Unity 5.x By Example

Unity is an exciting and popular engine in the game industry. Throughout this book, you’ll learn how to use Unity by making four fun game projects, from shooters and platformers to exploration and adventure games.

Unity 5.x By Example is an easy-to-follow guide to using Unity in a practical context, step by step, by making real-world game projects. Even if you have no previous experience of Unity, this book will help you understand the toolset in depth. You’ll learn how to create a time-critical collection game, a twin-stick space shooter, a platformer, and an action-fest game with intelligent enemies. Starting from the ground up and moving toward an intermediate level, this book will help you establish a strong foundation in making games with Unity 5.

Who this book is written for

The ideal target audience for this book is game developers. They need not have previous experience with Unity since this book will cover all the basics of game development with Unity.

What you will learn from this book

- Understand core Unity concepts, such as game objects, components, and scenes
- Learn level design techniques for building immersive and interesting worlds
- Learn to make functional games with C# scripting
- Use the toolset creatively to build games of different themes and styles
- Learn to handle player controls and input functionality
- Dive into the process of working with terrains and world-creation tools
- Import custom content into Unity from third-party tools, such as Maya and Blender
- Get to grips with making both 2D and 3D games
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'The Coin Collection Game – Part 1'
- A synopsis of the book’s content
- More information on Unity 5.x By Example
About the Author

Alan Thorn is an award-winning author, mathematician, and independent video game developer based in London, UK. He is the founder of the game development studio, Wax Lyrical Games, and the creator of the critically acclaimed PC adventure game, Baron Wittard: Nemesis of Ragnarok. Alan works freelance for some of the world’s largest entertainment corporations. He has lectured on game development at some of the most prestigious institutions in Europe and written nine books on games programming, including the highly popular Teach Yourself Games Programming, Game Engine Design and Implementation, and UDK Game Development. Some of Alan’s other interests include computing, mathematics, graphics, and philosophy. More information about his company, Wax Lyrical Games, can be found at http://www.waxlyricalgames.com/.
Video games are a cultural phenomenon that has captivated, entertained, and moved billions of people worldwide over the past fifty years. As an industry and movement, video games are an exciting place to be, both for the developer and the artist. In these roles your vision, ideas, and work can influence wide audiences, shaping and changing generation after generation in an unprecedented way. In more recent times, there's been a general movement towards democratizing game development, making the development process simpler, smoother and more accessible to a wider audience, including developers perhaps working from home on a very limited budget. Instrumental in this movement is the Unity engine, which forms the main subject of this book. The Unity engine is a computer program that works with your existing asset pipeline (such as 3D modeling software) and is intended for compiling video games that work seamlessly across multiple platforms and devices, including Windows, Mac, Linux, Android, iOS and Windows Phone. Using Unity, developers import ready-made assets (such as music, textures, and 3D models), and assemble them into a coherent whole, forming a game world that works by a unified logic. Unity is an amazing program. The latest version is free for most people to download and use, and it works well with many other programs, including free software such as GIMP and Blender. This book focuses on the Unity engine and how it can be used in a practical context for making playable and fun games. No prior knowledge of Unity is expected, although some knowledge of programming and scripting (such as JavaScript, ActionScript, C, C++, Java, or C#) would be beneficial. Let's now take a look at what this book covers, on a chapter-by-chapter basis.
What this book covers

This book explores how to use the Unity engine in a hands-on, practical way by looking at concrete examples that result in real-world playable games. Specifically, it focuses on the implementation of four distinct projects divided across eight chapters, two chapters per project. Let's take a look at what these projects are:

Chapter 1, The Coin Collection Game – Part 1, begins our journey into Unity by creating a first-person collection game. This is a great starting point if you're totally new to Unity and are ready to create your first game.

Chapter 2, Project A – the Collection Game Continued, continues from the previous chapter and completes the first project. It assumes that you have completed the first chapter and brings a closure to our project, leading neatly to the next chapter.

Chapter 3, Project B – the Space Shooter, marks the beginning of our second project, focusing on the creation of a space shooter game. Here, we'll create a project in which the player must shoot the oncoming enemies.

Chapter 4, Continuing the Space Shooter, completes the space shooter project, taking the project from its state in the previous chapter and adding final touches to it.

Chapter 5, Project C – a 2D Adventure, enters the world of 2D and UI functionality. Here, we'll explore Unity's wide range of 2D features in making a side-view platformer game that relies on 2D physics.

Chapter 6, Continuing the 2D Adventure, completes the 2D adventure game project that was started in the previous chapter, adding the final touches and linking it together with the overarching game logic. This is a great place to see how multiple parts and facets of a game come together to form a whole.

Chapter 7, Project D – Intelligent Enemies, focuses on artificial intelligence and creating enemies that can patrol, chase, and attack the player's character at relevant times, while cleverly navigating their way around the level.

