Make a Complete Unity Game
from Scratch Using C#

TWIN STICK SHOOTER TUTORIAL

By the raywenderlich.com Tutorial Team

Brian Moakley

Twin Stick Shooter Tutorial

Brian Moakley

Copyright ©2016 Razeware LLC.

Notice of Rights

All rights reserved. No part of this book or corresponding materials (such as text, images, or source code) may be reproduced or distributed by any means without prior written permission of the copyright owner.

Notice of Liability

This book and all corresponding materials (such as source code) are provided on an "as is" basis, without warranty of any kind, express of implied, including but not limited to the warranties of merchantability, fitness for a particular purpose, and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in action of contract, tort or otherwise, arising from, out of or in connection with the software or the use of other dealing in the software.

Trademarks

All trademarks and registered trademarks appearing in this book are the property of their own respective owners.

Dedications

"To Lizzie, Fiora, and Rowen — these words exist only from your sacrifice and blessings. My others find joy in them as I find joy in you."

— Brian Moakley

About the author

Brian Moakley is a writer and develeper at raywenderlich.com who produces video tutorials on iOS, Unity, and various other topics. When not writing or coding, Brian enjoys story driven first person shooters, reading genre fiction, and epic board game sessions with friends.

About the editors

Adrian Strahan is a tech editor of this book. He is a freelance iOS developer and Project Manager living in the South West of England. He's worked on iPhone and iPad apps since 2010 (iOS3) and specializes in mobile- and web-based application development.

Mitch Allen is a tech editor of this book. Mitch is an indie developer, maker and tech writer. You can find his games on iTunes and his modules on npmjs. mitchallen.com

Chris Belanger is an editor of this book. Chris Belanger is the Book Team Lead and Lead Editor for raywenderlich.com. If there are words to wrangle or a paragraph to ponder, he’s on the case. When he kicks back, you can usually find Chris with guitar in hand, looking for the nearest beach, or exploring the lakes and rivers in his part of the world in a canoe.

Wendy Lincoln is an editor of this book. She is a full-time project manager (PMP, actually) specializing in IT marketing and content development. She has an unusual background that involves a culinary degree, cooking show, writing and activism. Occasionally, she logs off help her husband with home improvement projects or enjoy beach life.

Ray Wenderlich is a final pass editor of this book. Ray is part of a great team - the raywenderlich.com team, a group of over 100 developers and editors from across the world. He and the rest of the team are passionate both about making apps and teaching others the techniques to make them. When Ray’s not programming, he’s probably playing video games, role playing games, or board games.

About the artists

Mike Berg made all of the 3D models, animations, and textures for this book. He is a full-time game artist who is fortunate enough to work with many indie game developers from all over the world. When he's not manipulating pixel colors, he loves to eat good food, spend time with his family, play games and be happy. You can check out his work at: www.weheartgames.com

Vinnie Prabhu created all of the music and sounds for the games in this book. Vinnie is a music composer/software engineer from Northern Virginia who has done music and sound work for concerts, plays and video games. He's also a staff member on OverClocked ReMix, an online community for music and video game fans. You can find Vinnie on Twitter as @palpablevt.

Table of Contents: Overview

Introduction 11
Chapter 1: Hello Unity 16
Chapter 2: GameObjects 38
Chapter 3: Components 58
Chapter 4: Physics 80
Chapter 5: Managers and Pathfinding 110
Chapter 6: Animation 134
Chapter 7: Sound 164
Chapter 8: Finishing Touches 189
Conclusion 216

Table of Contents: Extended

Introduction 11
- Why Unity? 12
- What you need 13
- Who this book is for 13
- How to use this book 13
- Book source code and forums 14
- Book updates 14
- License 15
- Acknowledgments 15

Chapter 1: Hello Unity 16
- Installing and running Unity 19
- Learning the interface 21
- Organizing your assets 25
- Importing Assets 27
- Add models to the Scene view 31
- Adding the hero 34
- Where to go from here? 37

