
30
MAKE A GAME LIKE
TIMBERMAN
JEREMY NOVAK

Table of Contents

Introduction 6

Required Software 6

Why learn SpriteKit? 7

Why learn Objective-C? 7

Naming conventions used in this book 8

Design conventions used in this book 9

Use of third party libraries 9

Adventure Time 10

Chapter 1 - Getting Started 11

Downloading the project from GitHub 11

Touring the Project 12

View Controller group 13

Utilities Group 16

Scenes Group 17

GameResources Folder 21

Supporting Files 23

Assets.xcassets 25

Build and Run 25

Conclusion 26

Chapter 2 - Game World 27

GameTextures 27

SpriteNames.h 33

Clouds 34

Update GameScene 37

Build and Run 38
Birds 39
Update GameScene 43
Build and Run 44
Forest 45
Build and Run 46
Conclusion 47
Need Help? 47
Chapter 3 - Log Stack....48
SpriteNames.h 48
Contact.h 49
Update PrefixHeader.pch 49
Branch 50
StackController 52
Update GameScene 58
Build and Run 59
Conclusion 60
Need Help? 60
Chapter 4 - Player....61
Updating SpriteNames.h 61
Smoke 62
Player 64
Updating GameScene 69
Build and Run 72
Conclusion 73
Need Help? 73

Chapter 5 - Game State....74
Update SpriteNames.h....75
TutorialButton....75
PlayButton....78
Update GameScene with GameState....80
Change touchesBegan:withEvent: to use GameState....82
Update update: to use GameState....83
Update didBeginContact: to use GameState....84
Update setupScene....84
One more thing… 85
Build and Run....87
Conclusion....88
Need Help? 88
Chapter 6 - Adding the Timer....89
Update SpriteNames.h....90
ScoreLabel....90
TimeBar....92
Updating GameScene....94
Build and Run....98
Conclusion....99
Need Help?....99
Chapter 7 - Wrapping Up....100
LoadScene....100
Update GameViewController....102
Update Utilities.h....103
Build and Run....103

GameSettings 103
Update Player 106
Update SpriteNames.h 107
PauseButton 107
Updating GameScene 109
Build and Run 113
Scoreboard 114
Update GameScene 116
Build and Run 117
Conclusion 118

Introduction

In this book, we will create a clone of the popular game Timberman with SpriteKit.

Don’t worry if you are brand new to Xcode, Objective-C, or even programming. No experience is necessary to complete this project. When you complete the final chapter you will have a fully functioning game that runs on iOS.

A complete discussion on SpriteKit, Objective-C, Xcode and git is beyond the scope of this book. After completing this book, I recommend spending time learning the more advanced features of each. You can do a lot with a little knowledge of each, but they become very powerful once you know how to use the advanced features.

Required Software

You will need a Mac running OS X 10.10 Yosemite or newer and Xcode 7 or newer. You can download the latest Xcode from the Mac App Store for free here.

Please install the latest version of Xcode now from the Mac App Store if you haven’t already done so.

Why learn SpriteKit?

SpriteKit has two really big advantages. It is developed by Apple and optimized for their devices. This opens up a fairly large range of opportunities for a game developer including Mac, iPhone, iPad, Apple TV and Apple Watch. The other is Metal, which means high a high level of performance for your games on devices that support it without actually needing to write code in Metal to get it. As your skills as a game developer increase this opens up a whole new world of cool effects and “bare metal” performance tweaks that is the envy of the industry right now.

Apple also has a 3D engine called SceneKit that you may be interested in exploring at some point as well. It is very similar to SpriteKit, so once you have it under your belt you are well prepared to take on a 3D game.

Why learn Objective-C?

Swift is Apple’s hot new programming language, and most of the quality tutorials, books and blogs are writing their content in Swift 2.0 these days. So learning Objective-C is going backwards, right? In my opinion, no, it is well worth your time if you are serious about developing Apps and Games for Apple devices.

Objective-C has a number of advantages. Here are just a few of them.

If you think you might want to get a job as an iOS or Mac developer in the next few years, you are going to need to know Objective-C. Some companies are not going to be writing any production code in Swift 2.0 for a while, at least not until it has more time to mature. Being an Ace programmer with Swift is great, but if you are good with both it will get you a job.

Most of what you are writing in Swift is from a framework written in Objective-C. Cocoa has been around since before OS X and isn’t going anywhere. Some may disagree with me here, but I think the Cocoa frameworks are sometimes more clear when working

with them in Objective-C. In any case working with them in Objective-C will strengthen your understanding of them when using them in Swift.

Currently, Xcode Instruments work best on Objective-C projects. They work in Swift, but several features are not quite there yet, at least not at the same level. I'm certain this will change over the next year or so, but for now it is at least something to consider.

Both Swift and Objective-C use ARC (Automatic Reference Counting). This is strictly my opinion, but if you understand how to use ARC in Objective-C you will have an easier time with it in Swift where it can be a little bewildering at times, to put it nicely.

Naming conventions used in this book

Classes and C functions are capitalized using *PascalCase*. The first letter of each word is capitalized. Variables, constants and method names are capitalized using *camelCase*. The first word is all lower case and subsequent words begin with a capitalized letter.

Apple has done a good job of coming up with naming conventions that help make code self-documenting. We will be following the recommendations in Apple's coding guidelines for Cocoa.

Design conventions used in this book

Our project will be created using a simplified version of Separation of Concerns, which also addresses our desire to encapsulate our classes. Each class will control as much of its own behavior as possible, and calling classes will simply ask the class to perform an action or tell it that something happened the class knows how to handle.

My simple analogy is a Traffic Cop. The Cop tells the cars to go, stop, turn left or right, but the Cop does not tell the cars *how* to do these things.

Our game will be laid out so that the scene is a collection of objects. In the scene we add each of the objects that make up the game and when appropriate the scene tells the objects to perform an action, or that something happened the object knows what to do with. We could technically stuff everything we need for the entire game in the scene, but that would become a tangled mess that is difficult to understand in no time. If we ever decided that we wanted more than one scene we'd have a lot of refactoring to do.

Singletons sometimes get a bad rap in the Software Engineering world. We will be using them for what they were designed for. Whenever we use a Singleton we want to make sure that something is initialized only once, and that we are using the already initialized instance.

Use of third party libraries

Our game uses two open source libraries because they are better options than what is available only from SpriteKit. Specifically we are using an excellent sound library called ObjectAL (same one used in Cocos2d) that makes playing music and sound effects as easy as it can be. We'll also be using a library that can do labels from bitmap fonts called BMGlyphLabel.

Adventure Time

In Chapter 1 we are going to get off to a good start by downloading the project source code from GitHub and taking a tour of the template you'll be using to create our game called *TimberJim*.

Because I am providing a starting point for the game I want to make sure you have an understanding of how I prepared the project before we move on to building up the game in Chapter 2.

Lets get started!

Chapter 1 - Getting Started

Downloading the project from GitHub

For this book I have provided a project that has two folders for each chapter . The “Start” folder is the state of the project where we left off in the previous chapter. The “Finish” folder is the project at the end of any given chapter. For each chapter in this book you’ll find both of these in the folder for the chapter.

Your first task is to get the source code for the book downloaded to your mac. You can either download the project as a ZIP file to your Downloads folder by going to https://github.com/spritekitbook/timberman-objc and clicking the button labeled **Download ZIP** or you can do it via command line in Terminal line like this.

```
cd ~/Downloads && git clone https://github.com/spritekitbook/timberman-objc.git
```

Copy the folder *~/Downloads/timberman-objc/Project/Chapter 1/Start/TimberJim* some place convenient like your Desktop. This is the starting point for the project. I recommend using a copy of this project and adding to it while you go through the book, typing in the code as it is given.

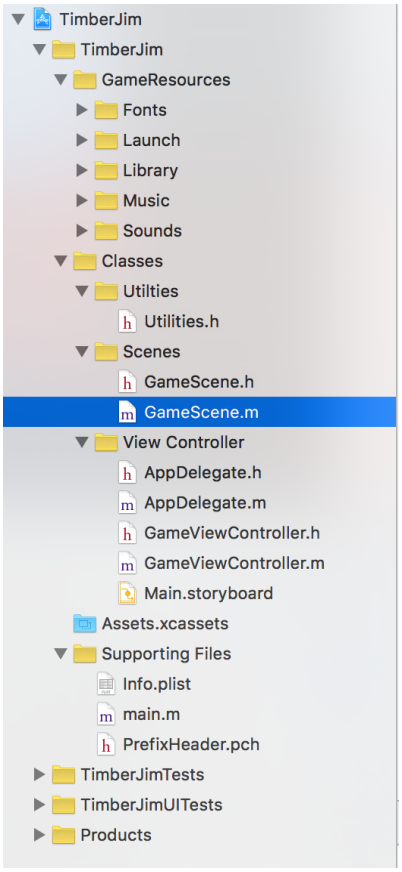
By the way, ~ represents your home directory. So ~ for me is `/Users/jeremy`, yours will be something similar for your account on your Mac.

Go to your desktop and look inside the TimberJim folder. Double click the file `TimberJim.xcodeproj` to open the project in Xcode.

Touring the Project

Now that we have our starting project open in Xcode, lets take a look at what we have so far.

In the Project Navigator pane on the left side of Xcode, expand the Classes folder.



Under the Classes folder we have three groups, Scenes, Utilities and View Controller. This is how I like to organize my code in Xcode. Groups are different from folders in that they don't create a folder on disk, but give you a handy way to group files together.

View Controller group

The View Controller group has five files that are essential to running an app.

We won't be touching **Main.storyboard** at all.

AppDelegate.m is the run when the game launches and gives us a way to control behavior when the app is coming into view or going into the background to be suspended. **AppDelegate.h** is the header file for the class.

GameViewController.m is a UIViewController class that creates an instance of an SKView that our game will run in. In other words, our game runs in a view controller. For the most part we don't need to do anything with the view controller once it is set up. Any views that you need to run in the game other than the scenes (Game Center, Ads) will need to be run from GameViewController.m because they will run over the top of the scene. **GameViewController.h** is the header file for the class.

Near the top of GameViewController.h, we need to import the header file for the GameScene class. This will give us access to any methods or properties of GameScene that have been exposed for other classes to use. In Objective-C we have to import the header of other classes we want to use, and if we need other classes to be able to call a method or work with a property (variable), those need to be exposed (made public) in the header.

```
#import "GameScene.h"
```

There are two methods in GameViewController that we'll talk about before moving on. The first is `viewDidLayoutSubviews`.

```
-(void)viewDidLayoutSubviews {

    [super viewDidLayoutSubviews];

    SKView *skView = (SKView *)self.view;

    if (!skView.scene) {

        if (kDebug) {
            skView.showsFPS = YES;
            skView.showsNodeCount = YES;
            skView.showsPhysics = YES;
        }

        CGSize viewSize = skView.bounds.size;

        GameScene *scene = [[GameScene alloc] initWithSize:viewSize];

        scene.scaleMode = SKSceneScaleModeAspectFill;

        SKTransition *transition = [SKTransition fadeWithColor:[SKColor
blackColor] duration:0.25];

        [skView presentScene:scene transition:transition];


        if (kDebug) {
            NSLog(@"Screen width: %.2f, Screen height: %.2f", viewSize.width,
viewSize.height);
        }

    }

}
```

This method is called when the UIViewController is finished laying out the views. In my experience this method works well in both Portrait and Landscape orientation games so it tends to be my choice from the methods provided by the UIViewController class.

Inside the method we cast the view as an SKView. We want to know if `skView` already has a `scene`, if it does we don't want to re-initialize it again. If it does not, then we can proceed to creating a new scene. We have a global debug constant called `kDebug` in Utilities.h that will let us turn code like this on or off in one place. If the value of `kDebug` is `YES`, we'll show the FPS, Node Count (how many nodes are in the scene), and draw a border on Physics bodies so we can see them.

We would like to know the size of the screen, so we'll create a variable that has the size of the view. Sometimes it is easier or more clear to work with a variable that has a good name, particularly when the full value it references is long. That is what `viewSize` is doing here.

Next we'll create an instance of GameScene called scene. We have our own custom init function in GameScene called `initWithSize:size` that lets us pass in the size of the screen. We want the scene to fill the screen, so we set the value of the `scaleMode` property to `SKSceneScaleModeAspectFill` which fills the screen.

We'll use a little SKTransition that is a simple fade to black that lasts a quarter second.

Then we tell the view to present the scene using the transition.

Finishing up `viewDidLayoutSubViews`, we print a message to the debug console with the size of the screen.

The other important function to go over in GameViewController is `supportedInterfaceOrientations`. Here we are telling the view that this game only supports one orientation, Portrait.

```
-(UIInterfaceOrientationMask)supportedInterfaceOrientations {

    return UIInterfaceOrientationMaskPortrait;
}
```

Utilities Group

The next group up is the Utilities group. Select the file **Utilities.h** and lets take a look.

```
#pragma mark - Debug
static const BOOL kDebug = YES;


#pragma mark - Screen Size and Position
static __inline CGSize ScreenSize() {
    return [UIScreen mainScreen].bounds.size;
}

#pragma mark - Device Type
static __inline__ BOOL DeviceIsPad() {
    return UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiomPad;
}

#pragma mark - Math Functions
static __inline__ int RandomIntegerBetween(int min, int max) {
    return min + arc4random_uniform(max - min + 1);
}

static __inline__ CGFloat RandomFloatRange(CGFloat min, CGFloat max) {
    return ((CGFloat)arc4random() / 0xFFFFFFFFu) * (max - min) + min;
}
```

This file is just a handy place to store a constant and some functions we can use to make our lives easier making TimberJim.

`kDebug` is a `BOOL` that we can use to put code we might like to run when debugging into an if statement. Doing it this way, we can turn that code on or off by simply setting this value to `YES` or `NO`.

`ScreenSize()` returns the size of the screen as a `CGSize`. We'll be using this a lot to position our nodes in the scene.

`DeviceIsPad()` returns YES if it is an iPad and NO if it is not. Sometimes we'll want to do something a little different depending on the type of device because the iPad has a 4:3 aspect ratio screen, and the iPhone has a 16:10 aspect ratio screen.

Our two math functions `RandomIntegerBetween(min, max)` and `RandomFloatRange(min, max)` let us pass in two numbers and return a random value between (and including) a minimum and maximum value.

Scenes Group

Select the Scenes group. In some games you may want more than one scene. For our game, TimberJim, we'll be using a single scene. Throughout this project I'll ask you to create groups that sometimes only have one file. This is because in my experience it keeps the project neat and easy to find your classes, so I'll suggest you do the same.

