12
Standing Out – Integrating Game Center and In-App Purchases

Welcome to the final chapter of this book. In this chapter, we're going to review a few topics that are not directly related to gameplay, but can definitely improve your game. We are going to create a new simple game using everything we've learned in the previous chapters, and we're going to add Game Center and In-App purchases to this game.

In this chapter, we will discuss the following topics:

- Creating an application in iTunes Connect
- Enabling Game Center
- Creating leaderboards and achievements
- Submitting scores and achievements
- Making In-App purchases
- Reviewing a list of useful resources

Creating a sample game

In order to test Game Center and In-App purchases in the game, we need to create an application in iTunes Connect.

> This means that you need to be a participant of the Apple iOS Developer program. So if you have not yet registered for the Apple iOS Developer program, unfortunately, you won't be able to test the code listed in this chapter.

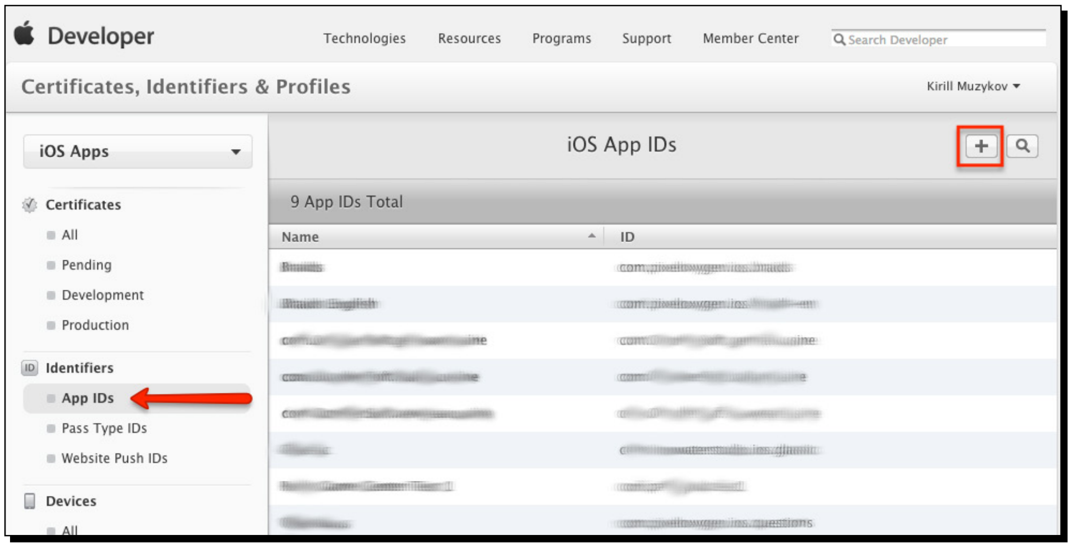
We could go on and register an app for the Cocohunt game that we created in the previous chapters, and add Game Center and In-App purchases to it. However, this would have caused a lot of time spent on housekeeping and supporting the old code.

Instead, we're going to create a simple new game so that we can concentrate on new topics.

Time for action – creating an application in iTunes Connect

In this recipe, we are going to create an **App ID** and an app in iTunes Connect using its bundle ID. If you're familiar with this, please skip to the next section, *Time for action – creating a game skeleton*.

1. First, we need to create an App ID. To do this, click on `https://developer.apple.com/account/ios/identifiers/bundle/bundleList.action` and log in to iOS Dev Center, as shown in the following screenshot:

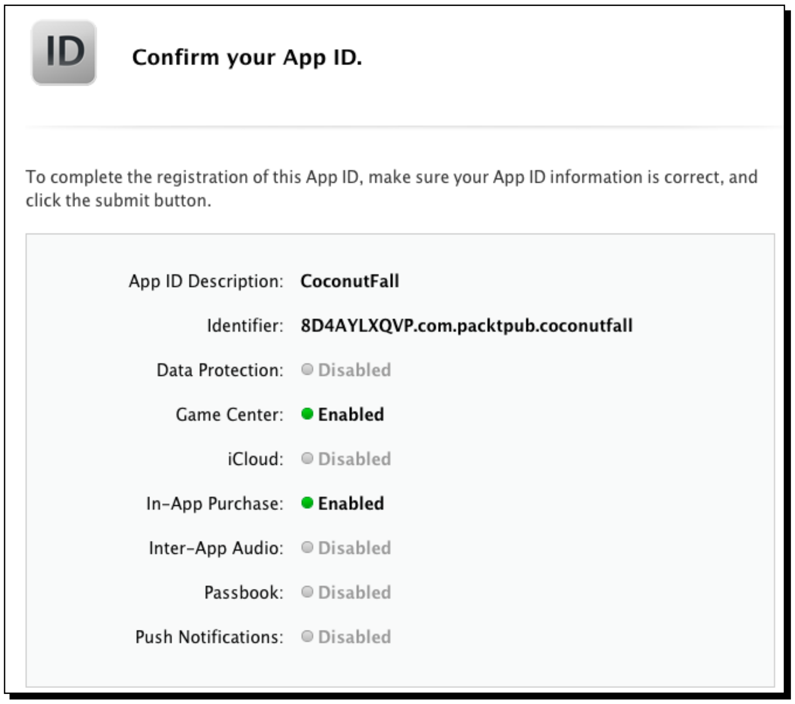


If the link is not working because Apple changed the URL, you can go to `https://developer.apple.com`, find iOS Dev Center, and search for a page where you can add an App ID.

2. Click on the **+** button to add a new App ID. You can enter any description in the **App ID Description** field; in my case, I've entered `CoconutFall`.
3. Then, you need to make sure the **Explicit App ID** option is selected, and enter a unique **Bundle ID** identifier. The easiest way to do this is to use a reverse domain name notation. In my case, this was `com.packtpub.coconutfall`, but you need to make up your own.

The bundle ID should be unique across the App Store. The easiest way to do this is to use a reverse domain name notation; however, the domain name doesn't have to exist or even work. You can use a combination of other data that has a high chance of being unique. For example, you might use your full name, name of the game, and a few random words. In any case, Apple won't allow you to enter an already taken bundle ID.

4. Click on the **Continue** button at the bottom. Make sure everything is correct and then click on the **Submit** button. Refer to the following screenshot:



Confirm your App ID.

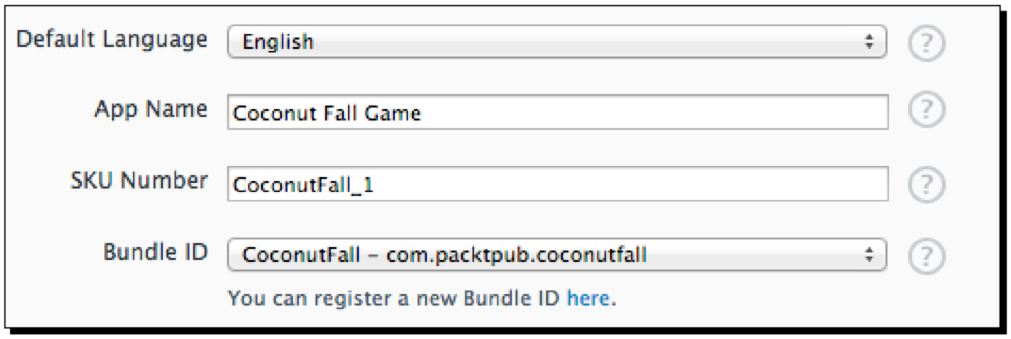
To complete the registration of this App ID, make sure your App ID information is correct, and click the submit button.

App ID Description:	CoconutFall
Identifier:	8D4AYLXQVP.com.packtpub.coconutfall
Data Protection:	● Disabled
Game Center:	● Enabled
iCloud:	● Disabled
In-App Purchase:	● Enabled
Inter-App Audio:	● Disabled
Passbook:	● Disabled
Push Notifications:	● Disabled

5. Now, when we have a bundle ID for the new app, we can go to iTunes Connect. Just click on `https://itunesconnect.apple.com`.
6. In iTunes Connect, click on **Manage Your Apps**, and then click on **Add New App** in the top-left corner.
7. On the **App Information** page, think of an **App Name** and **SKU Number**.

> The **Stock Keeping Unit** (**SKU**) number is just a value that will allow you to track your app. It will appear in reports and in several other places. You can just enter the name of the game and a suffix (for example, `CoconutFall_1`).

8. Now, select the **Bundle ID** that you've previously created and click on **Continue**. Refer to the following screenshot:



Default Language	English
App Name	Coconut Fall Game
SKU Number	CoconutFall_1
Bundle ID	CoconutFall – com.packtpub.coconutfall

You can register a new Bundle ID here.

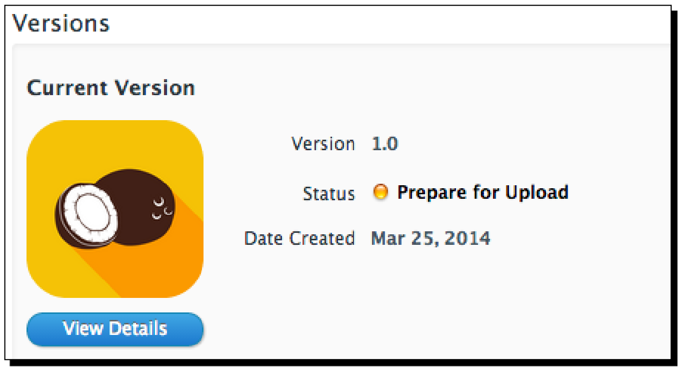
> If your bundle ID doesn't appear in the **Bundle ID** dropdown, you just need to wait for some time until the changes you've made in iOS Dev Center are propagated to iTunes Connect. You rarely have to wait for more than 15 minutes, and in most cases, it is already there, but in some cases, you might need to wait an hour or two.

9. On the next page, select any **Price Tier** and click on **Continue**. I've chosen this to be a **Free** app because this is only a test app, but you can set any price.

10. We need to provide a lot of information on the next page. It's a good thing that right now we can enter anything just to pass this page. All of this can be changed later. Just click on **Save** and see what is missing in the error message, then enter some test information until iTunes Connect is happy.

> You can find a stub **Large App Icon** and **Screenshots** files that you'll need to add at this stage in the `Chapter_12/Assets/AppStore` folder in the book's supporting files (which can be downloaded from `www.packtpub.com/support`).

11. After successfully submitting your app information, you should see it in the **Prepare for Upload** status:



Version 1.0

Status Prepare for Upload

Date Created Mar 25, 2014

What just happened?

That's it for now. We've created an app. With a few settings changed, we will be able to use Game Center and In-App purchase services.

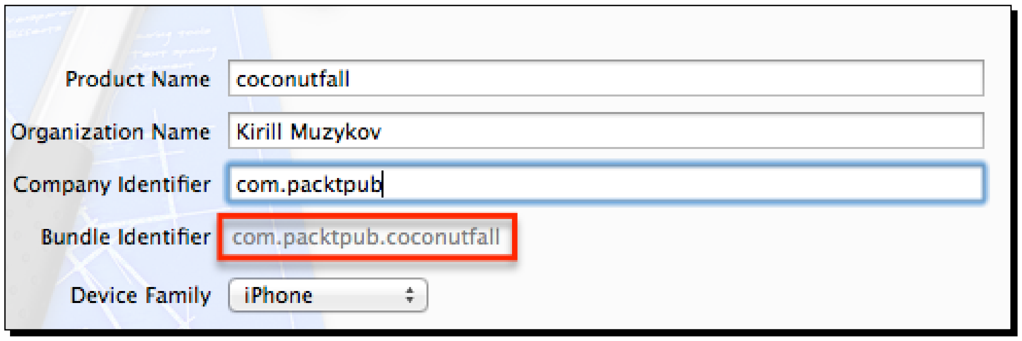
However, to use services in the game, we need a game, don't we? It is time to create a very simple game that we'll use to demonstrate the features of Game Center and In-App purchases.

> If you don't want to create this game from scratch, you can take the completed game project from the `Chapter_12/CoconutFall_Starter` folder and skip to the *Integrating Game Center* section. Starting from there, you will add Game Center and In-App purchases to this game. However, I would suggest that you spend a few minutes to at least copy and paste the code of the following sections and have a better understanding of the game structure.

Time for action – creating a game skeleton

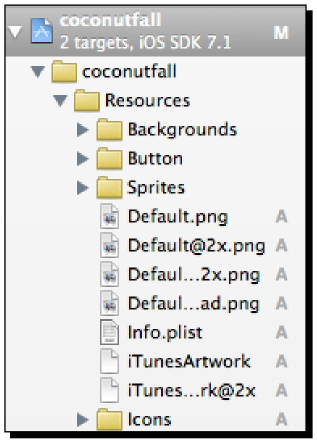
Let's start by creating a game skeleton. The game will consist of three scenes: `MenuScene`, `GameScene`, and `ShopScene`. In this part of the chapter, we're going to create a new Cocos2D project and add all the required scenes and navigation between them. In the next part, we're going to implement the gameplay process in the game scene.