Chapter 8, Continuing with Intelligent Enemies, brings closure to the AI project started in the previous chapter, as well as to the book content as a whole. Here, we'll see how to use finite-state machines to achieve powerful intelligence functionality that'll help us in a variety of scenarios.
This chapter starts the first project on our list, which will be a fun collection game. Remember, it doesn't matter if you've never used Unity before. We'll go through everything necessary step by step. By the end of the next chapter, you'll have pieced together a simple, but complete and functional, game. This is an important thing to achieve because you'll get familiar with a start-to-end game development workflow. This chapter will demonstrate the following topics:

- Game design
- Projects and folders
- Asset importing and configuration
- Level design
- Game objects
- Hierarchies
Game design

Let’s make a coin collection game. Here, the player should control a character in the first-person mode, and he must wander the level, collecting all coins before a time limit runs out. If the timer runs out, the game is lost. On the other hand, if all coins are collected before the timer expires, the game is won. The first-person controls will use the default WASD keyboard setup, where W moves forward, A and S move left and right, and D walks backward. Head movement is controlled using the mouse, and coins are collected by simply walking into them. See Figure 1.1, featuring the coin collection game in action in the Unity Editor. The great benefit in making this game is that it demonstrates all the core Unity features together, and we don’t need to rely on any external software to make assets, such as textures, meshes, and materials.

Figure 1.1: Preparing for a coin collection game (the completed game)

The completed CollectionGame project, as discussed in this chapter and the next, can be found in the book companion files in the Chapter01/CollectionGame folder.
Getting started – Unity and projects

Every time you want to make a new Unity game, including coin collection games, you'll need to create a **New Project**. Generally speaking, Unity uses the term **Project** to mean a **Game**. There are two main ways to make a new project, and it really doesn't matter which one you choose because both end up in the same place. If you're already in the Unity interface, looking at an existing scene or level, you can select **File** | **New Project** from the application menu. See **Figure 1.2**. It may ask if you want to save changes to the currently opened project and you should choose either **Yes** or **No**, depending on what you need. After selecting the **New Project** option, Unity leads you to the project creation wizard:

![Figure 1.2: Creating a new project via the main menu](image)
Alternatively, if you've just started Unity for the first time, you'll probably begin at the welcome dialog. See Figure 1.3. From here, you can access the new project creation wizard by choosing the NEW PROJECT button:

![Figure 1.3: The Unity welcome screen](image)

On reaching the NEW PROJECT creation wizard, Unity can generate a new project for you on the basis of some basic settings. Simply fill in the name of your project (such as CollectionGame), and select a folder on your computer to contain the project files that will be generated automatically. Finally, click on the 3D button to indicate that we're going to create a 3D game, as opposed to 2D, and then click on the Create project button to complete the project generation process. See Figure 1.4:

![Figure 1.4: Creating a new project](image)
Projects and project folders

Unity has now created a blank, new, and empty project. This represents the starting point for any game development project and is the place where development begins. The newly created project contains nothing initially: no meshes, textures, or any other Assets. You can confirm this by simply checking the Project panel area at the bottom of the editor interface. This panel displays the complete contents of the project folder, which corresponds to an actual folder on your local drive created earlier by the project wizard. This folder should be empty. See Figure 1.5. This panel will later be populated with more items, all of which we can use to build a game.

Figure 1.5: The Unity project panel docked at the bottom of the interface
If your interface looks radically different from Figure 1.5, in terms of its layout and arrangement, then you can reset the UI layout to its defaults. To do this, click on the Layout drop-down menu from the top-right corner of the editor interface, and choose Default. See Figure 1.6.
You can view the contents of your project folder directly via either Windows Explorer or Mac Finder, by right-clicking the mouse in the Project panel from the Unity Editor to reveal a context menu, and from there, choose the Show in Explorer (Windows) or Reveal in Finder (Mac) option. See Figure 1.7:

Figure 1.7: Displaying the project folder via the Project panel
Clicking on **Show in Explorer** displays the folder contents in the default system file browser. See Figure 1.8. This view is useful to inspect files, count them, or back them up. However, don't change the folder contents manually this way via Explorer or Finder. Specifically, don't move, rename, or delete files from here, because doing so can corrupt your Unity project irretrievably. Instead, delete and move files where needed within the **Project** panel in the **Unity Editor**. This way, Unity updates its metadata as appropriate, ensuring that your project continues to work properly.

![Figure 1.8: Viewing the Project panel from the OS file browser](image)

Viewing the project folder in the OS file browser will display additional files and folders not visible in the **Project** panel, such as **Library** and **ProjectSettings**, and maybe a **Temp** folder. Together, these are known as the project metadata. This is not directly a part of your project per se, but contains additional settings and preferences that Unity needs to work properly. These folders and their files should not be edited or changed.
Importing assets

Assets are the raw materials for games—the building blocks from which they're made. Assets include meshes (or 3D models), such as characters, props, trees, houses, and more: textures, which are image files such as JPEGs and PNGs (these determine how the surface of a mesh should look); music and sound effects to enhance the realism and atmosphere of your game, and finally, scenes, which are 3D spaces or worlds where meshes, textures, sounds, and music live, exist, and work together holistically as part of a single system. Thus, games cannot exist without assets—they would otherwise look completely empty and lifeless. For this reason, we'll need assets to make the coin collection game we're working toward. After all, we'll need an environment to walk around and coins to collect!