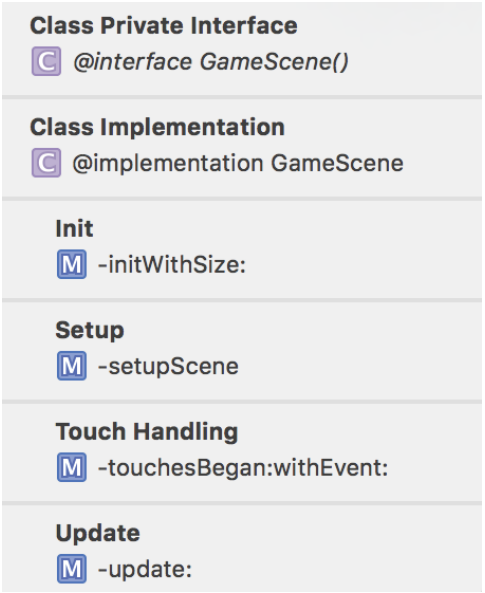
GameScene.m is the implementation file for the GameScene class. The scene is where we will organize all of the elements (nodes) of the game, handle touches, contact, and anything else the game will need.

I'd like to explain some of the basics of what is going on in GameScene.m while it is still pretty small.

```
#import "GameScene.h"

#pragma mark - Class Private Interface
@interface GameScene()

@end

#pragma mark - Class Implementation
@implementation GameScene

#pragma mark - Init
-(instancetype)initWithSize:(CGSize)size {

    if ((self = [super initWithSize:size])) {

        [self setupScene];
    }

    return self;
}
```

```
#pragma mark - Setup
-(void)setupScene {

    self.backgroundColor = [SKColor blackColor];

    BMGlyphFont *glyphFont = [BMGlyphFont fontWithName:@"GameFont"];
    BMGlyphLabel *label = [BMGlyphLabel labelWithText:@"Yay, it works!"
font:glyphFont];
    label.position = CGPointMake(ScreenSize().width / 2,
ScreenSize().height / 2);
    [self addChild:label];

    [label runAction:[SKAction scaleTo:1.1 duration:0.25] completion:^{
        [label runAction:[SKAction scaleTo:1.0 duration:0.25]];
    }];

}

#pragma mark - Touch Handling
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

}

#pragma mark - Update
-(void)update:(NSTimeInterval)currentTime {

}

@end
```

We've already seen it a few times, but I'll explain the use of `#pragma mark`. This creates breadcrumbs or jump points in your code that you can easily get to from the navigation bar in the editor window.



We will use a private `@interface` for the class that will let us declare some properties. These will be things like our nodes, and any variables we need in the class to make GameScene come together. These are private because they are in the implementation file (.m) and we won't be putting them in the header file (.h). As I mentioned in the Introduction, this is a good practice called encapsulation. We only expose things in the header that other classes must have access to. Notice that `@interface` ends with `@end`.

Next we have the `@implementation`, which also ends with `@end` at the bottom of the class. In simple terms, this is where your code goes in an Objective-C class.

`initWithSize:size` is a custom init method for our GameScene. We'll use this to initialize the scene with the size of the device screen. Because GameScene is a subclass of SKScene, we should call `[super init]`, which runs the init method in the super class (SKScene). Before leaving our init method, we call our setup method and return `self`.

In `setupScene` we are doing some really basic things to build up the scene. First we set the `backgroundColor` property to Black. We'll use our 3rd party library BMGlyphFont and BMGlyphLabel to initialize the Bitmap font included in the GameResources and then create a label using this font.

Whenever we create a node for it to display on screen we must add it as a child. Some times it is useful to create an empty node, add that to the scene and then add our other nodes to this empty node. For this game that is not necessary, so we just add the node as a child of the scene itself.

Before leaving `setupScene` we tell the label to run an action to scale up to 1.1, and when that is complete scale to 1.0. This will give the label a little animation that appears to bounce or "pop" when the scene loads. We'll be discarding this label in the next chapter, it is here for demonstration for one of the ways you can animate a node using SKActions.

The method `touchesBegan:touches withEvent:` is our touch handling method. We'll be adding code to this method to detect where the player touched on screen and using that to perform an action.

The method `update:currentTime` is very important. Whatever code we provide here will be run every single frame. For a 60 FPS game, that means `update:currentTime` is run 60 times a second.

In the Scenes group, select **GameScene.h**.

```
#import <SpriteKit/SpriteKit.h>

@interface GameScene : SKScene

-(instancetype)initWithSize:(CGSize)size;

@end
```

GameScene.h is a header file for the GameScene class. Note that we are importing the SpriteKit header file so the class has access to SpriteKit classes and functions.

The `@interface` tells the class it is of type SKScene. This means GameScene is a subclass of SKScene. One way to think about it is GameScene is a custom SKScene where we can provide our own code that should be executed while having all the characteristics of an SKScene. We don't need to explicitly write code for everything an SKScene can do, just the parts we want to customize for our use.

We have provided a method signature for `initWithSize:size` so that any class that imports GameScene.h knows how to call that function.

GameResources Folder

In the Project Navigator pane just above classes, find the GameResources folder and expand it.

The first folder is **Fonts**. In this case we are using a Bitmap font. This is a font that has been styled and then exported as images. SpriteKit cannot do this natively, so I have included a library that knows how to create labels from a Bitmap font. You've already seen an example of using this in GameScene.swift from the label "Yay, it works!".

The next folder is **Launch**. We get a couple of ways to create a splash or launch image for our games. I've chosen to use the older way, which is to provide an image for each device type that is scaled to the appropriate size and given a special name that Xcode knows what to do with. The table below lists the Default launch image for each device type.

Device	Launch Image Name
iPhone 4	Default@2x.png
iPhone 5	Default-568h@2x.png
iPhone 6	Default-667h@2x.png
iPhone 6 Plus	Default-736h@3x.png
iPad/iPad 2	Default~ipad.png
iPad Retina, iPad Mini	Default@2x~ipad.png

The next folder is **Library**. We are using two libraries in our game to extend SpriteKit. The first is for BMGlyphFont and BMGlyphLabel. This is what lets us use Bitmap font labels in the game.

The other library is ObjectAL, which is a very good Open Source sound library that will make playing music and sound effects in the game very simple.

The final two folders are **Music** and **Sounds**. As the names imply, these just have our music file and sound effect files respectively. I recommend that you use MP3 files for your music, and CAF files (Core Audio Format) for your sound effects. It is my experience that these are the best balance of file size and performance in SpriteKit games.

Supporting Files

Below the Classes folder, find the group named **Supporting Files** and expand it. The Info.plist file holds a number of important values about our game, but for this game we are going to leave it alone.

We'd like to make some of our headers automatically included with our classes. The best place to do this is in a precompile header.

Select **PrefixHeader.pch**.

```
#ifndef PrefixHeader_pch
#define PrefixHeader_pch

#ifdef __OBJC__
#import <Foundation/Foundation.h>
#import <UIKit/UIKit.h>
#import <SpriteKit/SpriteKit.h>
// Project
#import "Utilities.h"
// ObjectAL
#import "OALSimpleAudio.h"
// BMGlypFont / BMGlyphLabel
#import "BMGlyphFont.h"
#import "BMGlyphLabel.h"

#endif

#endif /* PrefixHeader_pch */
```

In order for this to work, we had to provide the path for our libraries in the Project settings under **Build Settings > Header Search Path > Search Paths**.

There is a magic string you can use to make this easy, as long as your Libraries are in the same place in your projects as I am using for TimberJim. Xcode will automatically expand this to the real location, so it is much better than hard coding the path.

```
$(PROJECT_DIR)/$PRODUCT_NAME/GameResources/Library
```

We also had to provide the path to the PrefixHeader.pch under **Build Settings > Apple LLVM 7.0 - Language > Prefix Header**.

The magic string that works well for providing this path is the following.

```
$(PROJECT_DIR)/$PRODUCT_NAME/PrefixHeader.pch
```

Now we can put the name of any header located at **GameResources > Library** in PrefixHeader.pch and it will be included in the project. In other words, any class we create will already know about things like OALSimpleAudio for example.

Assets.xcassets

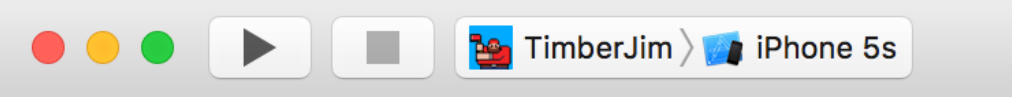
Just below the Supporting Files folder, select **Assets.xcassets**. This is an Asset Catalog. We are using this to provide the App Icon, and a Sprite Atlas.

The first thing at the top is the AppIcon. If you select it you'll see the icon in all of the sizes for the devices this game can support. The different devices display the icon in various sizes, and there are different sized icons for things like Spotlight vs. the Springboard (home screen).

Below that is a folder called **Sprites**. This is a Sprite Atlas. In the Sprite Atlas I have provided the images we need in each of the resolutions that the game may need for different devices. This is very handy, since starting with iOS9 the App Store can deliver just the images a device needs in a process called App Thinning to make the download smaller. Our game will be pretty tiny, but for bigger games with a lot of images this can be very helpful.

Build and Run

Lets see what the game looks like in this starting template state. Select the iPhone 5s simulator from the dropdown list, and click the play button to launch the game on the simulator. Alternatively you can use the command hotkey ⌘**R** to run the game.



After a brief compile, you should get a simulator launched to our GameScene.

Conclusion

In this chapter I wanted to make sure you know the basics of how the project is structured and what has already been added. With that done we are ready to start building TimberJim into a game.

You can download a template that is very similar to this for use in creating your own games from GitHub here.

https://github.com/spritekitbook/sktemplate-objc.git

Chapter 2 - Game World

In Chapter 2 we will get the game world working for TimberJim. The “World” for this game consists of a nice blue sky with clouds that will scroll continuously in the background, a flock of birds that will fly back and forth and a forest image.

Have a look in Assets.xcassets > Sprites at the images for Clouds, Birds0, Birds1 and Forest. These are the sprites that we will be working with.

Here are the things we need to get done before moving on Chapter 3.

Create a Singletons group in Xcode for our Singleton classes.

Create the GameTextures Singleton class for accessing our textures and sprites.

Create a SpriteName header that holds the string names of our images.

Create an Environment group in Xcode for our Clouds and Birds classes.

Create the Clouds class and add it to the GameScene.

Create the Birds class and add it to the GameScene.

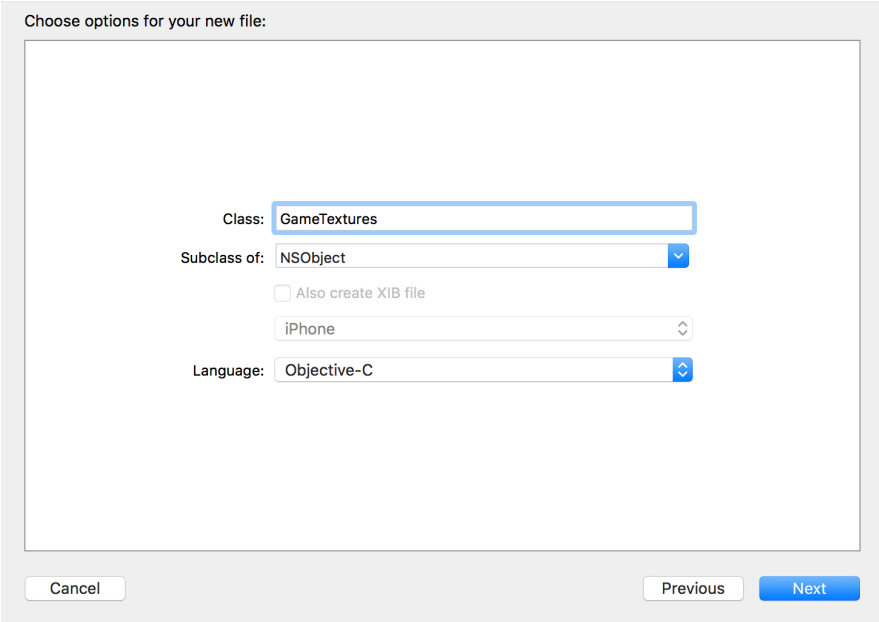
Add the Forest sprite to the GameScene.

GameTextures

The best practice for loading textures and sprites is to read them out of an SKTextureAtlas. We could reference them directly by name because they are in our Assets.xcassets in a Sprite Atlas, but doing it that way can produce some odd results.

There is also a performance advantage to getting textures from an SKTextureAtlas, particularly since we will be making this into a Singleton so it only needs to be loaded from disk once.

In Xcode, right click the Classes folder and choose **New Group**, name this new group **Singletons**. Right click the Singletons group and choose New File… In the wizard, choose **iOS > Source > Cocoa Touch Class**, and click **Next**. Name this class **GameTextures**, make it a subclass of **NSObject**, and click **Next** and **Create**.



Select GameTextures.m. Replace the temple code by typing in the following.

```
#import "GameTextures.h"

@interface GameTextures()

@property SKTextureAtlas *atlas;

@end


@implementation GameTextures

#pragma mark - Init
+(instancetype)sharedInstance {
    static GameTextures *sharedInstance = nil;
    static dispatch_once_t onceToken;
```

```
    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });

    return sharedInstance;
}

-(instancetype)init {
    if ((self = [super init])) {
        [self setup];
    }

    return self;
}

#pragma mark - Setup
-(void)setup {

    self.atlas = [SKTextureAtlas atlasNamed:@"Sprites"];
}

#pragma mark - Public Methods
-(SKSpriteNode *)spriteWithName:(NSString *)name {
    SKTexture *texture = [self.atlas textureNamed:name];

    return [SKSpriteNode spriteNodeWithTexture:texture];
}

-(SKTexture *)textureWithName:(NSString *)name {
    return [self.atlas textureNamed:name];
}

@end
```

Lets go over each part of this in some detail. At the top we are importing the header file for the class.

```
#import “GameTextures.h"
```

Every Objective-C class has a .m file (the implementation file) and a .h file (the header file). It is standard for the .m file to import its own header. We’ll sometimes also be importing the header of other classes to get access to their public properties and methods.

The next section of code is a private interface. This lets us declare some class properties that the entire class will know about, but they are private because they are in the .m file. If any of our properties or methods need to be public, visible to other classes by importing the header, we'll need to declare them in the .h file instead. The only property we need for the GameTextures class is an SKTextureAtlas that we'll call `atlas`.

```
@interface GameTextures()

@property SKTextureAtlas *atlas;

@end
```

As mentioned previously in the book, everything between `@implementation` and `@end` in the ..m file is essentially the code for the class. All of our methods go in the implementation.

