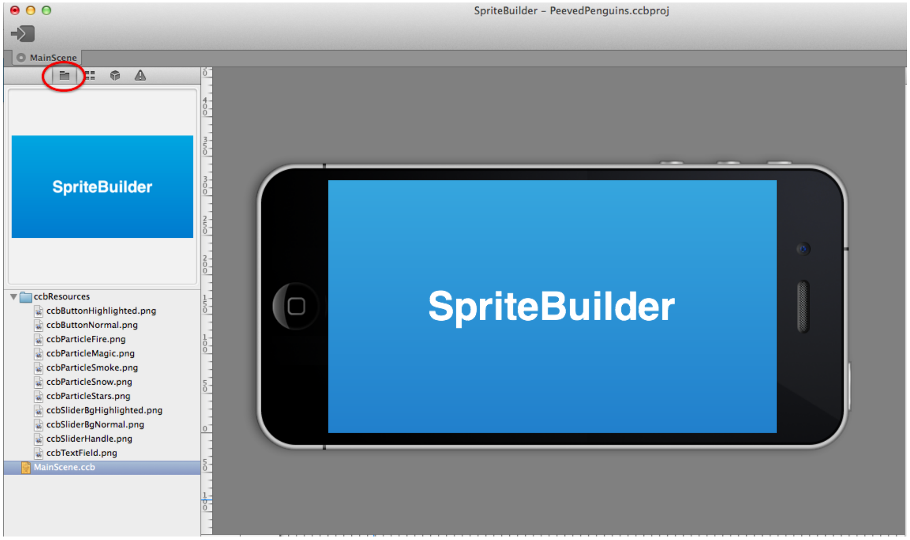


1. *Resource/Component Browser:* Here you can see the different resources and scenes you have created or added to your project. You can also select different types of Nodes and drag them into your scene.
2. *Stage:* The stage will preview your current scene. Here you can arrange all of the nodes that belong to a scene.
3. *Timeline:* The timeline is used to create animations within SpriteBuilder.
4. *Inspector:* Once you select a node in your scene, this detail view will display a lot of editable information about that node. You can modify positions, content (the text of a label, for example) and physics properties.

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

File View

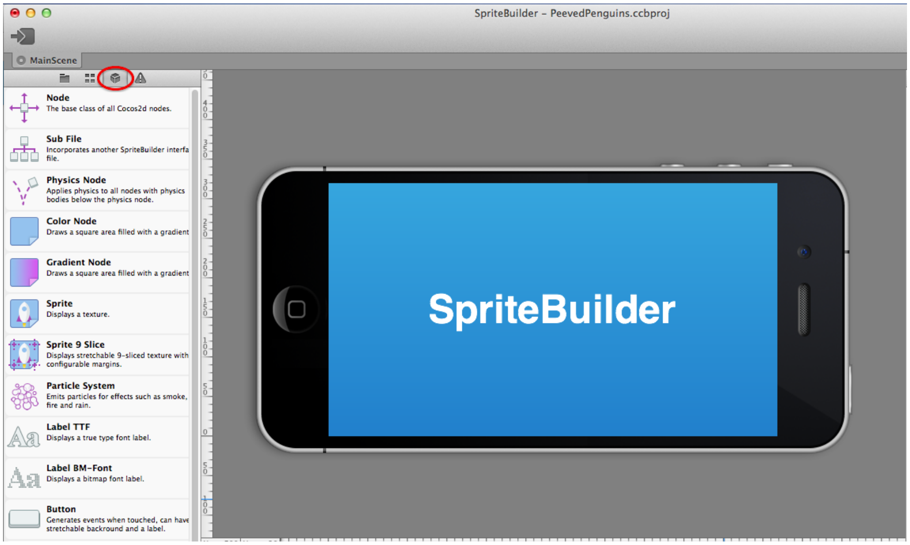
The first tab in the resource/component browser represents the *File View*. It lists all the *.ccb* files and resources that are part of the SpriteBuilder project:



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

Node Library

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



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

Inspector

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

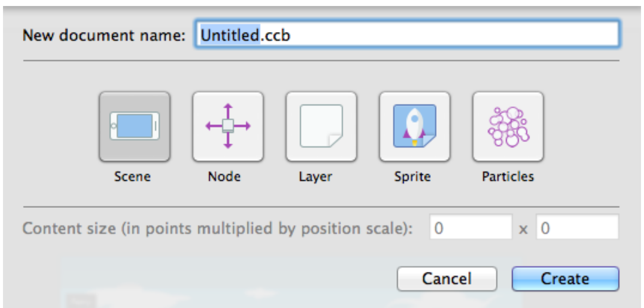
Code Connections

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

Code connections will be discussed in detail later on.

1.4.3 CCB Files

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



These are the different document types briefly explained:

Scenes will fill the full screen size of the device.

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

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

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

Particles is used to design particle effects.

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

1.4.4 How SpriteBuilder and Xcode work together

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

When creating a SpriteBuilder project, SpriteBuilder will create and maintain a corresponding Xcode project. In SpriteBuilder will you create multiple CCB Files that describe the content of the scenes in your game. You will also add the resources that you want to use in your game and set up code connections to interact with the code in your Xcode project. Xcode will be the place where you add code to your project and where you run the actual game.

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

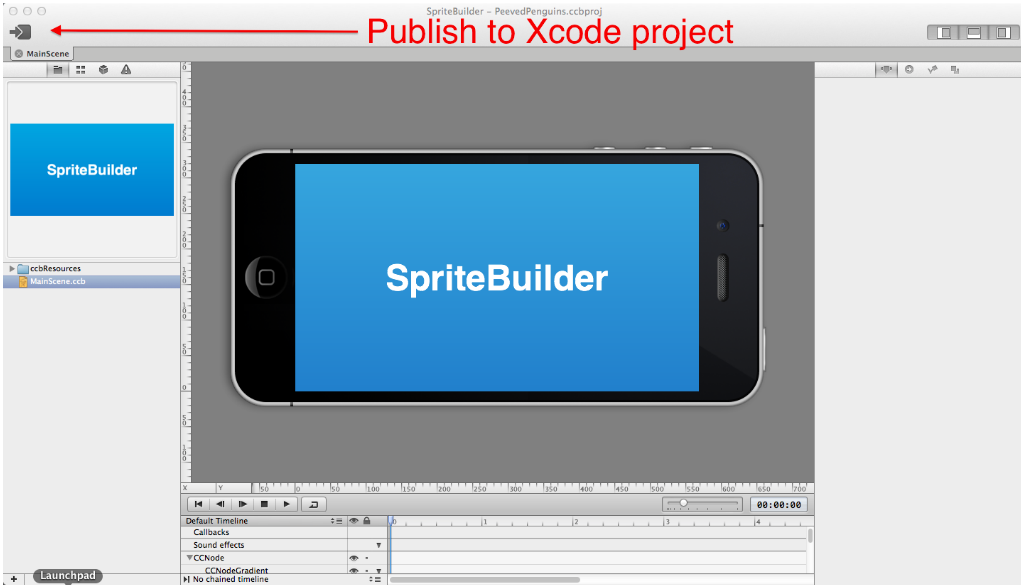


Figure 1.7: Use the publish button to update your Xcode project with the latest changes in your SpriteBuilder project.

