# 6
# Adding Game Controllers

In this chapter, we are going to discuss a new feature available in iOS 7—game controllers. Apple allowed video streaming from iOS devices to TV a long time ago, but the lack of hardware controller support left many players underwhelmed. Game controllers allow you to focus on the gameplay rather than controls, and if you are using your iPhone or iPad as a controller, you don't receive any physical feedback. This is one of the major drawbacks of touchscreen gaming—you won't be sure whether you tapped the right thing unless you are looking at the screen.

Some companies started to work on this problem, and different controllers such as iCade appeared. But they had one fundamental flaw in them—they had to be supported by the developers in order to work with the game. Another issue was connectivity. Some of them wanted to work over Bluetooth while others were connected directly, and each of these methods had their downsides.

There were very few games that supported hardware controllers. Developers didn't feel like supporting these controllers as the install base was far too small, which would affect the sales, thus increasing the time spent to support them.

Another problem with such a controller is the lack of uniformity. Everybody makes their own interface, and developers need to learn many different frameworks and APIs in order to make these things work.

Another issue is the availability and spread of such controllers—the company might go out of business, it may decide to not make more controllers, or their supply might run out.

That's why supporting third-party controllers didn't work too well.

## Native game controllers

Support for native game controllers first appeared in iOS 7. What are the advantages of native controllers? Here are some of them:

- **A universal API for every controller out there**: You will need to write the code only once to support a myriad of different controllers by different manufacturers.
- **An easy-to-use API written and supported by Apple engineers**: You know that your code will work on the latest versions of iOS and will generally be supported throughout your application's lifetime. Bugs will be removed and everyone will be happy.
- **A standard layout for buttons and joysticks**: You know what buttons each controller will have and where they are expected to be.

Now that we have found out why game controllers are useful, let's incorporate them into our project.

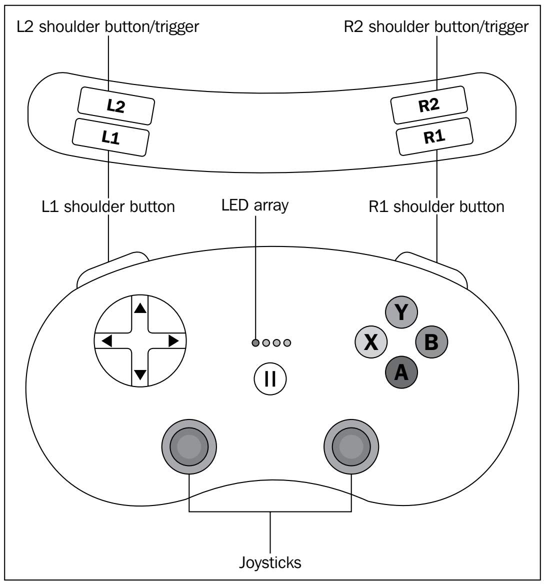
## Game controller basics

The Game Controller API is based on the Game Controller framework. Everything related to controllers happens there.

There are three types of controllers that are supported by the framework:

- **Standard form-fitting controller**: This is a controller in which the iOS device resides. This controller has a direction pad, a pause button, four buttons, and two shoulder buttons.
- **Extended form-fitting controller**: This is the same as the previous one, but it can also have up to two sticks and two more shoulder buttons.
- **Extended wireless controller**: This is the same as the previous one, but it works wirelessly and is an external device without holding your iOS device in it.

The following figure shows the various types of controllers:



Different controller types

Apple does not necessarily require a game controller to play a game—there should always be a way to play your game via touchscreen. Game controllers enhance your user's experience, but do not force them to do so.

Wired controllers are automatically discovered, but their wireless counterparts require a one-time setup and can be used freely after that. A controller and a device will automatically connect to each other once they are powered on. There is no difference from the code's perspective for wired or wireless controllers.

Each game controller is represented with the `GCController` object, which you can use in your game. You receive notifications when a new one is being connected or disconnected.

Each game controller has a profile that identifies it as a controller type. Currently, there are two profiles—`gamepad` and `extendedGamepad`.

You can check `GCController` for the properties of `gamepad` and `extendedGamepad`. If they exist, it means that the controller is one of those types. Otherwise, it will return `nil`.