Chapter 2: GameObjects 38
- Introducing GameObjects 38
- Creating a prefab 44
- Creating spawn points 55
- Where to go from here? 57

Chapter 3: Components 58
- Getting started 59
- Introducing scripting 63
- Creating your first script 64
- Managing Input 65
- Camera movement 71
- Adding Gunplay 74
- Where to go from here? 79

Chapter 4: Physics 80
Getting started 80
Destroying old objects 92
Collisions and layers 93
Joints 98
Raycasting 105
Where to go from here? 109
Chapter 5: Managers and Pathfinding 110
Introducing the GameManager 110
Pathfinding in Unity 118
Final touches 130
Where to go from here? 133
Chapter 6: Animation 134
Getting started 135
The animation window 135
Introducing keyframe animations 136
Your first animation 137
Animation states 141
Animation state transitions 146
Animation state transition conditions 148
Triggering animations in code 151
Animating models 153
Imported animations 155
Animating the space marine 160
Where to go from here? 163
Chapter 7: Sound 164
Getting started 164
Playing background music 166
Building a sound manager 168
Playing sounds 172
Adding power-ups 173
A power-up pick-up 176
Introducing the Audio Mixer 179

Isolating sounds 184
Adding audio effects 186
Where to go from here? 188

Chapter 8: Finishing Touches 189
Fixing the game manager 190
Killing the hero 193
Removing the bobblehead 196
Decapitating the alien 199
Adding particles 202
Activating particles in code 206
Winning the game 209
Animation events 211
Where to go from here? 215

Conclusion 216

Introduction

If you're reading this book, chances are that you have a dream of making your own video game. But if you're like I was a few years ago, you might be worried that this is too difficult, or something that's out of your reach.

Don't worry! The Unity development platform makes that dream a reality for aspiring game developers everywhere. Unity's aim is to "democratize" game development, by providing a AAA-level engine to independent game developers in a way that is both affordable and accessible.

And with this book, we'll show you how to make your own games with Unity step-by-step — even if you're a complete beginner.

You'll learn by doing. This tutorial will walk you through the process of creating a twin stick shooter. By the time you're done reading this book, not only will you have created a kick-ass game, but you'll be ready to create that dream game of your own!

Why Unity?

Unity is one of the most powerful and popular game frameworks used today. But why use it rather than other frameworks?

Well, here are a few good reasons:

- **It's free to use.** If you're an indie game developer, you can download and start using Unity for free, which is great when you're just learning. You do have to pay once your company earns $100K or more in a year or in certain other situations, but a lot of the time the free version is just fine. For example, the free version is all you need for this book!
- **It's cross-platform.** With Unity, you can make your game once and build it for a variety of platforms, including Windows, macOS, Linux, iOS, and more.
- **It's powerful.** Unity isn't just for indie games — it has been used by AAA game developers in popular games such as City Skylines, the Long Dark, Hearthstone, and more.
- **It has a visual editor.** Unlike other game platforms where you have to type tons of code before you see anything on the screen, with Unity you can simply import an asset and drag and drop. This visual style of development is great for beginners and professionals alike, and makes game development fast and fun.
- **Live debugging.** With Unity you can click a button to preview your game instantly in the editor, and you can even modify game objects on the fly. For example, you can drag new enemies onto the level as you play it, tweak gameplay values and more, allowing for an iterative game design process.
- **Asset store.** Need some functionality Unity doesn't provide on its own? Chances are somebody has provided the functionality through the Asset Store — a place where you can buy scripts, models, sounds, and more for your games. Or maybe, even, a tutorial or two? ;]
- **Unity is fun!** You can think of Unity like a box of LEGO: the only limits are those of your own imagination.