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

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

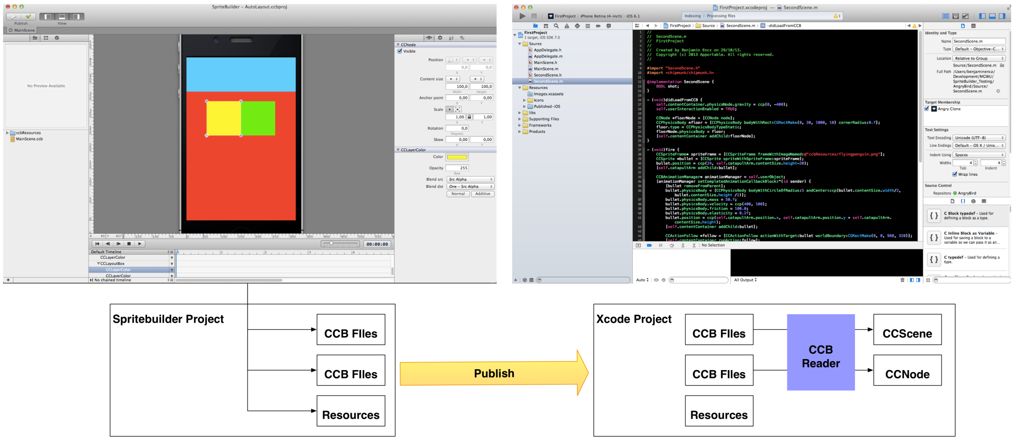


Figure 1.8: SpriteBuilder creates and organizes a Xcode project for you. Adding all the resources and scenes you have created.

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

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

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

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

1.4.5 Code Connections

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

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

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

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

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

1.5 A first SpriteBuilder project

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

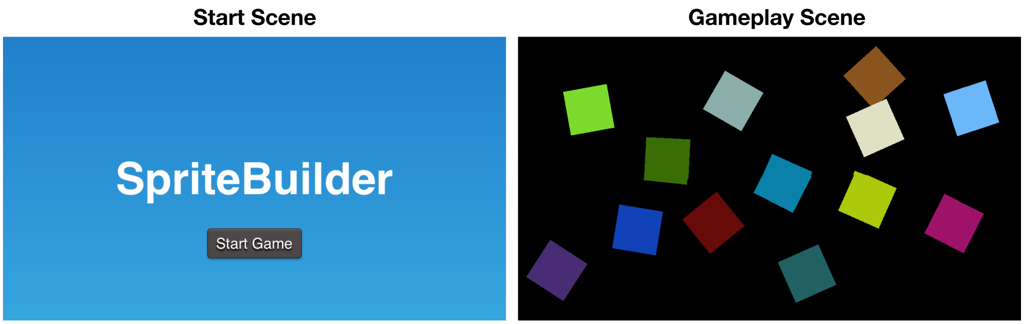


Figure 1.9: The project build throughout this chapter

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

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

1.5.1 Setting up the first scene

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

Positioning the first button

Start by adding a button to the first scene:

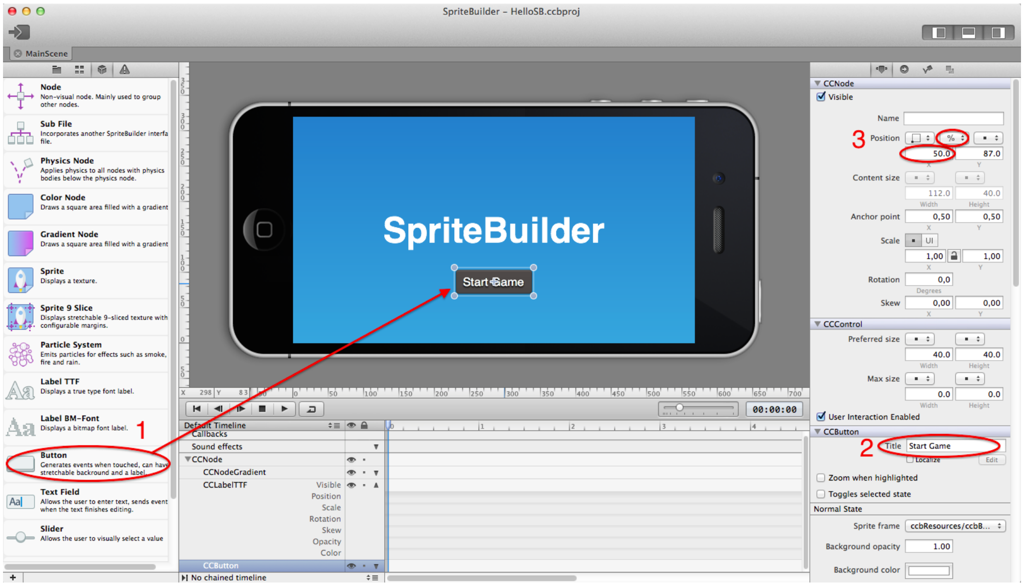


Figure 1.10: The project build throughout this chapter

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

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

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

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

Positioning System in Cocos2D and SpriteBuilder

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

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

Check the extra chapter on dynamic layouts in SpriteBuilder.

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

Setting up a code connection

Earlier you learned that SpriteBuilder has three types of code connections (1.4.5). Now we will use one of them in our project - *Callbacks*. Callbacks are only available to nodes that allow for some sort of user interaction (this means they need to be subclasses of `CCControl`). Buttons, next to Sliders and Text Fields are one of these types of nodes. Select the button we have added to the scene earlier and select the third tab of the right pane:

Figure 1.11: Nodes that allow user interaction can use callback methods to connect to the code base

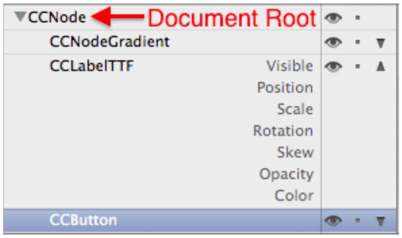
Inside the CCControll section you can see two options called *selector* and *target*. Here you can choose which method (selector) shall be called on which object (target) when this button is tapped by a user. As selector enter `startGame`. As target choose *Document Root*.