There are two ways to receive data from the controller, which are as follows:

- By reading the properties directly
- By assigning a handler (block) to each button to be executed on button press

The following is a sample method to read inputs if you want to read them at arbitrary times:

```
- (void) readControlInputs
{
    GCGamepad *gamepad = self.myController.gamepad;
    if (gamepad.buttonA.isPressed)
        [self jumpCharacter];
    if (gamepad.buttonB.isPressed)
        [self shootApple];
    [self moveBy: profile.leftThumbstick.xAxis.value];
}
```

Here we are checking if the buttons are pressed, and if they are, we execute some methods such as jumping or shooting. On the last line, we read the value of the x axis on the left thumbstick to move our character left or right.

The second way to set handlers is as follows:

```
- (void) setupController
{
    GCExtendedGamepad *gamepad = self.myController.extendedGamepad;
    gamepad.buttonA.valueChangedHandler = ^(GCControllerButtonInput
*button, float value, BOOL pressed)
    {
        if (pressed)
            [self shoot];
    };
    gamepad.buttonX.valueChangedHandler = ^(GCControllerButtonInput
*button, float value, BOOL pressed)
    {
        if (pressed)
            [self openInventory];
    };
}
```

What we do here is set the block to each button (in this case, two buttons) to execute a piece of code each time a button is pressed. One button shoots and the second button opens the inventory. These handlers also get called when the player releases the button, so you should check for the `pressed` variable.

Picking the positions of the thumbstick or the directional pad is a little more complex. Usually, you get x and y values and you can work from there:

```
       GCControllerDirectionPadValueChangedHandler dpadMoveHandler =
^(GCControllerDirectionPad *dpad, float xValue, float yValue) {
           if (yValue > 0)
           {
               player.accelerating = YES;
           } else {
               player.accelerating = NO;
           }
       };
```

In this block, we check the y value, and if it is higher then `0` (neutral), we start jumping.

Each controller also has a pause button, and you should implement the functionality of pausing your game when the user presses the pause button. The controller holds a block for the pause button that gets triggered each time it is pressed.

## Using a controller in our game

As our game is not too complicated, we use only one button. We will map the same thing to a few buttons so that the players can use any one that they like.

The first thing that we need to do is import the `GameController` framework. In order to do this, add the following line to `ERGMyScene.h` at the beginning of the file:

```
@import GameController;
```

The preceding line of code adds game controller headers and links our project with the game controller library while it is compiling. After this, we will need two new properties in the same file:

```
@property (strong, nonatomic) ERGPlayer *player;
@property (strong, nonatomic) SKLabelNode *pauseLabel;
```

We will need a way to access our `player` object from the code that sets up the game controllers, so we add the first property in the preceding code to hold it.

The second one is the label for the paused state of our game. We will show or hide it based on the current game state.

Right after the line where you add `player` as a child in the `initWithSize:` method in `ERGMyScene.m`, add the following line of code:

```
self.player = player;
```

Here, we set our property to hold a pointer to the `player` object. At the end of `initWithSize:`, we will add the code to handle controller discovery and setup, and create a `pauseLabel`, as shown in the following code:

```
        [self configureGameControllers];

        self.pauseLabel = [[SKLabelNode alloc] initWithFontNamed:@"Ch
alkduster"];
        self.pauseLabel.fontSize = 55;
        self.pauseLabel.color = [UIColor whiteColor];
        self.pauseLabel.position = CGPointMake(self.size.width / 2,
self.size.height / 2);
        self.pauseLabel.zPosition = 110;
        [self addChild:self.pauseLabel];
        self.pauseLabel.text = @"Pause";
        self.pauseLabel.hidden = YES;
```

Currently, we have not yet written the `configureGameControllers` method, but we will do this shortly. The code to create a label should be fairly familiar to you—we create a label node, set its properties, and set its position to the center of the screen and hide it, as we don't want it to be visible in the unpaused state of the game.

> Each scene has a `paused` property. When it is set to `YES`, the scene stops evaluating actions for all descendants. But the `update` method is still getting called! So, you may experience a very strange behavior. If we pause our game now, all animations will stop, but the background will still scroll.