Unity, however, is a game engine and not an asset creation program. This means that assets, such as characters and props, are typically made first by artists in external, third-party software. From here, they are exported and transferred ready-made to Unity, and Unity is responsible only for bringing these assets to life in a coherent game that can be played. Third-party asset creation programs include Blender (which is free of charge), Maya or 3DS Max to make 3D models, Photoshop or GIMP (which is free of charge) to create textures, and Audacity (which is free of cost) to generate audio. There are plenty of other options too. The details of these programs are beyond the scope of this book. In any case, Unity assumes that you already have assets ready to import to build a game. For the coin collection game, we'll use assets that ship with Unity. So let's import these to our project. To do this, select Assets | Import Package from the application menu. Then select Characters, ParticleSystems, Environment, and Prototyping. See Figure 1.9:
Each time you import a package from the menu, you'll be presented with an **Import** dialog. Simply leave all settings at their defaults, and click on **Import**. See Figure 1.10:

![Importing package](image)

Figure 1.10: Choosing Assets to import

By default, Unity decompresses all files from the package (a library of assets) into the current project. After importing, lots of different assets and data will have been added to the **Project**, ready for use. These files are copies of the originals. So any changes made to the imported files will not affect or invalidate the originals, which Unity maintains internally. The files include models, sounds, textures, and more. These are listed in the **Unity Editor** from the **Project** panel. See the following screenshot:
When selecting Assets | Import from the application menu, if you don't see all, or any, asset packages listed, you can download and install them separately from the Unity website at https://unity3d.com/. From the downloads page, choose the Additional Downloads option, and then select the Standard Assets package. See Figure 1.12.
The imported assets don't exist yet in our game. They don't appear on screen, and they won't do anything yet! Rather, they're simply added to the **Project** panel, which behaves as a library or repository of assets, from which we can pick and choose to build up a game. The assets imported thus far are built-in into Unity and we'll be continually using them in subsequent sections to make a functional coin collection game. To get more information about each asset, you can select the asset by clicking on it with the mouse, and asset-specific details will be shown on the right-hand side of the **Unity Editor** in the **Inspector**. The **Inspector** is a property sheet editor that appears on the right-hand side of the interface. It is context-sensitive and always changes to display properties for the selected object. See **Figure 1.13**.

Figure 1.13: The Inspector displays all the properties for the currently selected object
Starting a level

We've now created a Unity project and imported a large library of assets via the Unity Standard Asset package, including architectural meshes for walls, floors, ceilings, and stairs. This means that we're now ready to build our first level using these assets! Remember, in Unity, a scene means a level. The word scene and level can be used interchangeably here. They refer simply to a 3D space, that is, the space-time of the game world—the place where things exist. Since all games happen in space and time, we'll need a scene for the coin collection game. To create a new scene, select File | New Scene from the application menu, or press Ctrl + N on the keyboard. When you do this, a new and empty scene is created. You can see a visualization or preview of the scene via the Scene tab, which occupies the largest part of the Unity interface. See Figure 1.14.

Figure 1.14: The Scene tab displays a preview of a 3D world
As shown in Figure 1.14, other tabs besides the scene are visible and available in Unity. These include a Game tab and an Animator tab; in some cases, there could be more as well. For now, we can ignore all the tabs except Scene. The Scene tab is designed for quick and easy previewing of a level during its construction.

Each new scene begins empty; well, almost empty. By default, each new scene begins with two objects. Specifically, a Light to illuminate any other objects that are added, and a Camera to display and render the contents of the scene from a specific vantage point. You can view a complete list of all the objects existing in the scene using the Hierarchy panel, which is docked to the left-hand side of the Unity interface. See Figure 1.15. This panel displays the name of every GameObject in the scene. In Unity, the word GameObject simply refers to a single, independent, and unique thing that lives within the scene, whether visible or not: meshes, lights, cameras, props, and more. Thus, the Hierarchy panel tells us about everything in the Scene.
Next, let's add a floor to the scene. After all, the player needs something to stand on! We could build a floor mesh from scratch using third-party modeling software such as Maya, 3DS Max, or Blender. However, the Unity Standard Asset package, which was imported earlier, contains floor meshes that we can use. This is very convenient. These meshes are part of the Prototyping package. To access them via the Project panel, open the Standard Assets folder by double-clicking it, and then access the Prototyping | Prefabs folder. From here, you can select objects and preview them from the Inspector. See Figure 1.16.