Targets and Selectors

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

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

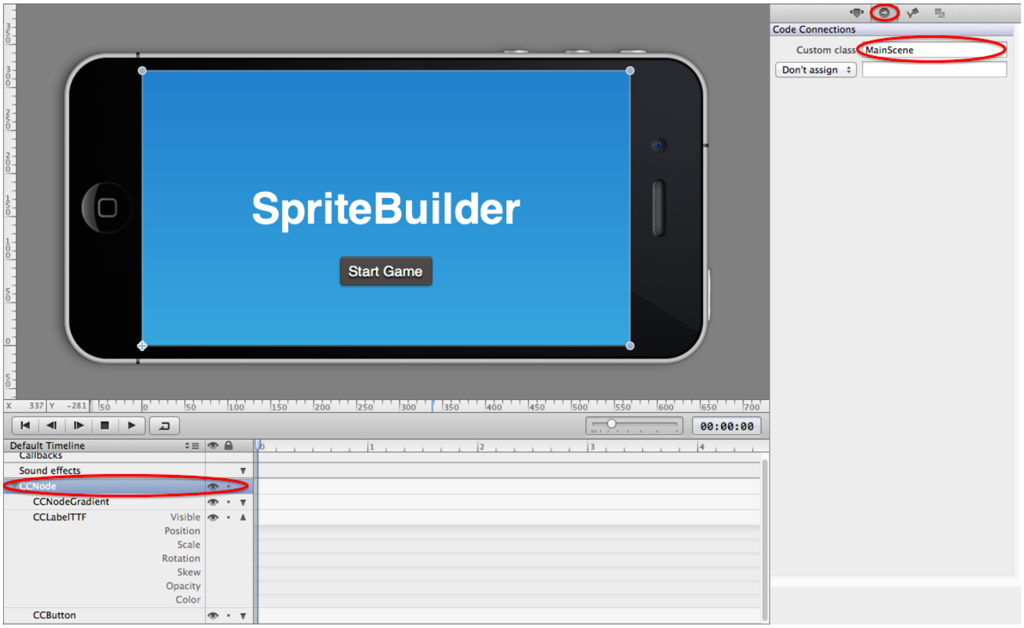
Document Root The document root is the highest node within the current CCB File. The hierarchy of the CCB File is shown in the SpriteBuilder timeline:



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

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

For our button we have decided that the `startGame` method should be called on the document root when the button is tapped. Next, we will have to implement this `startGame` method within our document root. But to which *class* could we add this method? In order to find that out we need to understand the concept of *Custom Classes* . Think about it - by default our document root is an instance of a plain CCNode class. Now we want to call a method called `startGame` on this object. Our problem: the CCNode class does not have a `startGame` method! This is where custom classes come to rescue us, they allow us to tell SpriteBuilder that our document root node should **not** be a plain CCNode but should be an instance of a class that we have created and that knows about our `startGame` method. To define a custom class for the document root you need to select the document root (the top-level CCNode) from the timeline and open the third tab in the right pane:



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

Requirements for Custom Classes

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

Adding Code to a SpriteBuilder project

When creating games with SpriteBuilder we are always working with two tools. SpriteBuilder to create interfaces and scenes (our game content) and Xcode to add code (game mechanics, etc.). Now we will add our first few lines of code to the `MainScene` class. Now it's time to publish the changes in our SpriteBuilder project, so that they are available in our Xcode project. Use the publish button in the top left corner of the SpriteBuilder interface (1.4.4).

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

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

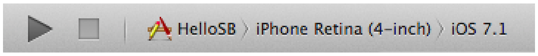
```
- (void)startGame {
  CCLOG(@"Start Button Pressed!");
}
```

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

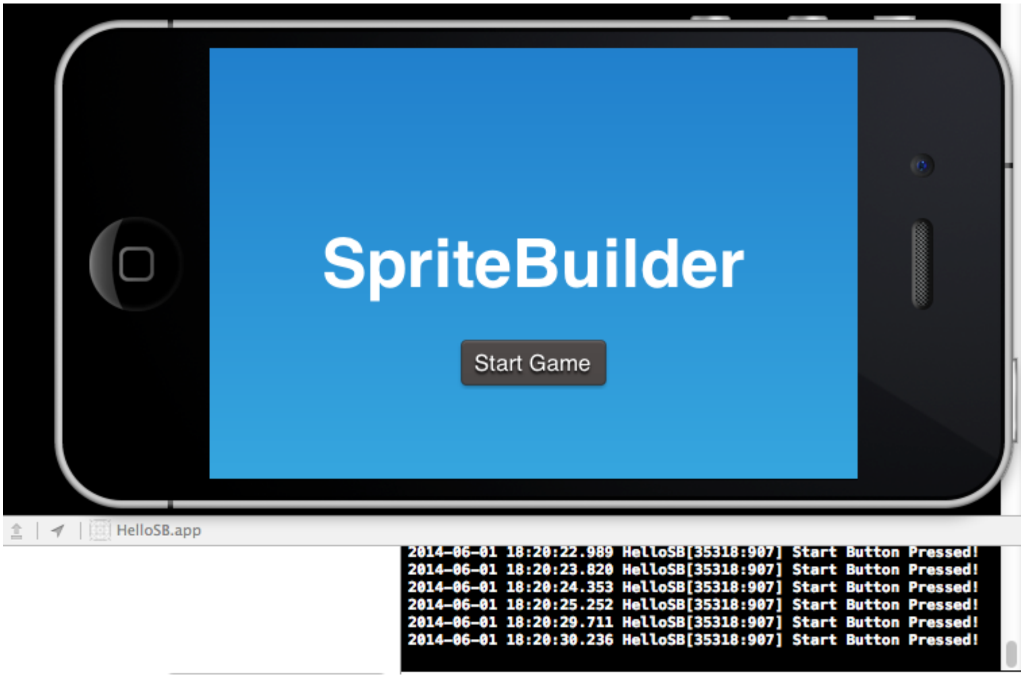
Displaying the console in Xcode

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

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



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



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

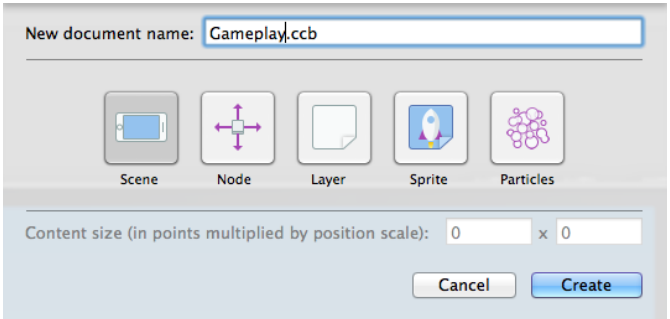
Common Error 1.1

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

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

1.5.2 Creating the Gameplay Scene

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



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

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

1.5.3 Adding a Scene Transition

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

Cocos2D has one central class that is responsible for displaying the active scene and generating transitions between different scenes: `CCDirector`. CCDirector is implemented as a singleton - thus there's only one CCDirector per Cocos2D game. The instance can be accessed through the class method `[CCDirector sharedInstance]`.