Following that we have a class method called sharedInstance. You can tell it is a class method because it begins with a +. This method returns an instance of the class, but it guarantees that it can only be initialized once. This bit of code is what makes GameTextures a Singleton.

```
+(instancetype)sharedInstance {
    static GameTextures *sharedInstance = nil;
    static dispatch_once_t onceToken;

    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });

    return sharedInstance;
}
```

Next we have a very standard init method that calls `[super init]`, runs our `setup` method and returns.

```
-(instancetype)init {
    if ((self = [super init])) {
        [self setup];
    }

    return self;
}
```

Our `setup` method initializes the SKTextureAtlas and populates it with the contents of the Sprite Atlas.

```
-(void)setup {

    self.atlas = [SKTextureAtlas atlasNamed:@"Sprites"];
}
```

The entire reason for making this class is to give ourselves an easy way to get textures and sprites directly from the atlas. The first method `spriteWithName:name` returns an SKSpriteNode from the `atlas` using the given name.

```
-(SKSpriteNode *)spriteWithName:(NSString *)name {
    SKTexture *texture = [self.atlas textureNamed:name];

    return [SKSpriteNode spriteNodeWithTexture:texture];
}
```

The final method returns an SKTexture from the atlas using the given name.

```
-(SKTexture *)textureWithName:(NSString *)name {
    return [self.atlas textureNamed:name];
}
```

Select GameTextures.h. We need to make our two methods and the `sharedInstance` class method public so that other classes who import the header know how to use them.

Add these three lines to the template code so that the contents of the .h file looks like this. The new lines go between `@interface` and `@end`.

```
#import <Foundation/Foundation.h>

@interface GameTextures : NSObject

+(instancetype)sharedInstance;

-(SKSpriteNode *)spriteWithName:(NSString *)name;
-(SKTexture *)textureWithName:(NSString *)name;

@end
```

Now classes that import the header know how to use `sharedInstance`, `spriteWithName:name`, and `textureWithName:name`.

We are going to be using the GameTextures class a lot. It would be convenient to add the .h file one time so that it is available to any of the classes we'll make, most of which will need access to it.

Under the Supporting Files group, select PrefixHeader.pch. Add this comment and import statement just after `#import "BMGlyphLabel.h"`.

```
// GameTextures Singleton
#import "GameTextures.h"
```

Now we can use the GameTextures class anywhere in the project.

SpriteNames.h

One of the easiest mistakes to make is misspelling a string name. Xcode has no idea that @“Payler” is a typo. Code with this misspelled word will compile just fine, and then crash and burn when the method tries to load a resource named “Payler” and can’t find it in the app bundle.

The best way to guard against that is to limit the number of times you provide a literal string by defining a constant. If you misspell the constant, Xcode will throw an error so you can catch your mistake.

Right click the Utilities group and choose **New File…** Select **iOS > Source > Header File** and click **Next**. Name the file **SpriteNames** and click **Create**.

Select SpriteNames.h. Before `#endif`, type in the following code.

```
// Environment
static NSString *const CLOUDS = @"Clouds";
static NSString *const BIRDS0 = @"Birds0";
static NSString *const BIRDS1 = @"Birds1";
static NSString *const FOREST = @"Forest";
```

My preference is to give string constants an all upper case name.

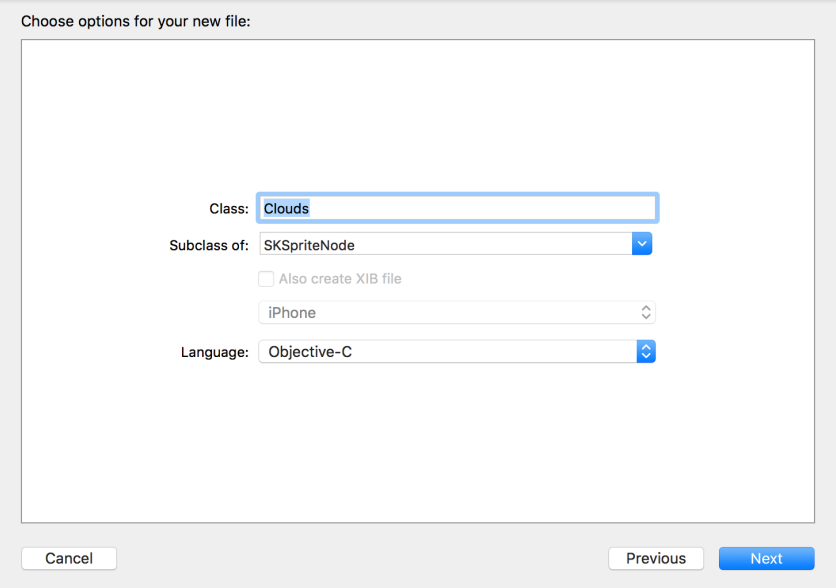
This is another header we’ll want to use in many of our classes, so lets add it to PrefixHeader.pch. Select PrefixHeader.pch under Supporting Files. Just after adding GameTextures.h, add this new comment and import statement.

```
// SpriteNames Constants
#import "SpriteNames.h"
```

Clouds

The background for our game is a full screen sprite that we will scroll across the screen forever. The Clouds sprite is double the width of the screen. We will move the sprite half its width to the left, and then reset. Because the image is seamless, this gives the illusion of a continuous scroll.

Right click the Classes folder and choose **New Group**, name this new group **Environment**. Right click the Environment group and choose **New File…** Select **iOS > Source > Cocoa Touch Class** and click **Next**. Name the class **Clouds** and make it a subclass of **SKSpriteNode**, then click **Next** and **Create**.



Select Clouds.m. Replace the template code by typing in the following.

```
#import "Clouds.h"

@implementation Clouds

#pragma mark - Init
-(instancetype)init {

    if ((self = [super  init])) {
        SKTexture *texture = [[GameTextures sharedInstance]
textureWithName:CLOUDS];
        self = [Clouds spriteNodeWithTexture:texture];

        [self setup];
    }

    return self;
}


#pragma mark - Setup
-(void)setup {
    self.anchorPoint = CGPointZero;
    self.position = CGPointZero;
}


#pragma mark - Update
-(void)update:(NSTimeInterval)delta {
    CGFloat speedX = DeviceIsPad() ? delta * 60 * 0.5 : delta * 60 * 0.25;

    self.position = CGPointMake(self.position.x - speedX, self.position.y);

    if (self.position.x < (0 - self.size.width  / 2)) {
        self.position = CGPointZero;
    }
}


@end
```

As with our other subclasses, we begin in int by calling `[super init]`. Then we get a texture out of the SKTextureAtlas from the GameTextures Singleton. Notice that Clouds is a subclass of SKSpriteNode, but to initialize the class with the right texture, we are calling the spriteNodeWithTexture method on Clouds not SKSpriteNode. This is the pattern when we subclass SKSpriteNode. Clouds inherits all of the attributes of an SKSpriteNode, so this is how we initialize it.

Before leaving init we call our setup function. In setup we are setting the `anchorPoint` to `CGPointZero`, which is an alias for `CGPointMake(0, 0)`. This puts the `anchorPoint` in the

lower left corner of the sprite. This is important because sprites are positioned using the `anchorPoint`, which by default is at 0.5, 0.5, the middle of the sprite. Next we set the `position` to also be CGPointZero (0,0). This means we are positioning using the lower left corner of the sprite to be in the lower left corner of the screen. Screen coordinates start in the lower left of the screen and go right on the X axis and up on the Y axis. If we didn't do this our sprite would be positioned using the center of the sprite, so the middle of the image is what would be placed at the lower left corner of the screen.

In update, we are taking an argument called `delta`. We'll look at this more in GameScene, but it is the elapsed time between frames. You might think that frame rate in a game is constant, but in practice there are small differences in time from frame to frame. Using `delta` we can smooth out these differences so that if the frame rate jumps around little the game still looks good.

The variable `speedX` is determined using our `DeviceIsPad()` function to determine if the game is being run on an iPad, if not for simplicity we assume it is an iPhone. What follows is called a ternary operator. If the expression is true, the statement before the : is assigned, if false the statement after the : is assigned. Our `speedX` variable determines how many pixels per frame we should move the clouds on the X axis.

We then subtract `speedX` from the x position of the sprite. This makes it move left. Following that we have an if statement that checks if the sprite is half its width off screen, if it is we reset the position back to 0, 0 (CGPointZero).

Select Clouds.h. Between @interface and @end, add this line.

```
-(void)update:(NSTimeInterval)delta;
```

Now classes that import the Clouds.h header file will know about the update:delta method.

Update GameScene

Before we move on to adding the other elements of the environment, lets an instance of our Clouds class to GameScene and make it scroll.

Select GameScene.m in Xcode. Just after importing GameScene.h, add the header for the Clouds.

```
#import “Clouds.h"
```

In the section just below that, between @interface GameScene() and @end, add these lines to create a member variable for tracking update time and an instance of Clouds.

```
// Scene member variables
@property NSTimeInterval lastUpdateTime;

// Node member variables
@property Clouds *clouds;
```

We don’t need any of the template code in `setupScene`. Lets replace the entire method with this code which instantiates our instance of Clouds and adds it to the scene.

```
#pragma mark - Setup
-(void)setupScene {
    self.lastUpdateTime = 0.0;

    self.clouds = [[Clouds alloc] init];
    [self addChild:self.clouds];
}
```

Scroll down to the `update:currentTime` method near the end of the class. We need to calculate the difference in time between this frame and the previous frame that we’ve been calling “delta”.

Replace the entire method which is empty right now, with the following code.

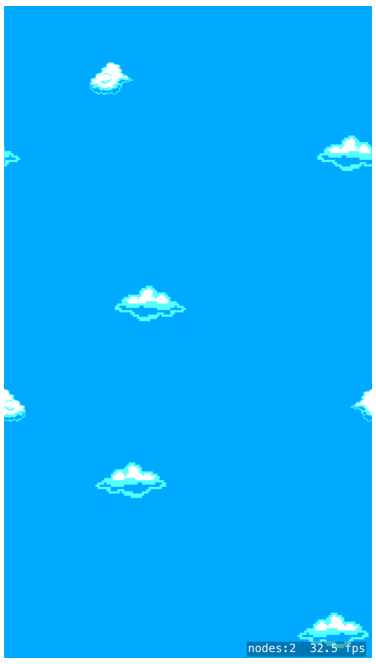
```
#pragma mark - Update
-(void)update:(NSTimeInterval)currentTime {
    // Calculate "Delta"
    NSTimeInterval delta = currentTime - self.lastUpdateTime;
    self.lastUpdateTime = currentTime;

    [self.clouds update:delta];
}
```

Build and Run

Build and Run the game by pressing the Play button in the toolbar of Xcode, or you can use the hotkey combination ⌘**R**.

When the game launches, you'll see the background continuously scroll. We know that it is really moving halfway off screen and then resetting back to 0,0 but in game it appears to just scroll forever.

The reason this illusion works is because the sprite is twice the width of the screen, so as it scrolls left, the portion that is off screen covers up the area that would be left blank. The image is also seamless, identical on both sides. One of the common sources of difficulty with this method is if the image is not seamless it will appear to jump or pop when it resets.



Birds

The next element we'll create for the environment of the game is a flock of birds that flies left off screen then flies right off screen, repeating this forever. The birds will also pick a random Y position each time they change direction to make it a little more interesting. In the Birds class we'll also get a look at a simple animation using different textures and an SKAction animation.

Right click the Environment group in Xcode, and choose **New File…** Select **iOS > Source > Cocoa Touch Class** and click **Next**. Name the class **Birds** and make it a subclass of **SKSpriteNode**, click **Next** and then **Create**.

Select Birds.m. Just after #import "Birds.h", we are going to need an enum we can use that holds our directions for the birds to fly, type in the following.

```
typedef NS_ENUM(NSInteger, Direction) {
    Left,
    Right
};
```

We'll use this NS_ENUM called Direction to hold our two possible directions.

Just below that, lets add our private interface for the Birds class.

```
@interface Birds()

@property SKAction *birdsAnimation;
@property BOOL moving;
@property Direction direction;

@end
```

The first of these is an SKAction that will hold our SKAction to animate through the textures that make up the flight animation. Next is a BOOL use to determine if the Birds should be moving. Finally a way to track our current direction.

In the implementation, between @implementation and @end type in the following code.

```
#pragma mark - Init
-(instancetype)init {
    if ((self = [super init])) {
        SKTexture *texture = [[GameTextures sharedInstance]
textureWithName:BIRDS0];
        self = [Birds spriteNodeWithTexture:texture];

        [self setupBirds];
        [self setupBirdsAnimation];
    }

    return self;
}

#pragma mark - Setup
-(void)setupBirds {
    self.moving = NO;

    self.direction = Left;

    CGFloat randomY = RandomFloatRange(ScreenSize().height * 0.45 ,
ScreenSize().height * 0.9);

    self.position = CGPointMake(ScreenSize().width, randomY);

    [self runAction:[SKAction waitForDuration:0.016] completion:^{
        self.moving = YES;
    }];
}

-(void)setupBirdsAnimation {
    SKTexture *frame0 = [[GameTextures sharedInstance]
textureWithName:BIRDS0];
    SKTexture *frame1 = [[GameTextures sharedInstance]
textureWithName:BIRDS1];

    _birdsAnimation = [SKAction animateWithTextures:@[frame0, frame1]
timePerFrame:0.1];

    [self runAction:[SKAction repeatActionForever:_birdsAnimation]];
}
```

In `init` we are initializing the Birds in the same way as we did with Clouds and any other subclasses of SKSpriteNode in this book.

In `setupBirds` we first set the BOOL `moving` to NO and the `direction` to Left. We are using our function `RandomFloatRange(min, max)` to get a random Y position between

45% and 90% up the screen. Then we set the position of the Birds to be at the far right edge of the screen on the X axis and this random Y position on the Y axis.

Then we run an SKAction to wait for 0.016 seconds before setting moving to YES. 0.016 happens to be about the length of time for 1 frame in a 60 FPS game. We are using this artificial delay to guarantee that the birds are in the scene and positioned before we start moving them.

In `setupBirdsAnimation` we get a texture for each of the two textures that make up the animation. Then we can initialize our `birdsAnimation` to use an `animateWithTextures:timePerFrame` SKAction to animate through them. Finally, we tell the Birds class to run this action forever with `repeatActionForever`.

Just after `setupBirdsAnimation`, add these two methods to move the Birds.

```
#pragma mark - Update
-(void)update:(NSTimeInterval)delta {
    if (self.moving) {
        CGFloat speedX = DeviceIsPad() ? delta * 60 * 3 : delta * 60 * 1.5;

        switch (self.direction) {

            case Left:
                self.position = CGPointMake(self.position.x - speedX,
self.position.y);

                if (self.position.x < (0 - self.size.width)) {
                    [self changeDirection];
                }

                break;

            case Right:
                self.position = CGPointMake(self.position.x + speedX,
self.position.y);

                if (self.position.x > (ScreenSize().width + self.size.width))
{
                    [self changeDirection];
                }

                break;
        }
    }
}
```

```
#pragma mark - Direction
-(void)changeDirection {
    CGFloat randomY = RandomFloatRange(ScreenSize().height * 0.45,
ScreenSize().height * 0.9);

    self.position = CGPointMake(self.position.x, randomY);

    self.xScale = self.xScale * -1;

    self.direction = self.direction == Left ? Right : Left;
}
```

In the `update:delta` method, the first thing we determine is if moving is true (YES). If int is not, we can immediately leave the method. If it is, we proceed with updating the position of the Birds. As we did with Clouds we get a CGFloat called `speedX` using a ternary operator to determine if the device is an iPad and assign the value multiplied by delta accordingly.

