Learning Unreal® Engine iOS Game Development

Unreal® Engine 4 has always been the best measurement for industry standards. With all its tools and power, game developers and designers can reach a new level of productivity for their games with the lowest cost ever.

Learning Unreal® Engine iOS Game Development teaches you the basics of iOS game development using Unreal® Engine 4. You will discover how to build an engaging Unreal iOS game, how to generate revenue, and how to optimize game performance using the tools and functionalities the Engine provides. To begin, you will start by preparing the required files for an iOS game using the developer’s portal. Next, you will go through purchasing, installing, and building game elements for different game types, picked from the chart of the latest trending games. By the end of the book, you will be able to add social and monetization functionalities to your game and submit it for review to the app store.

Who this book is written for

If you are a game developer, designer, artist, or a beginner in the gaming industry, and want to make iOS games efficiently at a low cost, this book is ideal for you.

What you will learn from this book

- Install, build, and compile UE4 on Mac OSX and prepare it for iOS development
- Understand the process of creating an iOS game and its requirements
- Build different types of modern gameplay mechanics for iOS devices
- Implement and build different asset types
- Control your game’s user interaction using the social functionalities inside UE4
- Develop iOS games using tools and functionalities provided by Unreal Engine 4.x
- Generate revenue from your games using the monetization capability

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 8 'iOS Debugging and Optimization'
- A synopsis of the book’s content
- More information on Learning Unreal® Engine iOS Game Development

About the Author

Muhammad A.Moniem started in the industry at a very early age. He taught himself everything related to the game development process even before he joined college. After being a software engineer, he started to teach himself the art of game design and game art techniques. As a self-taught person, he was able to find his way into the industry very easily, which led him to be hired for big, medium, and small companies, titles, and teams. Throughout his career, he was able to contribute as a full-time or part-time employee, or freelancer on games for a wide range of platforms, including Windows, Mac, iOS, Android, PS4, XBOXOne, and OUYA; he has also worked with technologies such as VR, AR, or Kinect. Finally, he was able to establish his own one-person game company/team as a part-time independent developer. Lots of his indie games got recognition or have been finalists in international indie game events such as IGF, Indie Showcase, IGC, and Tokyo Game Show. He has also designed an amazing website, www.mamoniem.com.
Acknowledgements

I would like to thank all those who helped me make this book possible. A special mention to all the team members at Epic Games for their hard work on Unreal® Engine and for the great decision of making the engine available to everyone by sharing the source code. Without their hard work and tough decisions, this book would not exist. I would like to thank them for their support along the way since I started using Unreal®, and all the people I knew at Epic were supportive and interested in my activities with the engine.

I would like to thank all the people at Packt Publishing who worked on this book. They helped me a lot and were patient when I got out of my schedule because of my personal obligations. A special thanks to the technical reviewers; they constantly checked my errors and omissions. This book has become better because of them; thanks to all their hard work. Richard, Arwa, Rebecca, and Faisal; thank you for bringing me on board at Packt and helping me put the book together.

Last but not least, I want to express my pleasure of having Lamiaa, Yusuf, and Mariam in my life. You are all my gifts. You are a very supportive family; thanks for giving me the best environment to write this book, and thanks for believing in all my projects. Father and mother, thanks for teaching me how to learn on my own; it helped me a lot to learn and pass on the knowledge.
Learning Unreal® Engine iOS Game Development

This book is meant to help you going into the process of creating an iOS game from scratch using one of the best and oldest engines in the industry—Unreal® Engine 4.

Although the engine was made from the beginning to make AAA games for PC and console, it has recently acquired the market of casual and mobile games with lots of power and creative tools.

During the course of this book, we will go in depth into how to make several casual game types that look a lot like the top selling and profiting games on the App Store.

What This Book Covers

Chapter 1, Prepare to Make Unreal Games with Unreal® Engine – Installing and Setting Up, will take you through the process of preparing your environment to develop an iOS game using Unreal® Engine 4. At the end of the day, making an iOS game is completely different from making any other game, even if from other mobile devices.