CCDirector is versatile!

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

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

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

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

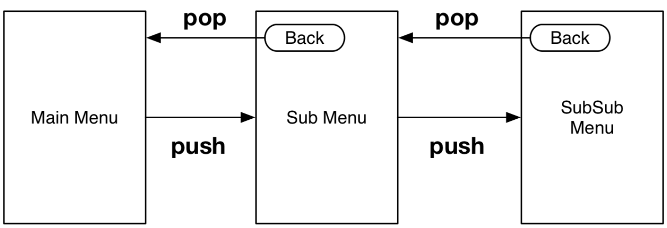
Before we look into using transition effects, let's take a look at the differences between pushing and replacing a scene.

Replacing scenes vs. pushing scenes

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

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

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



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

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

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

```
[[CCDirector sharedDirector] popScene];
```

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

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

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

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

Scene transitions - the right way

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

Adding transition effects

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

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

Implementing a scene transition for our game

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

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

```
    [[CCDirector sharedDirector] presentScene:gameplayScene
        ↪ withTransition:transition];
}
```

Now that you are familiar with scene transitions, the only interesting line should be the one where we use the `CCBReader` to load a CCB File. The CCBReader class was briefly introduced at the beginning of this chapter (1.4.4). It is capable of reading SpriteBuilders *.ccbi* files and creating the according Cocos2D classes from the information stored in them. Whenever we want to load a scene or any other type of node that we created in SpriteBuilder into code we use the CCBReader class. In the lines shown above, we load the content of our *Gameplay.ccb* into a variable called `gameplayScene`. The `loadAsScene:` method wraps whatever scene graph you load into an instance of `CCScene`, use it whenever you want to load a CCB File as a scene. Then we create a simple fade transition and store that object in the `transition` variable. Finally we use the `CCDirector` to present our loaded scene with the transition we just created.

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

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

.ccb and .ccbi

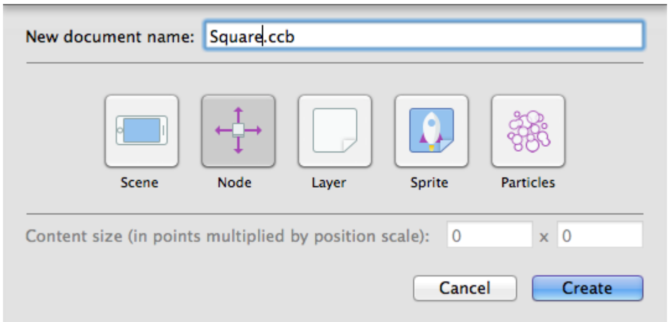
The files with the file extension *.ccb* are in XML-format and are used by SpriteBuilder to store and read information about a scene or node created in SpriteBuilder. When a SpriteBuilder project gets published, SpriteBuilder generates a binary version of each *.ccb* file. The file extension for these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

1.5.4 Implementing the Gameplay

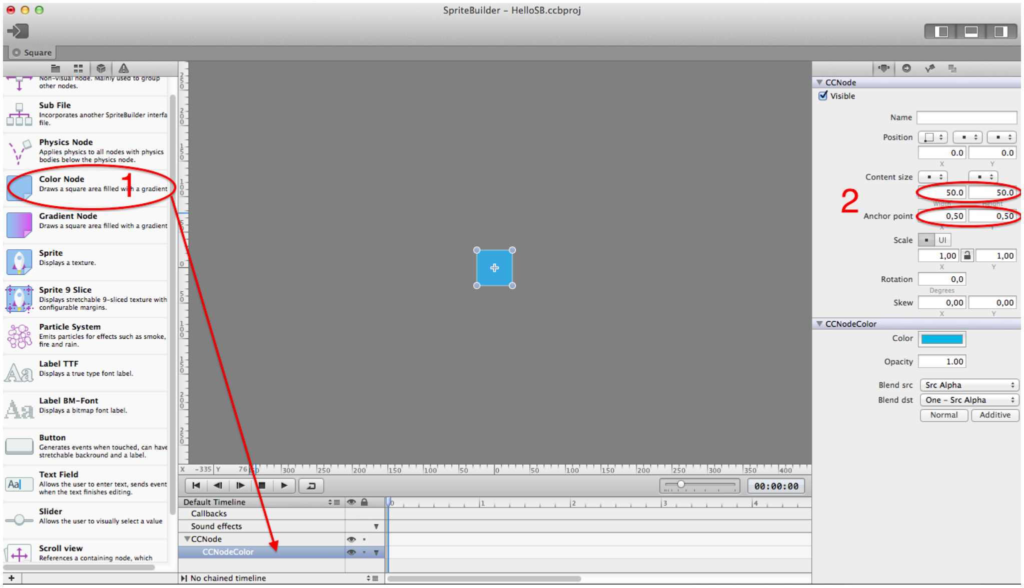
Now it's time to implement the actual gameplay. For our first project we want to keep that fairly simple. Whenever a user touches the screen, we want to add a rotating square with a random color to the gameplay scene. We position the square at the location of the touch.

Creating the Square CCB File

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



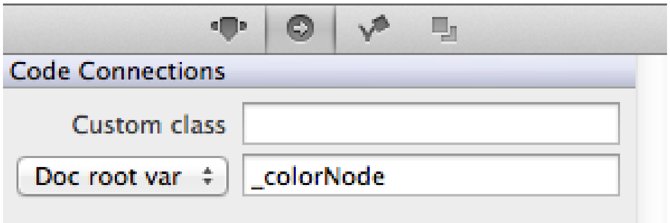
The squares we generate in the game shall have a color. A default CCNode cannot display a color. In order to display a color we need to use a `CCNodeColor`. The SpriteBuilder node for a CCNodeColor is called *Color Node*. The root node of every CCB File is a plain CCNode, that cannot be changed. This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



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

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

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

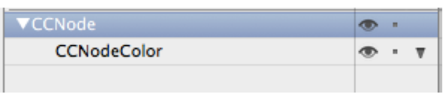


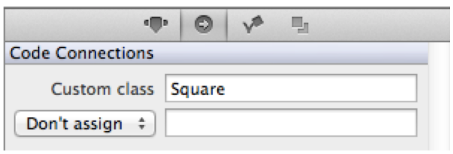
As the variable name (entered in the text field), choose *_colorNode*. As the second option you need to choose the object to which this variable will be assigned to. Just as for callbacks you can choose between the *Document Root* and the *Owner* (1.5.1). We choose the *Document Root*, which means that SpriteBuilder will attempt to store a reference to the *Color Node* in an instance variable called *_colorNode* on the root node object of this CCB File. We now face the same 'problem' as earlier when we set up a *Callback*. The root node of *Square.ccb* is a plain CCNode and a plain CCNode does not have an instance variable called *_colorNode*! We once again need to define a custom class for the root node of this CCB File.