You could also quickly add a floor to the scene by choosing GameObject | 3D Object | Plane from the application menu. However, this just adds a dull, grey floor, which isn't very interesting. Of course, you could change its appearance. As we'll see later, Unity lets you do this. However, for this tutorial, we'll use a specifically modeled floor mesh via the Standard Assets package from the Project panel.
The mesh named `FloorPrototype64x01x64` (as shown in Figure 1.16) is suitable as a floor. To add this mesh to the scene, simply drag and drop the object from the Project panel to the Scene view and then release the mouse. See Figure 1.17. When you do this, notice how the Scene view changes to display the newly added mesh within the 3D space, and the mesh name also appears as a listing in the Hierarchy panel:

![Figure 1.17: Dragging and dropping mesh assets from the Project panel to the Scene view will add them to the scene](image)

---

[16]
The floor mesh asset from the **Project** panel has now been instantiated as a **GameObject** in the scene. This means that a copy or clone of the mesh asset, based on the original in the **Project** panel, has been added to the scene as a separate **GameObject**. The **Instance** (or **GameObject**) of the floor in the scene still depends on the floor asset in the **Project** panel. However, the asset does not depend on the instance. This means that by deleting the floor in the scene, you will not delete the asset. However, if you delete the asset, you will invalidate the **GameObject**. You can also create more floors in **Scene** if you want by dragging and dropping the floor asset many times from the **Project** panel to the **Scene** view. Each time, a new instance of the floor is created in the **Scene** as a separate and unique **GameObject**, although all the added instances will still depend on the single floor asset in the **Project** panel. See Figure 1.18:

![Image of Unity interface showing multiple instances of the floor mesh]

Figure 1.18: Adding multiple instances of the floor mesh to the scene
We don’t actually need the duplicate floor pieces, however. So let’s delete them. Just click on the duplicates in the Scene view and then press Delete on the keyboard to remove them. Remember, you can also select and delete objects by clicking on their name in the Hierarchy panel and pressing Delete. Either way, this leaves us with a single floor piece and a solid start to building our scene. One remaining problem, though, concerns the floor and its name. By looking carefully in the Hierarchy panel, we can see that the floor name is FloorPrototype64x01x64. This name is long, obtuse, and unwieldy. We should change it to something more manageable and meaningful. This is not technically essential but is good practice to keep our work clean and organized. There are many ways to rename an object. One way is to first select it and then enter a new name in the name field in the Object Inspector. I’ll rename it WorldFloor. See Figure 1.19:
Transformations and navigation

A scene with a floor mesh has been established, but this alone is uninteresting. We need to add more, such as buildings, stairs, columns, and perhaps more floor pieces. Otherwise, there would be no world for the player to explore. Before building on what we've got, however, let's make sure that the existing floor piece is centered at the world origin. Every point and location within a scene is uniquely identified by a coordinate, measured as an \((X, Y, Z)\) offset from the world center (origin). The current position for the selected object is always visible from the Object Inspector. In fact, the Position, Rotation, and Scale of an object are grouped together under a category (component) called Transform. Position indicates how far an object should be moved in three axes from the world center. Rotation indicates how much an object should be turned or rotated around its central axes. Scale indicates how much an object should be shrunk or expanded to smaller or larger sizes. A default Scale of one means that an object should appear at normal size, two means twice the size, and 0.5 means half the size, and so on. Together, the Position, Rotation and Scale of an object constitute its transformation. To change the position of the selected object, you can simply type new values in the X, Y, and Z fields for Position. To move an object to the world center, simply enter \((0, 0, 0)\), as shown in Figure 1.20:

![Figure 1.20: Centering an object to the world origin](image)

Chapter 1
Setting the position of an object, as we've done here, by typing numerical values is acceptable and appropriate for the specifying of exact positions. However, it's often more intuitive to move objects using mouse-based controls. To do this, let's add a second floor piece and position it away from the first instance. Drag and drop a floor piece from the Project panel in Scene to create a second floor GameObject. Then click on the new floor piece to select it and switch to the Translate tool. To do this, press W on the keyboard or click on the translate tool icon from the toolbar at the top of the editor interface. The translate tool allows you to reposition objects in Scene. See Figure 1.21:

![Figure 1.21: Accessing the translate tool](image-url)
When the translate tool is active and an object is selected, a **Gizmo** appears centered on the object (three colored axes visible in the **Scene** tab). The translate Gizmo appears as three colored perpendicular axes: red, green, and blue corresponding to $X$, $Y$, and $Z$ respectively. To move an object, hover your cursor over one of the three axes (or planes between axes), and then click and hold the mouse while moving it to slide the object in that direction. You can repeat this process as often as needed to ensure that your objects are positioned where you need them to be. Use the translate tool to move the second floor piece away from the first. See **Figure 1.22**:

![Figure 1.22: Translate an object using the translate Gizmo](image)
You can also rotate and scale objects using the mouse, as with translate. Press \textit{E} to access the rotate tool or \textit{R} to access the scale tool, or you can activate these tools using their respective toolbar icons from the top of the editor. When these tools are activated, a Gizmo appears centered on the object, and you can click and drag the mouse over each specific axis to rotate or scale objects as needed. See \textit{Figure 1.23}:

\begin{center}
\includegraphics[width=\textwidth]{figure1_23.png}
\end{center}

\textit{Figure 1.23: Accessing the rotate and scale tools}

Being able to translate, rotate, and scale objects quickly through mouse and keyboard combinations is very important when working in Unity. For this reason, make using the keyboard shortcuts a habit as opposed to accessing the tools continually from the toolbar. However, in addition to moving, rotating, and scaling objects, you'll frequently need to move around yourself in the \textbf{Scene} view in order to see the world from different positions, angles, and perspectives. This means that you'll frequently need to reposition the scene preview camera in the world. You'll want to zoom in and zoom out of the world to get a better view of objects and change your viewing angle to see how objects align and fit together properly. To do this, you'll need to make extensive use of both the keyboard and mouse together.
To zoom closer or further from the object you're looking at, simply scroll the mouse wheel up or down—up zooms in and down zooms out. See Figure 1.24:

![Figure 1.24: Zooming in and out](image)

To pan the Scene view left or right, or up or down, hold down the middle mouse button while moving the mouse in the appropriate direction. Alternatively, you can access the pan tool from the application toolbar (or press Q on the keyboard) and then simply click and drag in the Scene view while the tool is active. Pan does not zoom in or out; it simply slides the camera left and right, or up and down.

![Figure 1.25: Accessing the Pan tool](image)
Sometimes, while building levels, you'll lose sight entirely of the object that you need. For example, your viewport camera could be focusing on a completely different place from the object you really want to click or see. In this case, you'll often want to shift the viewport camera automatically in order to focus on that specific object. Specifically, you'll want to reposition and rotate the viewport as necessary to bring a desired object to the center of the view. To do this automatically, select the object to focus on (or frame) by clicking on its name from the Hierarchy panel. Then, press the $F$ key on the keyboard. Alternatively, you can double-click its name in the Hierarchy panel. See Figure 1.26:
After framing an object, you'll often want to rotate around it in order to quickly and easily view it from all important angles. To achieve this, hold down the `Alt` key on the keyboard while clicking and dragging the mouse to rotate the view. See Figure 1.27:

![Figure 1.27: Rotating around the framed object](image)

Figure 1.27: Rotating around the framed object
Lastly, it's helpful to navigate a level in the Scene view using first-person controls, that is, controls that mimic how first-person games are played. This helps you experience the scene at a more personal and immersive level. To do this, hold down the right mouse button and (with the button depressed) use the WASD keys on the keyboard to control forward, backward, and strafing movement. Movement of the mouse controls head orientation. You can also hold down the Shift key while moving to increase movement speed. See Figure 1.28:

![Figure 1.28: Using first-person controls](image)

The great thing about learning the versatile transformation and navigation controls is that, on understanding them, you can move and orient practically any object in any way, and you can move and view the world from almost any position and angle. Being able to do this is critically important to build quality levels quickly. All of these controls, along with some others that we'll soon see, will be used frequently throughout this book to create scenes and work in Unity generally.
Scene building

Now that we've seen how to transform objects and navigate the scene viewport successfully, let's proceed to complete our first level for the coin collection game. Let's separate the two floor meshes apart in space, leaving a gap between them that we'll fix by creating a bridge, which the player will be able to cross, moving between the floor spaces like islands. We can use the translate tool (W) to move objects around. See Figure 1.29:

![Figure 1.29: Separating the floor meshes into islands](image)

If you want to create more floor objects, you can use the method that we've seen already by dragging and dropping the mesh asset in the Project panel in the Scene viewport. Alternatively, you can duplicate the selected object in the viewport by pressing Ctrl + D on the keyboard. Both methods produce the same result.
Next, we'll add some props and obstacles to the scene. Drag and drop some house objects onto the floor. The house object (HousePrototype16x16x24) is found in the Assets | Standard Assets | Prototyping | Prefabs folder. See Figure 1.30:

On dragging and dropping the house in the scene, it may align to the floor nicely with the bottom against the floor, or it may not align like that. If it does, that's splendid and great luck! However, we shouldn't rely on luck every time because we're professional game developers! Thankfully, we can make any two mesh objects align easily in Unity using vertex snapping. This feature works by forcing two objects into positional alignment within the scene by overlapping their vertices at a specific and common point.
For example, consider Figure 1.31. Here, a house object hovers awkwardly above the floor and we naturally want it to align level with the floor and perhaps over to the floor corner. To achieve this, start by selecting the house object (click on it or select it from the Hierarchy panel). The object to be selected is the one that should move to align and not the destination (which is the floor), which should remain in place.

Figure 1.31: Misaligned objects can be snapped into place with Vertex Snapping
Next, activate the translate tool (W) and hold down the V key for vertex snapping. With V held down, move the cursor around and see how the Gizmo cursor sticks to the nearest vertex of the selected mesh. See Figure 1.32. Unity is asking you to pick a source vertex for the snapping:

Figure 1.32: Hold down V to activate Vertex Snapping
With $V$ held down, move the cursor to the bottom corner of the house, and then click and drag from the corner to the floor mesh corner. The house will then snap align to the floor, corner to corner. When aligned this way, it release the $V$ key, and the two meshes get aligned exactly at the vertices. See Figure 1.33:

![Figure 1.33: Align two meshes by vertices](image-url)
Now you can assemble a complete scene using the mesh assets included in the Prototyping package. Drag and drop props in the scene, and using translate, rotate, and scale, you can reposition, realign, and rotate these objects; using vertex snapping, you can align them wherever you need. Give this some practice. See Figure 1.34 for the scene arrangement that I made using only these tools and assets:

![Figure 1.34: Building a complete level](image)

**Lighting and sky**

The basic level has been created in terms of architectural models and layout; this was achieved using only a few mesh assets and some basic tools. Nevertheless, these tools are powerful and offer us a multitude of combinations and options to create great variety and believability in game worlds. One important ingredient is missing for us, however. This ingredient is lighting. You'll notice from Figure 1.34 that everything looks relatively flat, with no highlights, shadows, or light or dark areas. This is because scene lighting is not properly configured for best results, even though we already have a light in the scene, which was created initially by default.
Let's start setting the scene for the coin collection game by enabling the sky, if it's not already enabled. To do this, click on the Extras drop-down menu from the top toolbar in the Scene viewport. From the context menu, select Skybox to enable Skybox viewing. A Skybox simply refers to a large cube that surrounds the whole scene. Each interior side has a continuous texture (image) applied to simulate the appearance of a surrounding sky. For this reason, clicking the Skybox option displays a default sky in the Scene viewport. See Figure 1.35:

Figure 1.35: Enabling the sky
Now, although the Skybox is now enabled and the scene looks better than before, it's still not being illuminated properly—the objects lack shadows and highlights. To fix this, be sure that lighting is enabled for the scene by toggling on the Lighting icon at the top of the Scene viewport. See Figure 1.36. This setting is for display purposes only. It only affects whether lighting effects are shown in the Scene viewport and not whether lighting is truly enabled for the final game.

Figure 1.36: Enabling scene lighting in the Scene viewport
Enabling lighting display for the viewport will result in some differences to the scene appearance and, again, the scene should look better than before. You can confirm that scene lighting is taking effect by selecting **Directional Light** from the **Hierarchy** panel and rotating it. Doing this controls the time of day, rotating the light cycles between day and night and changing the light intensity and mood. This changes how the scene is rendered. See **Figure 1.37**:

![Figure 1.37: Rotating the scene directional light changes the time of day](image)
Let’s undo any rotations to the **Directional Light** by pressing `Ctrl + Z` on the keyboard. To prepare for final and optimal lighting, all non-movable objects in the scene (such as walls, floors, chairs, tables, ceilings, grass, hills, towers, and more) should be marked as **Static**. This signifies to Unity that the objects will never move, no matter what happens during gameplay. By marking non-movable objects ahead of time, you can help Unity optimize the way it renders and lights a scene. To mark objects as **Static**, simply select all non-movable objects (which includes practically the entire level so far), and then enable the **Static** checkbox via the **Object Inspector**. Note that you don’t need to enable the **Static** setting for each object separately. By holding down the **Shift** key while selecting objects, you can select multiple objects together, allowing you to adjust their properties as a batch through the **Object Inspector**. See **Figure 1.38**:

![Figure 1.38: Enabling the Static option for multiple non-movable objects improves lighting and performance](image-url)
When you enable the Static checkbox for geometry, Unity auto-calculates scene lighting in the background—effects such as shadows, indirect illumination, and more. It generates a batch of data called the GI Cache, featuring Light Propagation Paths, which instructs Unity how light rays should bounce and move around the scene to achieve greater realism. Even so, enabling the Static checkbox as we've done still won’t produce cast shadows for objects, and this seriously detracts from realism. This happens because most mesh objects have the Cast Shadows option disabled. To fix this, select all meshes in the scene. Then, from the Object Inspector, click on the Cast Shadows checkbox from the Mesh Renderer component, and choose the On option from the context menu. When you do this, all mesh objects should be casting shadows. See Figure 1.39:

Voila! Your meshes now cast shadows. Splendid work: in reaching this far, you’ve created a new project, populated a scene with meshes, and successfully illuminated them with directional lighting. That’s excellent. However, it’d be even better if we could explore our environment in the first-person mode. We’ll see how next.
Play testing and the Game tab

The environment created thus far for the coin collection game has been assembled using only the mesh assets included with the native Prototyping package. My environment, as shown in Figure 1.40, features two main floor islands with houses, and the islands themselves are connected together by a stepping-stone bridge. Your version may be slightly different, and that's fine.

Figure 1.40: The scene created so far contains two island areas

Overall, the scene is good work. It's well worth saving. To save the scene, press Ctrl + S on the keyboard or else choose File | Save Scene from the application menu. See Figure 1.41. If you're saving the scene for the first time, Unity displays a pop-up Save dialog, prompting you to name the scene descriptively (I called it Level_01).
After saving the scene, it becomes scene asset of the project and appears in the Project panel. See Figure 1.42. This means that the scene is now a genuine and integral part of the project and not just a temporary work-in-progress as it was before. Notice also that saving a scene is conceptually different from saving a project. For example, the application menu has entries for Save Scene and Save Project. Remember, a Project is a collection of files and folders, including assets and scenes. A scene, by contrast, is one asset within the project and represents a complete 3D map that may be populated by other assets, such as meshes, textures, and sounds. Thus, saving a project saves the configuration between files and assets, including scenes. Saving a scene, in contrast, just retains the level changes within that specified scene.
You can see from Figure 1.42 that I've saved my scene in a folder named Scenes. Folders can be created in your project by right-clicking on any empty area in the Project panel and choosing New Folder from the context menu, or else choose Assets | Create | Folder from the application menu. You can easily move and rearrange assets among folders by simply dragging and dropping them.