The next thing that we need to do is stop updates if the scene is paused. To do that, change the `update` method in `ERGMyScene.m`—add a check for the paused property at the start of the `update` method and add the following code:

```
    if (self.paused) {
        return;
    }
```

This way, the `update` method will do something only if the scene is not paused, which is the intended behavior. If the scene is paused, it will skip all that code and drop out.

After this, we need the code to find any controllers and use them in our game. To do this, add the following code to `ERGMyScene.m`:

```
- (void)configureGameControllers {

    [[NSNotificationCenter defaultCenter] addObserver:self selector:@
selector(gameControllerDidConnect:) name:GCControllerDidConnectNotific
ation object:nil];
```

```
    [[NSNotificationCenter defaultCenter] addObserver:self selector:@
selector(gameControllerDidDisconnect:) name:GCControllerDidDisconnectN
otification object:nil];

    [self configureConnectedGameControllers];

    [GCController
startWirelessControllerDiscoveryWithCompletionHandler:^{

        // we don't use any code here since when new controllers are
found we will get notifications
    }];
}
```

The first thing that we do is subscribe for notifications related to the game controller. They are sent when a new controller is connected or disconnected from the device. Next, we call another method to configure the controller.

Also, notice that if you turn off Bluetooth on your device, the game will crash on the `configureGameControllers` method. The same thing will happen if you test on a simulator. Adding error-checking unnecessarily complicates the code in our example, so comment out `[self configureGameControllers];` in the `initWithSize:` method if you don't intend to use it on a Bluetooth-enabled device.

We also start searching for wireless controllers. We don't need any code inside the completion handler as we will be getting notifications when controllers connect or disconnect from the device. Let's get to the next method:

```
- (void)configureConnectedGameControllers {

    for (GCController *controller in [GCController controllers]) {
        [self setupController:controller forPlayer:self.player];
    }
}
```

This is a simple `for` loop, where we will go through all the available controllers and run a common setup code for each of them. If you are making a more complex or multiplayer game, you might want to differentiate the controllers here. There is a handy `playerIndex` property on the controller that may prove useful for you, but we don't use it here.

Most of the logic happens in the following method, where we set up actual button handlers:

```
- (void)setupController:(GCController *)controller
forPlayer:(ERGPlayer *)player
{
    GCControllerDirectionPadValueChangedHandler dpadMoveHandler =
^(GCControllerDirectionPad *dpad, float xValue, float yValue) {
        if (yValue > 0)
        {
            player.accelerating = YES;
        } else {
            player.accelerating = NO;
        }
    };
    if (controller.extendedGamepad) {
        controller.extendedGamepad.leftThumbstick.valueChangedHandler
= dpadMoveHandler;
    }
    if (controller.gamepad.dpad) {
        controller.gamepad.dpad.valueChangedHandler = dpadMoveHandler;
    }

    GCControllerButtonValueChangedHandler jumpButtonHandler =
^(GCControllerButtonInput *button, float value, BOOL pressed) {
        if (pressed) {
            player.accelerating = YES;
        } else {
            player.accelerating = NO;
        }
    };

    if (controller.gamepad) {
        controller.gamepad.buttonA.valueChangedHandler =
jumpButtonHandler;
        controller.gamepad.buttonB.valueChangedHandler =
jumpButtonHandler;
    }
    if (controller.extendedGamepad) {
        controller.extendedGamepad.buttonA.valueChangedHandler =
jumpButtonHandler;
```

```
        controller.extendedGamepad.buttonB.valueChangedHandler =
jumpButtonHandler;
    }

    controller.controllerPausedHandler = ^(GCController *controller) {
        [self togglePause];
    };

}
```

At the beginning of this method, we create a handler for the directional pad and thumbsticks—if the player presses the up button on the pad or moves the thumbstick up, we will have the Y value going up from zero. Thus, we start jumping by setting the accelerating property to `YES`.

After that, we check if the controller has an `extendedGamepad` profile, and if it does, we attach this handler to the thumbstick. If it is a regular gamepad, we attach the handler to the directional pad.

The same procedure is repeated when we declare a handler for a button—it checks if the button is pressed and our character jumps; if it is, we set the A and B button handlers to this handler.