Variable Assignments, Callbacks and Custom Classes

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

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





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

Setting up a custom class for the Gameplay

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

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

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

Creating the Square class

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

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

```
#import "Square.h"

@implementation Square {
  CCNodeColor *_colorNode;
}

@end
```

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

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

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

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

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

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

You will get to see lots of examples of how to use the lifecycle methods throughout this book, for now we know that we need to override `onEnter` to pick and apply a random color for our square as soon as it gets added to the Gameplay scene. It is also important to know that you need to call the `super` implementation if you override any of the `onEnter...` or `onExit...` methods. CCNode has its own implementation of these methods and they are important for the functionality of the framework - if you do not call them this will result in unexpected behaviour throughout your game.

Overriding Cocos2D lifecycle methods

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

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

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

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

  // arc4random_uniform(N) generates a random number between 0 and N-1
  float red =   arc4random_uniform(256) / 255.f;
  float green = arc4random_uniform(256) / 255.f;
```

```
    float blue =  arc4random_uniform(256) / 255.f;

    _colorNode.color = [CCColor colorWithRed:red green:green blue:
        ↪ blue];
}
```

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

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

The Action System consists of dozens of subclasses of `CCAction` - a majority of these actions represent some type of animated movement or transformation. `CCActionMoveTo` for example moves a node to a target position within a provided time interval. This is how to use it:

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

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

More about the Cocos2D Action System

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

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

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

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

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

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

Creating the Gameplay class

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

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

Adding Touch Handling to the Gameplay

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

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

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

touchBegan: called when a touch begins

touchMoved: called when the touch position of a touch changes

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

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

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

☞ **The Cocos2D Touch System**

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

Now that you know the basics, let's implement touch handling for the `Gameplay` class. First, we need to enable user interaction. A great place to do this is in the `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.m*:

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

    self.userInteractionEnabled = YES;
}
```

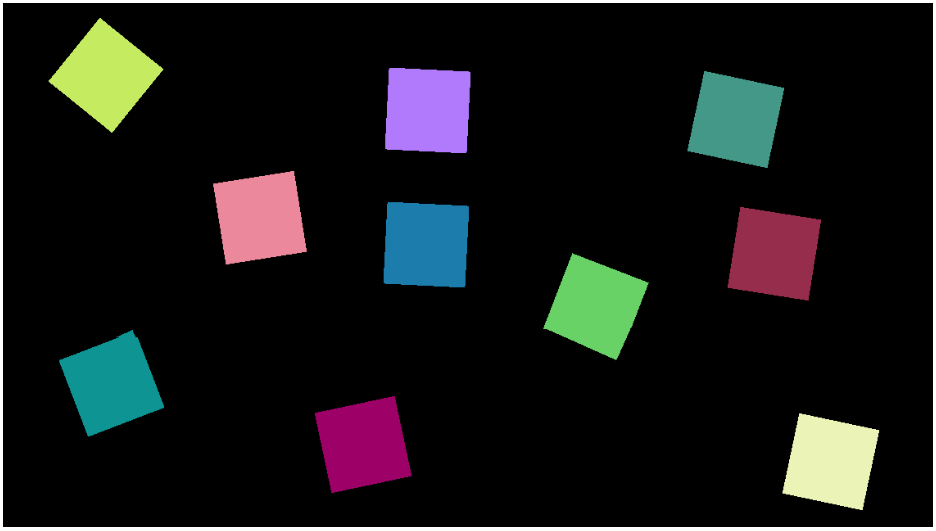
As discussed earlier you need to call the `super` implementation of the lifecycle method you are overriding. In the second step we are setting `userInteractionEnabled` to `YES`. Now Cocos2D knows that this node wants to receive touch events.

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

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

We have now implemented the `touchBegan:` method. It will be called every time the user taps onto the gameplay scene. As one parameter of this method we receive a `UITouch`. The `UITouch` stores all information about the touch. Cocos2D adds a method called `locationInNode:`. This method returns the touch position relative to the provided node. In the first line we call this method to receive the touch location within the gameplay scene (referred to by `self`). In the next line we load one `Square` node using the `CCBReader`.

Then we add that loaded square as a child to the gameplay scene. The `addChild:` method of CCNode will add the square to the node hierarchy of the gameplay scene. As soon as node becomes part of the node hierarchy of the currently active scene it will be displayed on the screen. Finally we choose a position for the square. We provide the touch position that we determined in the first line - this way the square will be spawned exactly at the position touched by the player. Now it's time to run your project again. Once the game started, select the *Start* button to go to the gameplay scene then click onto the screen multiple times to simulate touches. Every time you simulate a touch you should see a new square spawn at the touch position:



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

1.5.5 Exercises

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

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

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

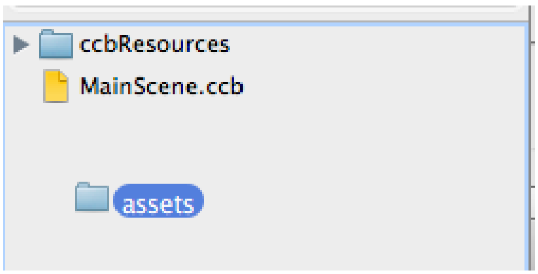
2 A Game with Assets in SpriteBuilder

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

2.1 Adding Assets to a SpriteBuilder project

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

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



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

2.2 Asset Handling in SpriteBuilder and Cocos2D

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

phone resolution for non-retina iPhone and Android devices

phone-hd retina resolution for iPhone and Android

tablet resolution for non-retina iPad and Android tablets

tablet-hd resolution for retina iPad and Android tablets

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

1x non-retina images

2x retina images

4x double sized retina images

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

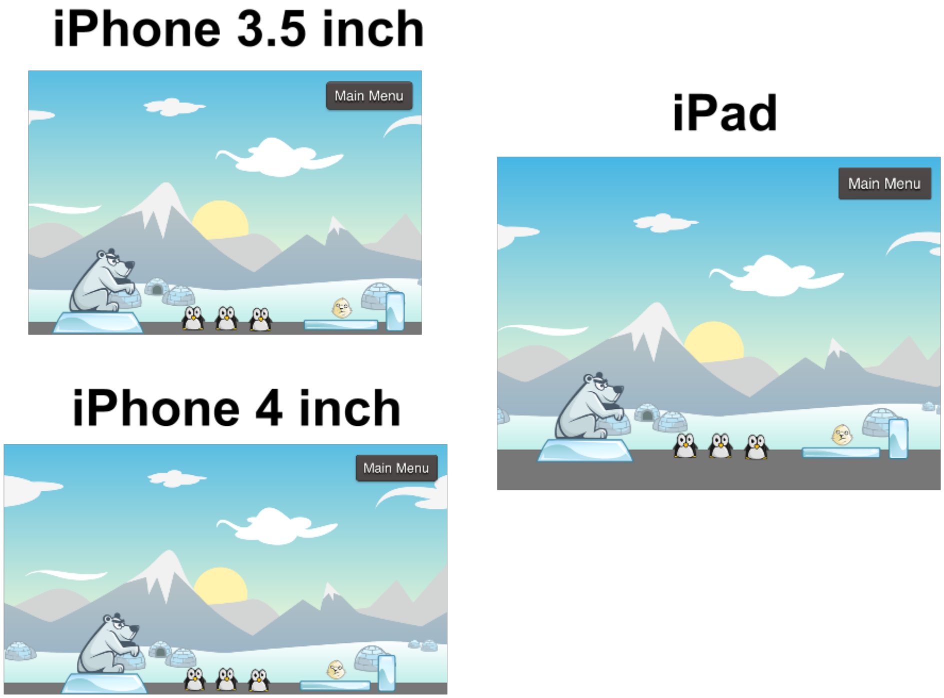
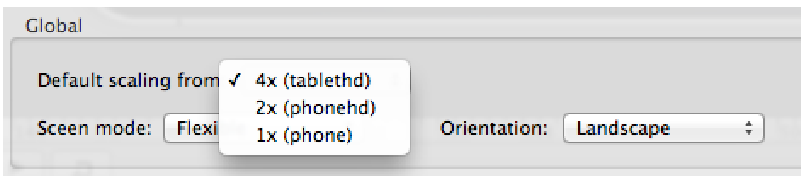


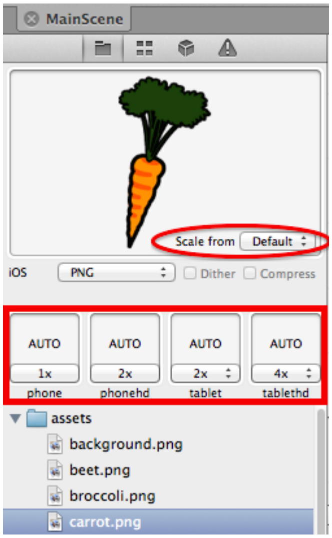
Figure 2.1: From our tutorial *Dynamic Layouts with SpriteBuilder and Cocos2D*

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



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

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



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

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

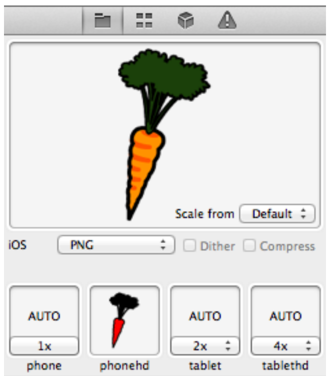
Device	Default Resolution Type	Size on Screen (points)	Size in Pixels
iPhone	1x	50x50	50x50
iPhone Retina	2x	50x50	100x100
iPad	2x	100x100	100x100
iPad Retina	4x	100x100	200x200

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

A last interesting case are background images that you want to work for all 4 resolutions. A solution is discussed in the Q&A section (3.2.1).

Different images for different devices

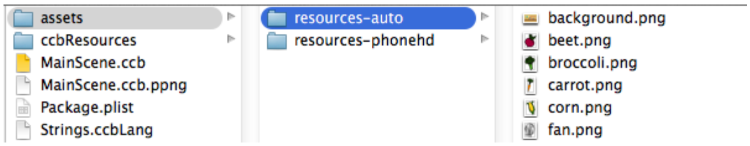
You can not only change the scaling option for an asset on different devices, you can even use an entirely different image for a certain resolution. You can do that by dragging an image **that is currently not part of the SpriteBuilder project** from Finder into one of the four boxes below the asset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



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

Behind the scenes

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



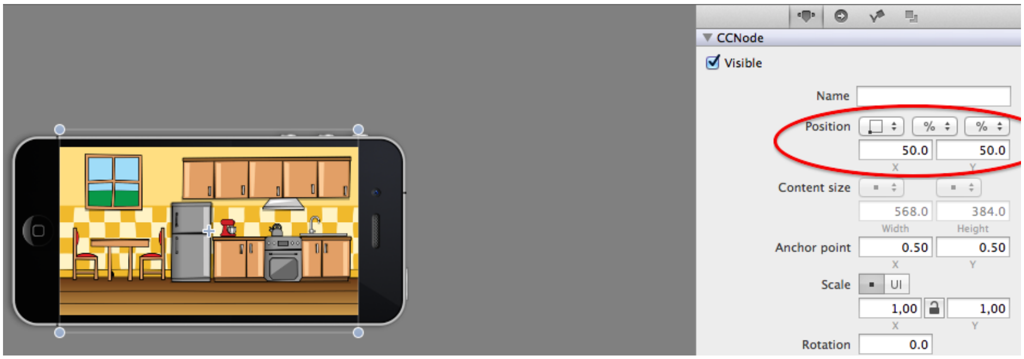
You will see that SpriteBuilder groups images inside the assets folder into a *resources-auto* folder, all images in that folder are subject to automatic downscaling. If you explicitly add images for a certain resolution as shown with the carrot in the above example, a new folder for that resolution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

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

2.3 Adding the background image

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

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



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

2.4 Create falling objects

Now let's dive into the implementation of the actual game. The next step should be adding falling objects. Our game will have two categories of objects, ones that should be caught (food) and ones that shouldn't (electronic devices).