Chapter 2, Methods and Tools to Create Your Games, discusses the different ways in which we can make a game using Epic's technology, what is the best, and why.

Chapter 3, Creating a Brick Breaking Game, contains the first game to be made in this book. Physics is one of the most important topics in any game engine, so in this chapter, will be making a game based on physics to understand how it works in Unreal® Engine.

Chapter 4, Advanced Game Content Generation with a Fruit Chopper Game, is the second game project in this book and will feature how to make a simple 3D game, which has more animations, effects, and complex swipe inputs.

Chapter 5, Building an Exciting Endless Runner Game, admits that the most famous genre in mobile games in the last few years was the endless runner games. So, we will be taking the ride of making a runner game with a randomly generated level.

Chapter 6, Designing an Advanced Game, tells you what a game is without enemies. All the past examples were based on scoring with some obstacles, but let's add some enemies in that chapter to interact with, in a simple touch-screen-friendly platformer game.

Chapter 7, Monetizing Your Game, will cover the easiest way to get profit from your free games. There are a few techniques that can be added to any mobile/iOS game to make it bring you money.
Chapter 8, *iOS Debugging and Optimization*, helps you understand why debugging is considered as a top priority when it comes to mobile devices, the pros of finding and fixing bugs, various debugging tools and techniques, and how to optimize performance.

Chapter 9, *Publishing*, will take you through the process of building and submitting the game to the store review because a game is not a game without an audience. This chapter is the final destination for any game release.

Appendix, *Nodes Database*, gives you a full database with explanation for each node we have used during the process of making all the four game examples. It will be helpful for you to start improvising and making your own unique games.
Debugging has always been a strong topic when it comes to game development. With every development, there is a debugging process to help the product's performance and push it to the best, make it as bug-free as possible, and remove the performance issues. However, when it comes to games and especially mobile games, debugging is the number one priority topic as these devices are not usually powerful enough to handle some aspects of the game.

By the end of this chapter, you will:

- Learn how to use the editor in different ways to debug your games
- Understand the different ways in which XCode can help in debugging your games
- Learn how to optimize the game performance
- Learn how to minimize the game size

**Blueprints Live view**

What could be better than having a visual representation of the game logic during runtime! Using blueprints, you guarantee that when it comes time to find a bug, it will be a very quick process.
Being able to see the code execution process, which is something that is running in the background, but you didn't think about it. If you have a dual screen or even if you just have a big screen and you can shrink your window size, try to put the viewport alongside the blueprint view. Then by pressing the play button, you will see a visual representation with flashing red and yellow lines into and out of the nodes that are being executed right now. Like **Event Begin Play**, when the game starts, it shows you the execution of its **SET** integer node.

![Blueprint View](image)

Usually, when you have a bug, it is because there is something that has not been executed or has been executed in the wrong way or with wrong values. Using this method is simple and easy. You can see if a specific node has been executed or not, and if it was executed, in which order or with what values.
Printing messages

Printing messages to the console has always been a quick and easy way to check the occurrence of a function or perhaps the changes in a variable over time. Unreal Engine has a very advanced **Print String** node. When you use the node for the first time, it appears that it'll be printing the request **In String** value to the console.

![Print String node](image)

However, if you press the little down arrow at the bottom of the node, it will show you more options. These options give you the chance to print this message to the console using **Print to Log**, print to the player game view using **Print to Screen**, or even use both!

The **Text Color** option is the color of the printed text on the screen. This only works if you have selected the **Print to Screen** option; otherwise it is useless as the console always prints messages in one color—black!
To be able to see the console/log messages during your development process, the best way is to watch it through the **Output Log** window that you can access from the **Window** panel.

The **Output Log** screen not only prints messages from your blueprints' logic, but also prints messages from the internal engine core. It is also a good place to follow the progress of building and packaging a playable version of the game. The **Output Log** screen is a log for anything related to your Unreal Engine project—not only code and engine-related stuff, but anything else too. It is always a good place to check for the cause of crashes or building and packaging failures.
The **Output Log** window has its own tagging system, and any printed line will have a tag for itself. For example, a printed message from a **Print String** node in your blueprint will usually start with **LogBlueprintUserMessages**:

### Breakpoints

As the name implies, a breakpoint is a point where the game will be forced to break for a while. If you are from a programming background, then you definitely know what breakpoints are.