Note that Unity has some cons to consider as well:

- **Learning curve.** It's not hard to learn Unity itself — this book has you covered in that department. :] But if you want to make a game, in addition to knowing how to build it in Unity, you'll also need 3D models, textures, and sounds. These are all made in different tools like Blender, Photoshop, and Audacity, and each of these are subjects of their own books. If you're lucky enough to work with a team of artists that specialize in these tools, then you're set — but if you're an indie developer trying to learn them all, it can be a challenge.

- **Can be expensive.** Although Unity starts out free, eventually you'll have to move to the paid version (such as if your company earns enough money, or if you want to get rid of the splash screen, or access certain other features). When you do move to the paid tier, it can get expensive quickly, especially as your team size grows.

What you need

To follow along with the tutorials in this book, you'll need the following:

- **A PC running Windows 7 or later or a Mac running Mountain Lion or later**. You'll need this to install the latest version of Unity. Note this book will work fine whether you prefer to develop on Windows or on the Mac, since Unity is cross-platform.
- **Unity 5.5 or later**. You'll need Unity 5.5 or later for all tasks in this book. You can download the latest version of Unity for free here: https://store.unity.com/

Once you have these items in place, you'll be able to follow along with every chapter in this book.

Who this book is for

This book is for complete beginners to Unity, or for those who'd like to bring their Unity skills to a professional level. The book assumes you have some prior programming experience (in a language of your choice).

If you are a complete beginner programming, we recommend you learn some basic programming skills first. A great way to do that is to watch our free Beginning C# with Unity series, which will get you familiar with programming in the context of Unity. You can watch it for free here:

- https://www.raywenderlich.com/category/unity

How to use this book

This book is designed to teach you Unity from the ground up. The following chapters, included with this release, are designed to give you a solid foundation in Unity:

- Chapter 1, "Hello Unity"
- Chapter 2, "GameObjects"
- Chapter 3, "Components"

- Chapter 4, "Physics"
- Chapter 5, "Managers and Pathfinding"
- Chapter 6, "Animation"
- Chapter 7, "Sound"
- Chapter 8, "Finishing Touches"

That covers the most important features of Unity covered in this tutorial.

Since you've downloaded this book as part of an asset package, you can complete this tutorial as part of the current project. You can also create a new project and import the assets as well. By doing this approach, you'll learn how to configure your project as well as how to import assets. You can download the assets over here:

- https://www.raywenderlich.com/downloads/twin_stick_shooter_unity_55.zip

The assset bundle is broken into chapters, representing each chapter in the book. You can use these as starting points, or for your own reference.

Book source code and forums

This asset comes with the source code, game assets, starter and completed projects for each chapter; these resources are shipped with the PDF.

We've also set up an official forum for the book at raywenderlich.com/forums. This is a great place to ask questions about the book, discuss making games with Unity in general, or to submit any errors you may find.

Book updates

Great news: since you purchased the PDF version of this book, you'll receive free updates of the book's content! Updates are managed through the Unity Asset store. Each update will correspond with newer versions of the Unity engine, keeping the book up to date with each major change.

If you enjoyed this book and are interested in similar books, feel free to sign up for our weekly newsletter. This newsletter includes a list of the tutorials published on raywenderlich.com in the past week, important news items such as book updates or new books, and a few of our favorite developer links. Sign up here:

- www.raywenderlich.com/newsletter

License

By purchasing *Twin Stick Shooter Tutorial*, you have the following license:

- You are allowed to use and/or modify the source code in *Twin Stick Shooter Tutorial* in as many games as you want, with no attribution required.
- You are allowed to use and/or modify all art, images, or designs that are included in *Twin Stick Shooter Tutorial* in as many games as you want, but must include this attribution line somewhere inside your game: "Artwork/images/designs: from the *Twin Stick Shooter Tutorial* book, available at www.raywenderlich.com".
- The source code included in *Twin Stick Shooter Tutorial* is for your own personal use only. You are NOT allowed to distribute or sell the source code in *Twin Stick Shooter Tutorial* without prior authorization.
- This book is for your own personal use only. You are NOT allowed to sell this book without prior authorization, or distribute it to friends, co-workers, or students; they must to purchase their own copy instead.