Now, the level, as it stands, contains nothing really playable. It's simply a static, lifeless, and non-interactive 3D environment made using the Editor tools. Let's correct this by making our scene playable, allowing the player to wander around and explore the world in first-person mode, controlled using the standard WASD keys on the keyboard. To achieve this, we'll add a first-person character controller to the scene. This is a ready-made asset included with Unity, which contains everything necessary to create quick and effective first-person controls. Open the Standard Assets | Characters | FirstPersonCharacter | Prefabs folder. Then drag and drop the FPSController asset from the Project panel in the scene. See Figure 1.43:
After adding the first-person controller, click on the play button from the Unity toolbar to play test the game in first-person mode. See *Figure 1.44*:
On clicking play, Unity typically switches from the **Scene** tab to the **Game** tab. As we’ve seen, the **Scene** tab is a *director’s-eye view* of the active scene; it’s where a scene is edited, crafted, and designed. In contrast, the **Game** tab is where the active scene is played and tested from the perspective of the gamer. From this view, the scene is displayed through the main game camera. While play mode is active, you can play test your game using the default game controls, provided that the **Game** tab is *in focus*. The first-person controller uses the WASD keys on the keyboard and mouse movement controls head orientation. See Figure 1.45:

![Figure 1.45: Play testing levels in the Game tab](image)

You can switch back to the **Scene** tab while in play mode. You can even edit the scene and change, move, and delete objects there too! However, any and all scene changes made during play mode will automatically revert back to their original settings when play mode ends. This behavior is intentional. It lets you edit properties during gameplay to observe their effects and debug any issues without permanently changing the scene.
Congratulations! Your level should now be walkable in first-person mode. When completed, you can easily stop playback by clicking on the play button again or by pressing Ctrl + P on the keyboard. Doing this will return you to the Scene tab.

Unity also features a **Toggle-Pause** button to suspend and resume gameplay.

You should notice that, on playing the level with a first-person controller, you receive an information message printed to the **Console** window. By default, this window appears at the bottom of the Unity Editor, docked beside the Project panel. This window is also accessible manually from the application menu, Window | Console. The **Console** window is where all encountered errors or warnings are displayed for your review as well as information messages. Errors are printed in red and warnings in yellow, and information messages appear as a default grey. Sometimes, a message appears just once, or sometimes it appears many times repeatedly. See Figure 1.46:

![Figure 1.46: The Console outputs information, warnings, and errors](image-url)
As mentioned, the **Console** window outputs three distinct types of message: information, warnings, and errors. Information messages are typically Unity's way of making best practice recommendations or suggestions based on how your project is currently working. Warnings are slightly more serious and represent problems either in your code or scene, which (if not corrected) could result in unexpected behaviors and suboptimal performance. Finally, errors describe areas in your scene or code that require careful and immediate attention. Sometimes, errors will prevent your game from working altogether and sometimes errors happen at runtime and can result in game crashes or freezes. The **Console** window, therefore, is helpful because it helps us debug and address issues with our games. *Figure 1.46* has identified an issue concerning duplicated audio listeners.

An audio listener is a component attached to a camera object. Specifically, each and every camera, by default, has an audio listener component attached. This represents an *ear point*, that is, the ability to hear sound within the scene from the position of the camera. Unfortunately, Unity doesn't support multiple active audio listeners in the same scene, which means that you can only hear audio from one place at any one time. This problem happens because our scene now contains two cameras, one that was added automatically when the scene was created, and the other that is included in the first-person controller. To confirm this, select the first-person controller object in the **Hierarchy** panel and click on the triangle icon beside its name to reveal more objects underneath, which are part of the first-person controller. See *Figure 1.47*:
Select the FirstPersonCharacter object, which is underneath the FPSController object (as shown in Figure 1.47). The FirstPersonCharacter object is a child of the FPSController, which is the parent. This is because FPSController contains or encloses the FirstPersonCharacter object in the Hierarchy panel. Child objects inherit the transformations of their parents. This means that as parent objects move and rotate, all transformations will cascade downwards to all children. From the Object Inspector, you can see that the object has an Audio Listener component. See Figure 1.48:
We could remove the **Audio Listener** component from the **FPSController**, but this would prevent the player hearing sound in first-person perspective. So, instead, we’ll delete the original camera created by default in the scene. To do this, select the original camera object in the hierarchy and press **Delete** on the keyboard. See Figure 1.49. This removes the **Audio Listener** warning in the **Console** during gameplay. Now give the game a play test!

![Deleting a camera object](image-url)
Adding a water plane

The collection game is making excellent progress. We now have something playable insofar as we can run around and explore the environment in first-person mode. However, the environment could benefit from additional polish. Right now, for example, the floor meshes appear suspended in mid-air with nothing beneath them to offer support. See Figure 1.50. Further, it's possible to walk over the edge and fall into an infinite drop. So let's add some water beneath the floors to complement the scene as a complete environment.