The last handler that can be found in this method is the pause handler. It will get called when a player presses the pause button. The following is the code for this method:

```
- (void) togglePause
{
    if (self.paused) {
        self.pauseLabel.hidden = YES;
        self.paused = NO;
    } else {
        self.pauseLabel.hidden = NO;
        self.paused = YES;
    }
}
```

Here we check if the scene is paused, and if it wasn't paused, we pause it and show the **Pause** label. If it was paused, we continue the scene and hide the label.

## Handling controller notifications

Another thing that we would want to handle is new controllers. We need to do something when a controller connects or disconnects.

We can create a complex interface for that, but for our project, simple alerts will do.

The easy part is that when the controller disconnects, we won't get any new input, so we don't need to do anything; we just show an alert and are done with it:

```
- (void)gameControllerDidDisconnect:(NSNotification *)notification {

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Warning"
                                                   message:@"Game
controller has disconnected."
                                                  delegate:nil
                                         cancelButtonTitle:@"Ok"
                                         otherButtonTitles:nil, nil];
    [alert show];
}
```

We create the `alertView` object and show it. The user cancels it and goes on with their game with touch controls.

But if we connect the controller, we need to ask the user if they actually want to use it in the game. In order to do that, we need to set our scene as a delegate to `UIAlertView` and change the `@interface` line in `ERGMyScene.h` to the following:

```
@interface ERGMyScene : SKScene <SKPhysicsContactDelegate,
UIAlertViewDelegate>
```

This means that we conform to `UIAlertViewDelegate` and that we will implement certain methods to be executed after the user taps a button in the alert view.

When a new controller connects, we will get a notification and the following method will get called, which should be added to the `ERGMyScene.m` file:

```
- (void)gameControllerDidConnect:(NSNotification *)notification {

    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Warning"
                                                   message:@"Game
controller connected. Do you want to use it?"
                                                  delegate:self
                                         cancelButtonTitle:@"No"
                                         otherButtonTitles:@"Yes",
nil];
    [alert show];
}
```

We create the alert view as before, but here we set its delegate to our scene. The following is the `delegate` method:

```
-(void)alertView:(UIAlertView *)alertView
clickedButtonAtIndex:(NSInteger)buttonIndex
{
    if (buttonIndex == 1) {
        [self configureConnectedGameControllers];
    }
}
```

When you press a button in the alert view, and if it has the delegate set, it will call this method on its delegate. The button's index starts at zero, and zero is the cancel button that is always present. If there are any other buttons, they will have the subsequent indexes.

Usually, if we have multiple alert views, we have to understand which alert view called what, and what the button at said index should do.

But in our situation, we have only one alert view that has more than one button, and we know what it is, so we don't actually need to check anything here. If the user tapped the **Yes** button on the alert view, we will configure all the connected controllers. If they click on **No**, which is at index `0`, nothing will happen.

## Adding sound and music

Our game feels a bit bland without sound effects, so we are going to add some. We will add background music and sound effects.

Sprite Kit is mostly a graphics library, and sound support is limited, but there are ways to add music to our game.

The first thing that we need to do is add music files to the project. To do this, drag-and-drop `Kick_Shock.mp3`, `shieldCharged.wav`, and `shieldSmashed.wav` to our project in Xcode. Select **Copy items into destination group's folder** and make sure that **Add to targets** has your project selected.

In order to play background music, we will need to add the `AVFoundation` library to our project.

The `AVFoundation` library is a collection of classes that allows for audio and video playback on iOS devices. To add `AVFoundation` to our project, add the `@import AVFoundation;` line to `ERGMyScene.h`. We will also need a new property to play music. To do this, add the following property to `ERGMyScene.h`:

```
@property (strong, nonatomic) AVAudioPlayer* musicPlayer;
```

After this, we need to create a new method to set up the playback sound. Add this new method to `ERGMyScene.m`:

```
- (void) setupMusic
{
    NSString *musicPath = [[NSBundle mainBundle]
pathForResource:@"Kick_Shock" ofType:@"mp3"];
    self.musicPlayer = [[AVAudioPlayer alloc]
initWithContentsOfURL:[NSURL fileURLWithPath:musicPath] error:NULL];
    self.musicPlayer.numberOfLoops = -1;
    self.musicPlayer.volume = 1.0;
    [self.musicPlayer play];
}
```

In the first line, we created a path to our background music file. Next, we initialized `AVAudioPlayer`—a class that plays music from `AVFoundation`. Then, we set `numberOfLoops`, the property that says how many times the music should be repeated.