Then we switch on the value of `direction`. If it is Left, we execute the block of code that moves the Birds to the left. If it is Right, we execute the block of code that moves the Birds right. This time instead of resetting the position, when the Birds are off screen we change direction.

In the `changeDirection` method we first get a randomY position again so that the Birds will fly across the screen at a random height each pass. We then “flip” the `xScale` by multiplying it by -1. If the Birds are facing left, they will face right. Finally we use another ternary operator to determine what direction should be. If the current value of `direction` is Left, it becomes Right, otherwise it becomes Left.

The final thing we need to do for Birds is add the update:delta signature to the header file so that other classes can use it.

Select Birds.h and add this line between @interface and @end.

```
-(void)update:(NSTimeInterval)delta;
```

Update GameScene

At the top of the file we need to import the header for Birds so we can use it, add this line just after importing the header for Clouds.

```
#import “Birds.h"
```

Next we’ll add a property for Birds to the GameScene class. In the @interface GameScene(), add this line just after adding a property for the Clouds.

```
@property Birds *birds;
```

In `setupScene`, just after adding the clouds as a child of the scene, add this new code.

```
self.birds = [[Birds alloc] init];
[self addChild:self.birds];
```

In update:currentTime, just after calling update:delta on the clouds, add this new line.

```
[self.birds update:delta];
```

Build and Run

Build and Run the game (⌘**R**) and have a look. We've got Birds flying back and forth in the scene.

Forest

Our forest is static so it can be a sprite we create and add to the scene directly. We won't need to subclass it.

In GameScene.m, go to `setupScene`. Just after instantiating the `birds` and adding them to the scene, add this new code to create a forest sprite and add it to the scene.

```
    SKSpriteNode *forest = [[GameTextures sharedInstance]
spriteWithName:FOREST];
    forest.anchorPoint = CGPointZero;
    forest.position = CGPointZero;
    [self addChild:forest];
```

We are once again setting the anchorPoint to 0,0 (CGPointZero) so when we position it at 0,0 the lower left corer is being used.

Build and Run

Build and Run (⌘**R**) the game a final time for this chapter and have a look at what our completed environment looks like.

Starting to look pretty good!

Conclusion

In Chapter 2 we got the environment for TimberJim working and got the important foundation for our game in place.

The Clouds and Birds don’t contribute to the game play itself, but certainly make the game more interesting to look at.

In Chapter 3 we will get our log stack working.

Need Help?

If you got stuck on any of these steps for some reason, check your work against the project in the source code you downloaded for this book at **Project > Chapter 2 > Finish**.

Chapter 3 - Log Stack

In Chapter 3 we need to get the core mechanic of TimberJim working, the StackController that spawns and moves the logs for our player to chop.

We aren't ready to add the Player class yet, so we'll artificially move the stack every second in this chapter. This is something we'll get rid of once the Player class is in the game, but for now it will let us see how it works.

Here are the things we need to get done before moving on to Chapter 4.

Update the SpriteName header to have our Branch and Log sprite names.

Create a Contact header that stores our Contact bit masks.

Add the Contact header to PrefixHeader.pch.

Create a Stack group in the project.

Create a Branch class that has a physics body.

Create a StackController class that spawns the log stack and branches.

Add an instance of StackController to GameScene and test it out.

SpriteNames.h

In the Utilities group, select the SpriteNames.h header and add the following new constants before #endif.

```
// Logs
static NSString *const BRANCH = @"Branch";
static NSString *const LOG0 = @"Log0";
static NSString *const LOG1 = @“Log1";
```

Contact.h

We need a way to tell instances of Branch, Player and GameScene who is who in a physics simulation when contact between two bodies occurs. A handy way to do that is to define them as constants in a header file.

Right click the Utilities group and choose **New File…** Select **iOS > Source > Header File** and click **Next**. Name it **Contact** and click **Create**.

Select Contact.h and add the following lines before #endif.

```
static const uint32_t ContactPlayer = 0x1 << 0;
static const uint32_t ContactBranch = 0x1 << 1;
```

Update PrefixHeader.pch

In Supporting Files, select PrefixHeader.pch. Add the following lines just after importing SpriteNames.h.

```
// Category bitmask Constants
#import "Contact.h"
```

Branch

Right click the Classes folder and choose **New Group**, name this new group **Stack**. Right click the Stack group, choose **New File…** Select **iOS > Source > Cocoa Touch Class**, click **Next**. Name it **Branch** and make it a subclass of **SKSpriteNode**, click **Next** and **Create**.

Select Branch.h, between the @interface and @end, add this code.

```
typedef NS_ENUM(NSInteger, BranchType) {
    BranchLeft,
    BranchRight
};

-(instancetype)initWithBranch:(BranchType)branchType;
```

We are adding these to the header first this time because both are meant to be public, we want them to be available to any class that imports Branch.h.

The first is an NS_ENUM named BranchType that is similar to what we used for the Birds class. We'll put this in the header this time because we want calling classes to know about BranchType.

The next is our signature for our special init method called `initWithBranch:branchType`. We just added this now since we are already in the header file.

Select Branch.m and type in the following code.

```
#pragma mark - Init
-(instancetype)initWithBranch:(BranchType)branchType {
    if ((self = [super init])) {
        SKTexture *texture = [[GameTextures sharedInstance]
textureWithName:BRANCH];
        self = [Branch spriteNodeWithTexture:texture];

        switch (branchType) {
            case BranchLeft:
```

```
                break;

            case BranchRight:
                self.xScale = self.xScale * -1;
                break;
        }

        [self setupPhysics];

    }

    return self;
}

#pragma mark - Setup
-(void)setupPhysics {
    self.physicsBody = [SKPhysicsBody bodyWithRectangleOfSize:self.size
center:self.anchorPoint];
    self.physicsBody.categoryBitMask = ContactBranch;
    self.physicsBody.collisionBitMask = 0x0;
    self.physicsBody.contactTestBitMask = ContactPlayer;
    self.physicsBody.affectedByGravity = NO;
}
```

In `initWithBranch:branchType` we are initializing the Branch class with a texture as we've been doing so far. Then we switch on the `branchType` passed in. If it is BranchLeft, we just break, exit the block. If it is BranchRight, we "flip" the image. In a similar way to how we handle changing directions in the Birds class, we can treat it like we have two textures with this trick.

In `setupPhysics` we create a physics body that is a rectangle using the size of the sprite as the size of the rectangle. The categoryBitMask property gives this physics body an ID, it tells the physics engine what it is. We don't need or want collisions so we set the collisionBitMask to 0x0. We do want to test for contact with our Player, so we provide that bit mask for contactBitMask.

By default SpriteKit simulates Earth gravity, -9.8 on the Y axis. Physics bodies are affected by gravity by default. Here we are stating that this physics body is not affected by gravity. If we didn't set this here, we would have to turn off gravity SKPhysicsWorld in the scene or the branches would just fall off screen as soon as they are added.

StackController

Out StackController actually moves the pieces, and spawns the Branches on them. For this chapter we'll move the stack every second so we can see how it works. When we get the Player created in the next chapter we'll move the stack when the player "chops".

Right click the Stack group and choose **New File…** Select **iOS > Source > Cocoa Touch Class**, click **Next**. Name it **StackController**, make it a subclass of **SKNode**, click **Next** and **Create**.

Select StackController.m. The first thing we'll need to do is import the header for the Branch class. Just after importing the header for StackController add this line.

```
#import "Branch.h"
```

Our logs in the stack can have a branch on the Left, Right or None at all. Let's make an NS_ENUM called Side that declares these. Add it just after importing the headers.

```
typedef NS_ENUM(NSInteger, Side)  {
    Left,
    Right,
    None
};
```

We need a few member variables. Two mutable arrays for holding our sprites (logs and branches), a CGPoint that we can use to build the stack, and a way to determine what side to spawn a branch on. We'll also make a CGFloat for tracking how many frames have passed so we can move the stack every second.

Just after our NS_ENUM, add this new code for the private interface.

```
@interface StackController()

@property NSMutableArray *branchArray;
@property NSMutableArray *pieceArray;
@property CGPoint basePosition;
@property Side currentSide;
@property CGFloat frameCount;

@end
```

Just after @implementation, lets add our code for the class.

```
#pragma mark - Init
-(instancetype)init {
    if ((self = [super init])) {

        [self setup];

    }

    return self;
}

#pragma mark - Setup
-(void)setup {
    Branch *leftBranch = [[Branch alloc] initWithBranch:BranchLeft];
    Branch *rightBranch = [[Branch alloc] initWithBranch:BranchRight];

    self.branchArray = [[NSMutableArray alloc] init];

    [self.branchArray addObject:leftBranch];
    [self.branchArray addObject:rightBranch];

    self.pieceArray = [[NSMutableArray alloc] init];

    self.basePosition = CGPointMake(ScreenSize().width / 2,
ScreenSize().height * 0.3);

    self.currentSide = Left;

    [self spawnStack];
}



#pragma mark - Stack and Branch
-(void)spawnStack {
    SKSpriteNode *piece0 = [[GameTextures sharedInstance]
spriteWithName:LOG0];
    SKSpriteNode *piece1 = [[GameTextures sharedInstance]
spriteWithName:LOG1];

    NSArray *spriteArray = @[piece0, piece1];
```

```
    for (int i = 0; i < 12; i++) {
        SKSpriteNode *log = [[spriteArray objectAtIndex:i % 2] copy];

        log.position = CGPointMake(self.basePosition.x, self.basePosition.y +
log.size.height * i);

        [self spawnBranch:log];

        [self addChild:log];

        [self.pieceArray addObject:log];
    }
}

-(void)chooseSide {
    CGFloat randomNumber = RandomFloatRange(0.0, 1.0);


    if (self.currentSide != None) {
        self.currentSide = None;

    } else {
        if (randomNumber < 0.45) {
            self.currentSide = Left;

        } else if (randomNumber < 0.9) {
            self.currentSide = Right;

        } else {
            self.currentSide = None;

        }
    }
}

-(void)spawnBranch:(SKSpriteNode *)log {

    [self chooseSide];

    if (self.currentSide == Left) {
        CGPoint position = CGPointMake(-log.size.width, 0);
        Branch *branch = [[self.branchArray objectAtIndex:0] copy];
        branch.position = position;

        [log addChild:branch];

    } else if (self.currentSide == Right) {
        CGPoint position = CGPointMake(log.size.width, 0);
        Branch *branch = [[self.branchArray objectAtIndex:1] copy];
        branch.position = position;

        [log addChild:branch];
    }
}
```

That was a lot to type in at once. Lets go over each part of it starting with `setup`.

In `setup` we first create a `leftBranch` and `rightBranch` instance of our Branch class. Remember each of these is not just a sprite, but a physics body. Next we need to instantiate our `branchArray` before we can use it . Then we add each branch to the `branchArray`. We'll set the `basePosition` to be in the center of the screen on the X axis and 30% up the screen on the Y axis, this will let us start the stack on the base of the tree in the Forest sprite. We set the initial `side` to Left, we'll use this to make sure the first log cannot have a branch in the `chooseSide` method below. Then we call `spawnStack`.

In spawnStack we get each of our two log sprites and assign then to a local variable. We'll then create an immutable array called spriteArray and instantiate it with these two logs (pieces). Once we spawn the stack the spriteArray is no longer needed. Then run a for loop that will create pieces.

For each piece we get a log out of the spriteArray using the remainder of i % 2 to determine if it is 0 or 1. The works because an even number % 2 is always 0 so it is the same thing as saying objectAtIndex:0; and an odd number % 2 is always 1 so it is the same thing as saying objectAtIndex:1.

We need to call the copy method when we do it like this. If we do not call copy, then it will crash after the first of each sprite is added, complaining that the sprite already has a parent and cannot be added again.

We'll use the value of i to multiply the position of the sprite in the stack, adding the height of the sprite * i + basePosition.y. This is a neat and easy way to stack each object.

We'll then call spawnBranch before adding the log as a child of StackController and the pieceArray.

In chooseSide we first get a random number between 0 and 1.0. 45% of the time we want the branch to be on the left, 45% we want the branch to be on the right, and 10% of the time we want no branch. Because the game will immediately end if the Player and a Branch make contact, we want to make sure that the first log cannot have a branch when the stack is created.

If randomNumber is less than 0.45, currentSide is set to Left. If randomNumber is less than 0.9, currentSide is set to Right. Otherwise it is set to None.

In spawnBranch: we are passing in an SKSpriteNode, this is the log that we want to possibly add a branch as a child to. First we'll run chooseSide to determine which side to put the branch on.

If the currentSide is Left, we make the position of the branch on the left side of the log. We'll grab the left branch out of the array as a copy and then add it as a child. If the currentSide is Right, we'll do the same thing but getting a right branch and positioning it on the right side of the log.

Just after the spawnBranch: method, add these two final methods.

```
#pragma mark - Stack Management
-(void)moveStack {
    NSTimeInterval oneFrame = 0.016;

    [self runAction:[SKAction waitForDuration:oneFrame] completion:^{
        SKSpriteNode *log = [self.pieceArray objectAtIndex:0];

        [self.pieceArray addObject:log];

        [self.pieceArray removeObjectAtIndex:0];

        [log removeAllChildren];

        [self spawnBranch:log];

        for (int i = 0; i < self.pieceArray.count; i++) {
            SKSpriteNode *chunk = [self.pieceArray objectAtIndex:i];

            chunk.position = CGPointMake(self.basePosition.x,
self.basePosition.y + chunk.size.height * i);
        }
    }];
}
```

```
#pragma mark - Update
-(void)update:(NSTimeInterval)delta {
    self.frameCount += delta;

    if (self.frameCount >= 1.0) {
        [self moveStack];

        self.frameCount = 0.0;
    }
}
```

In moveStack we start by creating an NSTimeInterval local variable called oneFrame that holds the approximately time in seconds of a single frame in a 60 FPS game. We'll be moving our Player by setting the position, but we want the physics engine to detect contact so we can end the game. Usually when you are using physics it causes problems to move nodes with SKActions or by setting the position. In our game it makes no sense to move the Player with physics, so we'll create an artificial delay for one frame to make sure the physics engine "catches" the contact between the Player and a Branch. If we didn't do this, it would be possible for the Player to move on top of a Branch and it would not be detected.

After waiting for the delay we'll get the first log out of the pieceArray. This is the log at the bottom of the stack. We then add this log to the end of the pieceArray, and remove it from the beginning of the pieceArray. We'd like to randomize the branches, we we'll strip the log of any children and pass it back into spawnLog to get a new random branch (or none).

Then we loop over the pieceArray. We are using the name "chunk" just to make sure that we don't have anything strange happening from using an already used name like "log". Don't be confused by this, "chunk" is a log. For each iteration through the loop we get a the log at the corresponding index in the array, we then set the position in the same way we did in spawnStack.

Finally we have our update:delta. In this method we simply add delta to frameCount. When frameCount reaches 1.0 approximately 1 second has passed and we moveStack before resetting the frameCount back to 0.