1. Open Xcode and create a new project by navigating to **File** | **New** | **Project**, select the **cocos2d v3.x** section on the left-hand side, and select the project type as **cocos2d iOS**. Then, click on the **Next** button.
2. In the next window, you need to enter **Product Name** and **Company Identifier** so that the resulting **Bundle Identifier** matches with the bundle ID of the app that we created in iTunes Connect. This is required as the game will be recognized using this bundle ID. In my case, the bundle ID was `com.packtpub.coconutfall`; this way, I've used `coconutfall` as **Product Name** and `com.packtpub` as **Company Identifier**, as shown in the following screenshot:

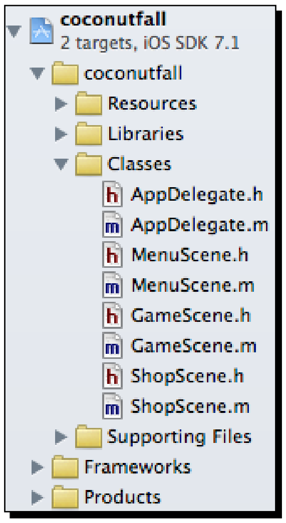


Product Name	coconutfall
Organization Name	Kirill Muzykov
Company Identifier	com.packtpub
Bundle Identifier	com.packtpub.coconutfall
Device Family	iPhone

3. Click on the **Next** button and choose a folder to save the new project.
4. Now, let's add all the images that we will use in this game. Open the `Chapter_12/Assets/CoconutFallResources` folder and drag all the three folders from there to the `Resources` group in Xcode, as shown in the following screenshot:



5. Create three new scenes: `MenuScene`, `GameScene`, and `ShopScene`. Make sure you make them subclasses of the `CCScene` class. Also, it is a good idea to place them in the `Classes` group just to keep things organized.
6. Remove the `HelloWorldScene` and `IntroScene` files from the `Classes` group since we don't need them. Here is what you should get in the end:



7. Open the `AppDelegate.m` file and remove the following `#import` directives, since they will be causing errors:

```
#import "IntroScene.h"
#import "HelloWorldScene.h"
```

8. Instead, import the `MenuScene.h` header:

```
#import "MenuScene.h"
```

9. Find the `startScene:` method and change it to return a `MenuScene` class instead of the `IntroScene` class:

```
-(CCScene *)startScene
{
   return [[MenuScene alloc] init];
}
```

10. Open the `MenuScene.m` file and import the following headers:

```
#import "cocos2d.h"
#import "cocos2d-ui.h"
#import "GameScene.h"
#import "ShopScene.h"
```

11. Then, add the following `#define` statements that are the buttons that the menu scene will have:

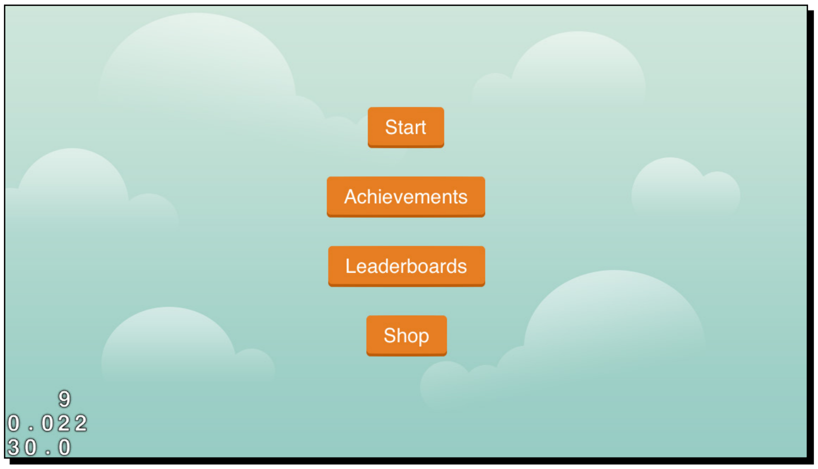
```
#define kButtonStart        @"Start"
#define kButtonAchievements @"Achievements"
#define kButtonLeaderboards @"Leaderboards"
#define kButtonShop         @"Shop"
```

12. After that, add the following methods to create a background scene and buttons, and handle buttons touch:

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

    return self;
}

-(void)addBackground
{
    CCSprite *bg =
      [CCSprite spriteWithImageNamed:@"menu_bg.png"];
    bg.positionType = CCPositionTypeNormalized;
    bg.position = ccp(0.5f, 0.5f);
    [self addChild:bg];
}

-(void)addButtons
{
    CCLayoutBox *buttonsLayout = [CCLayoutBox node];
    buttonsLayout.spacing = 20.0f;
    buttonsLayout.direction = CCLayoutBoxDirectionVertical;

    NSArray *buttons = @[kButtonStart,
                         kButtonAchievements,
                         kButtonLeaderboards,
                         kButtonShop];

    CCSpriteFrame *btnNormal =
      [CCSpriteFrame frameWithImageNamed:@"btn_9slice.png"];
    CCSpriteFrame *btnPressed =
      [CCSpriteFrame
        frameWithImageNamed:@"btn_9slice_pressed.png"];

    for (NSString *btnName in
          [buttons reverseObjectEnumerator])
    {
        CCButton *button = [CCButton buttonWithTitle:btnName
                                         spriteFrame:btnNormal
                              highlightedSpriteFrame:btnPressed
                                 disabledSpriteFrame:nil];
        button.name = btnName;
        button.horizontalPadding = 12.0f;
        button.verticalPadding = 4.0f;

        [button setTarget:self selector:@selector(onBtnTap:)];

        [buttonsLayout addChild:button];
    }

    [buttonsLayout layout];

    buttonsLayout.anchorPoint = ccp(0.5f, 0.5f);
    buttonsLayout.positionType = CCPositionTypeNormalized;
    buttonsLayout.position = ccp(0.5f, 0.5f);
    [self addChild:buttonsLayout];
}

-(void)onBtnTap:(CCButton *)btn
{
    if ([btn.name isEqualToString:kButtonStart])
    {
        [[CCDirector sharedDirector]
          replaceScene:[GameScene node]];
    }
    else if ([btn.name isEqualToString:kButtonAchievements])
    {
        //Nothing to do yet
    }
```

```
            else if ([btn.name isEqualToString:kButtonLeaderboards])
            {
                //Nothing to do yet
            }
            else if ([btn.name isEqualToString:kButtonShop])
            {
                [[CCDirector sharedDirector]
                  replaceScene:[ShopScene node]];
            }
            else
            {
                CCLOG(@"Unknown button:  %@", btn.name);
            }
        }
```

13. Build and run the game. You should see a scene with the following menu. You can even tap the **Start** and the **Shop** buttons, but currently they lead to empty scenes and you won't be able to return back.



What just happened?

Everything that we've done in this part should already be familiar to you from previous chapters. So let's just briefly review the code in the `MenuScene` class.

In the `init` method, we simply called the `addBackground` and `addButtons` methods that add a background image and create a menu using the `CCLayoutBox` class.

Note that we assigned the name of the button to the `name` property. In the `onBtnTap:`, we used the `name` property value to understand which button was pressed.

If the **Start** or **Shop** button is pressed, we simply show the corresponding scene.

The only important and new part here is creating the app with the bundle ID corresponding to the app we created in iTunes Connect. This will allow us to test Game Center and In-App purchases.

You don't need to create a new application from scratch to set the bundle ID. You can change it later in the project settings. However, since we decided to create a simpler game to demonstrate new features anyway, we just set it to the required bundle ID right from the start.

Time for action – implementing GameScene

In the game that we're going to create, coconuts will fall from the sky and the player will need to tap them before they fall. This will be just enough to demonstrate the features of Game Center while still having a very simple game. Perform the following steps:

1. Create a new class called `Coconut` and make it a subclass of the `CCSprite` class.
2. Open the `Coconut.h` file and replace its contents with the following code:

```
#import "CCSprite.h"

@protocol CoconutDelegate <NSObject>

-(void)coconutRemovedAt:(CGPoint)position;

-(void)fallenOffScreenAt:(CGPoint)position;

@end

@interface Coconut : CCSprite

@property (nonatomic, weak) id<CoconutDelegate> delegate;

@end
```

3. Then, open the `Coconut.m` file and replace its contents with the following code:

```
#import "Coconut.h"
#import "cocos2d.h"

@implementation Coconut

-(instancetype)init
{
    if (self = [super initWithImageNamed:@"coconut.png"])
    {
        self.userInteractionEnabled = YES;

        float coconutX = clampf(CCRANDOM_0_1(), 0.05f, 0.95f);
        self.positionType = CCPositionTypeNormalized;
        self.position = ccp(coconutX, 1.1f);
    }

    return self;
}

-(void)onEnter
{
    [super onEnter];

    float coconutSpeed = 3.0f + CCRANDOM_0_1() * 2.0f;
    CCActionMoveTo *moveDown =
      [CCActionMoveTo
        actionWithDuration:coconutSpeed
                  position:ccp(self.position.x, -0.1f)];
    CCActionEaseIn *moveDownEased =
      [CCActionEaseIn actionWithAction:moveDown rate:2.0f];
    CCActionCallFunc *notify =
      [CCActionCallFunc actionWithTarget:self
                        selector:@selector(fallenOffScreen)];
    CCActionSequence *fallDownAndNotify =
      [CCActionSequence actions:moveDownEased, notify, nil];
    [self runAction:fallDownAndNotify];
}

-(void)touchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
    [self.delegate coconutRemovedAt:self.position];
    [self removeFromParentAndCleanup:YES];
}

-(void)fallenOffScreen;
{
    [self.delegate fallenOffScreenAt:self.position];
    [self removeFromParentAndCleanup:YES];
}
@end
```

4. Now, open the `GameScene.h` file and import the `Coconut.h` header:

```
#import "Coconut.h"
```

5. After that, make the `GameScene` class conform to the `CoconutDelegate` protocol:

```
@interface GameScene : CCScene<CoconutDelegate>
```

6. Switch to the `GameScene.m` file and import the following headers:

```
#import "MenuScene.h"
#import "cocos2d.h"
```

7. Then, add `enum` that contains the states of the game:

```
typedef NS_ENUM(NSUInteger, GameState)
{
    GameStateInit,
    GameStatePlaying,
    GameStateLost
};
```

8. Add the following instance variables that will be used in the game:

```
@implementation GameScene
{
    int _lives;
    int _points;
    int _pointsPerCoconut;

    GameState _gameState;
    float _timeUntilNextCoconut;

    CCLabelTTF *_lblPoints;
    CCLabelTTF *_lblLives;
}
```

9. Add the `init` method and the methods to initialize the game defaults along with all the scene elements:

```
-(instancetype)init
{
    if (self = [super init])
    {
        [self setupGameDefaults];
        [self addBackground];
        [self addLabels];

        self.userInteractionEnabled = YES;
    }

    return self;
}

-(void)onEnterTransitionDidFinish
{
    [super onEnterTransitionDidFinish];

    _gameState = GameStatePlaying;
}

-(void)setupGameDefaults
{
    _gameState = GameStateInit;
    _timeUntilNextCoconut = 0;

    _lives = 3;
    _points = 0;
    _pointsPerCoconut = 5;
}

-(void)addBackground
{
    CCSprite *bg =
      [CCSprite spriteWithImageNamed:@"game_bg.png"];
    bg.positionType = CCPositionTypeNormalized;
    bg.position = ccp(0.5f, 0.5f);
    [self addChild:bg];
}

-(void)addLabels
{
    _lblPoints = [CCLabelTTF labelWithString:@"Points: 0"
                                    fontName:@"Helvetica"
                                    fontSize:14];
    _lblPoints.anchorPoint = ccp(0, 0.5f);
    _lblPoints.color = [CCColor redColor];
    _lblPoints.positionType = CCPositionTypeNormalized;
    _lblPoints.position = ccp(0.05f, 0.95f);
    [self addChild:_lblPoints];

    NSString *lives =
      [NSString stringWithFormat:@"Lives: %d", _lives];

    _lblLives = [CCLabelTTF labelWithString:lives
                                   fontName:@"Helvetica"
                                   fontSize:14];

    _lblLives.anchorPoint = ccp(1, 0.5f);
    _lblLives.color = [CCColor redColor];
    _lblLives.positionType = CCPositionTypeNormalized;
    _lblLives.position = ccp(0.95f, 0.95f);
    [self addChild:_lblLives];
}
```

10. Add the `update:` method to instantiate coconuts:

```
-(void)update:(CCTime)dt
{
    if (_gameState == GameStatePlaying)
    {
        _timeUntilNextCoconut -= dt;

        if (_timeUntilNextCoconut <= 0)
        {
            Coconut *coconut = [Coconut node];
            coconut.name = @"coconut";
            coconut.delegate = self;
            [self addChild:coconut];

            _timeUntilNextCoconut =
              0.5f + arc4random_uniform(2.0f);
        }
    }
}
```

11. Add the following implementation of the methods from the `CoconutDelegate` protocol:

```
-(void)coconutRemovedAt:(CGPoint)position
{
    if (_gameState == GameStatePlaying)
    {
        _points += _pointsPerCoconut;
        _lblPoints.string =
          [NSString stringWithFormat:@"Points: %d", _points];
    }
}

-(void)fallenOffScreenAt:(CGPoint)position
{
    [self displayMissedCoconutAt:position.x];

    _lives--;
    if (_lives < 0)
        _lives = 0;

    _lblLives.string =
      [NSString stringWithFormat:@"Lives: %d", _lives];

    if (_lives <= 0 && _gameState == GameStatePlaying)
    {
        _gameState = GameStateLost;

        CCLabelTTF *youLoseLabel =
            [CCLabelTTF labelWithString:@"You lose!"
                               fontName:@"Helvetica-Bold"
                               fontSize:48];

        youLoseLabel.positionType = CCPositionTypeNormalized;
        youLoseLabel.position = ccp(0.5f, 0.5f);
        youLoseLabel.color = [CCColor whiteColor];
        [self addChild:youLoseLabel];
    }
}
```

12. Add a method that will display a red cross in the place where a coconut has fallen off the screen:

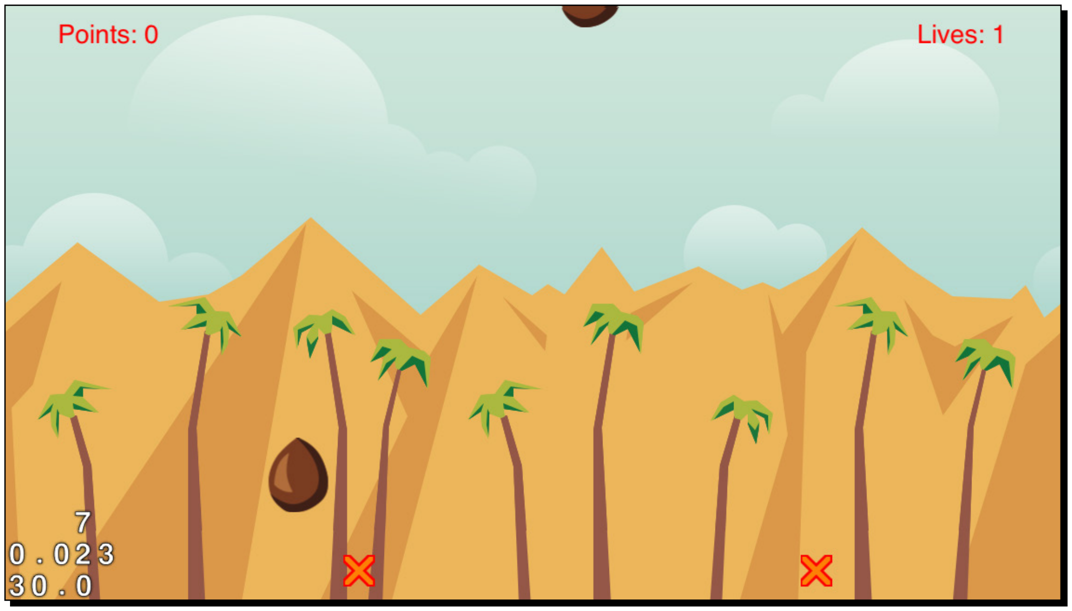
```
-(void)displayMissedCoconutAt:(float)coconutX
{
    CCSprite *cross =
      [CCSprite spriteWithImageNamed:@"cross.png"];
    cross.positionType = CCPositionTypeNormalized;
    cross.position = ccp(coconutX, 0.05f);
    [self addChild:cross];

    CCActionDelay *delay =
     [CCActionDelay actionWithDuration:1.0f];
    CCActionFadeOut *fadeOut =
      [CCActionFadeOut actionWithDuration:0.5f];
    CCActionRemove *remove = [CCActionRemove action];

    CCActionSequence *displayFadeThenRemove =
      [CCActionSequence actions:delay,fadeOut,remove, nil];
    [cross runAction:displayFadeThenRemove];
}
```

13. Finally, when the player loses the game, add a method to leave the game by touching the game screen:

```
-(void)touchBegan:(UITouch *)touch withEvent:(UIEvent *)event
{
    if (_gameState == GameStateLost)
        [[CCDirector sharedDirector]
           replaceScene:[MenuScene node]];
}
```

Note that we added the `self.userInteractionEnabled = YES;` line in the `init` method. Otherwise, the `touchBegan:` method won't be called.

14. Build and run the game. Tap on the **Start** button to start the game. Tap on the coconuts falling from the sky to remove them and earn points. If you lose the game, just tap anywhere on the screen to leave the game and navigate to the main menu. Refer to the following screenshot:



What just happened?

Let's start reviewing the code from the existing code in the `GameScene.m` file. The game has three states: `GameStateInit`, `GameStatePlaying`, and `GameStateLost`.

The `GameStateInit` state simply means that the game is not started yet. We start the game by changing the game state to the `GameStatePlaying` state in the `onEnterTransitionDidFinish` method, so that coconuts start to fall only after the enter transition is finished.

> We don't use transitions in this sample game, but still use `onEnterTransitionDidFinish` just to follow best practices or in case you want to add transitions later.

The game will remain in the `GameStatePlaying` state until the player loses all the lives. In this state, new coconuts are spawned, and the player can earn points by hitting them.

Then, when the player loses all the lives, the game will switch to the `GameStateLost` state. This will stop the spawning of coconuts, and if the player touches the screen in this state, we simply return to the main menu.

In a normal game, we would have created some kind of dialog that offers to restart the game or return to the main menu. For this simple game, we just need a way to return to the main menu when we lose, to open Game Center, and check that we submitted the score and achievements successfully.

Instance variables should be pretty clear. They are required to keep the count of lives and points, the current game state, and a timer variable to spawn coconuts. In addition to this, there are two labels to display points and lives, so we need to store a reference for them somewhere.

The code of the `init` method simply sets the game defaults, and adds the background and labels.

Note that we set `userInteractionEnabled` to `YES` and added the `touchBegan:` method so that the player can tap anywhere on the screen when he or she loses the game and goes back to the main menu. Touching coconuts is handled in the `Cocohunt` class and not in the `GameScene` class.

The code to spawn coconuts is placed in the `update:` method. There, we simply create an instance of the `Coconut` class that contains all the code to instantiate, move, and remove coconuts.

If you open the `Coconut.m` class, you will see that it simply creates itself at random positions and starts moving down.

We didn't use physics; we simply used `CCActionEaseIn` to speed up the coconut's movement as it falls down just as if it were affected by gravity.

Note that we used `CCActionCallFunc` to call the `fallenOffScreen` method at the end of the action sequence. Calling this method will mean that the coconut reached the bottom of the screen without the player tapping on it.

We set `userInteractionEnabled` to `YES` in the `init` method, since we need to detect touches on the coconut. One of the main reasons why we subclassed the `CCSprite` class is that we can override the `touchBegan:` method and detect touches on each coconut node individually.

However, if the `touchBegan:` method is called, this means the player hit the coconut, and it should be removed. This will stop the sequence, and the `fallenOffScreen` method won't be called.

If you call the `removeFromParent` or `removeFromParentAndCleanup:` methods with the `YES` parameter, which is the same as just calling the `removeFromParent` method, it will stop all the running actions. So you don't have to call `stopAllActions` explicitly.

In both cases, if the coconut has fallen off the screen or was hit by the player, we notify the `GameScene` class using one of the `CoconutDelegate` protocol methods.

If the player hits the coconut, the `coconutRemovedAt:` method is called. In this case, we increase points and update the **Points** label. Otherwise, if the coconut falls off the screen, the `fallenOffScreenAt:` method is called and we decrease the lives count by updating the **Lives** label and checking whether the player has lost the game.

The `displayMissedCoconutAt:` method simply shows a red cross at the place where the coconut has fallen so that the player can see where this occurred.

Integrating Game Center

Now that we have a simple game in our hands, it is time to integrate Game Center into it.

If you've skipped the previous part of the chapter and took the starter game project from the book's supporting files, don't forget to update the **Bundle Identifier** in the project settings to match the bundle ID of the app you created in iTunes Connect.

Enabling Game Center and authenticating the player

Before we can use any Game Center services, we need to enable it in iTunes Connect and add leaderboards and achievements that we want to use. After that, we need to add the code to authenticate the player and display Game Center views.

Looks like a lot of work, so let's get started.

Time for action – creating achievements and a leaderboard

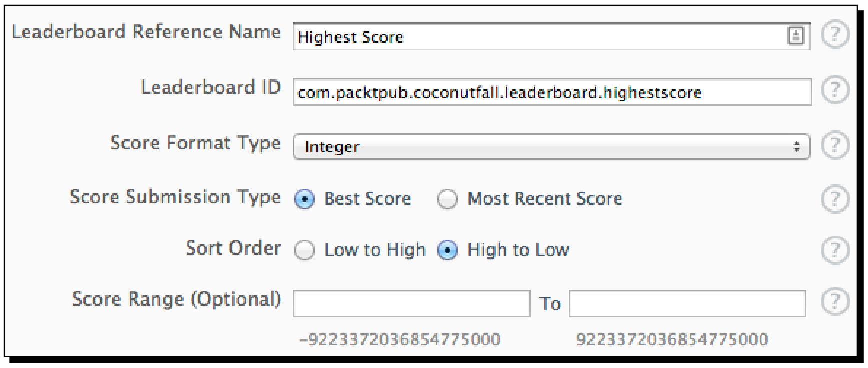
Before writing any code, let's enable Game Center and create a few achievements and a leaderboard.

1. Sign in to **iTunes Connect** using the link `https://itunesconnect.apple.com`. Select **Manage Your Apps** and click on the sample game that you created.
2. Then, click on the **Manage Game Center** button to enable Game Center. When you click on this button for the first time, it will ask whether you want to enable Game Center for a single game or a group of games. Click on the **Enable for Single Game** button.

> Enabling Game Center for a group of games is useful if you want to share leaderboards and achievements between two or more separate games. For example, you might want to register the iPhone and iPad versions of the game as separate applications with different bundle IDs, but you want them to have the same leaderboards and achievements.

3. Click on the **Add Leaderboard** button to create a leaderboard. Choose **Single Leaderboard**.
4. Fill out the leaderboard's properties as follows:
   - **Leaderboard Reference Name**: Select **Highest Score**
   - **Leaderboard ID**: Enter your bundle ID plus `.leaderboard.highestscore` (for example, `com.packtpub.coconutfall.leaderboard.highestscore`)
   - **Score Format Type**: Select **Integer**
   - **Score Submission Type**: Select **Best Score**
   - **Sort Order**: Select **High to Low**

These properties can be seen in the following screenshot:

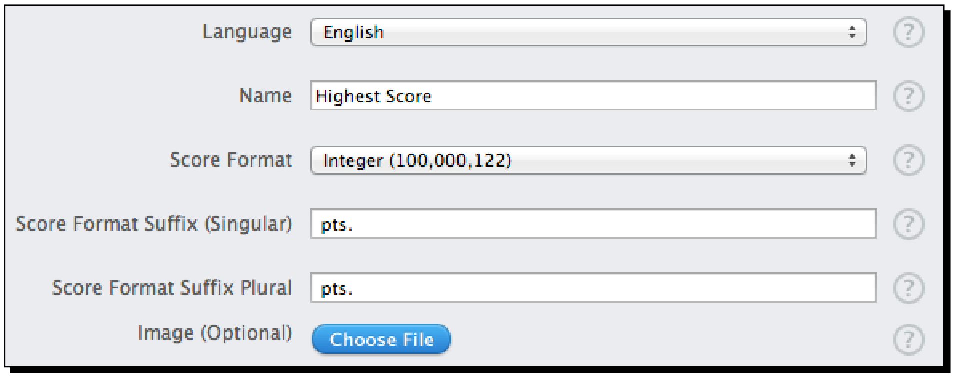


Leaderboard Reference Name	Highest Score
Leaderboard ID	com.packtpub.coconutfall.leaderboard.highestscore
Score Format Type	Integer
Score Submission Type	[✓ Best Score] Most Recent Score
Sort Order	Low to High [✓ High to Low]
Score Range (Optional)	-9223372036854775000 To 9223372036854775000

In fact, you don't have to prefix the leaderboard ID or achievement ID with the bundle ID. However, since the leaderboard ID and achievement ID should be unique among all of your games and applications, it is a good idea to use the bundle ID as a prefix to guarantee a higher level of uniqueness.

5. Click on the **Add Language** button, and add **English** as the localization of the leaderboard. Fill out the fields as follows:
   - **Language**: Select **English**
   - **Name**: Enter `Highest Score`
   - **Score Format**: Choose any option from the drop-down menu
   - **Score Format Suffix (Singular)**: Enter `pts`
   - **Score Format Suffix Plural**: Enter `pts`

   Refer to the following screenshot:



Language	English
Name	Highest Score
Score Format	Integer (100,000,122)
Score Format Suffix (Singular)	pts.
Score Format Suffix Plural	pts.
Image (Optional)	Choose File

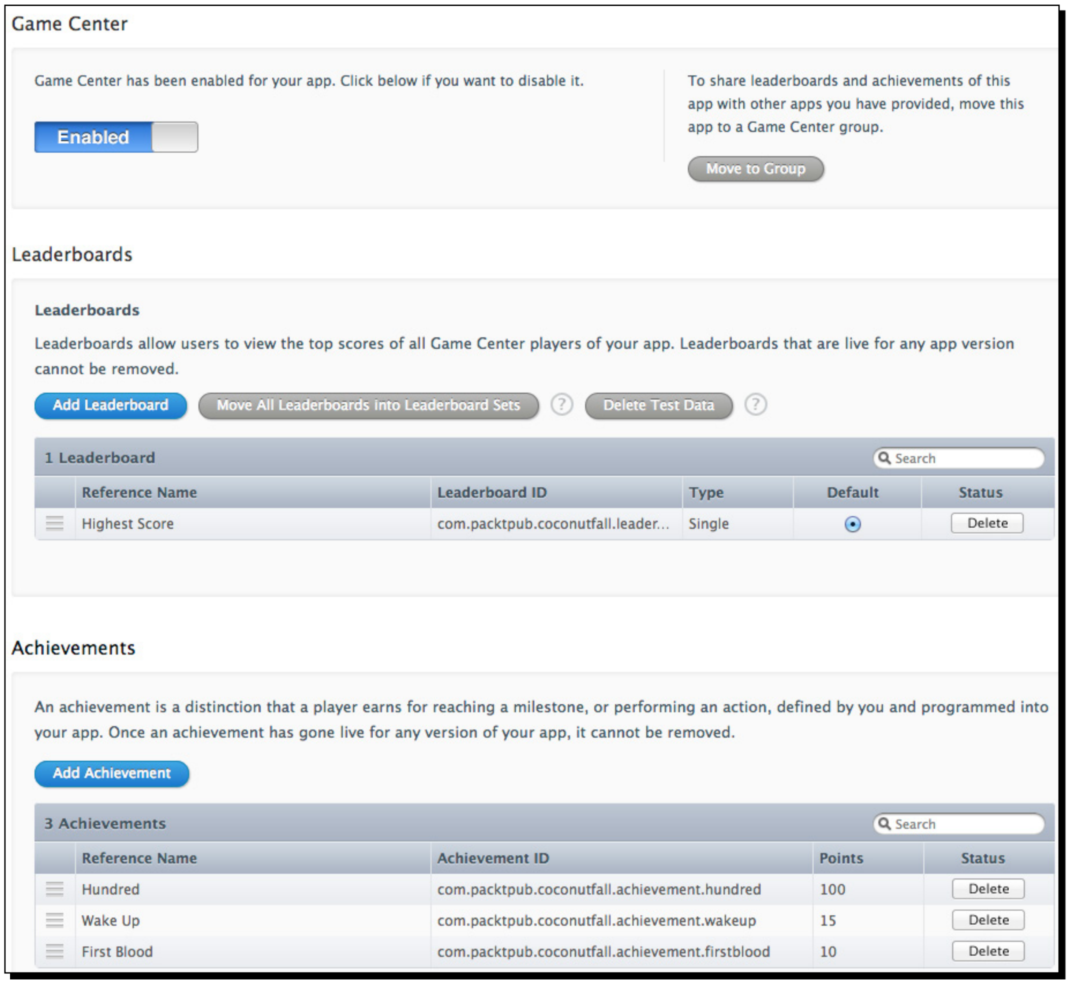
6. Click on **Save** to save the localization and then click on **Save** again to save the leaderboard.

You can add as many languages as you want, but I recommend completing this chapter first.

7. Now, let's add a few achievements. Click on the **Add Achievement** button and enter the following achievement properties:
   - **Achievement Reference Name**: Enter `Hundred`
   - **Achievement ID**: Enter your bundle ID plus `.achievement.hundred` (for example, `com.packtpub.coconutfall.achievement.hundred`)
   - **Point Value**: Enter `100`
   - **Hidden**: Select **No**
   - **Achievable More Than Once**: Select **No**
8. Then, click on **Add Language** and add a localization:
   - **Language**: Select **English**
   - **Title**: Enter `Hundred`
   - **Pre-earned Description**: Enter `Score 100 points`
   - **Earned Description**: Enter `You scored a hundred points!`
   - **Image**: Use the `Achievement_Icon.png` image from the `Chapter_12/Assets/AppStore` folder
9. Save the achievement and add two more achievements using the following properties (use the same image for all achievements):
   - Wake-up achievement:
     - **Achievement Reference Name**: Enter `Wake Up`
     - **Achievement ID**: Enter your bundle ID plus `.achievement.wakeup`
     - **Point Value**: Enter `15`
     - **Hidden**: Select **Yes**
     - **Achievable More Than Once**: Select **No**
     - **Language**: Select **English**
     - **Title**: Enter `Wake Up`
     - **Pre-earned Description**: Enter anything since it won't be visible until achieved
     - **Earned Description**: Enter `Lost a game without scoring a single point!`
   - First-blood achievement:
     - **Achievement Reference Name**: Enter `First Blood!`
     - **Achievement ID**: Enter your bundle ID plus `.achievement.firstblood`
     - **Point Value**: Enter `10`
     - **Hidden**: Select **No**

- **Achievable More Than Once**: Select **No**
- **Language**: Select **English**
- **Title**: Enter `First Blood!`
- **Pre-earned Description**: Enter `Hit your first coconut!`
- **Earned Description**: Enter `You've hit your first coconut!`

The following screenshot shows how the configuration page of Game Center should look after adding a leaderboard and three achievements:



Game Center

Game Center has been enabled for your app. Click below if you want to disable it.

To share leaderboards and achievements of this app with other apps you have provided, move this app to a Game Center group.

Leaderboards

Leaderboards

Leaderboards allow users to view the top scores of all Game Center players of your app. Leaderboards that are live for any app version cannot be removed.

Reference Name	Leaderboard ID	Type	Default	Status
Highest Score	com.packtpub.coconutfall.leader...	Single	●	Delete

Achievements

An achievement is a distinction that a player earns for reaching a milestone, or performing an action, defined by you and programmed into your app. Once an achievement has gone live for any version of your app, it cannot be removed.

Reference Name	Achievement ID	Points	Status
Hundred	com.packtpub.coconutfall.achievement.hundred	100	Delete
Wake Up	com.packtpub.coconutfall.achievement.wakeup	15	Delete
First Blood	com.packtpub.coconutfall.achievement.firstblood	10	Delete

Note that you need to replace the `com.packtpub.coconutfall` part of the leaderboard ID and achievement IDs with your bundle ID.

What just happened?

We've just created the Game Center elements that we'll use in the game.

Leaderboards allow players to view the highest scores, best time, longest distance, and other measurable characteristics of your game and compete with other Game Center players of your game. You can have many leaderboards in your game, for example, players can compete for the highest score, see who lasts longer without dying, and so on, and all of it within one game.

[You can have the highest score, but another player scored fewer points while lasting longer than you on a level. You will be a leader in one leaderboard, but the other player will be a leader in another one.]

The **Highest Score** leaderboard will contain the highest points scored while hitting coconuts.

Achievements are a way to reward the player for performing some special actions, reaching a milestone in the game, and so on. I'm sure you've gained some achievements if you played at least one game with Game Center integrated.

We will have the following three achievements:

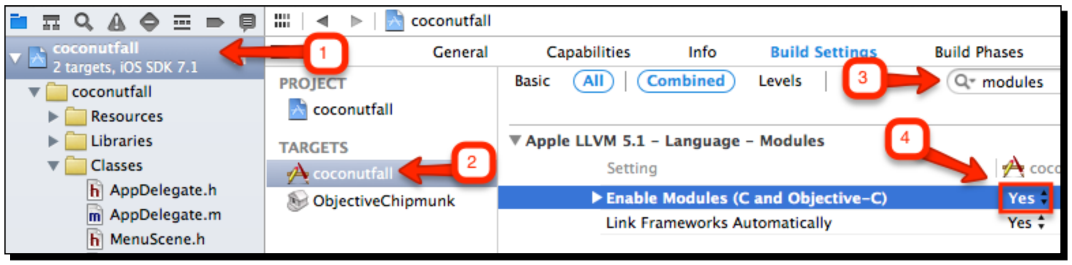
- `Hundred`: This will be awarded to the player who scores 100 points or more. The interesting thing about this achievement is that we'll update the `percent completed` property of this achievement. This means that the player who scored 60 points will see that the achievement is 60 percent completed. It might not be very useful in our case, but there are times when you need to perform many actions, such as killing 1,000 monsters in any amount of tries, and you want to know your progress.
- `Wake Up`: This will be awarded to the player who scores zero points. The interesting part about this achievement is that it is hidden. This means you don't know how to achieve it. This boils up the player's interest and they play your game more and more, trying to figure out what they should do to unlock this achievement.
- `First Blood`: This is the easiest one to achieve. To achieve it, you simply need to tap one coconut. It is a good idea to make a few easily attainable achievements and award the player immediately after they start playing your game so that the player knows there are achievements and may open Game Center to see them.

Time for action – authenticating the player

Only authenticated players can compete with other players and earn achievements. So the first step of integrating Game Center in the game is letting the player sign in.

> For those players who don't want to use Game Center, you can create a type of alternate highscores table and achievements, just as we created a simple offline highscores table in *Chapter 9*, *User Interface and Navigation*, which of course is too primitive to even compare with Game Center.

1. Before we can use Game Center in the game, we need to add the `GameKit` framework. Instead of adding it manually, we're going to enable modules and use an `@import` statement. To enable modules, go to the Xcode project's settings, select the **coconutfall** target, and open the **Build Settings** tab. Type `modules` in the search box and change **Enable Modules (C and Objective-C)** to **Yes**.



Enable Modules (C and Objective-C) Yes
Link Frameworks Automatically Yes

2. To organize our code, we're going to create a `GCManager` class that will contain most of the code related to Game Center. So go ahead and create a new **Objective-C class** called `GCManager` and make it a subclass of the `NSObject` class.

3. Open the `GCManager.h` file and import `GameKit` using the `@import` statement:

```
@import GameKit;
```

4. Add the following method's declarations:

```
-(void)loginToGameCenter;

+(GCManager *)sharedInstance;
```

5. Switch to the `GCManager.m` file and add the `_isUserLoggedIn` instance variable:

```
@implementation GCManager
{
    BOOL _isUserLoggedIn;
}
```

6. Then, add the following methods that we'll use to authenticate the player and track the changes in the authentication status:

```
-(id)init
{
    if (self =[super init])
    {
        _isUserLoggedIn = NO;
        [self subscribeToAuthStatusChange];
    }

    return self;
}

-(void)loginToGameCenter
{
    if ([GKLocalPlayer localPlayer].authenticated == NO)
        [[GKLocalPlayer localPlayer]
          authenticateWithCompletionHandler:nil];
    else
        NSLog(@"loginToGameCenter: User Already Logged In");
}

-(void)subscribeToAuthStatusChange
{
    [[NSNotificationCenter defaultCenter]
     addObserver:self
     selector:@selector(authenticationChanged)
     name:GKPlayerAuthenticationDidChangeNotificationName
     object:nil];
}

-(void)authenticationChanged {
    if ([GKLocalPlayer localPlayer].isAuthenticated &&
        !_isUserLoggedIn)
    {
        _isUserLoggedIn = YES;
        NSLog(@"User logged in to Game Center");
    }
    else if (![GKLocalPlayer localPlayer].isAuthenticated &&
             _isUserLoggedIn)
    {
        _isUserLoggedIn = NO;
        NSLog(@"User logged out from Game Center");
    }
}

+(GCManager *)sharedInstance
{
    static dispatch_once_t pred;
    static GCManager * _sharedInstance;
    dispatch_once(&pred, ^{
      _sharedInstance = [[self alloc] init];
    });
    return _sharedInstance;
}
```

7. Now, open the `AppDelegate.m` file and import the `GCManager.h` header:

   ```
   #import "GCManager.h"
   ```

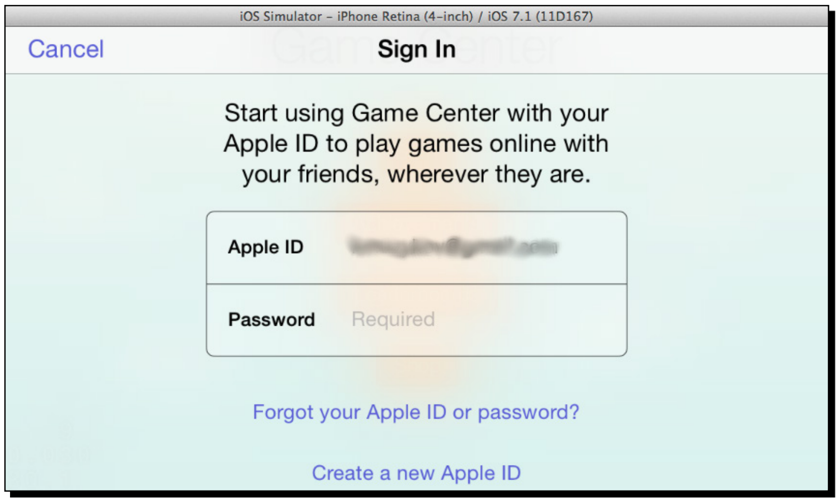
8. Then, add the following line to the beginning of the `application:didFinishLaunchingWithOptions:` method:

   ```
   -(BOOL)application:(UIApplication *)application didFinishLaunching
   WithOptions:(NSDictionary *)launchOptions
   {
       [[GCManager sharedInstance] loginToGameCenter];

       //..skipped..
   }
   ```

9. Build and run the game. When the game is launched, wait for the Game Center login dialog box to appear.

> It might take some time for it to appear. The longest time I had to wait was for around 30 seconds.

Create a new Apple ID in this dialog box even if you already have one, since this will be your sandbox mode's Game Center Apple ID. Refer to the following screenshot:


iOS Simulator – iPhone Retina (4-inch) / iOS 7.1 (11D167)
Cancel
Sign In

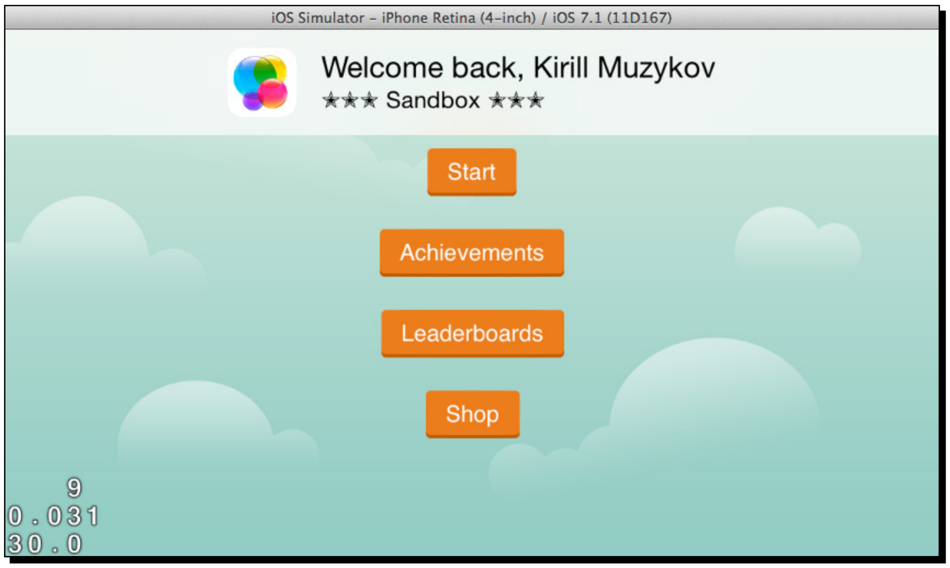
Start using Game Center with your Apple ID to play games online with your friends, wherever they are.

Apple ID

Password Required

Forgot your Apple ID or password?
Create a new Apple ID

After logging in, you will see a **Welcome Back** banner on the top, as shown in the following screenshot:


iOS Simulator – iPhone Retina (4-inch) / iOS 7.1 (11D167)
Welcome back, Kirill Muzykov
★★★ Sandbox ★★★
Start
Achievements
Leaderboards
Shop
9
0.031
30.0

What just happened?

When the player starts the game, we need to prompt him or her to log in to Game Center as early as possible. We need to do this because it takes some time to authenticate even the previously authenticated player, and we have buttons that require an authenticated player as early as on the menu scene (the **Achievements** and **Leaderboards** buttons).

To authenticate the player, we used the `loginToGameCenter` method of the `GCManager` singleton. We called this method as early as we could from the `application:didFinishLaunchingWithOptions:` method.

The `loginToGameCenter` method simply checks whether the player is not already authenticated and calls the `authenticateWithCompletionHandler:` method of the local player.

```
    [[GKLocalPlayer localPlayer] authenticateWithCompletionHandler:nil];
```

The `GKLocalPlayer` class represents the player that currently plays the game on the current device.

We can get information about other players in Game Center; this is why it is required to understand the concept of the local player.

The `authenticateWithCompletionHandler:` method simply authenticates the player in Game Center. If the player is not previously authenticated in Game Center, it shows a **Sign In** window. If the player is already authenticated in Game Center, maybe in some other game or using Game Center app, it will show the **Welcome Back** banner.

[You will see that once authenticated, you won't have to sign in each time; now, you will only see the **Welcome Back** banner, unless you sign out in the Game Center app.]

You might wonder why we passed a `nil` value to the `authenticateWithCompletionHandler:` method as if we didn't care about when it completes.

The reason for this is that the player can sign in and sign out from Game Center outside of our game. For example, the player might simply push the **Home** button, go to the Game Center app, sign out there, and then return to our game, and our game needs to know that.

To get notified about the player authentication changes, we created and used the `subscribeToAuthStatusChange` method, which uses `NSNotificationCenter` to get the `GKPlayerAuthenticationDidChangeNotificationName` notification in the `authenticationChanged` method, shown as follows:

```
-(void)subscribeToAuthStatusChange
{
    [[NSNotificationCenter defaultCenter]
     addObserver:self
     selector:@selector(authenticationChanged)
     name:GKPlayerAuthenticationDidChangeNotificationName
     object:nil];
}
```

We called this `subscribeToAuthStatusChange` method from the `init` method, which is executed when we access the singleton for the first time, so it gets executed even before we call the `loginToGameCenter` method.

This way, the `authenticationChanged` method is called each time the user authentication changes, including the change after we call the `authenticateWithCompletionHandler:` method. This is why we don't need the completion handler, since we'll know when the player will be authenticated anyway.

Time for action – displaying the Game Center view

After the player is authenticated, we can display the Game Center dialogs with achievements and leaderboards. Right now, the leaderboard will be empty, and no achievements will be completed. However, let's display the dialogs before we can concentrate on submitting the score and reporting achievements. Perform the following steps:

1. Open the `MenuScene.h` file and import the `GCManager.h` header:

   ```
   #import "GCManager.h"
   ```

2. Then, make the `MenuScene` class conform to three protocols:

   ```
   @interface MenuScene : CCScene
     <GKLeaderboardViewControllerDelegate,
      GKGameCenterControllerDelegate,
      GKAchievementViewControllerDelegate>
   ```

3. Switch to the `MenuScene.m` file and add the following two methods to display achievements and leaderboards:

   ```
   -(void)displayGCAchievements
   {
       //1
       if (NSFoundationVersionNumber <
            NSFoundationVersionNumber_iOS_6_0)
       {
           //2
           GKAchievementViewController *achievements =
             [[GKAchievementViewController alloc] init];

           //3
           achievements.achievementDelegate = self;

           //4
           [[CCDirector sharedDirector]
             presentModalViewController:achievements
                                 animated:YES];
       }
       else
       {
           //5
           GKGameCenterViewController *achievements =
             [[GKGameCenterViewController alloc] init];

           //6
           achievements.gameCenterDelegate = self;

           //7
   ```

```
        achievements.viewState =
          GKGameCenterViewControllerStateAchievements;
        //8
        [[CCDirector sharedDirector]
          presentModalViewController:achievements
                            animated:YES];
    }
}

-(void)displayGCLeaderboard
{
    if (NSFoundationVersionNumber <
          NSFoundationVersionNumber_iOS_6_0)
    {
        GKLeaderboardViewController *leaderboard =
          [[GKLeaderboardViewController alloc] init];
        leaderboard.leaderboardDelegate = self;
        [[CCDirector sharedDirector]
           presentModalViewController:leaderboard
                            animated:YES];
    }
    else
    {
        GKGameCenterViewController *leaderboard =
          [[GKGameCenterViewController alloc] init];
        leaderboard.gameCenterDelegate = self;
        leaderboard.viewState =
          GKGameCenterViewControllerStateLeaderboards;
        [[CCDirector sharedDirector]
          presentModalViewController:leaderboard
                            animated:YES];
    }
}
```

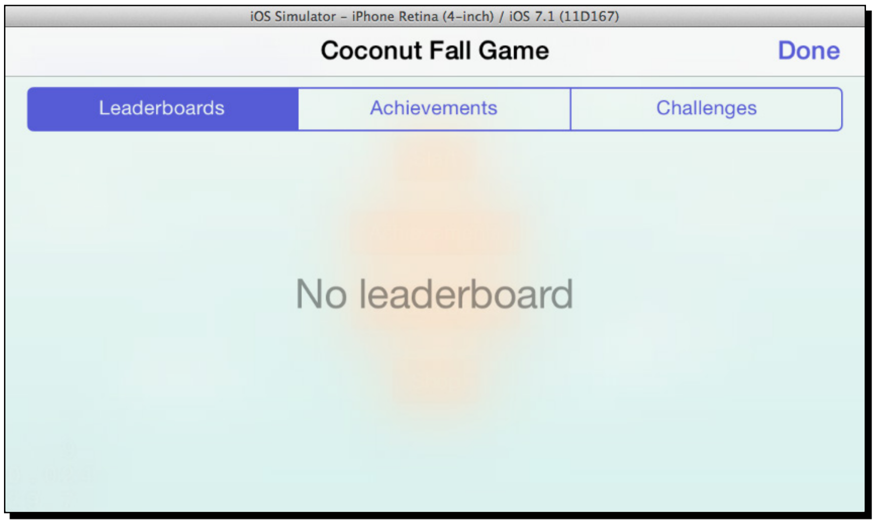
4. Then, we need to add the implementation of each method from three protocols. This means we need to add three methods as follows:

```
-(void)achievementViewControllerDidFinish:
         (GKAchievementViewController *)viewController
{
    [viewController.presentingViewController
      dismissModalViewControllerAnimated:YES];
}

-(void)leaderboardViewControllerDidFinish:
         (GKLeaderboardViewController *)viewController
{
    [viewController.presentingViewController
      dismissModalViewControllerAnimated:YES];
}

-(void)gameCenterViewControllerDidFinish:
          (GKGameCenterViewController *)gameCenterViewController
{
    [gameCenterViewController.presentingViewController
      dismissModalViewControllerAnimated:YES];
}
```

5. Finally, modify the `onBtnTap:` method to call the `displayGCAchievements` and `displayGCLeaderboard` methods when the corresponding button is tapped:

```
-(void)onBtnTap:(CCButton *)btn
{
    //..skipped..

    else if ([btn.name isEqualToString:kButtonAchievements])
    {
        [self displayGCAchievements];
    }
    else if ([btn.name isEqualToString:kButtonLeaderboards])
    {
        [self displayGCLeaderboard];
    }

    //..skipped

}
```

6. Build and run the game. You should see a **Welcome Back** banner since you've already authenticated previously. If you tap the **Achievements** or **Leaderboard** buttons, you should see the Game Center dialog box. Refer to the following screenshot:



Don't be surprised to see a **No leaderboard** label instead of the leaderboard. The leaderboard will become visible when at least one player reports a score to it.

What just happened?

Now, when you press the **Achievements** or **Leaderboards** button, you actually execute the code. Depending on which button was pressed, you either call the `displayGCAchievements` or `displayGCLeaderboard` method.

Both methods are very similar, especially for iOS 6.0 and later, so we'll review only the `displayGCAchievements` method. Refer to the following numbered comments:

1. This code checks whether the device is running an iOS version earlier than iOS 6.

   Since Cocos2D-Swift v3.x supports iOS 5 and later, this will mean that the device is running one of the iOS v5.x.

2. In iOS 5, we need to use the `GKAchievementViewController` class to display achievements, and the `GKLeaderboardViewController` class to display leaderboards. While in iOS 6 or later, we can use the new `GKGameCenterViewController` class in both cases.
3. Set `achievementDelegate` to `self` in order to get notified when the Game Center view controller needs to be closed. This is also the reason we made the `MenuScene` class conform to the `GKAchievementViewControllerDelegate` protocol. Two other protocols are serving the same purpose.
4. Since the `CCDirector` class is the game's view controller (inherited from `UIViewController`), we used it to display the Game Center view controller.
5. In the case of iOS 6 or higher, we used the `GKGameCenterViewController` class to display both achievements and leaderboards.
6. Set the `gameCenterDelegate` to `self` so that we got notified when the Game Center view controller needs to be closed.
7. This is the only line that differs if you want to display achievements or leaderboards on iOS 6 or higher. In this case, we set it to `GKGameCenterViewControllerStateAchievements` since we want to display achievements.
8. Display the Game Center view controller.

This is everything we need to authenticate the player and let the player take a look at the current Game Center information.

However, right now, there is nothing to look at. The leaderboard is empty and you can't complete any achievements. Let's fix this!

Reporting scores and achievements to Game Center

To see some information in the Game Center view, we first need to report this information to Game Center. Let's begin with reporting the score.

Time for action – reporting the score

We will report the score to Game Center at the moment the player has already lost the game because this means the player cannot increase the score anymore.

To report scores, we'll need to extend our `GCManager` class as follows:

1. Open the `GCManager.h` file and use the `#define` statements to store the leaderboard ID and achievement IDs:

   [Don't forget to change the values with your leaderboard and achievement IDs.]

```
#define kLeaderboardID  @"com.packtpub.coconutfall.leaderboard.
highestscore"

#define kAchievementHundred     @"com.packtpub.coconutfall.
achievement.hundred"

#define kAchievementWakeUp      @"com.packtpub.coconutfall.
achievement.wakeup"

#define kAchievementFirstBlood  @"com.packtpub.coconutfall.
achievement.firstblood"
```

[Note that there is no line break inside the string value. Each `#define` statement is placed on a single line.]

2. Add the `reportScore:` method declaration:

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

3. Switch to the `GCManager.m` file and add the implementation of the `reportScore:` method:

   ```
   -(void)reportScore:(int)score
   {
       //1
       if (!_isUserLoggedIn)
           return;

       //2
       GKScore *gkScore = [[GKScore alloc]
                           initWithCategory:kLeaderboardID];
       //3
       gkScore.value = score;

       //4
       id completionHandler = ^(NSError * error) {
           if (error)
               NSLog(@"Error reporting score: %@", error);
       };

       //5
       if (NSFoundationVersionNumber <
             NSFoundationVersionNumber_iOS_6_0)
       {
           [gkScore
             reportScoreWithCompletionHandler:completionHandler];
       }
       else
       {
           [GKScore reportScores:@[gkScore]
              withCompletionHandler:completionHandler];
       }
   }
   ```

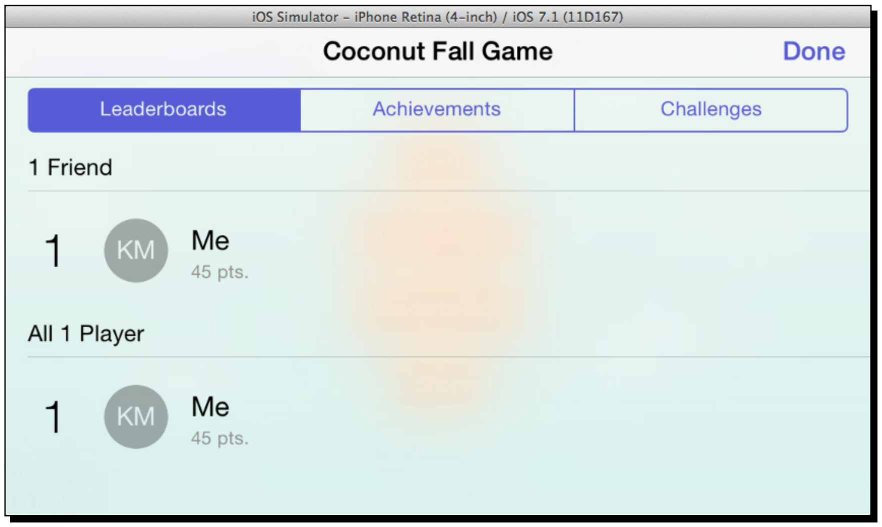
4. Now, you need to actually report the score. Open the `GameScene.m` file and import the `GCManager.h` header:

   ```
   #import "GCManager.h"
   ```

5. Then, search for the `fallenOffScreenAt:` method. Inside the `if` statement, add the following line to report the score:

```
-(void)fallenOffScreenAt:(CGPoint)position
{
    //..skipped..

    if (_lives <= 0 && _gameState == GameStatePlaying)
    {
        [[GCManager sharedInstance]
         reportScore:_points];

        //..skipped..
    }
}
```

6. Build and run the game. Play one game to score some points. After you lose, click anywhere within the game scene to return to the main menu and tap the **Leaderboards** button to open the leaderboards. Refer to the following screenshot:



What just happened?

It is nice to see some scores. We only had to write a few lines of code for that.

Let's review the code of the `reportScore:` method:

1. If the user is not authenticated in Game Center, then we cannot report the score.
2. The `GKScore` class is used to report the score. Note that we passed the `kLeaderboardID` leaderboard ID as a category. Since there can be several leaderboards, you should specify which one you want to submit the score to.

[The category is just another way of saying leaderboard ID.]

3. Set the score value to report.
4. There are two different ways to report scores depending on the iOS version, but we can use the same `completionHandler` method, so we just store it in a variable here. This time, we specified the completion handler to log the error to a console, if there is one.
5. We used the `reportScoreWithCompletionHandler:` (instance) method in the case of iOS 5 and the `reportScores:withCompletionHandler:` (class) method for iOS 6 and later to report the score.

[In the case of iOS 6 and later, you can report an array of scores at once. This might be useful if you have many different leaderboards.]

That's it. Now, you can report and beat your own scores, and when the game goes live, you will be able to compete with other players.

[You can delete test data in the leaderboard in the iTunes Connect Game Center management page.]

Time for action – awarding achievements

Reporting achievements is pretty similar to reporting scores. However, there are several differences that we will see in a moment. Perform the following steps:

1. Open the `GCManager.h` file and add the `reportAchievement:` method declaration:

```
-(void)reportAchievement:(NSString *)achievementId
                progress:(double)progress;
```

2. Then, switch to the `GCManager.m` file and add the implementation of that method:

```
-(void)reportAchievement:(NSString *)achievementId
                progress:(double)progress
{
    //1
    GKAchievement *achievement =
          [[GKAchievement alloc] initWithIdentifier:achievementId];

    //2
    achievement.percentComplete = progress;

    //3
    achievement.showsCompletionBanner = YES;

    //4
    id completionHandler = ^(NSError * error) {
        if (error)
            NSLog(@"Error reporting achievements: %@", error);
    };

    //5
    if (NSFoundationVersionNumber <
           NSFoundationVersionNumber_iOS_6_0)
    {
        [achievement reportAchievementWithCompletionHandler:
          completionHandler];
    }
    else
    {
        [GKAchievement reportAchievements:@[achievement]
                    withCompletionHandler:completionHandler];
    }
}
```

3. Now, you need to add the code that will detect when the player unlocks an achievement. Open the `GameScene.m` file and search for the `fallenOffScreenAt:` method. In this method, place the following code at the beginning of the `if` block, right before we report the score. This code will check whether the player unlocked the `Wake Up` and `Hundred` achievements:

```
-(void)fallenOffScreenAt:(CGPoint)position
{
    //..skipped..

    if (_lives <= 0 && _gameState == GameStatePlaying)
    {
        //1
        if (_points == 0)
            [[GCManager sharedInstance]
              reportAchievement:kAchievementWakeUp
```

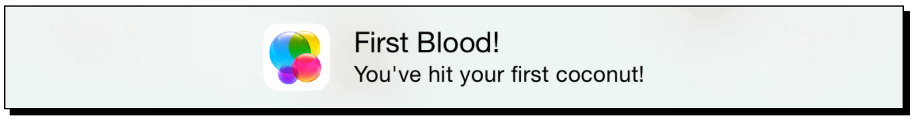
```
                            progress:100];

        //2
        if (_points > 100)
            [[GCManager sharedInstance]
              reportAchievement:kAchievementHundred
                         progress:100];
        else
            [[GCManager sharedInstance]
              reportAchievement:kAchievementHundred
                         progress:_points];

        //..skipped..

}
}
```

4. Then, add the code to award the `First Blood!` achievement in the `coconutRemovedAt:` method:

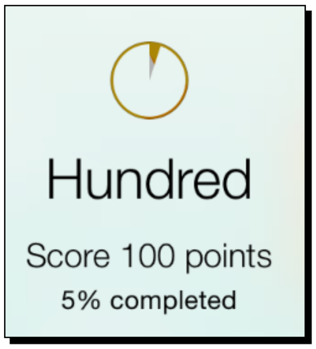
```
-(void)coconutRemovedAt:(CGPoint)position
{
    if (_gameState == GameStatePlaying)
    {
        [[GCManager sharedInstance]
          reportAchievement:kAchievementFirstBlood
                   progress:100];

        //..skipped..
    }
}
```

5. Build and run the game. Here is one of the suggested ways to test achievements:

   1. Try to click on several coconuts almost at once; this way, you will see that the **First Blood!** achievement banner appears two or more times, as shown in the following screenshot:


First Blood!
You've hit your first coconut!

2. Don't score more than 100 points in your first run to see the progress in the **Hundred** achievement, as shown in the following screenshot:



3. Lose a game without scoring a single point to unlock the **Wake Up** achievement.
4. Score more than 100 points to fully unlock the **Hundred** achievement.

There is no button to remove the achievements in iTunes Connect, but you can do this with the following code. Just add the `debugResetAchievements` method to the `GCManager` class and call it from somewhere:

```
-(void)debugResetAchievments
{
     [GKAchievement
       resetAchievementsWithCompletionHandler:
         ^(NSError *error) {
            if (error != nil)
                 NSLog(@"Resetting error: %@", error);
      }];
}
```

For example, you can call it at the start of the game to clear out achievements each time. Just don't forget to remove it.

What just happened?

First, we're going to discuss how to report achievements and review the code of the `reportAchievement:` method:

1. The `GKAchievement` class is used to work with achievements. To create an achievement, we passed the achievement identifier, which we created earlier in iTunes Connect.

2. As you can see with our **Hundred** achievement, the achievement can be partially completed. Setting the `percentComplete` property to `100` means that the achievement is fully completed, while setting the `percentComplete` property between `0` and `100` will mark it as partially completed.
3. The `showsCompletionBanner` property controls whether the banner should be shown when the achievement is completed. The banner only shows if `percentComplete` is `100`.

> You can disable the banner and show your own completion banner. This is done in many games where the default Game Center banner doesn't fit into the overall design and atmosphere of the game.

4. Once again, we used one completion handler for both cases, just as with the reporting score.
5. The `reportAchievementWithCompletionHandler:` (instance) method was used in case of iOS 5, and the `reportAchievements:withCompletionHandler:` (class) method was used in case of iOS 6 and higher to submit the score to Game Center.

Now, let's review the code we added to the `fallenOffScreenAt:` method.

1. At this stage, the player already lost the game and won't score any more points. If the player scored **0** points, then we award the **Wake Up** achievement.
2. This `if` block checks whether the player fully completed the **Hundred** achievement or we should just report the progress.

Finally, in the `coconutRemovedAt:` method, we simply report the **First Blood!** achievement every time the player hits a coconut.

> There is nothing bad with reporting an already completed achievement. The only issue might be the banner being shown first several times, but we'll talk about this in a moment.

That's it! Now you know how to add Game Center to your games.

> If you want some more information about Game Center, please visit `https://developer.apple.com/game-center/`. In addition to that, there are some wonderful videos from WWDC that show examples of using Game Center.

Have a go hero

As you can see, reporting scores and achievements is quite easy. However, there are lots of things you can improve:

- The first thing that comes to mind is adding more achievements and leaderboards.
- Right now, the **First Blood!** achievement banner appears on top of the screen and hides the points and the lives labels, which is an example of bad design. You can think of how to solve it. Maybe you just need to remove this achievement and show it at a different time or change the user interface design.
- Another possible issue is the achievement banner that appear multiple times for the same achievement. This happens because you manage to report the achievement multiple times before it is marked as completed. This can be fixed by keeping a local cache of achievements.

> You can see one way of doing this in the GKTapper sample project by Apple at `https://developer.apple.com/library/ios/samplecode/GKTapper/Introduction/Intro.html` (short link: `http://bit.ly/1o6UaLS`).

Making In-App purchases

You can pursue a different goal by integrating In-App purchases in your game, irrespective of whether you want to ride on a wave of freemium model popularity or simply add a few different hats for your character in a paid game. However, in both cases, you will need to know how to integrate In-App purchases.

Listing products and making a purchase

In this section of the chapter, we're going to add a shop to the sample game we created earlier in this chapter.

> Unfortunately, to test In-App purchases, you will need an actual device. However, you cannot test In-App purchase-related code on a simulator.

The players will be able to see all the products they can buy and make a purchase.

Time for action – adding In-App purchases in iTunes Connect

Before we can make purchases in our game, we need to add them in iTunes Connect. Perform the following steps:

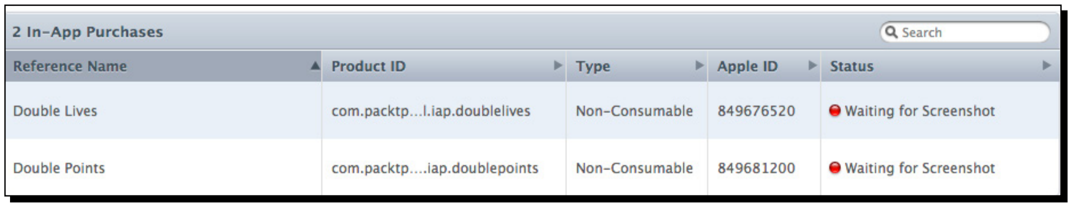
1. Sign in to iTunes Connect using the link `https://itunesconnect.apple.com`.
2. Click on **Manage Your Apps** and then click on the app we created at the start of this chapter. Click on the **Manage In-App Purchases** button.
3. Click on the **Create New** button and select the **Non-Consumable** In-App purchase type.

> We are only going to review the **Non-Consumable** In-App purchases, since all other types are pretty similar but require you to write more code to store and manage the purchases.
>
> For example, you might need to keep track of consumable purchases in iCloud or on your server, so that the player doesn't lose all of the purchases when reinstalling the game and can also use them on other devices. And using subscriptions in games is not very common.

4. Enter the following information (note that I included the **Localization** fields):
   - **Reference Name**: Enter `Double Lives`
   - **Product ID**: Enter your bundle ID plus `.iap.doublelives` (for example, `com.packtpub.coconutfall.iap.doublelives`)
   - **Cleared for Sale**: Select **Yes**
   - **Price Tier**: Choose any option from the drop-down menu
   - **Language**: Select **English**
   - **Display Name**: Enter `Double Lives`
   - **Description**: Enter `Double your lives!`
   - **Hosting Content with Apple**: Select **No**
   - **Review Notes**: Leave this field empty
   - **Screenshot for Review**: Don't provide a screenshot for now

> You might wonder what screenshot you should provide. Well, if your In-App purchase unlocks several new levels in the game, you might provide a screenshot where levels are shown unlocked.
> If your In-App purchase doesn't make any visual changes (for example, doubles points earned per coconut), you can simply attach a screenshot of the game scene and write how the In-App works in the **Review Notes** field.

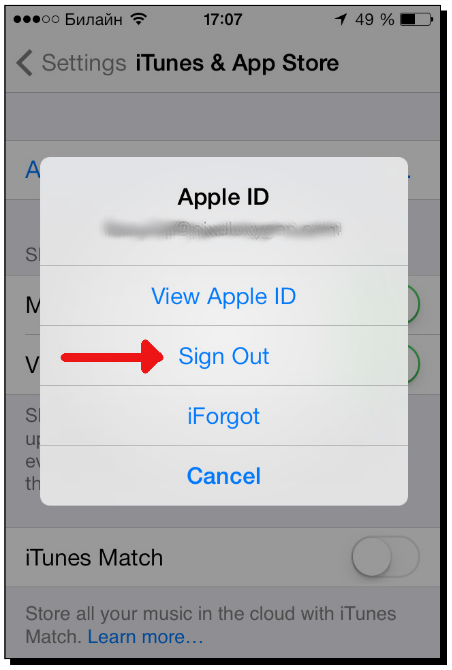
5. Click on the **Save** button.
6. Add one more **Non-Consumable** In-App purchase using the following data:
   - **Reference Name**: Enter `Double Points`
   - **Product ID**: Enter your bundle ID plus `.iap.doublepoints` (for example, `com.packtpub.coconutfall.iap.doublepoints`)
   - **Cleared for Sale**: Select **Yes**
   - **Price Tier**: Choose any option from the drop-down menu, except the previous one
   - **Language**: Select **English**
   - **Display Name**: Enter `2x Points`
   - **Description**: Enter `Get 2x points for every coconut!`
   - **Hosting Content with Apple**: Select **No**
   - **Review Notes**: Leave this field empty
   - **Screenshot for Review**: Don't provide a screenshot for now
7. Click on the **Save** button. The following screenshot shows what you should see in iTunes Connect:



2 In-App Purchases

Reference Name	Product ID	Type	Apple ID	Status
Double Lives	com.packtp...l.iap.doublelives	Non-Consumable	849676520	● Waiting for Screenshot
Double Points	com.packtp....iap.doublepoints	Non-Consumable	849681200	● Waiting for Screenshot

8. Before leaving iTunes Connect, return to the main menu of iTunes Connect. The simplest way to do this is by simply going to `https://itunesconnect.apple.com`.
9. Click on the **Manage Users** button and select **Test User**. Click on **Add New User** and create a new user. Remember this user's e-mail address and password; this will be the user you will use to make purchases in the sandbox environment.
10. Now you need to log out from your real Apple ID from the App Store if you're logged in. To do this, take the device that you will use to test and go to the **Settings** app. Find the **iTunes & App Store** item and tap on it. In the **iTunes & App Store** view, click on your current **Apple ID** and choose **Sign Out**. Refer to the following screenshot:



What just happened?

We've just added two In-App purchases that the player can buy to get higher scores in the game. Purchasing the `Double Lives` In-App will double the player's lives, and purchasing the `2x Points` In-App will obviously make each coconut give double points.

Time for action – displaying a list of purchases

Before the player can purchase anything, we need to display a list of purchases. Finally, we are going to fill our `ShopScene` class with some content. Just as with Game Center, we're going to create a singleton to manage In-App purchases.

1. Create a new **Objective-C class** called `IAPManager` and make it a subclass of `NSObject`.
2. Open the `IAPManager.h` file and replace its contents with the following code:

```
@import StoreKit;

#define kInAppPoints @"com.packtpub.coconutfall.iap.doublepoints"
```

```
#define kInAppLives  @"com.packtpub.coconutfall.iap.doublelives"

@protocol IAPManagerDelegate

-(void)productsLoaded:(NSArray *)products;

@end

@interface IAPManager : NSObject<SKProductsRequestDelegate>

@property (nonatomic, weak) id<IAPManagerDelegate> delegate;

-(void)retrieveProducts;

+(IAPManager *)sharedInstance;

@end
```

Don't forget to replace the `com.packtpub.coconutfall` part with your bundle ID, and also note that there is no line-break between the name and the value in the `#define` statement. In other words, each `#define` statement takes a single line.

3. Then, open the `IAPManager.m` file and replace its contents with the following code:

```
#import "IAPManager.h"

@implementation IAPManager

-(void)retrieveProducts
{
    //1
    NSSet *products =
      [NSSet setWithArray:@[kInAppLives, kInAppPoints]];

    //2
    SKProductsRequest *productsRequest =
      [[SKProductsRequest alloc]
        initWithProductIdentifiers:products];

    //3
    productsRequest.delegate = self;

    //4
    [productsRequest start];
}

- (void)productsRequest:(SKProductsRequest *)request
     didReceiveResponse:(SKProductsResponse *)response
```

```
{
    [self.delegate productsLoaded:response.products];
}

- (void)request:(SKRequest *)request
     didFailWithError:(NSError *)error
{
    NSLog(@"Request error %@", error.localizedDescription);
}

+(IAPManager *)sharedInstance
{
    static dispatch_once_t pred;
    static IAPManager * _sharedInstance;
    dispatch_once(&pred, ^{ _sharedInstance = [[self alloc] init];
    });
    return _sharedInstance;
}

@end
```

4. Open the `ShopScene.h` file and import the `IAPManager.h` header:

```
#import "IAPManager.h"
```

5. Then, make the `ShopScene` class conform to the `IAPManagerDelegate` protocol:

```
@interface ShopScene : CCScene<IAPManagerDelegate>
```

6. Switch to the `ShopScene.m` file and add the following headers:

```
#import "MenuScene.h"
#import "IAPManager.h"
#import "cocos2d.h"
#import "cocos2d-ui.h"
```

7. Then, add instance variables for a label and a list of items to buy as follows:

```
@implementation ShopScene
{
    CCLabelTTF *_lblLoading;
    CCLayoutBox *_items;
}
```

8. Add the `init` method and methods used to create a background, back button, and a label:

```
-(instancetype)init
{
    if (self = [super init])
    {
        [self addBackground];
        [self addLoadingLabel];
```

```
            [self addBackButton];

            [IAPManager sharedInstance].delegate = self;
            [[IAPManager sharedInstance] retrieveProducts];
        }

        return self;
    }

    -(void)addBackground
    {
        CCSprite *bg =
          [CCSprite spriteWithImageNamed:@"shop_bg.png"];
        bg.positionType = CCPositionTypeNormalized;
        bg.position = ccp(0.5f, 0.5f);
        [self addChild:bg];
    }

    -(void)addLoadingLabel
    {
        _lblLoading =
          [CCLabelTTF labelWithString:@"Loading..."
                             fontName:@"Helvetica"
                             fontSize:48];
        _lblLoading.positionType = CCPositionTypeNormalized;
        _lblLoading.position = ccp(0.5f, 0.5f);
        [self addChild:_lblLoading];
    }

    -(void)addBackButton
    {
        CCSpriteFrame *normal =
          [CCSpriteFrame frameWithImageNamed:@"btn_9slice.png"];
        CCSpriteFrame *pressed =
          [CCSpriteFrame
            frameWithImageNamed:@"btn_9slice_pressed.png"];

        CCButton *btnBack =
          [CCButton buttonWithTitle:@"Back"
                        spriteFrame:normal
             highlightedSpriteFrame:pressed
                disabledSpriteFrame:nil];
        btnBack.block = ^(id sender)
        {
            [IAPManager sharedInstance].delegate = nil;
            [[CCDirector sharedDirector]
              replaceScene:[MenuScene node]];
        };

        btnBack.horizontalPadding = 12.0f;
        btnBack.verticalPadding = 4.0f;

        btnBack.anchorPoint = ccp(0,1);
        btnBack.positionType = CCPositionTypeNormalized;
        btnBack.position = ccp(0.05f, 0.95f);

        [self addChild:btnBack];
    }
```

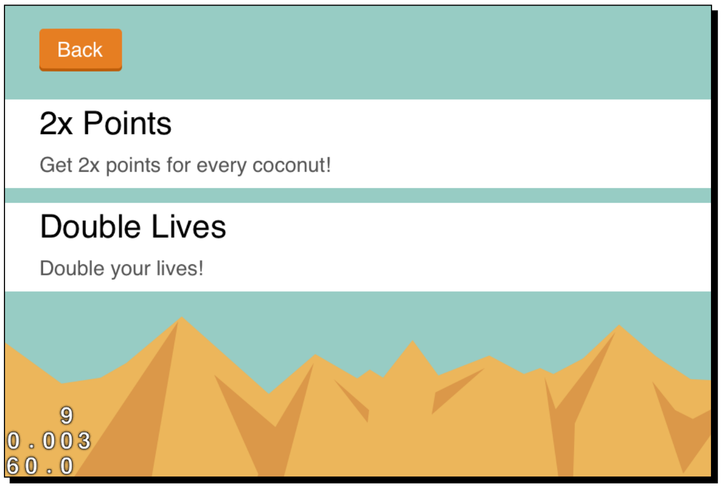
9. Add the `productsLoaded:` method implementation of the `IAPManager` protocol, which is called when the products are loaded:

```
-(void)productsLoaded:(NSArray *)products
{
    //1
    _lblLoading.visible = NO;

    //2
    _items = [CCLayoutBox node];
    _items.direction = CCLayoutBoxDirectionVertical;
    _items.spacing = 10.0f;

    //3
    for (SKProduct *product in products)
    {
        CCNode *item =
          [self createPurchaseItemWithProduct:product];
        [_items addChild:item];
    }

    //4
    [_items layout];
    _items.anchorPoint = ccp(0.5f, 1.0f);
    _items.positionType = CCPositionTypeNormalized;
    _items.position = ccp(0.5f, 0.8);
    [self addChild:_items];
}
```

10. Now, add the `createPurchaseItemWithProduct:` helper method, which simply creates a node with the information (title, description, and so on) of each item that the player can buy:

```
-(CCNode *)createPurchaseItemWithProduct:(SKProduct *)product
{
    CGSize viewSize = [CCDirector sharedDirector].viewSize;
    CCNodeColor *item =
      [CCNodeColor nodeWithColor:[CCColor whiteColor]
                           width:viewSize.width
                          height:60.0f];

    CCLabelTTF *productName =
      [CCLabelTTF labelWithString:product.localizedTitle
                         fontName:@"Helvetica"
                         fontSize:22];
    productName.color = [CCColor blackColor];
    productName.anchorPoint = ccp(0,1);
    productName.positionType = CCPositionTypeNormalized;
    productName.position = ccp(0.05, 0.95);
    [item addChild:productName];

    CCLabelTTF *productDescription =
      [CCLabelTTF labelWithString:product.localizedDescription
                         fontName:@"Helvetica"
                         fontSize:14];
    productDescription.color = [CCColor darkGrayColor];
    productDescription.anchorPoint = ccp(0,1);
    productDescription.positionType = CCPositionTypeNormalized;
    productDescription.position = ccp(0.05, 0.4f);
    [item addChild:productDescription];

    return item;
}
```

11. Build and run the game **on your device**. Tap on the **Shop** button, and you will see a list of possible In-App purchases that we created in iTunes Connect. Refer to the following screenshot:



What just happened?

We just saw a lot of code, but most of it just required us to glue everything together and display the products' list.

Let's start reviewing the code in a chronological order. When the `ShopScene` class is created, it adds a background, a back button, and a **Loading** label in the `init` method. In this way, we show the player that the product list is loading.

Then, it asks the `IAPManager` singleton to load products by calling the `retrieveProducts` method, and sets the `delegate` property to `self` to get notified by the `IAPManager` singleton when the products are loaded. This is why we created the `IAPManagerDelegate` protocol and made the scene conform to it.

Let's review the `retrieveProducts` method, which initiates the product retrieval, to understand why we need to use the `delegate` property and how things work:

1. Here, we created a set of product IDs that we want to retrieve. Most of the time, you will create a set that contains all your product IDs to get information about all the products. However, in some cases, you might want to remove one or two product IDs from the set.

   You might have a server with a web service that returns a set of active product IDs. This way, you can immediately remove some products from the shop on your server by simply not returning them via the web service, without the need to modify anything in iTunes Connect. You can even automate this on your server and create a schedule, such as making one product active only during Christmas day, each year.

2. To retrieve the list of products, you need to use the `SKProductsRequest` class. It takes a set of products IDs and returns information about each product from this set. Note that it will return only products from this set.
3. Products are retrieved asynchronously, so we specified the `delegate` property of the `SKProductsRequest` class. Also, this is why we made the `IAPManager` class conform to the `SKProductsRequestDelegate` protocol.
4. Use the `start` method to start the request.

This request can end up in two methods: in the `request:didFailWithError:` method in the case of an error or in the `productsRequest:didReceiveResponse:` method in the case of success.

Currently, we simply log the error in the console, but in a real game, you should notify the user that the loading of the products failed.

In the `productsRequest:didReceiveResponse:` method, we simply pass the products to the current `delegate` property, which is our `ShopScene` class in this case.

So we need to go back to the `ShopScene` class in the `productsLoaded:` method, which just received the list of available products.

In the `productsLoaded:` method, we placed the code to display the products' list. Let's review it in more detail:

1. The **Loading** label is no longer needed.
2. To display the products, we used the `CCLayoutBox` class.
3. The product item is created in the `createPurchaseItemWithProduct:` method. This method simply creates a `CCNodeColor` node, which is a simple node with a colored background, and adds two labels to that node: the product title and the product description.
4. All product items are laid out vertically and added to the scene.

The result can be seen in the preceding screenshot.

Time for action – making a purchase

Now, when we have a list of products, we can make a purchase. We are going to add the code in the `IAPManager` class to make a purchase and check whether a product has been purchased.

Then, we're going to add a button next to each product in the shop and will be able to actually purchase something. Refer to the following steps:

1. Open the `IAPManager.h` file and add the `purchaseCompleted` method to the `IAPManagerDelegate` protocol:

```
@protocol IAPManagerDelegate

-(void)productsLoaded:(NSArray *)products;

-(void)purchaseCompleted:(BOOL)success;

@end
```

2. Then, make the `IAPManager` class conform to the `SKPaymentTransactionObserver` protocol:

```
@interface IAPManager : NSObject<SKProductsRequestDelegate,
                                SKPaymentTransactionObserver>
```

3. Add the following two methods' declaration to the `IAPManager` class:

```
-(BOOL)isProductPurchased:(NSString *)productIdentifier;

-(void)buyProduct:(SKProduct *)product;
```

4. Switch to the `IAPManager.m` file and add the `_purchasedProducts` instance variable to hold the already purchased products:

```
@implementation IAPManager
{
    NSMutableSet *_purchasedProducts;
}
```

5. Then, add the `init` method to initialize this variable:

```
-(instancetype)init
{
    if (self = [super init])
    {
        _purchasedProducts = [NSMutableSet set];
    }

    return self;
}
```

6. Add the `isProductPurchased:` method, which will return `YES` if a product has already been purchased:

```
-(BOOL)isProductPurchased:(NSString *)productIdentifier
{
    return [_purchasedProducts
              containsObject:productIdentifier];
}
```

7. After that, add the methods shown in the following code:

```
-(void)buyProduct:(SKProduct *)product
{
    if ([self isProductPurchased:product.productIdentifier])
    {
        NSLog(@"You've already purchased this item!");
        return;
    }

    SKPayment *payment =
      [SKPayment paymentWithProduct:product];
    [[SKPaymentQueue defaultQueue] addPayment:payment];
}

- (void)paymentQueue:(SKPaymentQueue *)queue
 updatedTransactions:(NSArray *)transactions
{
    for (SKPaymentTransaction *transaction in transactions)
    {
        switch (transaction.transactionState)
        {
            case SKPaymentTransactionStatePurchased:
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored:
                [self restoreTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed:
                [self failedTransaction:transaction];
                break;
            default:
                break;
        }
    }
}

- (void)completeTransaction:(SKPaymentTransaction *)transaction {
    NSLog(@"Complete transaction: %@",
      transaction.payment.productIdentifier);

    [self
      purchaseSuccess:transaction.payment.productIdentifier];
    [[SKPaymentQueue defaultQueue]
      finishTransaction: transaction];
}

- (void)restoreTransaction:(SKPaymentTransaction *)transaction {
    NSLog(@"Restore transaction: %@",
      transaction.payment.productIdentifier);

    [self
      purchaseSuccess:transaction.payment.productIdentifier];
    [[SKPaymentQueue defaultQueue]
      finishTransaction: transaction];
}

-(void)purchaseSuccess:(NSString *)productIdentifier
{
    [_purchasedProducts addObject:productIdentifier];
    [self.delegate purchaseCompleted:YES];
}

- (void)failedTransaction:(SKPaymentTransaction *)transaction
{
    NSLog(@"Failed transaction: %@",
      transaction.payment.productIdentifier);

    if (transaction.error.code != SKErrorPaymentCancelled)
    {
        NSLog(@"Transaction error: %@",
          transaction.error.localizedDescription);
        UIAlertView *alert =
         [[UIAlertView alloc]
                initWithTitle:@"Purchase failed!"
                 message:transaction.error.localizedDescription
                     delegate:nil
            cancelButtonTitle:nil
            otherButtonTitles:@"OK", nil];

        [alert show];
    }

    [[SKPaymentQueue defaultQueue]
      finishTransaction: transaction];

    [self.delegate purchaseCompleted:NO];
}
```

8. Open the `ShopScene.m` file and add the following code to the end of the `createPurchaseItemWithProduct:` method to add a button next to each product item:

```
-(CCNode *)createPurchaseItemWithProduct:(SKProduct *)product
{
    //..skipped..

    //1
    BOOL purchased =
      [[IAPManager sharedInstance]
        isProductPurchased:product.productIdentifier];

    if (purchased)
    {
        //2
        CCLabelTTF *purchased =
         [CCLabelTTF labelWithString:@"Purchased!"
                             fontName:@"Helvetica"
                             fontSize:18];
        purchased.color = [CCColor grayColor];
        purchased.anchorPoint = ccp(1, 0.5f);
        purchased.positionType =
          CCPositionTypeNormalized;
        purchased.position = ccp(0.95f, 0.5f);
```

```
            [item addChild:purchased];
        }
        else
        {
            //3
            NSNumberFormatter *formatter =
              [[NSNumberFormatter alloc] init];
            [formatter setNumberStyle:
              NSNumberFormatterCurrencyStyle];
            [formatter setLocale:product.priceLocale];
            NSString *price = [formatter
              stringFromNumber:product.price];

            //4
            CCSpriteFrame *normal =
              [CCSpriteFrame
                frameWithImageNamed:@"btn_9slice.png"];
            CCSpriteFrame *pressed =
              [CCSpriteFrame frameWithImageNamed:
                @"btn_9slice_pressed.png"];
            CCButton *btnPurchase =
              [CCButton buttonWithTitle:price
                          spriteFrame:normal
                 highlightedSpriteFrame:pressed
                    disabledSpriteFrame:nil];
            btnPurchase.horizontalPadding = 12.0f;
            btnPurchase.verticalPadding = 12.0f;
            btnPurchase.anchorPoint = ccp(1, 0.5f);
            btnPurchase.positionType =
              CCPositionTypeNormalized;
            btnPurchase.position = ccp(0.95f, 0.5f);
            [item addChild:btnPurchase];

            //5
            btnPurchase.userObject = product;

            //6
            [btnPurchase
              setTarget:self
               selector:@selector(onPurchaseTap:)];
        }

        return item;
    }
```

9. Then, add the following two methods to initiate and handle the purchase:

```
-(void)onPurchaseTap:(CCButton *)btn
{
    //1
    SKProduct *product = btn.userObject;

    //2
    [_items removeFromParent];

    //3
    _lblLoading.string = @"Purchasing...";
    _lblLoading.visible = YES;

    //4
    [[IAPManager sharedInstance] buyProduct:product];
}

-(void)purchaseCompleted:(BOOL)success
{
    if (success)
    {
        _lblLoading.string = @"Refreshing products...";
        [[IAPManager sharedInstance] retrieveProducts];
    }
    else
    {
        _lblLoading.string = @"Purchase failed!";
    }
}
```

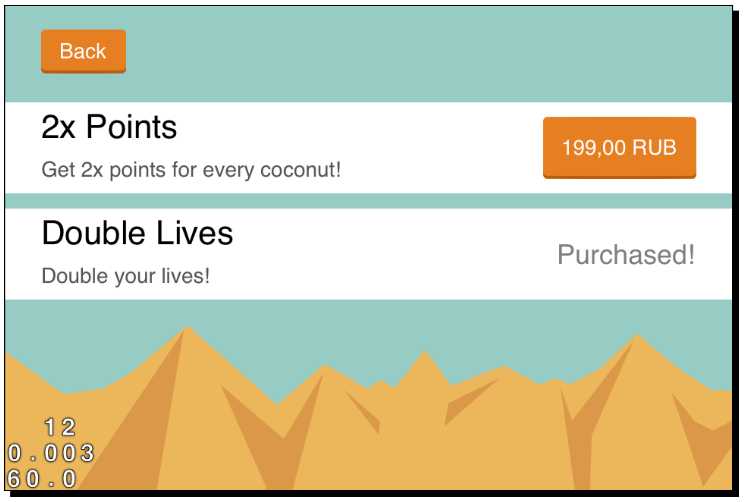
10. Open the `AppDelegate.m` file and import the `IAPManager.h` header as follows:

```
#import "IAPManager.h"
```

11. Then, add the following line at the start of the `application:didFinishLaunchingWithOptions:` method:

```
-(BOOL)application:(UIApplication *)application
   didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [[SKPaymentQueue defaultQueue]
       addTransactionObserver:
         [IAPManager sharedInstance]];

    //..skipped..
}
```

12. Build and run the game. Open the shop scene and purchase one of the items by clicking on the orange button on the right-hand side. Refer to the following screenshot:



Don't forget to use the Apple ID of the test user that we created earlier in this chapter.

What just happened?

Once again, let's review how the code works in a chronological order.

Initiating the purchase

To allow the player to purchase a product, we added a button next to each product in the `ShopScene` class. Let's review the code we added to the `createPurchaseItemWithProduct:` method to create this button:

1. First of all, we needed to know whether the item had already been purchased. To do this, we used the `isProductPurchased` method of the `IAPManager` class, which we'll review later. For now, all we need to know is that it returns `YES` for already purchased items and `NO` if the item is not yet purchased.
2. If the item is already purchased, the **Purchased!** label is displayed instead of the button because the player already purchased this item.

3. If this item had not yet been purchased, we used `NSNumberFormatter` to format the price of `SKProduct`, which is contained in the `price` property, using the `priceLocale` property. This will display the price on the button in the player's locale.
4. This code simply creates and positions the orange button on the right-hand side.
5. Here, we used the `userObject` property to store the reference to the initial `SKProduct` product so that we could take it later in the button-tap handler method.
6. Set the `onPurchaseTap:` method to be called when the button is pressed.

When the player taps the button, the `onPurchaseTap:` method is called. In this method, we get the `SKProduct` product saved in the `userObject` property of the button.

Then, we remove the items from the screen and return the `_lblLoading` label by making it visible, but this time, we change its text to **Purchasing**.

We pass the product to the `buyProduct:` method of the `IAPManager:` class. This means it is time to review the code of the `buyProduct:` method.

The `buyProduct:` method checks whether the product has not already been purchased and places `SKPayment` in the default payment queue to initiate the payment:

```
SKPayment *payment = [SKPayment paymentWithProduct:product];
[[SKPaymentQueue defaultQueue] addPayment:payment];
```

It will ask the player to enter the Apple ID and will ask whether the player really wants to buy this product.

Handling the purchase result

However, we didn't set any delegates or something like this to get notified about the purchase result. How do we know when the payment is completed or canceled?

To be notified, we added this line to the `AppDelegate` class of the `application:didFinishLaunchingWithOptions:` method as follows:

```
    [[SKPaymentQueue defaultQueue]
      addTransactionObserver:[IAPManager sharedInstance]];
```

This way, we let the payment queue know that we want it to notify the `IAPManager` singleton about all transactions. This is also why we made the `IAPManager` class to conform to the `SKPaymentTransactionObserver` protocol.

After we subscribed to receive all the news about transactions, the `paymentQueue:updatedTransactions:` method is called when the transaction is updated.

The following is the code for this method:

```
- (void)paymentQueue:(SKPaymentQueue *)queue
 updatedTransactions:(NSArray *)transactions
{
    for (SKPaymentTransaction *transaction in transactions)
    {
        switch (transaction.transactionState)
        {
            case SKPaymentTransactionStatePurchased:
                [self completeTransaction:transaction];
                break;
            case SKPaymentTransactionStateFailed:
                [self failedTransaction:transaction];
                break;
            case SKPaymentTransactionStateRestored:
                [self restoreTransaction:transaction];
            default:
                break;
        }
    }
}
```

As you can see, we simply enumerate all the updated transactions and call the corresponding method for each transaction state. We have a total of three methods that can be called: `failedTransaction:`, `completeTransaction:`, and `restoreTransaction:`.

The `completeTransaction:` method is called when the player successfully purchases the product. The `restoreTransaction:` method is called when the player restores their old purchase, for example, after reinstalling the game or on another device.

We've added the `restoreTransaction:` method here just to handle all transaction states. However, we will actually use it a bit later in this chapter.

Both these methods mean that the player owns this item. So in both cases, we call the `purchaseSuccess:` method, which simply adds the purchased product to the `_purchasedProducts` set and notifies the `delegate` property.

The `failedTransaction:` method is called if an error occurred during the purchase process. In this case, we display an alert view with the error.

We don't display the alert view if the player cancelled their purchase, since this is not an error from the player's point of view.

After displaying the error, we notify the `delegate` property, but this time, we pass `NO` as an argument to indicate that the purchase completed with an error.

Before going back to the `ShopScene` class, there is one more important moment. Note that all the three methods (`failedTransaction:`, `completeTransaction:`, and `restoreTransaction:`) call the `finishTransaction:` method to remove the transaction from the queue. Refer to the following code:

```
[[SKPaymentQueue defaultQueue] finishTransaction: transaction];
```

By removing the transaction from the queue, we tell Store Kit that we processed the transaction independently and whether it was completed successfully or failed. If we don't remove the transaction from the queue, we will get notified about this transaction again and again every time the game launches.

Updating the product status in the ShopScene class

Back to the code. We stopped at the moment when the `purchaseCompleted:` method of the `delegate` property was called. Let's return to the `ShopScene.m` file and review the `purchaseCompleted:` method:

```
-(void)purchaseCompleted:(BOOL)success
{
    if (success)
    {
        _lblLoading.string = @"Refreshing products...";
        [[IAPManager sharedInstance] retrieveProducts];
    }
    else
    {
        _lblLoading.string = @"Purchase failed!";
    }
}
```

As you can see, it is quite simple. In the case of success, we display the **Refreshing products** label and re-retrieve the products. However, this time, the `isProductPurchased:` method in the `createPurchaseItemWithProduct:` method will return `YES` as follows:

```
BOOL purchased = [[IAPManager sharedInstance]
                    isProductPurchased:product.productIdentifier];
```

This means that instead of the button to purchase the product, we will add a **Purchased!** label.

In the case of a purchase failure, we simply change the label to **Purchase failed!** and do not re-retrieve the products.

> Of course, this is not what you need to do in the real game, but for simplicity's sake, we're just leaving the player with this level.

Saving and restoring In-App purchases

If the player restarted the game right now and re-entered the shop, the player would be shocked! All the purchased items are not marked as **Purchased!** anymore. I can almost hear the player think – "Do I need to pay again?".

The good thing is that the answer is no. If the player tries to buy an already purchased item, there will be a message saying that the player already owns this and this one will be free.

However, not all players know this, so it is better for us to just store all the purchased items in a persistent storage to keep the purchase status between the game restarts.

Another good idea would be to add a **Restore Purchases** button that the player can tap and restore all purchases after reinstalling the game or launching the game on another device.

Time for action – storing purchases

Let's start with saving the purchased items between the game launches. We will simply store all the purchased products in the user defaults using the `NSUserDefaults` class, just as we did with our audio settings in *Chapter 8*, *Adding Sound Effects and Music*.

In iOS 7, you can persist purchases using App Receipts, but we'll write some code to work on iOS 5 or higher, since iOS 6 will be here for some time.

If you wish to know more about App Receipts, please go to this link `https://developer.apple.com/library/ios/releasenotes/General/ValidateAppStoreReceipt/Introduction.html#//apple_ref/doc/uid/TP40010573` (shortened link: `http://bit.ly/1l9urhe`).

Perform the following steps:

1. Open the `IAPManager.m` file and add a `#define` statement for a key that we'll use to store products in user defaults:

```
#define kUserDefaultsIAPKey @"IAP_USER_DEFAUTLS_KEY"
```

2. Add the following code to the end of the `purchaseSuccess:` method:

```
-(void)purchaseSuccess:(NSString *)productIdentifier
{
    [_purchasedProducts addObject:productIdentifier];
    [self.delegate purchaseCompleted:YES];

    [[NSUserDefaults standardUserDefaults]
       setObject:[_purchasedProducts allObjects]
```

```
            forKey:kUserDefaultsIAPKey];

        [[NSUserDefaults standardUserDefaults]
            synchronize];

    }
```

It is important to put this code after you add the product to the `_purchasedProducts` set.

3. Make the following changes to the `init` method:

```
-(instancetype)init
{
    if (self = [super init])
    {
        NSArray *tempArray =
         [[NSUserDefaults standardUserDefaults]
            objectForKey:kUserDefaultsIAPKey];

        if (tempArray)
            _purchasedProducts =
              [NSMutableSet setWithArray:tempArray];
        else
            _purchasedProducts = [NSMutableSet set];
    }

    return self;
}
```

4. Run the game on your device. Make a few purchases and then restart the game. You will see them marked as purchased right after the game starts.

What just happened?

When the player purchases something, we put this product identifier in the `_purchasedProducts` set and save this set into `NSUserDefaults`. Then, when the singleton is created when the game starts, we load this set and know which products are already purchased.

We need to save and load the set as `NSArray` because you cannot store `NSSet` in `NSUserDefaults`. We could archive it manually, but I find it easier to just convert it to `NSArray` while storing, and create a set using an array while loading.

Time for action – restoring purchases

We solved the issue with missing purchases after the game restarts. However, if the player removes and reinstalls the game, it will remove the user defaults, so we need another way to restore purchases. Fortunately, it is quite easy to do. Perform the following steps:

1. Open the `IAPManager.h` file and add the `purchasesRestored:` method to the `IAPManagerDelegate` protocol:

```
-(void)purchasesRestored:(BOOL)success;
```

2. Then, add the `restorePurchases` method to the `IAPManager` class itself:

```
-(void)restorePurchases;
```

3. Switch to the `IAPManager.m` file and add the following methods:

```
-(void)restorePurchases
{
    [[SKPaymentQueue defaultQueue]
      restoreCompletedTransactions];
}

-(void)paymentQueue:(SKPaymentQueue *)queue
   restoreCompletedTransactionsFailedWithError:(NSError *)error
{
    NSLog(@"Restoring purchases failed: %@", error);
   [self.delegate purchasesRestored:NO];
}

-(void)paymentQueueRestoreCompletedTransactionsFinished:
   (SKPaymentQueue *)queue
{
    [self.delegate purchasesRestored:YES];
}
```

4. Now we need to add a button to restore purchases. Open the `ShopScene.m` file and add the `addRestoreButton` method:

```
-(void)addRestoreButton
{
    CCSpriteFrame *normal =
      [CCSpriteFrame frameWithImageNamed:@"btn_9slice.png"];
    CCSpriteFrame *pressed =
      [CCSpriteFrame
        frameWithImageNamed:@"btn_9slice_pressed.png"];

    CCButton *btnRestore =
      [CCButton buttonWithTitle:@"Restore Purchases"
                    spriteFrame:normal
         highlightedSpriteFrame:pressed
```

```
            disabledSpriteFrame:nil];

    btnRestore.block = ^(id sender)
    {
        _lblLoading.visible = YES;
        _lblLoading.string = @"Restoring...";
        [_items removeFromParent];

        [[IAPManager sharedInstance] restorePurchases];
    };

    btnRestore.horizontalPadding = 12.0f;
    btnRestore.verticalPadding = 4.0f;

    btnRestore.anchorPoint = ccp(1,1);
    btnRestore.positionType = CCPositionTypeNormalized;
    btnRestore.position = ccp(0.95f, 0.95f);

    [self addChild:btnRestore];
}
```

5. Add a call to the `addRestoreButton` method in the `init` method in order to add the button to the scene:

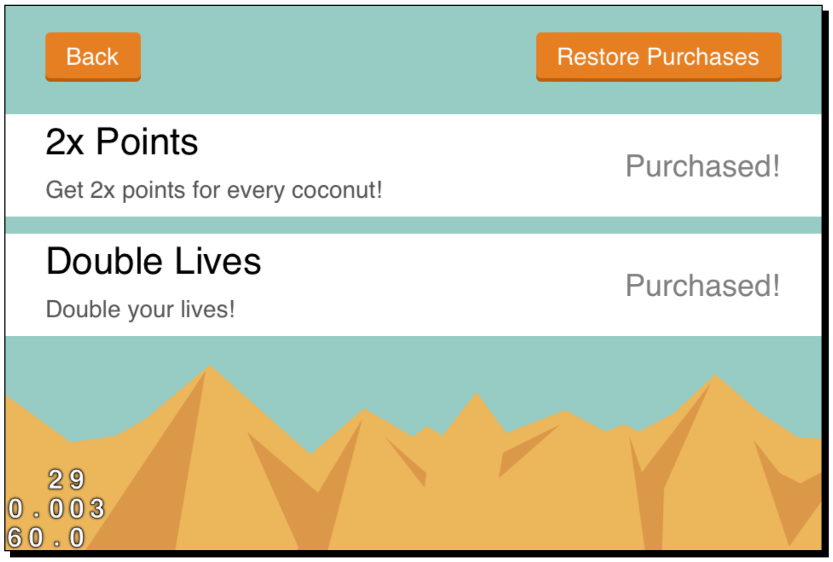
```
-(instancetype)init
{
    if (self = [super init])
    {
        [self addBackground];
        [self addLoadingLabel];
        [self addBackButton];
        [self addRestoreButton];

        //..skipped..
    }

    return self;
}
```

6. Then, we need to implement the `purchasesRestored:` method of the `IAPManagerDelegate` protocol. Add the following method to the `ShopScene` class:

```
-(void)purchasesRestored:(BOOL)success
{
    if (success)
    {
        _lblLoading.string = @"Refreshing products...";
        [[IAPManager sharedInstance] retrieveProducts];
    }
```

```
        else
        {
            _lblLoading.string = @"Restore failed!";
        }
    }
```

7. Remove the game from your device before running it. Then, build and run the game and open the shop scene. Tap on the **Restore Purchases** button and wait until all purchases are restored. Refer to the following screenshot:



What just happened?

To restore purchases, we added the `restorePurchases` method to the `IAPManager` class and the `purchasesRestored` method to the `IAPManagerDelegate` protocol to notify the `ShopScene` class when we finish restoring the purchases.

The process of a purchase restoration is quite easy. All we need to do is call the `restoreCompletedTransactions` method of the `SKPaymentQueue` class:

```
[[SKPaymentQueue defaultQueue] restoreCompletedTransactions];
```

It will restore all transactions and call `paymentQueue:updatedTransactions:` for each restored transaction with `SKPaymentTransactionStateRestored`, which will further lead to calling `purchaseSuccess:` and storing the purchase in the `_purchasedProducts` set.

> It is important to understand that `purchaseSuccess:` is called for every restored transaction, just as though the player just bought it, so we reuse the code to save purchases in `NSUserDefaults`.

However, we want one place where we can understand when all purchases were restored or failed. This is why we added two more optional methods from the `SKPaymentTransactionObserver` delegate.

The `paymentQueue:restoreCompletedTransactionsFailedWithError:` method is called if there is an error during purchases restoration. In this case, we simply log the error in the console and notify the delegate that there was an error:

```
-(void)paymentQueue:(SKPaymentQueue *)queue
restoreCompletedTransactionsFailedWithError:(NSError *)error
{
    NSLog(@"Restoring purchases failed: %@", error);
    [self.delegate purchasesRestored:NO];
}
```

In case of success, we simply notify the delegate that purchases are successfully restored. Refer to the following code:

```
-(void)paymentQueueRestoreCompletedTransactionsFinished:
  (SKPaymentQueue *)queue
{
    [self.delegate purchasesRestored:YES];
}
```

To let the player restore purchases, we added a button to the shop scene, which simply calls the `restorePurchases` method of `IAPManager`, removes the products' list, and displays the **Restoring** label.

Then, when the purchases are either restored or failed, the `purchasesRestored:` method of the `ShopScene` class is called. If purchases are successfully restored, we reload the products' list, but this time, the already purchased products will be marked as **Purchased!**.

> Just as with Game Center integration, we only reviewed the basics of working with Store Kit. In your real game, you might want to check whether there is an Internet connection, provide timeout for the purchase, and of course create a better user interface.
>
> Also, there are many different libraries that solve a lot of problems that you might face when working with Game Center and Store Kit. So maybe you can search for some that will suit your needs and use them. However, I believe that understanding how things work internally is still very important.

Affecting the game

When the player purchases something in the game, the player expects to get something in return. Of course, having a **Purchased!** label displayed next to each purchase is great, but it is definitely not enough.

In this short section, we're going to add the code to affect the gameplay when the player purchases something.

Time for action – doubling the lives and points

What we need to do is to add a small piece of code to the game scene and double the initial number of lives and points per coconut if the player purchased the corresponding item in the shop.

1. Open the `GameScene.m` file and import the `IAPManager.h` header:

   ```
   #import "IAPManager.h"
   ```

2. Then, add the following code to the end of the `setupGameDefaults` method:

   ```
   -(void)setupGameDefaults
   {
       _gameState = GameStateInit;
       _timeUntilNextCoconut = 0;

       _lives = 3;
       _points = 0;
       _pointsPerCoconut = 5;

       if ([[IAPManager sharedInstance]
              isProductPurchased:kInAppPoints])
           _pointsPerCoconut *= 2;

       if ([[IAPManager sharedInstance]
              isProductPurchased:kInAppLives])
           _lives *= 2;
   }
   ```

3. Run the game. Make sure you've purchased both items in the shop and start the game. Now, you should have 6 lives and receive 10 points per coconut.

What just happened?

This short piece of code simply checks whether the player purchased a corresponding item in the shop and doubles the points or lives.

Some might say that it is unfair, but this is only a demonstration. I'm sure you can come up with better ideas for your In-App purchases.

Useful resources

There are some truly-gifted people who can write code, create graphics, and even write music for their games. However, if you're like me, and my drawings can be used to frighten kids, then you will need some resources for your games.

Here is a small list of resources that can provide you with graphic assets, sound effects, and music.

Graphics

For free sprites, backgrounds, and animations, check out the following websites:

- `opengameart.org`: This will provide you with lots of graphics, sound, and music resources
- `widgetworx.com/spritelib/`: This will provide you with a collection of static and animated sprites
- `hasgraphics.com/free-sprites/`: This will provide you with sprites that are free to use for commercial and non-commercial projects
- `teh_pro.webs.com/sprites.htm`: This will provide you with free sprites; no permissions are required
- `spritedatabase.net`: This will provide you with sprites, background images, and other resources
- `lostgarden.com/search/label/free%20game%20graphics`: This will provide you with lots of free game graphics
- `reinerstilesets.de`: This will provide you with tilesets and sprites mostly for isometric games
- `pixelprospector.com/indie-resources/`: This will provide you with a huge list of other websites, resources, and tutorials

Sound effects

To get sound effects for your games, see the following websites:

- `freesfx.co.uk`: This will provide you with a great categorized list of sound effects
- `freesound.org`: This will provide you with a huge collection of sound effects
- `partnersinrhyme.com`: This will provide you with royalty-free music and sound effects
- `bfxr.net`: This will provide you with a sound effects generator

Music

The list of websites that provide music that you can use in your games is as follows:

- `incompetech.com`: This will provide you with great royalty-free music
- `jamendo.com`: This will provide you with music tracks uploaded by artists
- `freemusicarchive.org`: This will provide you with an interactive library of legal audio downloads directed by the WFMU radio station
- `nosoapradio.us`: This will provide you with lots of old-school music tracks

Fonts

The list of websites that provide True Type and bitmap fonts, which you can use in your games, is as follows:

- `dafont.com`: This will provide you with freeware, shareware, demo versions, or public domain fonts, including bitmap fonts
- `fontsquirrel.com`: This will provide you with free and almost-free fonts
- `openfontlibrary.org`: This will provide you with an open fonts library with high-quality fonts
- `theleagueofmoveabletype.com`: This will provide you with only the most well-made, free, and open source fonts

Summary

This is the final chapter of this book. In this final chapter, we reviewed features that are not directly related to Cocos2D. However, your game can definitely benefit from using Game Center and In-App purchases.

From the previous chapters, you learned all the basics of the process of making games with Cocos2D. Now, you know how to render and manipulate sprites, handle touches, use actions, animate your sprites, use particle systems for cooling effects, add sound and music, use physics, and even to create a user interface.

However, this is only the beginning of your way as a game developer. It is simply impossible to fit everything you need to know in one book. And not the least role in it plays the fact that you need to build on your own experience in addition to following the steps from the books or reading theoretical materials.

I recommend that you put your knowledge into action and practice creating games. Creating a simple game and publishing it will give you an experience that cannot be overrated.

You don't need to try to create a megahit with your first attempt, since there is a very small chance that you will do this with your first game. Just practice and success will come.

While practicing, you might benefit from reading some books in cookbook format, which contain short recipes that assume you know all the basics and can put the remaining code around them. Also, there are several resources that can help you if you get stuck:

- `cocos2d-iphone.org`: This is the official Cocos2D-iPhone website, and there is a great forum with a very responsive community
- `kirillmuzykov.com/forums`: This is where I've set up a forum where you can ask questions related to this book and I'll try to answer them as soon as I can
- `raywenderlich.com`: This website is full of great step-by-step tutorials for beginners and intermediate developers
- `stackoverflow.com`: This is the best questions-and-answers website, although it is not solely related to Cocos2D
- `spritebuilder.com`: This has a forum that will be useful when you start using SpriteBuilder

Good luck!