Figure 1.50: The world floor appears to float and have no support
To add water, we can use another ready-made Unity asset included in the Project panel. Open the Standard Assets | Environment | Water | Water | Prefabs folder. Then drag and drop the WaterProDaytime asset from the Project panel in the scene. See Figure 1.51. This appears as a circular object, which is initially smaller than needed.

Figure 1.51: Adding water to the environment
After adding the Water prefab, position it below the floor level and use the scale tool (R) to increase its planar size \((X, Z)\) to fill the environment outward into the distant horizon. This creates the feel that the floor meshes are smaller islands within an expansive world of water. See Figure 1.52:

![Figure 1.52: Scaling and sizing water for the environment](image-url)
Now let's take another test run in the **Game** tab. Press play on the toolbar and navigate the character around in first-person mode. See Figure 1.53. You should see the water in the level. Of course, you can't walk on the water! Neither can you swim or dive beneath it. If you try walking on it, you'll simply fall through it, descending into infinity as though the water had never been there. Right now, the water is an entirely cosmetic feature, but it makes the scene look much better.
The water is really a substanceless, ethereal object through which the player can pass easily. Unity doesn't recognize it as a solid or even a semi-solid object. As we'll see in more detail later, you can make an object solid very quickly by attaching a **Box Collider** component to it. Colliders and physics is covered in more depth from *Chapter 3, Project B – the Space Shooter* onward. For now, however, we can add solidity to the water by first selecting the *Water* object from the *Hierarchy* panel (or in the *Scene* viewport) and then by choosing **Component | Physics | Box Collider** from the application menu. See *Figure 1.54*. Attaching a component to the selected object changes the object itself; it changes how it behaves. Essentially, components add behavior and functionality to objects, making them behave in different ways. Even so, resist the temptation to add lots of components to an object without reason and with the view that it makes them more versatile or powerful. It's better to have as few components on an object as necessary. This strategy of preferring relevant simplicity keeps your workflow neater, simpler, and optimized.
When a **Box Collider** is added to the water, a surrounding green cage or mesh appears. This approximates the volume and shape of the Water object and represents its physical volume, namely, the volume of the object that Unity recognizes as solid. See Figure 1.55:

![Box Colliders approximate physical volume](image)

Figure 1.55: Box Colliders approximate physical volume
If you play the game now, your character will walk on water as opposed to falling through. True, the character should be able to swim properly, but walking might be better than falling. To achieve full swimming behavior would require significantly more work and is not covered here. If you want to remove the Box Collider functionality and return the water back to its original, ethereal state, then select the Water object, click on the cog icon on the Box Collider component, and then choose Remove Component from the context menu. See Figure 1.56:

![Figure 1.56: Removing a component](image)
Adding a coin to collect

On reaching this far, our game has many features, namely, a complete environment, first-person controller, and water. However, we’re supposed to be making a coin collection game and there aren’t any coins for the player to collect yet. Now, to achieve fully collectible coins, we’ll need to write some C# script, which will happen in the next chapter of this book. However, we can at least get started here at creating the coin object itself. To do this, we’ll use a Cylinder primitive that’s scaled to form a coin-looking shape. To create a cylinder, select GameObject | 3D Object | Cylinder from the application menu:

![Create a Cylinder](image-url)
Initially, the cylinder looks nothing like a coin. However, this is easily changed by scaling non-uniformly in the $Z$ axis to make the cylinder thinner. Switch to the scale tool (R) and then scale the **Cylinder** inward. See Figure 1.58:
After rescaling the coin, its collider no longer represents its volume. It appears much larger than it should (see Figure 1.58). By default, the Cylinder is created with a Capsule Collider as opposed to a Box Collider. You can change the size of the Capsule Collider component by adjusting the Radius field from the Object Inspector when the coin is selected. Lower the Radius field to shrink the collider to a more representative size and volume. See Figure 1.59. Alternatively, you could remove Capsule Collider altogether and add Box Collider instead. Either way is fine; generally choose the simpler shape where possible. The colliders will be used in script in the next chapter to detect when the player collides with the coin to collect them:

Figure 1.59: Adjusting the Capsule Collider for the coin

Here we are! We now have the basic shape and structure for a coin. We will, of course, improve it carefully and critically in many ways in the next chapter. For example, we'll make it collectible and assign it a material to make it look shiny. However, here, using only a basic Unity primitive and scale tool, we're able to generate a shape that truly resembles a coin.
Summary

Congratulations! On reaching this point, you have laid the foundations for a coin collection game that will be complete and functional in the next chapter. Here, we've seen how to create a Unity project from scratch and populate it with assets, such as meshes, textures, and scenes. In addition, we've seen how to create a scene for our game and use a range of assets to populate it with useful functionality that ships out of the box with the Unity engine, such as water, first-person controllers, and environment prototyping assets. In the next chapter, we'll resume work from where we ended here by making a coin that is collectible, and establishing a set of rules and logic for the game, making it possible to win and lose.
Where to buy this book
You can buy Unity 5.x By Example 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.