Select StackController.h. GameScene will need to know about our moveStack and update:delta methods. Between @interface and @end, add these lines.

```
-(void)moveStack;
-(void)update:(NSTimeInterval)delta;
```

Update GameScene

Select GameScene.m. Just after importing the header for the Birds class, add the header for our StackController class.

```
#import “StackController.h"
```

Just after adding the property for Birds in @interface GameScene() add this new line.

```
@property StackController *stackController;
```

In `setupScene`, just after adding the forest add these new lines.

```
self.stackController = [[StackController alloc] init];
[self addChild:self.stackController];
```

In update:currentTime, add this new line just after calling update:delta on the Birds.

```
[self.stackController update:delta];
```

Build and Run

Build and run the game ⌘**R** and let’s see the StackController move the logs.

Conclusion

In Chapter 3 we got the core game mechanic working. Once we have the player in the game we can make a small change to move the stack when the player taps the screen.

In Chapter 4 we'll focus on getting the Player class created and updating GameScene to get the "chop" working when the player taps the screen on the left or right.

Need Help?

If you got stuck on any of these steps for some reason, check your work against the project in the source code you downloaded for this book at **Project > Chapter 3 > Finish**.

Chapter 4 - Player

In Chapter 4 we will get our Player class in the game and chopping logs. The core game play of Timberman is the player must chop logs, avoid branches and stay ahead of a timer that is constantly running down. If the player is hit by a branch or time runs out, it is game over.

We'll be able to do some clean up on the cycle of losing and restarting the game in Chapter 5 once we have these basic features working.

Here are the things we need to do before moving on to Chapter 5.

Add our string names for the images to the SpriteName header.

Create a Player group to organize our classes in Xcode.

Create a Smoke class we'll use to animate a smoke ring when the player loses.

Create the Player class and add it to GameScene.

Update GameScene to use SKPhysicsContactDelegate.

Update touchesBegan:withEvent: to chop on the side the player touched.

Updating SpriteNames.h

In the Utilities group, select SpriteNames.h. Add the following new constants for our image names before #endif.

```
// Player
static NSString *const PLAYER0 = @"Player0";
static NSString *const PLAYER1 = @"Player1";
static NSString *const PLAYER2 = @"Player2";
static NSString *const SMOKE0 = @"Smoke0";
static NSString *const SMOKE1 = @"Smoke1";
static NSString *const SMOKE2 = @"Smoke2";
static NSString *const SMOKE3 = @"Smoke3";
static NSString *const SMOKE4 = @"Smoke4";
static NSString *const TOMBSTONE = @“Tombstone";
```

Smoke

Our Smoke class is an SKSpriteNode with an animation that plays through the textures that make the smoke ring animation. We'll make it invisible to start with and provide a public method called animateSmoke that will make it visible and run the animation.

For simplicity we'll add it as a child of the Player class.

Right click the Classes folder and choose **New Group**, name it **Player**. Right click the Player group and choose **New File…** Select **iOS > Source > Cocoa Touch Class**, name it **Smoke** and make it a subclass of **SKSpriteNode**, then click **Next** and **Create**.

Select Smoke.m and just after importing Smoke.h add the following private interface.

```
@interface Smoke()

@property SKAction *animation;

@end
```

Between @implemenation and @end, type in the following code.

```
#pragma mark - Init
-(instancetype)init {
    if ((self = [super init])) {
        SKTexture *texture = [[GameTextures sharedInstance]
textureWithName:SMOKE0];
        self = [Smoke spriteNodeWithTexture:texture];

        [self setup];
    }

    return self;
}

#pragma mark - Setup
-(void)setup {
    SKTexture *frame0 = [[GameTextures sharedInstance]
textureWithName:SMOKE0];
    SKTexture *frame1 = [[GameTextures sharedInstance]
textureWithName:SMOKE1];
    SKTexture *frame2 = [[GameTextures sharedInstance]
textureWithName:SMOKE2];
    SKTexture *frame3 = [[GameTextures sharedInstance]
textureWithName:SMOKE3];
    SKTexture *frame4 = [[GameTextures sharedInstance]
textureWithName:SMOKE4];

    self.animation = [SKAction animateWithTextures:@[frame0, frame1, frame2,
frame3, frame4] timePerFrame:0.1];

    self.alpha = 0.0;
}

#pragma mark - Animations
-(void)animateSmoke {
    self.alpha = 1.0;

    [self runAction:self.animation completion:^{
        self.alpha = 0.0;
    }];
}
```

In the `setup` method, we create an SKTexture for each of the images that make up the animation. We can then use those images in an animateWithTextures SKAction. Each will be displayed for 0.1 seconds. We set the alpha to 0.0 to make it invisible.

In `animateSmoke` we start by setting the `alpha` to 1.0, fully visible. We can then run the animation with a completion block that sets the `alpha` back to 0.0. The completion block is pretty handy, it runs whatever is in the block of code when the action is done.

Select Smoke.h. Between @interface and @end, add the following line so the signature for animateSmoke is visible to classes that import the header.

```
-(void)animateSmoke;
```

Player

The Player class for our game is somewhat simple. It needs an idle animation, a chop animation, a physics body that will test for contact with a Branch and a way to move to one of two fixed spots (left and right). We'll also give the Player class a method to call when the game ends.

Right click the Player group and choose **New File…** Select **iOS > Source > Cocoa Touch** Class. Name the class **Player** and make it a subclass of **SKSpriteNode**, then click **Next** and **Create**.

Just after importing Player.h, let's add the header for the Smoke class.

```
#import "Smoke.h"
```

Just after importing the header, lets add our private interface for the class member variables.

```
@interface Player()

@property CGPoint leftSide;
@property CGPoint rightSide;

@property SKAction *animationIdle;
@property SKAction *animationChop;

@property int taps;

@end
```

Before we get to the actions, lets cover the setup of the Player class. Just after @implementation type in this code to set up the Player class.

```
-(instancetype)init {
    if ((self = [super init])) {
        SKTexture *texture = [[GameTextures sharedInstance]
textureWithName:PLAYER0];
        self = [Player spriteNodeWithTexture:texture];

        [self setupPlayer];
        [self setupAnimations];
        [self setupPhysics];
    }

    return self;
}

#pragma mark - Setup
-(void)setupPlayer {
    self.leftSide = CGPointMake(ScreenSize().width * 0.22,
ScreenSize().height * 0.27);
    self.rightSide = CGPointMake(ScreenSize().width * 0.78,
ScreenSize().height * 0.27);

    self.position = self.leftSide;
}

-(void)setupAnimations {
    SKTexture *frame0 = [[GameTextures sharedInstance]
textureWithName:PLAYER0];
    SKTexture *frame1 = [[GameTextures sharedInstance]
textureWithName:PLAYER1];
    SKTexture *frame2 = [[GameTextures sharedInstance]
textureWithName:PLAYER2];

    self.animationIdle = [SKAction animateWithTextures:@[frame0, frame1]
timePerFrame:0.25];
```

```
    self.animationChop = [SKAction animateWithTextures:@[frame2]
timePerFrame:0.032];

    [self runAction:[SKAction repeatActionForever:self.animationIdle]];
}


-(void)setupPhysics {
    self.physicsBody = [SKPhysicsBody bodyWithCircleOfRadius:self.size.height
/ 3];
    self.physicsBody.categoryBitMask = ContactPlayer;
    self.physicsBody.collisionBitMask = 0x0;
    self.physicsBody.contactTestBitMask = ContactBranch;
    self.physicsBody.affectedByGravity = NO;
}
```

The init is the typical thing we've done getting and assigning a base texture for the sprite and initializing with it.

In `setupPlayer` we define what CGPoint the `leftSide` and `rightSize` variables hold. The `leftSide` is 22% of the width of the screen in from the left, and the `rightSide` is 22% of the screen in from the right. We finish off by setting the position to leftSide. When the Player is instantiated and added to the scene it will start with this position.

In `setupAnimations` we are getting our three textures that make up the Player animations. Our `animationIdle` is just a loop of playing the first two textures for a quarter second each. Our `animationChop` is playing the third texture for about two frames. We finish up the method by telling the Player to repeat the `animationIdle` animation forever.

In `setupPhysics` we are first creating the body to be a circle one-third of the hight of the Player texture. This makes it a nice tight circle that won't accidentally contact a branch that is just overhead but too far away to be a "hit". The `categoryBitMask` is the ID of this physics body, when the physics engine processes a contact it will use this to identify which bodies were involved. We don't want the Player to collide with anything so we set `collisionBitMask` to 0x0. We do want the Player to test for contact against a Branch, so we set the `contactTestMask` to `ContactBranch`. The Player should not be affected by gravity, if it is and we don't tell the physics engine to turn off gravity it will fall right off the screen.

Lets get the remaining action methods in the class. Just after `setupPhysics`, type in the following code.

```
#pragma mark - Actions
-(void)chopLeft {
    self.taps++;

    self.position = self.leftSide;

    self.xScale = 1;

    [self runAction:self.animationChop];
}

-(void)chopRight {
    self.taps++;

    self.position = self.rightSide;

    self.xScale = -1;

    [self runAction:self.animationChop];
}

#pragma mark - Get Taps
-(int)getTaps {
    return self.taps;
}

#pragma mark - Animation
-(void)animateSmoke {
    Smoke *smoke = [[Smoke alloc] init];

    [self addChild:smoke];

    [smoke animateSmoke];
}

#pragma mark - Game Over
-(void)gameOver {
    [self removeAllActions];

    SKTexture *texture = [[GameTextures sharedInstance]
textureWithName:TOMBSTONE];

    if (self.position.x > ScreenSize().width / 2) {
        self.texture = texture;
        self.xScale = -1;
    } else {
        self.texture = texture;
    }

    [self animateSmoke];
}
```

The chopLeft and chopRight methods do almost the same thing. First we increment taps, this is the number of times the player has “chopped” while the game is running. I debated whether to call it chops or taps, in the end I decided that taps is more clear.

Each sets the position to the side of the screen where the player tapped. In GameScene we’ll call the appropriate method depending on whether the player is tapping left or right. Other than running `animationChop` in each, we are also setting the scale to “flip” the image so that it faces right or left. We could have made this into another method, but it is as easy in this case to just write the single line of code.

The `getTaps` method returns the value of the taps member variable to calling classes. The idea here is that we don’t want other classes to directly manipulate this value. This is part of our concept of encapsulation. Other classes need access to the value, but they don’t need to access the variable directly.

The `animateSmoke` method instantiates an instance of Smoke adds it as a child of Player and runs the `animateSmoke` method. In case I haven’t already mentioned it, when you add a child to a node the child is always drawn on top of it by default. If we wanted our instance of Smoke to draw under the Player we would have to set the zPosition to be one lower than the Player.

The `gameOver` method will be called when the game is over. First we will remove all running actions on Player. In this case this will stop either of our animations that could be running. Then we get a texture of the Tombstone image. We are going to replace the texture used by Player with the Tombstone texture after determining which side of the screen the Player is on. We finish up by calling `animateSmoke`.

The chopLeft, chopRight, getTaps and gameOver methods need to be public so that other classes know how to call them. Select Player.h and add these signatures between @interface and @end.

```
-(void)chopLeft;
-(void)chopRight;
-(int)getTaps;
-(void)gameOver;
```

Updating GameScene

In the Scenes group, select GameScene.m. Near the top of the file just after importing the header for StackController we need to import the header for Player.

```
#import “Player.h”
```

We need to make GameScene a delegate for SKPhysicsContactDelegate. Change @interface GameScene to this.

```
#pragma mark – Class Private Interface
@interface GameScene() <SKPhysicsContactDelegate>
```

In the @interface GameScene() <SKPhysicsContactDelegate>, just after creating the property for StackController, add this new property for Player.

```
@property Player *player;
```

At the beginning of `setupScene` we need to set the contactDelegate to the scene.

```
self.physicsWorld.contactDelegate = self;
```

In `setupScene` just after adding the `stackController` lets add player as a child of the scene.

```
self.player = [[Player alloc] init];
[self addChild:self.player];
```

The next thing we need to do is handle touches on the screen. In our game if the player touches anywhere on the left half of the screen we count it as a touch left. If the player touches anywhere on the right side of the screen we count it as a touch right.

Scroll down to `touchesBegan:withEvent` and change the entire method to this code.

```
#pragma mark - Touch Handling
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

    UITouch *touch = [touches anyObject];
    CGPoint touchLocation = [touch locationInNode:self.scene];

    if (touchLocation.x > ScreenSize().width / 2) {
        [self.player chopRight];
        [self.stackController moveStack];
    } else if (touchLocation.x < ScreenSize().width / 2) {
        [self.player chopLeft];
        [self.stackController moveStack];
    }

}
```

The first thing we are doing here is getting the touch and converting it to a CGPoint on the screen. Those two lines will be the same in just about any game where the place the player touched on screen matters.

Then we determine if the `touchLocation.x` is more than half the width of the screen. If it is we count it as a touch right, tell the player instance to run the `chopRight` method and tell the `stackController` instance to run the `moveStack` method. We do the same thing for a left touch, calling the player instance `chopLeft` method and the `stackController` instance `moveStack` method.

The final thing we need to do for this chapter is to tell the physics simulation what to do when contact is detected.

Go down to the `update:` method. We no longer want to move the stack based on time, we want the player “taps” to move the log. Comment out the last line that calls `update:` on the `stackController`.

```
//    [self.stackController update:delta];
```

Just after the `update:` method at the bottom of GameScene, before @end add this new method to handle physics contact.

```
#pragma mark - Contact
-(void)didBeginContact:(SKPhysicsContact *)contact {

    SKPhysicsBody *other = contact.bodyA.categoryBitMask == ContactPlayer ?
contact.bodyB : contact.bodyA;

    if (other.categoryBitMask == ContactBranch) {
        [self.player gameOver];
    }
}
```

Here we are using our friend the ternary operator to determine which SKPhysicsBody is not the Player (`ContactPlayer`). Then we ask if the `categoryBitMask` of this other body is `ContactBranch`. If it is, we run the `gameOver` method on `player`.

When we get our game state code working in the next chapter this contact will end the game, for now it will just change the texture to the tombstone and animate the smoke.

Build and Run

Build and run (⌘**R**) the game and try it out. Tap on the screen on the left and right side. See what happens when the player is hit by a branch.

Conclusion

In Chapter 4 we got the Player in the GameScene and got contact detection working. Tapping the screen moves the player to the side the player is tapping, and moves the lock stack.

The main issue we have right now is that the game never ends. We'd like to end the game and give the player a way to restart it when they lose. It would also be nicer if the game started up in a sort of waiting state, and started running when the player taps the screen.

In Chapter 5 we'll implement these features by making our game work in states. We'll also need some buttons. With the combination of these things we can start to make TimberJim into an actual playable game.

Need Help?

If you got stuck on any of these steps for some reason, check your work against the project in the source code you downloaded for this book at **Project > Chapter 4 > Finish**.

Chapter 5 - Game State

We have the core elements of our game working and in the scene. The main thing we need right now is to be able to switch between different states. When the player makes contact with a branch, we want to end the game. We'll also need a way to restart the game.