All materials provided with this book are provided on an "as is" basis, without warranty of any kind, express or implied, including but not limited to the warranties of merchantability, fitness for a particular purpose and noninfringement. In no event shall the authors or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the software or the use or other dealings in the software.

All trademarks and registered trademarks appearing in this guide are the property of their respective owners.

Acknowledgments

We would like to thank many people for their assistance in making this possible:

- **Our families:** For bearing with us in this crazy time as we worked all hours of the night to get this book ready for publication!
- **Everyone at Unity Technologies:** For developing an amazing platform, for constantly inspiring us to improve our games and skill sets and for making it possible for many developers to make a living doing what they love!
- **And most importantly, the readers of raywenderlich.com — especially you!** Thank you so much for reading our site and purchasing this book. Your continued readership and support is what makes all of this possible!

Chapter 1: Hello Unity

By Brian Moakley

To say game development is a challenge would be the understatement of the year.

Until recently, making 3D games required low-level programming skills and advanced math knowledge. It was akin to a black art reserved only for super developers that never saw the sunlight.

That all changed with Unity. Unity has made this game programming into a craft that's now accessible to mere mortals. Yet, Unity still contains those complicated AAA features, so as you grow as a developer you can begin to leverage them in your games.

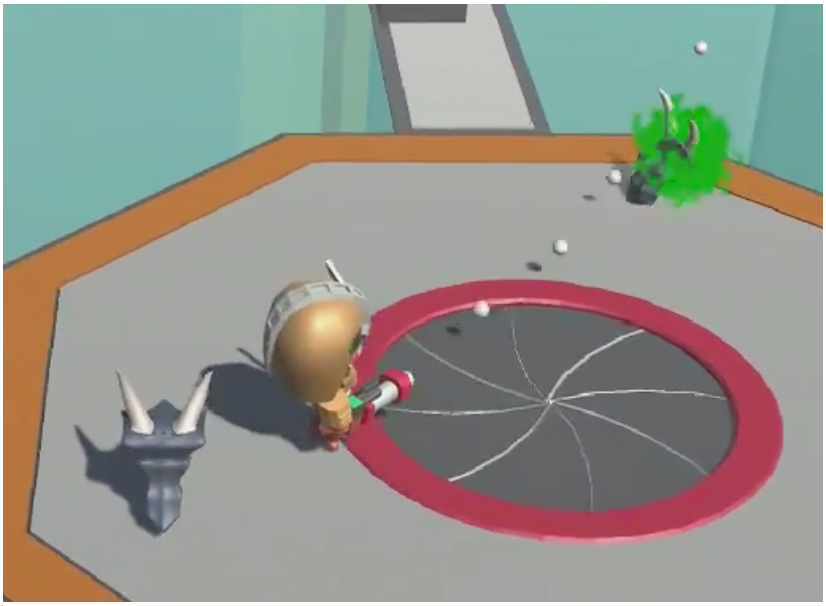
Just like every game has a beginning, so does your learning journey — and this one will be hands-on. Sure, you *could* pore over pages and pages of brain-numbing documentation until a lightbulb appears above your head, or you can learn by creating a game.

You obviously would prefer the latter, so you'll build a game named **Bobblehead Wars**.

In this game, you take the role of a kickass space marine, who just finished obliterating an alien ship. You may have seen him before; he also starred in our book *2D iOS & tvOS Games* in a game called Drop Charge.

After destroying the enemy ship, our space marine decides to vacation on a desolate alien planet. However, the aliens manage to interrupt his sun tan - and they are out for blood. After all, space marines are delicacies in this parts of the galaxy!

This game is a twin-stick shooter, where you blast hordes of hungry aliens that relentlessly attack:



You'll toss in some random powerups to keep gameplay interesting, but success lies in fast footwork and a happy trigger finger.