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

2.4.1 Create a falling object class

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

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

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

Create a new class called `SBBFallingObject` and make it a subclass of CCSprite. The best way to represent different types in Objective-C is using enumerations. Add this enum definition to *SSBFallingObject.h*, above the `@interface` block:

```
typedef NS_ENUM(NSInteger, SBBFallingObjectType) {
    SBBFallingObjectTypeGood,
    SBBFallingObjectTypeBad
};
```

Additionally we add a property to store the type and an initializer that allows us to create falling objects with a certain type. Your complete *.h* file should look similar to this:

```
#import "CCSprite.h"

typedef NS_ENUM(NSInteger, SBBFallingObjectType) {
    SBBFallingObjectTypeGood,
```

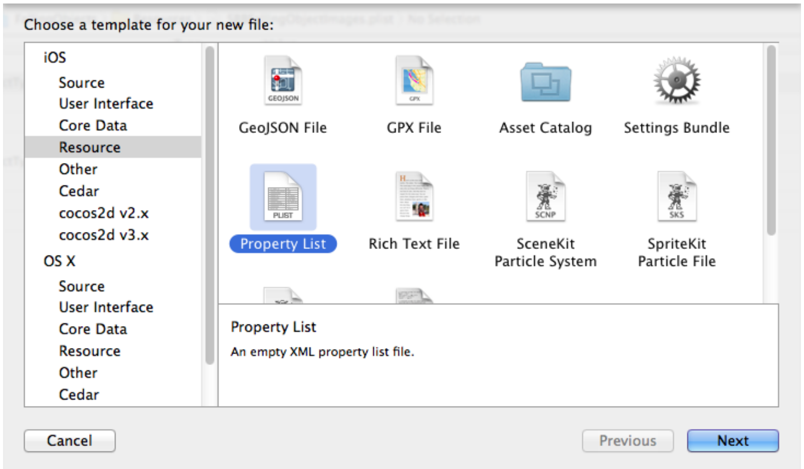
```
    SBBFallingObjectTypeBad
};

@interface SBBFallingObject : CCSprite

@property (nonatomic, readonly) SBBFallingObjectType type;

- (id)initWithType:(SBBFallingObjectType)fallingObjectType;

@end
```

We define the property to be *readonly* because will not support changing the type of a falling object after it has been created.

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 `SBBFallingObject` class. When `initWithType:` gets called we choose a random asset and apply it as a texture to the `SBBFallingObject`.

How can we know which assets we have available and which of them represent objects that we should catch and that we shouldn't? 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 Xcodes menu. Then you need to select *Resource* from the left panel and choose *Property List* on the right:



As the name choose *SBBFallingObjectImages*. 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 `SBBFallingObject` 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 Objective-C as well - there are some very convenient methods to load *plists* in code. During each game the player plays 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 `SBBFallingObject`. Instead we should only load it once and then keep a reference to it for future use. We can create `static` variables to hold the content of the *plist* once it is loaded.

Static variables in Objective-C

In Objective-C the concept of class variables does not exist but the C-keyword `static` makes a variable only accessible from the file it is defined in. Therefore defining a `static` variable in a *.m* file is very similar to a class variable.

We can then override the class method `initialize` to load the *plist* and store the results in the `static` variables. The `initialize` method is only called once per class and is guaranteed to be called before the class is used, so this is the perfect place to do some class level preparation work.

Open *SBBFallingObject.m* to start implementing this. First, define to `static` array references:

```
static NSArray *fallingObjectTypeGoodImageNames;
static NSArray *fallingObjectTypeBadImageNames;
```

Then implement the `initialize` method to create two arrays and assign them to these variables:

```
+ (void)initialize {
  // load possible images from plist
```

```
  NSURL *plistURL = [[NSBundle mainBundle] URLForResource:@"
      SBBFallingObjectImages" withExtension:@"plist"];
  NSDictionary *fallingObjectImageNames = [NSDictionary
      dictionaryWithContentsOfURL:plistURL];
  fallingObjectTypeGoodImageNames = fallingObjectImageNames[@"
      FallingObjectTypeGoodImages"];
  fallingObjectTypeBadImageNames = fallingObjectImageNames[@"
      FallingObjectTypeBadImages"];
}
```

First we need to get the path to the *plist*. Then we can load the *plist* using a convenience method of `NSDictionary`, this is possible because the root object in our *plist* is a dictionary. From that dictionary we can extract the two arrays with the images for good objects and bad objects and assign them to our `static` variables. When the `SBBFallingObject` class is used the first them this code will run and set up the two arrays for us!

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

- Pick a random image based on the object type
- Call the `initWithImageNamed:` of CCSprite

This is what the initializer should look like:

```
- (id)initWithType:(SBBFallingObjectType)fallingObjectType {
  // pick a random image name based on the FallingObjectType chosen
  NSString *imageName = nil;

  if (fallingObjectType == SBBFallingObjectTypeGood) {
    int randomImageIndex = arc4random_uniform((int)[
        fallingObjectTypeGoodImageNames count]);
    imageName = fallingObjectTypeGoodImageNames[randomImageIndex];
  } else if (fallingObjectType == SBBFallingObjectTypeBad) {
    int randomImageIndex = arc4random_uniform((int)[
        fallingObjectTypeBadImageNames count]);
```

```
    imageName = fallingObjectTypeBadImageNames[randomImageIndex];
  }

  // call super initializer with selected image
  self = [super initWithImageNamed:imageName];

  if (self) {
    self.anchorPoint = ccp(0, 0);
    _type = fallingObjectType;
  }

  return self;
}
```

Overall a prety straightforward method. We first check which type of object we are initializing. Based on the type we pick one of the arrays that have been populated with the content from the *plist*. Then we pick a random element from that array. That random element is our file name, we pass that on to the CCSprite initializer `initWithImageNamed:`. The we initialize two values. We set the `anchorPoint` of the `SBBFallingObject` to the bottom left corner, that will make it easier to determine the spawn position later on. Additionally we store the object type that has been passed in as a parameter to the initializer. That's all we need in order to use `SBBFallingObject`! Now we can move on and spawn some objects.

Accessing properties in initializers

In Objective-C it is common to access properties in initializers through their instance variables. The goal is to avoid potential side effects that could occur by calling setter/getter methods in the class itself or in subclasses while the object is not entirely initialized yet.

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 *SBBMainScene.m*. Since we will be using the `SBBFallingObject` class, start by adding an `import` statement:

```
#import "SBBFallingObject.h"
```

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 increasingly amount of memory we will need to take care to remove 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. Add the array definition to the private interface of the class in *SBBMainScene.m*:

```
@interface SBBMainScene()

@property (nonatomic, strong) NSMutableArray *fallingObjects;

@end
```

We 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. Let's create two constants at the top of the file:

```
static const CGFloat kFallingSpeed = 100.f;
static const CGFloat kSpawnFrequency = 0.5f;
```

Naming constants in Objective-C

A common way to name constants in Objective-C is using a leading small *k* for private constants (e.g. `kFallingSpeed`) and using the class name as prefix for public constants which are declared in the *.h* file (e.g. `SBBFallingObjectFallingSpeed`.