One way that I like to handle these tasks is with states. We'll add states to our game for Waiting, Running, Paused and GameOver. This will allow us to execute certain code, or nothing at all depending on what the state of the game is currently.

It would be nice to have a simple tutorial that tells the player how to play, and we'll need a button the player can tap to restart the game.

Here are the things we need to get done before moving on to Chapter 6.

Add our Button image names to the SpriteName header.

Create a Buttons group to store our new classes.

Create a TutorialButton class.

Create a PlayButton class.

Create a GameState NS_ENUM in GameScene.

Add our buttons to GameScene.

Create functions for setting the states and performing state-specific tasks.

Update SpriteNames.h

In the Utilities group, select SpriteNames.h and add these new constants just before #endif.

```
// Buttons
static NSString *const TAPLEFT = @"TapLeft";
static NSString *const TAPRIGHT = @"TapRight";
static NSString *const PLAYBUTTON = @"PlayButton";
```

TutorialButton

Right click the Classes folder in Xcode and choose **New Group**, name the group **Buttons**. Right click the Buttons group and choose **New File…** Select **iOS > Source > Cocoa Touch Class**, click **Next.** Name the class **TutorialButton**, make it a subclass of **SKNode**, and click **Next** and **Create**.

Our TutorialButton is an SKNode that will have two child sprites with our TapLeft and TapRight images. We'll give TutorialButton a couple of animations that will make it more interesting than just a static sprite on screen. It will animate in, bounce a little, and animate out when the player starts the game by touching anywhere on screen.

Select TutorialButton.m and type in the following for our private interface just after importing the header.

```
@interface TutorialButton()

@property SKSpriteNode *leftTap;
@property SKSpriteNode *rightTap;

@property CGPoint startLeftTapPosition;
@property CGPoint endLeftTapPosition;
@property CGPoint startRightTapPosition;
@property CGPoint endRightTapPosition;

@end
```

We'll start with just the init and setup sections and then look at the action functions.

Just after @implememntation, add this new code.

```
#pragma mark - Init
-(instancetype)init {
    if ((self = [super init])) {
        [self setup];
    }

    return self;
}

#pragma mark - Setup
-(void)setup {
    self.leftTap = [[GameTextures sharedInstance] spriteWithName:TAPLEFT];
    self.rightTap = [[GameTextures sharedInstance] spriteWithName:TAPRIGHT];

    self.startLeftTapPosition = CGPointMake(0 - self.leftTap.size.width,
ScreenSize().height * 0.2);
    self.endLeftTapPosition = CGPointMake(ScreenSize().width * 0.15,
ScreenSize().height * 0.2);
    self.startRightTapPosition = CGPointMake(ScreenSize().width +
self.rightTap.size.width, ScreenSize().height * 0.2);
    self.endRightTapPosition = CGPointMake(ScreenSize().width * 0.85,
ScreenSize().height * 0.2);

    self.leftTap.position = self.startLeftTapPosition;
    self.rightTap.position = self.startRightTapPosition;

    [self addChild:self.leftTap];
    [self addChild:self.rightTap];
}
```

TutorialButton is an SKNode, which is basically an empty node. All we need to do in the `init` is call `[super init]` and run our `setup` method.

In setup we are getting our two sprits for the LeftTap and RightTap images. We have a starting point and ending point for each button. This will make our lives a little easier when we animate them here in a bit. The starting position for each is off screen left and right + the width of the sprite. This will make them fully off screen. The ending position for each is 15% in from the sides on the let and right.

Before leaving setup we set the initial position for `leftTap` and `rightTap` to the starting position we just defined for each. Finally we add them as children of TutorialButton.

Next we just need to add our methods for animating these sprites in and out. Just below the setup `method`, add this new code.

```
#pragma mark - Animations
-(void)animateIn {
    SKAction *leftAction = [SKAction moveTo:self.endLeftTapPosition duration:
0.5];
    SKAction *rightAction = [SKAction moveTo:self.endRightTapPosition
duration:0.5];

    [self.leftTap runAction:leftAction];
    [self.rightTap runAction:rightAction];

    SKAction *scaleUp = [SKAction scaleTo:1.1 duration:0.25];
    SKAction *scaleNormal = [SKAction scaleTo:1.0 duration:0.25];
    SKAction *scaleSequence = [SKAction sequence:@[scaleUp, scaleNormal]];

    [self.leftTap runAction:[SKAction repeatActionForever:scaleSequence]];
    [self.rightTap runAction:[SKAction repeatActionForever:scaleSequence]];
}

-(void)animateOut {
    [self.leftTap removeAllActions];
    [self.rightTap removeAllActions];

    [self.leftTap runAction:[SKAction moveTo:self.startLeftTapPosition
duration:0.25]];
    [self.rightTap runAction:[SKAction moveTo:self.startRightTapPosition
duration:0.25]];

}
```

In animateIn we first create a moveTo:duration: action for each of our sprites that moves them to the end position over half a second. Then we tell each to run that action.

Then we create an action to scale up, an action to scale to the normal size, and create a sequence made up o those actions. Finally we'll tell each sprite to run this sequence forever.

In the `animateOut` method we first will remove all actions, this will cause each sprite to stop doing the scaleSequence we created above. Then we create a moveTo:duration: action on each that moves them out to the staring position.

We will want animateIn and animateOut to be visible to calling classes, so we'll add the signature of each to the header. Select TutorialButton.h and add these two lines between @inerface and @end.

```
-(void)animateIn;
-(void)animateOut;
```

PlayButton

Right click the Buttons group and choose **New File…** Select **iOS > Source > Cocoa Touch Class** and click **Next**. Name the class **PlayButton** and make it a subclass of **SKSpriteNode**, then click **Next** and **Create**.

Select PlayButton.m and type in the following code between @implementation and @end.

```
#pragma mark - Init
-(instancetype)init {
    if ((self = [super init])) {
        SKTexture *texture = [[GameTextures sharedInstance]
textureWithName:PLAYBUTTON];
        self = [PlayButton spriteNodeWithTexture:texture];

        [self setup];
    }

    return self;
}

#pragma mark - Setup
-(void)setup {
    self.position = CGPointMake(ScreenSize().width / 2, ScreenSize().height *
0.15);
    [self setScale:0];
}

#pragma mark - Animations
```

```
-(void)animateIn {
    [self runAction:[SKAction scaleTo:1.1 duration:0.25] completion:^{
        [self runAction:[SKAction scaleTo:1.0 duration:0.25]];
    }];
}


-(void)animateOut {
    [self runAction:[SKAction scaleTo:0 duration:0.25]];
}
```

This one is pretty simple. We are positioning it in the middle of the screen horizontally, and 15% up the screen vertically and setting the scale to 0.

In the `animateIn` method we run an action to scale it up to 1.1 and in the completion block scale it back to 1.0. This overcalling before going back to a normal scale will make it “pop” on screen.

In `animateOut` we scale back down to 0, becoming invisible again.

We’ll want both of our animation methods to be visible to calling classes, so lets add it to the header.

Select PlayButton.h and add these two method signatures between @interface and @end.

```
-(void)animateIn;
-(void)animateOut;
```

Update GameScene with GameState

We need to import the header for each of the buttons we just created so we can use them. Just after importing the header for the Player add these two new lines.

```
#import "TutorialButton.h"
#import "PlayButton.h"
```

In the Scenes group select GameScene.m. Near the top of the class, just after importing the headers add our new NS_ENUM.

```
typedef NS_ENUM(NSInteger, GameState) {
    Waiting, Running, Paused, GameOver
};
```

We are going to need two new member variables of type GameState. The first will have our current state, and the second will hold the previous state. This will be useful for pausing the game later on.

These are properties of the scene, so to keep it need lets add them just after lastUpdateTime in the @interface GameScene().

```
@property GameState state;
@property GameState previousState;
```

We are going to also need member variables for our two buttons. Just after creating the member variable for the player in @interface GameScene() add these.

```
@property TutorialButton *tutorialButton;
@property PlayButton *playButton;
```

We need some methods to handle our various states. To keep it neat and easy to understand we'll separate each thing we need into a method we can call when appropriate.

At the bottom of the class, just before @end add this new code for our states.

```
#pragma mark - State Functions
-(void)switchToWaiting {
    self.state = Waiting;

    [self.tutorialButton animateIn];
}

-(void)switchToRunning {
    self.state = Running;

    [self.tutorialButton animateOut];
}

-(void)switchToPaused {
    self.previousState = self.state;

    self.state = Paused;
}

-(void)switchToResume {
    self.state = self.previousState;
}

-(void)switchToGameOver {
    self.state = GameOver;

    [self.player gameOver];

    [self displayGameOver];
}

-(void)displayGameOver {
    [self runAction:[SKAction waitForDuration:0.5] completion:^{
        [self addChild:self.playButton];
        [self.playButton animateIn];
    }];
}

-(void)resetGame {
    [self removeAllChildren];

    SKScene *scene = [GameScene sceneWithSize:ScreenSize()];

    SKTransition *transition = [SKTransition fadeWithColor:[SKColor
blackColor] duration:1.0];
```

```
    [self.view presentScene:scene transition:transition];
}
```

Each of these are really simple. It will be apparent why that is a good thing and quite powerful shortly.

Change touchesBegan:withEvent: to use GameState

Scroll up in GameScene to the `touchesBegan:withEvent:` method and change the entire method to this code.

```
#pragma mark - Touch Handling
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

    UITouch *touch = [touches anyObject];
    CGPoint touchLocation = [touch locationInNode:self.scene];

    switch (self.state) {
        case Waiting:
            [self switchToRunning];
            break;

        case Running: {
            if (touchLocation.x > ScreenSize().width / 2) {
                [self.player chopRight];
                [self.stackController moveStack];
            } else if (touchLocation.x < ScreenSize().width / 2) {
                [self.player chopLeft];
                [self.stackController moveStack];
            }
            break;
        }

        case Paused:
            break;

        case GameOver:
            if ([self.playButton containsPoint:touchLocation]) {
                [self resetGame];
            }
    }
}
```

Now this method does something different depending on what our state is. If it is Waiting, we call `switchToRunning` to start the game. If it is Running, we handle chops and moving the stack. If is Paused, we do nothing at all. If it s GameOver and the player touches the playButton, we reset the game.

Update update: to use GameState

In GameScene, replace the entire update: method with this new code.

```
-(void)update:(NSTimeInterval)currentTime {

    switch (self.state) {
        case Waiting:
        case Running:
        case GameOver:
        {
            NSTimeInterval delta = currentTime - self.lastUpdateTime;
            self.lastUpdateTime = currentTime;

            [self.clouds update:delta];
            [self.birds update:delta];

            break;
        }

        case Paused:
            break;
    }
}
```

If the state is Waiting, Running or GameOver we calculate delta and manually run the update: method on clouds and birds. There is no reason to do anything if the game is Paused so we just break out.

Update didBeginContact: to use GameState

Scroll down to didBeginContact: and replace the entire method with this new code.

```
-(void)didBeginContact:(SKPhysicsContact *)contact {

    if (self.state != Running) {
        return;
    } else {
        SKPhysicsBody *other = contact.bodyA.categoryBitMask == ContactPlayer
? contact.bodyB : contact.bodyA;

        if (other.categoryBitMask == ContactBranch) {
            [self switchToGameOver];
        }
    }

}
```

Now if the state is anything other than Running, we jus return and exit. There shouldn't be any contact happening in any other state, and for simplicity we'll only process it at all if the state is Running.

If the contact is a ContactBranch categoryBitMask, we end the game by calling `switchToGameOver`.

Update setupScene

The last change we need to make to GameScene is to add our new buttons and run the `switchToWaiting` method.

Near the end of the setupScene method, add these new lines after adding the player as a child of the scene.

```
self.tutorialButton = [[TutorialButton alloc] init];
[self addChild:self.tutorialButton];

self.playButton = [[PlayButton alloc] init];


[self switchToWaiting];
```

One more thing…

It would be nice to start hearing some sound. Fortunately because we are using ObjectAL this is going to be just a few lines of code to get some sound effects going.

In the View Controller group, select GameViewController.m.

In the `viewDidLayoutSubviews` method, just after presenting the scene near the bottom of the method, add this new code (before the debug statement).

```
// Preload Sound Effects
[[OALSimpleAudio sharedInstance] preloadEffect:@"Chop.caf"];
[[OALSimpleAudio sharedInstance] preloadEffect:@"GameOver.caf"];
[[OALSimpleAudio sharedInstance] preloadEffect:@"Pop.caf"];

// Preload and play Music
[[OALSimpleAudio sharedInstance] playBg:@"SpiritualMoments.mp3" loop:YES];
```

We are “pre-loading” our sound effects and music. If we didn’t there would be a slight delay the first time we tried to play each as it is loaded out of the app bundle.

Under the Player group, select Player.m. Add this line to the end of the `chopLeft` and `chopRight` methods.

```
[[OALSimpleAudio sharedInstance] playEffect:@“Chop.caf"];
```

In Player.m, add this line to the end of the `animateSmoke` method.

```
[[OALSimpleAudio sharedInstance] playEffect:@“GameOver.caf"];
```

Select TutorialButton.m. Add this line to the end of the `animateOut` method.

```
[[OALSimpleAudio sharedInstance] playEffect:@“Pop.caf"];
```

In GameScene.m scroll down to the `resetGame` method and add this line as the first line of code in the method.

```
[[OALSimpleAudio sharedInstance] playEffect:@“Pop.caf"];
```

By the way, we are putting this here in `resetGame` because if we put it in the button it would be removed before it has a chance to play the sound.

Earlier in the book I talked at length about literal strings being a big source of bugs. We used them here, and it would be better to come up with a header file that has the names of all our sound effects and music as constants for the same reason. I am going to leave that up to you as an exercise. Refer to how we added SpriteNames.h to do the same thing for your audio file names.

Build and Run

Build and run the game ⌘**R**.

Play the game for a while. The changes we made in Chapter 5 really start to make this feel like a game someone might actually want to play.

We start off in a waiting state with a nice little hint on how to play the game provided by TutorialButton. We chop either side of the screen, and when we get hit by a branch, oops, it ends the game. Now we get a button that animates in we can tap to restart the game and try again.

Conclusion

Our changes in Chapter 5 have brought TimberJim a long way towards being a playable game. Now we can play the game and restart when we lose.

There is one more mechanic we need to get in the game, the time bar. We want to give the player just enough time that they can tap to chop logs, but take that time away fast enough that they must keep up a pretty fast pace or the time runs out and they lose.

In Chapter 6 we'll add the time bar, and update GameScene to use this new mechanic. We'll also add a label that shows the player how many taps they have racked up. This makes the game more visually interesting and a way to see how well they are doing. The tap count will serve as the players "score" for the rest of the book.

Need Help?

If you got stuck on any of these steps for some reason, check your work against the project in the source code you downloaded for this book at **Project > Chapter 5 > Finish**.