You'll build the game across the next eight chapters, in stages:

1. **Chapter 1, Hello Unity**: You are here! You'll start by exploring the Unity interface and you'll learn how to import assets into your project.
2. **Chapter 2, GameObjects**: Learn about two critical concepts in Unity – GameObjects and Prefabs – by adding and laying out the initial objects for Bobblehead Wars.
3. **Chapter 3, Components**: In Unity, you build up your game objects by combining a set of components. In this chapter, you'll use components to give your hero the ability to walk and blast away at the oncoming horde.
4. **Chapter 4, Physics**: Learn the basics of game physics by adding collision detection and giving the hero the ability to turn on a dime.
5. **Chapter 5, GameManager and Pathfinding**: This is where it gets rough for the space marine. In this chapter, you'll create the tools that spawn the aliens, and then enable them to chase after the hero.
6. **Chapter 6, Animations**: It's time for some shooting and chomping! Learn how to add animations to the marine and the aliens.
7. **Chapter 7, Sounds**: Bring your game game to life by adding background music and a variety of sound effects.
8. **Chapter 8, Finishing Touches**: Games are pointless without winners and losers. In this chapter, you'll add a winning and losing condition, and wrap up the game with some classy touches.

Installing and running Unity

Before you can take on aliens, you need to download the Unity engine itself.

In case you are curious, the Unity engine comes in a variety of versions. Here there are:

- **Unity Personal:** This edition allows you to create a complete game and distribute it without paying Unity anything. However, your company must make less than $100,000/year. The other catch is that each game will present a *Made by Unity* splash screen that you can't remove.
- **Unity Plus:** This edition costs $35/month. It comes with performance reporting tools, the Unity Pro skin and some additional features. This version requires your company make less than $200,000/year, and allows you to either disable or customize the "Made by Unity" splash screen.
- **Unity Pro:** This is the highest tier available. It costs $125 per month and comes with useful Unity services, professional iOS and Android add-ons, and has no splash screen. There is no revenue cap either.

There's also an enterprise edition for large organizations that want access to the source code and enterprise support.

> **Note**: Recently Unity switched from a "perpetual" model, where you paid a one-time fee, to a subscription-based model. If you prefer the perpetual model, note that Unity will offer this payment option until the end of 2017.

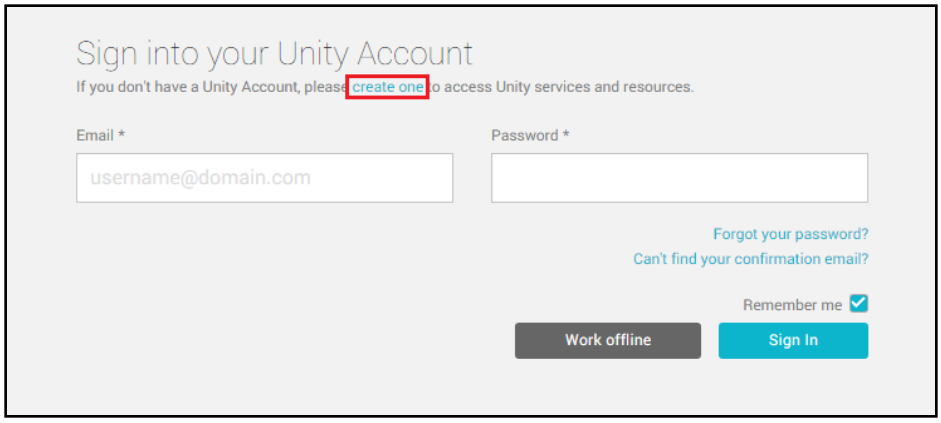
At this point, you'll need to create a new project. This project will be entirely new and walk you through the process of importing new assets.