In the `init` method we need to initialize our array of falling objects:

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

    if (self) {
        _fallingObjects = [NSMutableArray array];
    }

    return self;
}
```

We are going to spawn falling objects with the frequency that we have defined in the constant `kSpawnFrequency`. 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:

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

    // spawn objects with defined frequency
    [self schedule:@selector(spawnObject) interval:kSpawnFrequency];
}
```

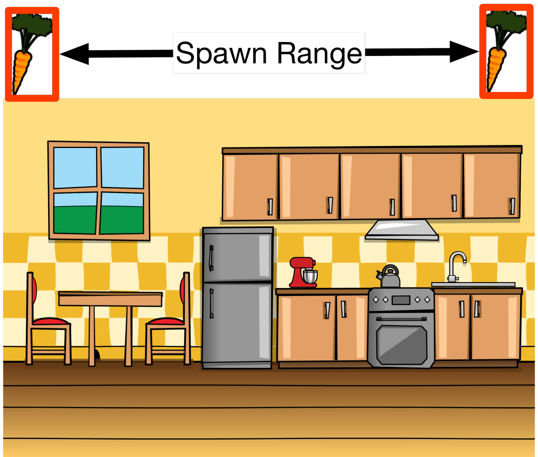
All we need to provide is a *selector*, which simply means a method name, and a frequency at which it shall be called. Now, as soon as the `SBBMainScene` is presented on the stage, the `spawnObject` method will be called twice a second. To complete the spawning functionality we will have to implement 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:

```
- (void)spawnObject {
```

```
    // generate a random number to decide wether to spawn a positive or a negative object
    NSInteger randomNumber = arc4random_uniform(2);

    SBBFallingObject *fallingObject = nil;

    if (randomNumber == 0) {
        fallingObject = [[SBBFallingObject alloc] initWithType:SBBFallingObjectTypeBad];
    } else if (randomNumber == 1) {
        fallingObject = [[SBBFallingObject alloc] initWithType:SBBFallingObjectTypeGood];
    }

    // add all spawning objects to an array
    [self.fallingObjects addObject:fallingObject];
    // spawn all obejcts at top of screen and at a random x position with scene bounds
    CGFloat xSpawnRange = ( (CGFloat) self.contentSizeInPoints.width) - CGRectGetMaxX(fallingObject.boundingBox);
    CGPoint spawnPosition = ccp(arc4random_uniform(xSpawnRange), self.contentSizeInPoints.height);
    fallingObject.position = spawnPosition;
    [self addChild:fallingObject];
}
```

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:

```
- (void)update:(CCTime)delta
```

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:

```
- (void)update:(CCTime)delta {
    // use classic for loop so that we can remove objects while iterating over the array
    for (int i=0; i < [self.fallingObjects count]; i++) {
        SBBFallingObject *fallingObject = self.fallingObjects[i];
```

```

        // check if falling object is below the screen boundary
        if (CGRectGetMaxY(fallingObject.boundingBox) <
          ↪ CGRectGetMinY(self.boundingBox)) {
            // if object is below screen, remove it
            [fallingObject removeFromParent];
            [self.fallingObjects removeObject:fallingObject];
        } else {
            // else, let the object fall with a constant speed
            fallingObject.position = ccp(fallingObject.position.x,
              ↪ fallingObject.position.y - (kFallingSpeed * delta
              ↪ ));
        }
    }
}
```

The interesting aspects of the code snippet above are 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` methods to determine...

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/.

3 Q&A

3.1 Core Cocos2D and SpriteBuilder

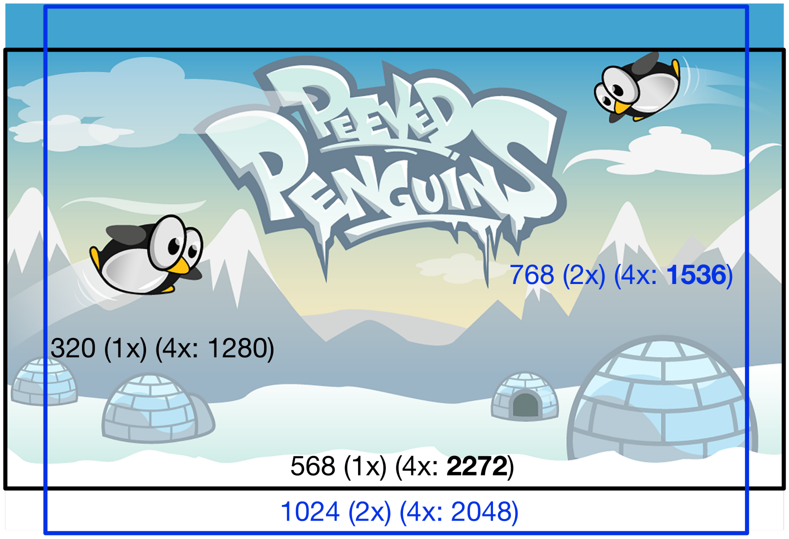
3.1.1 How can I programmatically change the texture of an existing CCSprite

CCSprite provides a `spriteFrame` property. You can initialize a `CCSpriteFrame` with a new image and use it to set the `spriteFrame` property.

3.2 Designing for multiple devices

3.2.1 Which size does a background image for all four resolutions need to have?

Since the iPad and the iPhone have different aspect ratios your background images need to be high enough for the iPad and wide enough for the iPhone (assuming you have a landscape game). The perfect image size is 2272px X 1536px, here is an image that explains why:



In blue you can see the outline of an iPad. An iPad has a resolution of 1024px x 768px, SpriteBuilder uses *2x* resolution for the iPad. To get the required *4x* resolution we need to multiply the value by 2, resulting in 2048px x 1536px. In black you can see the outline of an iPhone. A 3.5-inch iPhone has a resolution of 480px x 320px, but we want our image to also fit the 4-inch iPhone therefore we calculate with a resolution of 568px x 320px. This is a *1x* resolution, to get to a *4x* resolution we need to multiply the values by 4, resulting in a required resolution of 2272px x 1280px. You can see, the iPhone requires a wider image, the iPad a higher one. A resolution of 2272px X 1536px satisfies both requirements.

3.3 Advance Techniques

3.3.1 Can I take a screenshot in code or even generate a level preview?

Yes you can! Cocos2D has a awesome class called `CCRenderTexture` that allows you to render a node graph into an image instead of directly to the screen. You can use this for all kinds of features. Taking screenshots, generating level previews or even rendering screen-in-screen! And it is pretty simple to use, here's the outline:

```
CCRenderTexture *renderTexture = [CCRenderTexture
    ↪ renderTextureWithWidth:2048 height:self.level.
    ↪ contentSizeInPoints.height];
[renderTexture beginWithClear:0 g:0 b:0 a:1.f];
[self visit];
[renderTexture end];
[renderTexture saveToFile:@"Screenshot.jpeg"];
```

Index