Chapter 6 - Adding the Timer

A key mechanic of the game is that you can't just chop the logs forever at whatever pace you want. That makes the game pretty trivial. We'll make the game challenging by giving the player a 6 second time window that is constantly ticking down. The result will be that the player has to make fast decisions and eventually will make a mistake which ends the game.

The score for our version of the game is simply the number of taps the player achieved before the game ends. We are already incrementing this value, we just haven't put in a way to display it yet. This visual feedback makes the game more fun and gives the player a goal to reach for.

Before moving on to the final chapter of this book, here are the things we need to get done.

Add the names of our new sprites to the SpriteNames.h header.

Create the ScoreLabel group and class.

Create a TimeBar group and class.

Add the new nodes to GameScene.

Add the `addSecondsToTime` method to GameScene.

Update GameScene to use our time mechanic and end the game when out of time.

Update SpriteNames.h

In the Utilities group, select SpriteName.swift and add this new code near the end of the class.

In the Utilities group, select SpriteNames.h. Add these new lines before #endif.

```
// Time Bar
static NSString *const TIMEBARBACKGROUND = @"TimeBarBackground";
static NSString *const TIMEBAR = @“TimeBar";
```

ScoreLabel

Our ScoreLabel will be a Bitmap font label. We’ll make this as an SKNode and give it a method calling classes can use to update the text value.

Right click Classes and choose **New Group**, name this group **ScoreLabel**. Right click the ScoreLabel group, choose **New File…** Select **iOS > Source > Cocoa Touch Class**, click **Next**, name it **ScoreLabel** and make it a subclass of **SKNode**, then click Next and **Create**.

Select ScoreLabel.m. Just after importing the header, type in the following code for our private interface.

```
@interface ScoreLabel()

@property BMGlyphLabel *label;

@end
```

Between @implementation and @end type in the following code.

```
#pragma mark - Init
-(instancetype)init {
    if ((self = [super init])) {
        [self setup];
    }

    return self;
}


#pragma mark - Setup
-(void)setup {
    BMGlyphFont *font = [BMGlyphFont fontWithName:@"GameFont"];
    self.label = [BMGlyphLabel labelWithText:@"0" font:font];
    self.label.position = CGPointMake(ScreenSize().width / 2,
ScreenSize().height * 0.7);

    [self.label setScale:2.0];

    [self addChild:self.label];
}


#pragma mark - Public Actions
-(void)updateLabel:(int)score {
    self.label.text = [NSString stringWithFormat:@"%d", score];
}
```

In `setup` we first need to instantiate a BMGlyphFont that has our Bitmap font. Once that is done we can create the label with our initial string and using the font. We'll put this label in the center of the screen horizontally and 70% up the screen vertically. Because our Bitmap font is one size, and to my eye this label looks better large, we'll scale it up to 2.0 (200%) and then add it as a child node.

We'll need a method that calling classes can use to update the text of the label. In `updateLabel:` we take in the score and convert it to an NSString.

Select ScoreLabel.h and add the method signature between @interface and @end.

```
-(void)updateLabel:(int)score;
```

TimeBar

Our TimeBar will be another SKNode. It consists of the background with a nice red bar that we will scale to represent the amount of time the player has left.

Right click Classes and choose **New Group**, name this group **TimeBar**. Right click the TimeBar group, choose **New File…** Select **iOS > Source > Cocoa Touch Class**, click **Next.** Name it **TimeBar,** make it a subclass of **SKNode**, click **Next** and **Create**.

Select TimeBar.m. Just after importing the header, lets add our private interface.

```
@interface TimeBar()

@property SKSpriteNode *timeBar;

@end
```

Between @implementation and @end, add the following code.

```
#pragma mark - Init
-(instancetype)init {

    if ((self = [super init])) {
        [self setup];
    }

    return self;
}

#pragma mark - Setup
-(void)setup {
    // Background
    SKSpriteNode *background = [[GameTextures sharedInstance]
spriteWithName:TIMEBARBACKGROUND];
    background.position = CGPointMake(ScreenSize().width / 2,
ScreenSize().height * 0.85);
    [self addChild:background];

    // Time Bar
    self.timeBar = [[GameTextures sharedInstance] spriteWithName:TIMEBAR];
    self.timeBar.anchorPoint = CGPointMake(0, 0.5);
    self.timeBar.position = CGPointMake(-self.timeBar.size.width / 2, 0);
    [background addChild:self.timeBar];
}
```

```
#pragma mark - Actions
-(void)updateTimeBar:(NSTimeInterval)seconds {

    if (seconds > 6.0) {
        return;
    }

    if (seconds > 0) {
        self.timeBar.xScale = seconds / 6;
    } else {
        self.timeBar.hidden = YES;
    }

}
```

This probably all looks really familiar up until working with the timeBar in `setup`. If you recall, sprites are positioned using their `anchorPoint`. We want to scale the `timeBar` so that it grows from the left. We set the `anchorPoint` to 0 on the X axis, which means the left side of the sprite. We set the `anchorPoint` to 0.5 on the Y axis, which means the middle of the sprite on that axis. So this sprite is positioned using the left center of the sprite, exactly what we need for a sprite that will grow and shrink from the left.

The other thing to mention is that we are adding `timeBar` as a child of `background`. When you add a sprite as a child of another sprite, it uses the parent sprites anchor point to position it. If we just added the `timeBar` to `background` and did nothing else, it would put the left edge of the `timeBar` in the middle of the `background`.

Knowing that child sprites are positioned using the parent sprites `anchorPoint`, we can fix this for our time bar by positioning its X position as `-self.timeBar.size.width / 2`. This means place this sprite half its width to the left of the `anchorPoint`.

Doing so we will get this.

We need to provide the signature of updateTimeBar: in the header file. Select TimeBar.h and add this line between @interface and @end.

```
-(void)updateTimeBar:(NSTimeInterval)seconds;
```

Updating GameScene

We need to get our ScoreLabel and TimeBar in the scene. We also need to have a variable that tracks the time left that is always ticking down, and a way to add time when the player taps the screen.

The first thing we need to do is import the headers for the two classes that we just created. In the Scenes group, select GameScene.m. Just after importing the header for the PlayButton, add the following.

```
#import "ScoreLabel.h"
#import “TimeBar.h"
```

We need a member variable that we can use to track how much time the player has left.

In @interface GameScene() just after the property for previousState, add this new property.

```
@property NSTimeInterval timeLeft;
```

Still inside @interface GameScene(), we need a member variable for our ScoreLabel and TimeBar. Just after adding the property for playButton add these two new properties.

```
@property ScoreLabel *scoreLabel;
@property TimeBar *timeBar;
```

Scroll down to the `setupScene` method. Just before the end of the method, after calling `switchToWaiting` add these new lines of code.

Scroll down to `switchToRunning`. Just before the end of the method add this new line.

```
self.timeLeft = 6.0;
```

We need a way to give the player some time back when they chop logs. Scroll to the end of the class. Just before @end, add this new method.

```
#pragma mark - Timer Functions
-(void)addSecondsToTime {

    if (self.timeLeft < 6.0) {
        self.timeLeft += 1.0;
    } else {
        self.timeLeft = 6.0;
    }

    [self.timeBar updateTimeBar:self.timeLeft];
}
```

We need to refactor our update: method. We want the Birds and Clouds to scroll normally in all states except Pause. In Running we need to tick down timeLeft and update the TimeBar. To make that work we need to change what we have in update:

Scroll to the update: method and change the entire thing to this code.

```
#pragma mark - Update
-(void)update:(NSTimeInterval)currentTime {

    switch (self.state) {
        case Waiting:
        case GameOver: {
            NSTimeInterval delta = currentTime - self.lastUpdateTime;
            self.lastUpdateTime = currentTime;

            [self.clouds update:delta];
            [self.birds update:delta];

            break;
        }
        case Running: {
            NSTimeInterval delta = currentTime - self.lastUpdateTime;
            self.lastUpdateTime = currentTime;

            [self.clouds update:delta];
            [self.birds update:delta];

            self.timeLeft -= delta * 4;

            [self.timeBar updateTimeBar:self.timeLeft];

            if (self.timeLeft <= 0) {
                [self switchToGameOver];
            }

            break;
        }

        case Paused:
            break;
    }
}
```

It is basically the same code, we've just broken out the Running state to include code to update timeLeft, update the timeBar and end the game if we are out of time.

Our final task for this chapter is to credit the player for chops by adding time. Scroll to touchesBegan:withEvent: add replace the entire method with this code.

```
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

    UITouch *touch = [touches anyObject];
    CGPoint touchLocation = [touch locationInNode:self.scene];

    switch (self.state) {
        case Waiting:
            [self switchToRunning];
            break;

        case Running: {
            if (touchLocation.x > ScreenSize().width / 2) {
                [self.player chopRight];
                [self.stackController moveStack];
                [self.scoreLabel updateLabel:[self.player getTaps]];
                [self addSecondsToTime];

            } else if (touchLocation.x < ScreenSize().width / 2) {
                [self.player chopLeft];
                [self.stackController moveStack];
                [self.scoreLabel updateLabel:[self.player getTaps]];
                [self addSecondsToTime];

            }
            break;
        }

        case Paused:
            break;

        case GameOver:
            if ([self.playButton containsPoint:touchLocation]) {
                [self resetGame];
            }
    }
}
```

This code is nearly identical to what we had. Now we are updating the scoreLabel with the player's taps and adding time to timeLeft for each chop.

Build and Run

Lets build and run the game ⌘**R**.

Play the game some. Let the time run out and notice that it ends the game. Play it “properly” for a while and try to stay ahead of the timer. I think the TimeBar does a really good job of making the game feel fast paced. The ScoreLabel really adds information to the game, I find myself wanting to beat my previous score each play through.

Conclusion

We have the final game mechanic working in the game. I think this really took the game to the next level and makes it pretty fun too.

As fun as the game is right now it has a couple of problems we need to solve before we can call this book done.

Right now, your score is literally temporary. It never gets saved anywhere. The second is that we never display the score anywhere. You can see it because nothing is covering the score, but you'd have to write it down if you are going for a high score. The game could use a PauseButton. The final thing we need to address is that the game shows a gray blank screen very briefly while GameScene loads.

In the final chapter of this book, Chapter 7, we'll address these things and wrap up.

Need Help?

If you got stuck on any of these steps for some reason, check your work against the project in the source code you downloaded for this book at **Project > Chapter 6 > Finish**.

Chapter 7 - Wrapping Up

The final things we need to do for this game are fix the gray screen, save the score to disk, add a score board and a pause button.

Here are the steps we need to do to finish the project.

Add a LoadScene that has a black background and loads GameScene.

Update GameViewController to load LoadScene at launch.

Add a GameSettings Singleton class that can save the player's best score to disk.

Update the Player class to check for and save a best score using GameSettings.

Update SpriteNames.h with our PauseButton and Resume button images.

Create a PauseButton class and add it to GameScene.

Add a `pauseButtonPressed` method to GameScene.

Add Paused code to `touchesBegan:withEvent:` in GameScene.

LoadScene

You have probably noticed that our game briefly displays a blank gray screen during launch before it displays the GameScene. This is because it takes just a moment for GameScene to fully load and in the brief time that it takes we are seeing what an SKScene looks like with nothing in it. The default background color of SKScene happens to be medium gray.

The best default backgroundColor would be black. I have filed a bug report with Apple, but in the meantime we can work around this by launching the game to an empty SKScene that has a black backgroundColor and then loads GameScene.

Right click the Scenes group and choose **New File…** Select **iOS > Source > Cocoa Touch Class**, name it **LoadScene** and make the subclass **SKScene**. Click **Next** and **Create**.

Select LoadScene.m. Near the top of the file, just after importing the header for LoadScene import the header for GameScene.

```
#import “GameScene.h"
```

We are going to set the backgroundColor of this scene to black. Then we’ll run an SKAction that waits for approximately one frame (0.016) and then creates an instance of GameScene and loads it. Between @implementation and @end, type in the following code.

```
-(void)didMoveToView:(SKView *)view {
    self.backgroundColor = [SKColor blackColor];

    [self runAction:[SKAction waitForDuration:0.016] completion:^{
        CGSize viewSize = CGSizeMake(ScreenSize().width,
ScreenSize().height);

        GameScene *scene = [[GameScene alloc] initWithSize:viewSize];
        scene.scaleMode = SKSceneScaleModeAspectFill;

        SKTransition *transition = [SKTransition fadeWithColor:[SKColor
blackColor] duration:0.25];

        [self.view presentScene:scene transition:transition];
    }];
}
```

Update GameViewController

Now we need to update our GameViewController to load our LoadScene. Select GameViewController.m in the ViewController group.

Near the top of the file, replace `#import “GameScene.h”` with the header for our LoadScene.

```
#import “LoadScene.h"
```

We can simplify our code for `viewDidLayoutSubviews`, replace the entire method with this new code.

```
-(void)viewDidLayoutSubviews {

    [super viewDidLayoutSubviews];

    SKView *skView = (SKView *)self.view;

    if (!skView.scene) {

        if (kDebug) {
            skView.showsFPS = YES;
            skView.showsNodeCount = YES;
            skView.showsPhysics = YES;
        }

        LoadScene *scene = [[LoadScene alloc] init];

        [skView presentScene:scene];

        // Preload Sound Effects
        [[OALSimpleAudio sharedInstance] preloadEffect:@"Chop.caf"];
        [[OALSimpleAudio sharedInstance] preloadEffect:@"GameOver.caf"];
        [[OALSimpleAudio sharedInstance] preloadEffect:@"Pop.caf"];

        // Preload and play Music
        [[OALSimpleAudio sharedInstance] playBg:@"SpiritualMoments.mp3"
loop:YES];
    }

}
```

Update Utilities.h

We no longer need to see debug information. Lets turn that off by setting the kDebug value to NO in Utilities.h.

```
#pragma mark - Debug
static const BOOL kDebug = NO;
```

Build and Run

Build and Run the game and notice that we have about the same delay while GameScene loads, but now it is much less noticeable with the black background.

GameSettings

We need a way to save the player's high score to disk. One way we can do that is to store it in the application plist using NSUserDefaults. We'll make this a Singleton and call it from the `gameOver` method in the Player class.

Right click the Singletons group and choose **New File…** Select **iOS > Source > Cocoa Touch Class** and click **Next**. Name the class **GameSettings** and make it a subclass of **NSObject**, then click **Next** and **Create**.