The assets with the project demonstrate the starter and completed projects in various states. The assets can be found in the folder maked Assets. You can also download the assets here:

https://www.raywenderlich.com/downloads/twin_stick_shooter_unity_55.zip

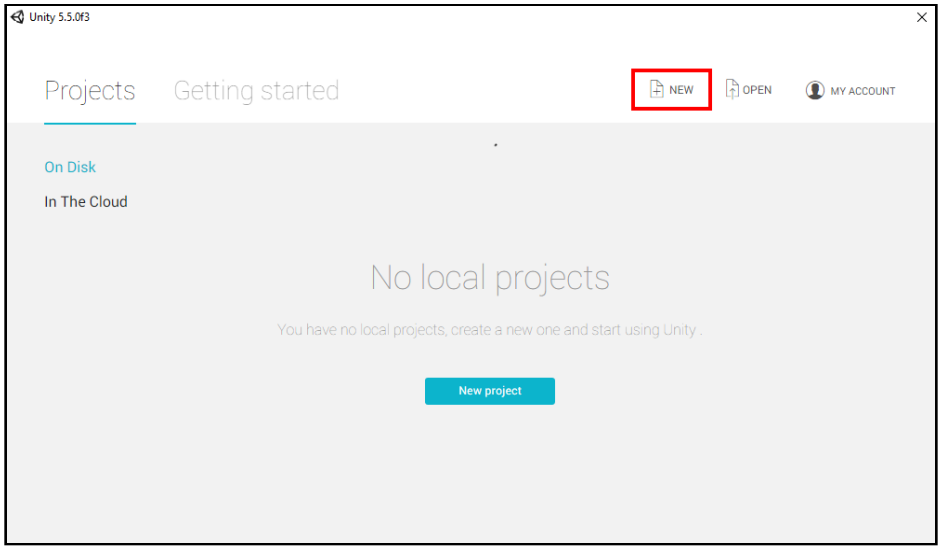
Download these files to your desktop and unzip them. Next, start or restart Unity. If you are using the current project, you can skip to the *"Learning the interface"* section.

Run the program once installation completes. The first thing you'll see is a dialog box asking for your Unity credentials.

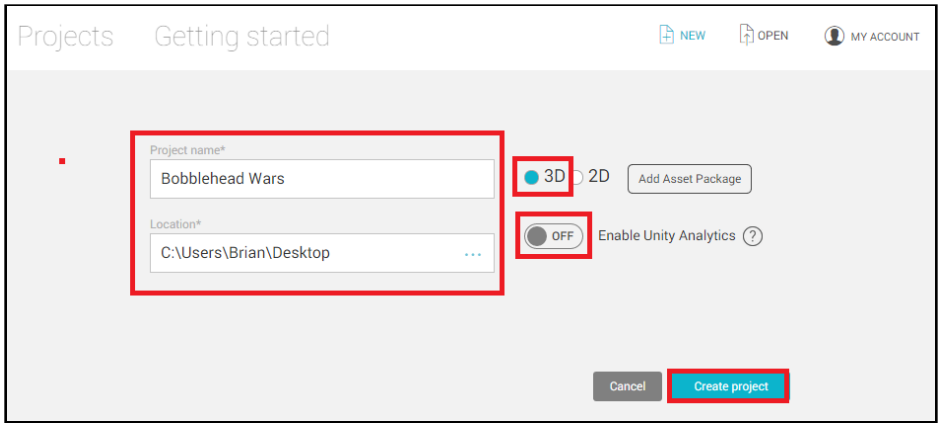


If you don't have an account, click **create one** and follow the steps. Unity accounts are free. You'll have to log in every time you fire it up, but the engine does have an offline mode for those times when you have no network.

Once you're logged in, you'll be presented with a project list that provides an easy place to access all of your projects. Since you don't have any projects, click the **New** button.