Next up is volume; it is set to `1.0` and sends play messages to the player.

After you have done this, call `[self setupMusic]` at the end of the `initWithSize:` method of `ERGMyScene.m` but before the `return` statement.

In a few lines, we managed to add background music to our game. But we will want to stop music playback if the **Game Over** screen is called. To do this, locate the `- (void) gameOver` method in `ERGMyScene.m`, and add the `[self.musicPlayer stop];` line at the beginning of it. This will stop music playback.

We will also add short playback sounds to our game. We will have them for shield charging and depleting. We won't use `AVFoundation` for this, as there is a simple sound playback action in Sprite Kit. It is used in this way (don't add this code to the project):

```
SKAction *soundAction = [SKAction playSoundFileNamed:@"name.mp3"
waitForCompletion:YES];
[self runAction:soundAction];
```

It works as any other action. You create an action and run it against the node. Wait for completion means that the action is considered done as soon as it runs if `waitForCompletion` is `NO`, and the action is considered running if you pass `YES` to it.

Why haven't we used this action to play background music? It is used for playing short sounds, as somehow removing this action does nothing. It is unclear if it is a bug or a feature, but it is present in the previous version of Sprite Kit. The music just keeps on playing.

As we handle shield effects in `Player.m`, it is a good place to add playback sound. We will change the `setShielded:` method by adding music actions to it. Replace your method with the following:

```
- (void) setShielded:(BOOL)shielded
{
    if (shielded) {
        if (![self.shield actionForKey:@"shieldOn"]) {
            [self.shield runAction:[SKAction
repeatActionForever:[SKAction animateWithTextures:self.shieldOnFrames
timePerFrame:0.1 resize:YES restore:NO]] withKey:@"shieldOn"];
            SKAction *musicAction = [SKAction playSoundFileNamed:@"shi
eldCharged.wav" waitForCompletion:NO];
            [self runAction:musicAction];
        }
    } else if (_shielded) {
        [self blinkRed];
        [self.shield removeActionForKey:@"shieldOn"];
        [self.shield runAction:[SKAction animateWithTextures:self.
shieldOffFrames timePerFrame:0.15 resize:YES restore:NO]
withKey:@"shieldOff"];
        SKAction *musicAction = [SKAction playSoundFileNamed:@"shieldS
mashed.wav" waitForCompletion:NO];
        [self runAction:musicAction];
    }
    _shielded = shielded;
}
```

As before, it handled the shield logic, but there is a little bit of playback sound added to it.

Build and run the project; it should run and you should have background music and sound effects fully working.

There is one more issue with our code. If you receive a call or exit from the application, it will crash. This happens due to the fact that we need to close `AVAudioSession` before our application resigns active. This class is used underneath `SKAction` to play sounds.

To fix this, add the `@import AVFoundation;` statement and add the following code to your `AppDelegate.m` file:

```
- (void)applicationWillResignActive:(UIApplication *)application
{
    // prevent audio crash
    [[AVAudioSession sharedInstance] setActive:NO error:nil];
}

- (void)applicationWillEnterForeground:(UIApplication *)application
{
    // resume audio
    [[AVAudioSession sharedInstance] setActive:YES error:nil];
}
```

This code handles `AVAudioSession` and prevents your application from terminating.

## Summary

In this chapter, we have learned what advantages hardware game controllers have and how to work with them. Adding game controller support to your game can certainly provide a better user experience for your players. Some people prefer playing on actual controllers than on touch-screen devices, since the screen is always flat and doesn't provide feedback to touches. A player can use the controller without looking at it, which completely changes the experience of playing if he/she is playing on a TV. One might argue that with game controllers, Apple is making a move into the console gaming territory, and a huge library of games on the iTunes App Store playable on your TV or iPad with a separate hardware controller is opening another new way to experience iOS games. Adding game controller support to your game is not difficult, and if you are not already supporting them, you should really consider doing that. We have also learned how to add background music and sound effects to your game. In the next chapter, we will discuss how you can add your game to the App Store, what provisioning profiles are, and how to prepare your application bundle.

In the next chapter, we will discuss how to prepare your application for the App Store. We will also learn about provisioning profiles and procedures to have your application approved by Apple.