Select GameSettings.m. Just after importing the header lets add our private interface for the class.

```
@interface GameSettings()

@property NSUserDefaults *localDefaults;
@property NSString *keyFirstRun;
@property NSString *keyBestScore;
@property int bestScore;

@end
```

Between @implementation and @end, type in the following code.

```
#pragma mark - Init
+(instancetype)sharedInstance {
    static GameSettings *sharedInstance = nil;
    static dispatch_once_t onceToken;

    dispatch_once(&onceToken, ^{
        sharedInstance = [[self alloc] init];
    });

    return sharedInstance;
}

-(instancetype)init {

    if ((self = [super init])) {
        [self setup];

        if ([self.localDefaults objectForKey:self.keyFirstRun] == nil) {
            [self firstLaunch];
        }
    }

    return self;
}

#pragma mark - Setup
-(void)setup {
    self.localDefaults = [NSUserDefaults standardUserDefaults];
    self.keyFirstRun = @"FirstRun";
    self.keyBestScore = @"BestScore";
}

-(void)firstLaunch {
    [self.localDefaults setInteger:0 forKey:self.keyBestScore];
    [self.localDefaults setBool:NO forKey:self.keyFirstRun];
    [self.localDefaults synchronize];
}

#pragma mark - Public Saving Functions
-(void)saveBestScore:(int)score {
    [self.localDefaults setInteger:score forKey:self.keyBestScore];
    [self.localDefaults synchronize];
```

```
}


-(int)getBestScore {
    return (int)[self.localDefaults integerForKey:self.keyBestScore];
}
```

The firstLaunch method determines if we have run the game before by checking for a one of our keys. If it does not exist we populate the plist with some good defaults. For this game we only need to save the high score and a value that lets us know that the plist exists.

We don't want calling classes to manipulate the plist directly, or the values. We provide two methods that calling classes can use to save and get the value of score.

Whenever we do something with NSUserDefaults we need to call `synchronize` to actually write it to disk. If we didn't call `synchronize`, each time we try to get the value of the key we would get whatever was last saved in the plist. It is basically always a good idea to `synchronize` whenever you change any value from NSUserDefaults.

We need our `saveBestScore:` and `getBestScore` methods to be in the header so calling classes know how to use them. Select GameSettings.h and add the following lines for our method signatures between @interface and @end.

```
+(instancetype)sharedInstance;
-(void)saveBestScore:(int)score;
-(int)getBestScore;
```

Update Player

We have a great method to use to set the player's best score when the game ends already with `gameOver`.

In the Player group select Player.m and import the header for Gamettings just after importing the header for Smoke.

```
#import "GameSettings.h"
```

Scroll down to the end of the class to the gameOver method. Add the following lines to the end of the `gameOver` method.

```
if (self.taps > 0 && self.taps > [[GameSettings sharedInstance]
getBestScore]) {
     [[GameSettings sharedInstance] saveBestScore:self.taps];
}
```

If the `taps` are greater than 0 and `taps` are greater than the value returned from `getBestScore`, save it to disk as the new best score. Pretty simple, but effective.

Update SpriteNames.h

In the Utilities group, select SpriteNames.h. Just after adding the constant for PLAYBUTTON, add these two new constants for our PauseButton textures.

```
static NSString *const PAUSEBUTTON = @"PauseButton";
static NSString *const RESUMEBUTTON = @"ResumeButton";
```

PauseButton

Select the Buttons group. Right click and choose **New File…** Select **iOS > Source > Cocoa Touch Class** and click **Next.** Name it **PauseButton** and make it a subclass of **SKSpriteNode**, then click **Next** and **Create**.

Select PauseButton.m. Just after importing the header add our private class interface.

```
@interface PauseButton()

@property BOOL gamePaused;
@property SKTexture *pauseTexture;
@property SKTexture *resumeTexture;

@end
```

Between @implementation and @end, add the following code.

```
#pragma mark - Init
-(instancetype)init {
    if ((self = [super init])) {
        SKTexture *texture = [[GameTextures sharedInstance]
textureWithName:PAUSEBUTTON];
        self = [PauseButton spriteNodeWithTexture:texture];

        [self setup];
    }

    return self;
```

```
}

#pragma mark - Setup
-(void)setup {
    self.gamePaused = NO;

    self.pauseTexture = [[GameTextures sharedInstance]
textureWithName:PAUSEBUTTON];
    self.resumeTexture = [[GameTextures sharedInstance]
textureWithName:RESUMEBUTTON];

    self.anchorPoint = CGPointMake(1.0, 1.0);
    self.position = CGPointMake(ScreenSize().width, ScreenSize().height);
}

#pragma mark - Actions
-(void)tapped {
    self.gamePaused = !self.gamePaused;

    [self setTexture:(self.gamePaused ? self.resumeTexture :
self.pauseTexture)];
}

-(BOOL)getPaused {
    return self.gamePaused;
}
```

The main thing I want to point out is in the `setup` method. We are setting the `anchorPoint to` 1.0, 1.0, which is the top right corner. Remember that sprites are positioned using their `anchorPoint`. Then we can set the position to the width of the screen on the X axis and the height of the screen on the Y axis. This puts the top right corner of the button in the top right corner of the screen.

Lets add the signatures for tapped and getPaused to the header. Select PauseButton.h and add these two lines between @interface and @end.

```
-(void)tapped;
-(BOOL)getPaused;
```

Updating GameScene

At the top of the file just after importing the header for TimeBar, add the header for PauseButton.

```
#import “PauseButton.h"
```

In the @interface GameScene(), add this new property for our PauseButton just after the one for TimeBar.

```
@property PauseButton *pauseButton;
```

We need to pause and resume the game. This works best if all of the nodes that update, move, run actions or need to process physics are in a separate node. Pausing the scene directly doesn’t work as well as you might think it would. To do this we will just add an empty node to the scene, and add all of our nodes to this new empty node.

Just after our property for the pauseButton, add this new property we’ll call gameNode.

```
@property SKNode *gameNode;
```

We have a little refactoring to do. Change the entire setupScene method to this new code.

```
#pragma mark - Setup
-(void)setupScene {
    self.physicsWorld.contactDelegate = self;

    self.lastUpdateTime = 0.0;

    // All nodes are now a child of gameNode
    self.gameNode = [SKNode node];
    [self addChild:self.gameNode];

    // Clouds
    self.clouds = [[Clouds alloc] init];
    [self.gameNode addChild:self.clouds];
```

```
    // Birds
    self.birds = [[Birds alloc] init];
    [self.gameNode addChild:self.birds];

    // Forest
    SKSpriteNode *forest = [[GameTextures sharedInstance]
spriteWithName:FOREST];
    forest.anchorPoint = CGPointZero;
    forest.position = CGPointZero;
    [self.gameNode addChild:forest];

    // Stack Controller
    self.stackController = [[StackController alloc] init];
    [self.gameNode addChild:self.stackController];

    // Player
    self.player = [[Player alloc] init];
    [self.gameNode addChild:self.player];

    // Tutorial Button
    self.tutorialButton = [[TutorialButton alloc] init];
    [self.gameNode addChild:self.tutorialButton];

    // Play Button, not added to scene until switchToGameOver
    self.playButton = [[PlayButton alloc] init];

    // Score Label
    self.scoreLabel = [[ScoreLabel alloc] init];
    [self.gameNode addChild:self.scoreLabel];

    // Time Bar
    self.timeBar = [[TimeBar alloc] init];
    [self.gameNode addChild:self.timeBar];

    // Pause Button
    self.pauseButton = [[PauseButton alloc] init];
    [self.gameNode addChild:self.pauseButton];

    // Set the state to Waiting
    [self switchToWaiting];
}
```

It is basically the same thing we had, but now we are adding a gameNode to the scene and adding everything as a child of this new node.

Now we need a method that we can call that does the pausing. We want every node on screen to pause, as well as the background music.

Scroll down to the end of the class. Just before @end add this new method we'll call `pauseButtonPressed`.

```
#pragma mark - Pause Function
-(void)pauseButtonPressed {
    [self.pauseButton tapped];

    if (self.pauseButton.getPaused) {

        self.gameNode.paused = YES;

        [self switchToPaused];

        [OALSimpleAudio sharedInstance].bgPaused = YES;

    } else {

        self.gameNode.paused = NO;

        [self switchToResume];


        [OALSimpleAudio sharedInstance].bgPaused = NO;
    }
}
```

Our Player could touch anywhere on screen. We want to make sure that if they touch inside the bounds of the PauseButton that we interpret that as pausing the game.

Change the entire touchesBegan:withEvent: method to this new code.

```
#pragma mark - Touch Handling
-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

    UITouch *touch = [touches anyObject];
    CGPoint touchLocation = [touch locationInNode:self.scene];

    switch (self.state) {
        case Waiting:
            if ([self.pauseButton containsPoint:touchLocation]) {
                [self pauseButtonPressed];
            } else {
                [self switchToRunning];
            }

            break;

        case Running: {

            if ([self.pauseButton containsPoint:touchLocation]) {
                [self pauseButtonPressed];

            } else {

                if (touchLocation.x > ScreenSize().width / 2) {
```

```
                    [self.player chopRight];
                    [self.stackController moveStack];
                    [self.scoreLabel updateLabel:[self.player getTaps]];
                    [self addSecondsToTime];

                } else if (touchLocation.x < ScreenSize().width / 2) {
                    [self.player chopLeft];
                    [self.stackController moveStack];
                    [self.scoreLabel updateLabel:[self.player getTaps]];
                    [self addSecondsToTime];

                }

            }

            break;
        }

        case Paused:
            if ([self.pauseButton containsPoint:touchLocation]) {
                [self pauseButtonPressed];
            }

            break;

        case GameOver:
            if ([self.pauseButton containsPoint:touchLocation]) {
                [self pauseButtonPressed];
            } else if ([self.playButton containsPoint:touchLocation]) {
                [self resetGame];
            }

            break;
    }
}
```

Now in each state we check first if the touchLocation is in the bounds of the PauseButton and can handle that by calling our `pauseButtonPressed` method.

The last thing we need to do to make our pauseButton work as intended is do a little refactoring of update:. Change the entire update: method to this new code.

```
-(void)update:(NSTimeInterval)currentTime {
    // Calculate "delta"
    NSTimeInterval delta = currentTime - self.lastUpdateTime;
    self.lastUpdateTime = currentTime;

    switch (self.state) {
        case Waiting:
        case GameOver: {
            if (!self.gameNode.paused) {
```

```
                [self.clouds update:delta];
                [self.birds update:delta];
            }

            break;
        }
        case Running: {
            if (!self.gameNode.paused) {

                [self.clouds update:delta];
                [self.birds update:delta];

                self.timeLeft -= delta * 4;

                [self.timeBar updateTimeBar:self.timeLeft];

                if (self.timeLeft <= 0) {
                    [self switchToGameOver];
                }
            }

            break;
        }

        case Paused:
            break;
    }
}
```

We moved calculating `delta` outside the switch statement. Doing it the way we had it previously is fine if you don't need to pause the game. In some games there is no point to pausing as the amount of time takes someone to tap a pause button is sufficient to consistently lose. In our game we want this to work and we don't want the `delta` to lurch forward or lag behind during pause and resume states. Then in our switch statement, we check to see if `gameNode` is paused, if it is not we proceed with our update code.

Build and Run

Build and Run the game and play around with pausing and resuming the game.

Scoreboard

Before we add the Scoreboard class lets add the sprite name to the SpriteNames.h header.

In the Utilities group, select SpriteNames.h and add this new constant before #endif.

```
// Score Board
static NSString *const SCOREBOARD = @“ScoreBoard";
```

Right click the Classes folder and choose **New Group**, name the group **Scoreboard**. Right click the Scoreboard group and choose **New File…** Select **iOS > Source > Cocoa Touch Class** and click **Next**. Name it **Scoreboard** and make it a subclass of **SKSpriteNode**, then click **Next** and **Create**.

Select Scoreboard.m. Just after importing the header for Scoreboard, add the header for GameSettings.

```
#import "GameSettings.h"
```

Between @implementation and @end, type in this code.

```
#pragma mark - Init
-(instancetype)initWithScore:(int)score {

    if ((self = [super init])) {

        [self setupWithScore:score];
    }

    return self;
}

#pragma mark - Setup
-(void)setupWithScore:(int)score {

    self.position = CGPointMake(ScreenSize().width / 2, ScreenSize().height *
0.6);
```

```
    // Scoreboard Sprite
    SKSpriteNode *background = [[GameTextures sharedInstance]
spriteWithName:SCOREBOARD];
    [self addChild:background];

    // Glyph Font
    BMGlyphFont *font = [BMGlyphFont fontWithName:@"GameFont"];

    // Score Label
    BMGlyphLabel *currentScore = [BMGlyphLabel labelWithText:[NSString
stringWithFormat:@"%d", score] font:font];
    currentScore.position = CGPointMake(0, background.size.height * 0.15);
    currentScore.textJustify = BMGlyphJustifyLeft;

    // Best Score Label
    int highScore = [[GameSettings sharedInstance] getBestScore];
    BMGlyphLabel *bestScore = [BMGlyphLabel labelWithText:[NSString
stringWithFormat:@"%d", highScore] font:font];
    bestScore.position = CGPointMake(0, -background.size.height * 0.25);
    bestScore.textJustify = BMGlyphJustifyLeft;

    [background addChild:currentScore];
    [background addChild:bestScore];

    [self setScale:0];
}

-(void)animateIn {
    [self runAction:[SKAction scaleTo:1.1 duration:0.25] completion:^{
        [self runAction:[SKAction scaleTo:1.0 duration:0.25]];
    }];
}
```

As a reminder, we have to position child nodes around the parent nodes anchorPoint. In this case the anchorPoint for the Scoreboard is in the center (0.5, 0.5), so our labels are positioned using the center of the scoreboard background.

We need to make our setupWithScore: and animateIn methods public to calling classes. Select Scoreboard.h and add the signatures between @interface and @end.

```
-(instancetype)initWithScore:(int)score;
-(void)animateIn;
```

Update GameScene

The final thing we need to do for TimberJim is get the Scoreboard in the game.

In the Scenes group select GameScene.m. We need to import the header for Scoreboard. Just after importing the header for the PauseButton, add this line.

```
#import “Scoreboard.h"
```

We don't need the Scoreboard in the scene at all until we run the `displayGameOver` method. Scroll down to this method and change the entire thing to this new code.

```
-(void)displayGameOver {
    [self runAction:[SKAction waitForDuration:0.5] completion:^{
        [self.gameNode addChild:self.playButton];
        [self.playButton animateIn];

        Scoreboard *scoreboard = [[Scoreboard alloc] initWithScore:
[self.player getTaps]];
        [self.gameNode addChild:scoreboard];
        [scoreboard animateIn];
    }];
}
```

Now in our block we are also adding an instance of `scoreboard`. We pass in the current player taps with the `getTaps` method from the Player class. This lets us display a scoreboard with both the current and best scores each time.

Build and Run

Congratulations! You've done a lot of hard work, but the project is done. Build and Run and enjoy your creation.

Conclusion

I want to congratulate you on finishing what you started with this book. I hope you learned a lot and had fun building TimberJim.

In this book we covered some practices that I hope you find useful. You can take some of these things too far, but I think we got the right balance of object oriented design, encapsulation, performance and being easy to understand.

In this book we used modern Objective-C styling throughout. One of the things I really enjoy about the language is the verbosity mixed with some good naming conventions that makes the code quite readable.

If you got stuck on any steps in this chapter check your work against the source code you downloaded from the book at **Project > Chapter 7 > Finish**.

I want to close by wishing you luck with your journey and thanking you for reading my book.

- Jeremy Novak