A breakpoint is something you add to your logic to ask the game/app to pause the execution process once it reaches this point. To add a breakpoint to a certain node, all that you need to do is right-click on the node itself and choose **Add Breakpoint**.
Once you finish adding the breakpoint, the node will be tagged with the breakpoint icon. However, the first time, you’ll see that the breakpoint icon looks strange.

The icon shown in the preceding screenshot means that you’ve successfully added the breakpoint to the node. However, it will not be effective until you compile the blueprint logic. Once you compile your blueprint, the breakpoint icon will change to show the usual red icon.

Now you have a breakpoint in the game that should take place once this **Destroy Actor** node has been called. While running the game, when this node is executed, you'll find:

- The game pauses
- The blueprint opens on that node and marks it with giant red arrow
- The app shows the flow input to the current breakpoint node
Now comes the most interesting part of debugging. By opening the Blueprint Debugger window from the Window toolbar, you will have access to the Blueprint Debugger.
This window gives you a lot of information about the execution of the nodes and how long it takes. It also shows you an execution trace for all of the nodes and their execution order. It also shows you the nodes tagged by blueprint and you can use this window as a quick way to disable or enable breakpoints.
Once you find the issue, you need to get rid of the breakpoint. All that you need to do is right-click on the node again and select **Remove Breakpoint**.

You can also remove the breakpoint but keep a reference for it. Let's say this node was often a cause of different issues and you want to remember that. So you could choose **Toggle breakpoint** and make it nonfunctional, but it will still be marked as a breakpoint entity. So once you see it, you'll know that it is a switched off breakpoint that is not active right now and you can re-enable it whenever you want.
If you have lots of breakpoints, usually while tracking a very serious bug, you'll keep adding breakpoints everywhere in all of the blueprints. However, once you fix the problem and need to clean your logic of those breakpoints, it does not make any sense to open the blueprints one-by-one and manually remove them. That's the reason behind adding the Disable All Breakpoints and the Delete All Breakpoints options in the Debug menu. Also, once you use Disable All Breakpoints, the Enable All Breakpoints option will be enabled to give you the opportunity to reactivate all of them again if you want.

XCode tools
As long as you are developing for iOS or OS X, you must use XCode for one task at least. Although Unreal Engine can directly give you a running game in your device or a final cooked IPA, you can still run the game via XCode.

Any game that is cooked using Unreal Engine either for direct play on the device or an IPA as a result of a project generated and compiled by XCode.

To find this autogenerated project, you can just browse your Unreal Engine's project directory and find it alongside the Unreal project file. It is usually named with the same name.
When you open this project, you will find it is set to **UE4CmdLineRun** by default. Usually, it will not run once you try to build in your device. So first you need to change the scheme to **UE4Game - iOS** and then choose your connected device from the list.

![Scheme Selection](image)

**Debug Navigator**

To access the debugging tools, you need to click the sixth icon on the left-hand side panel icon bar, which usually shows a Show the Debug Navigator tool tip, and this will give you direct access to the debugging tools.

Understand that accessing this panel without running a game on the device will show you a totally empty panel. You must have a running game on the device to start seeing the tools shown here:

![Debug Navigator](image)

The first part called **UE4Game** has general information about the game. It is not much, but it is useful anyway. It gives you the **PID** value, which is the process ID of **601**, and the current state of this process, which is obviously **Running**.
The second row of the toolset is the **CPU**. Accessing it will show you how many threads the device is running at the moment, the performance of each thread, and the utilization of each running process. Because you are running **UE4Game**, it will be the main focus here; any other running processes on the device will be listed in **Other Processes**. The amount of free CPU capacity will be listed in **Free percentage**.

The third row is **Memory** and it looks a lot like the **CPU** tool; in fact, it works in the same way, and the only difference is that the memory tool measures the memory capacity in MB.
The last row is meant for the **FPS** measurement. **FPS** is short for **frames per second**. A good game should be running on 60, but since complex games usually can’t reach this frame rate, 30 is a good frame rate. The highest frame rate is the best and the lowest frame time is the best as well, because a low frame time would mean a high frame rate. As you can see, the frame rate is **60** and the frame time is **6.0ms**, which means rendering the frame took 6 milliseconds from the device. This is very fast. Usually, its value would be something near 15 to 20 milliseconds for more complex games.
Capturing frames

You can also use the **Capture OpenGL ES Frame** option from the **Debug** menu, which will freeze the game for a while and totally change the content of the **Debug** menu to accommodate the new debugging environment. Finally, it shows you a frame debugger that contains all of the draw calls for OpenGL ES where you can see the construction and the rendering pipeline of the chosen frame. Also, you can check the used texture, shaders, and objects one by one. It is a more advanced tool, but it is very useful to use when you want to really understand how a frame gets rendered step by step.

From the changed **Debug** menu, you can go back and forth between the draw calls. Once you have finished investigating, you can go to the **Debug** menu again and choose **Release OpenGL ES Frame**. This will let the game keep running again on the device. Keep in mind that these options are available only when you have a game running on the device.
Instruments

Instruments has always been the most famous debugging tool and the most famous savior of so many games and apps for iOS and Mac OS X. As long as you are building iOS apps from OS X, you must use Instruments to check your game performance, memory leaks, allocations, system trace, and lots of other features. From the XCode menu, you can choose Instruments from the Open Developer Tool submenu.
Once it is open, you can choose to profile apps from different types of platforms and different types of metrics to profile. It is very easy-to-use and handy tool. Its values are very easy to read.

**Performance optimization**

iOS devices and mobiles in general can run games with a high frame rate, but there are some disadvantages that you need to take care of to make sure that the game will be running at its best frame rate. Here are some points that you need to check while working with Unreal games for iOS:

- Try not to use real-time lights and try to use baked lightmaps
- Make sure that you build all of the lighting before building to the device
- Pooling the actors is better than destroying and spawning them
- Don't use post process effects on the camera
- Make sure to have less than 700 draw calls in the camera view
- Use as few materials as possible to get fewer draw calls
- Use as few textures as possible; using atlases is a great way to optimize the textures' amount and size
- Square texture (power of two) is the best texture for iOS devices
- The tries count should not be more than 500,000 in the camera view
- Use `PrecomputedVisibilityVolumes` if your game has lots of 3D meshes
- Try not to use HDR or LDR if possible
- Try not to use lots of masked or translucent materials, as iOS devices are not friendly with opaque surfaces

**Minimizing the game size**

When it comes to packaging a game with an engine like Unreal Engine, there is not much that you can do to reduce the final installer file size, as there are so many files, headers, and things related to the engine itself that need to be shipped along with the game. Still, here are some hints that could help in reducing the final game size:

- Using less textures and game contents will guarantee that you have a smaller content folder within the IPA.
- Any unused content must have no reference. For example, if there is a testing level and it has testing content, then it needs to be removed from any logic; because any content referenced in the game will get cooked into the IPA.
Switching the project to C++ is a great way to reduce the size a little bit, because you'll be able to disable some of the plugins such as Slate. Normally, the project cannot make any assumptions about which plugins and libraries you are using.

Zipping the final IPA file before submitting to the App Store will shrink its size a little bit.

When you make your custom Loading screen and game icon, save them as .png files and try to make them as simple as possible and as few as possible.

Summary

Optimizations and bug fixing are a major part of game development; in fact, what makes one logic better than another is how bug free it is. What makes a game better for a player (regardless of the game design) is how fast the game runs. Now you have got your hands on the majority of the tools and techniques the pros use to find and fix their bugs, optimize their iOS game performance, and reduce the build size.

With all of that in mind, I would recommend that you go directly to the next chapter as you will be using this bug-free optimized game with its small installer file to submit it for review to the App Store.
Where to buy this book
You can buy Learning Unreal® Engine iOS Game Development from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.