You should be looking at the project creation dialog. You'll notice that you have a few options, so fill them in as follows:



Here's what everything on this screen means:

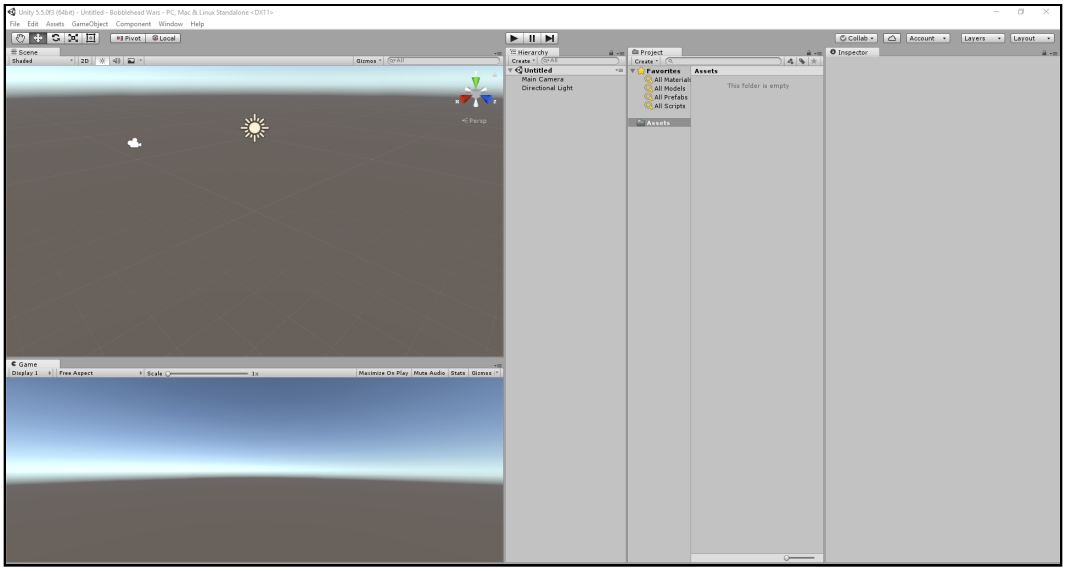
- The **Project name** represents the internal name of the game. It's not published with your final game, so you can name your projects whatever you like. Give this one the name **Bobblehead Wars**.
- The **Location** field is where you'll save the project and related items. **Click the three dots** in the **Location field** to choose a location on your computer.
- The **3D** option determines whether the game is 3D or 2D — it just configures the editor for that mode. You can switch between the two without starting a new project. For Bobblehead Wars, you want **3D**.
- The **Add Asset Package** button allows you to include additional assets in your game or any others you download from the Unity Asset Store. You don't need to do anything with this for now.
- Finally, you have the option to **Enable Unity Analytics**, which give you insight into your players' experiences. By reading the data, you can determine areas where players struggle and make changes based on the feedback. This book will not delve into analytics, so set the switch to set it to **off**.

Once you're ready, click the **Create project** button. Welcome to the world of Unity!

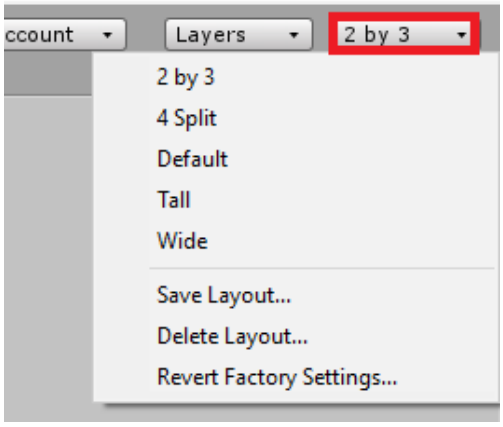
Learning the interface

When your project loads, you'll see a screen packed full of information. It's perfectly normal to feel a little overwhelmed at first, but don't worry - you'll get comfortable with everything as you work through the first few chapters.

Your layout will probably look like this:

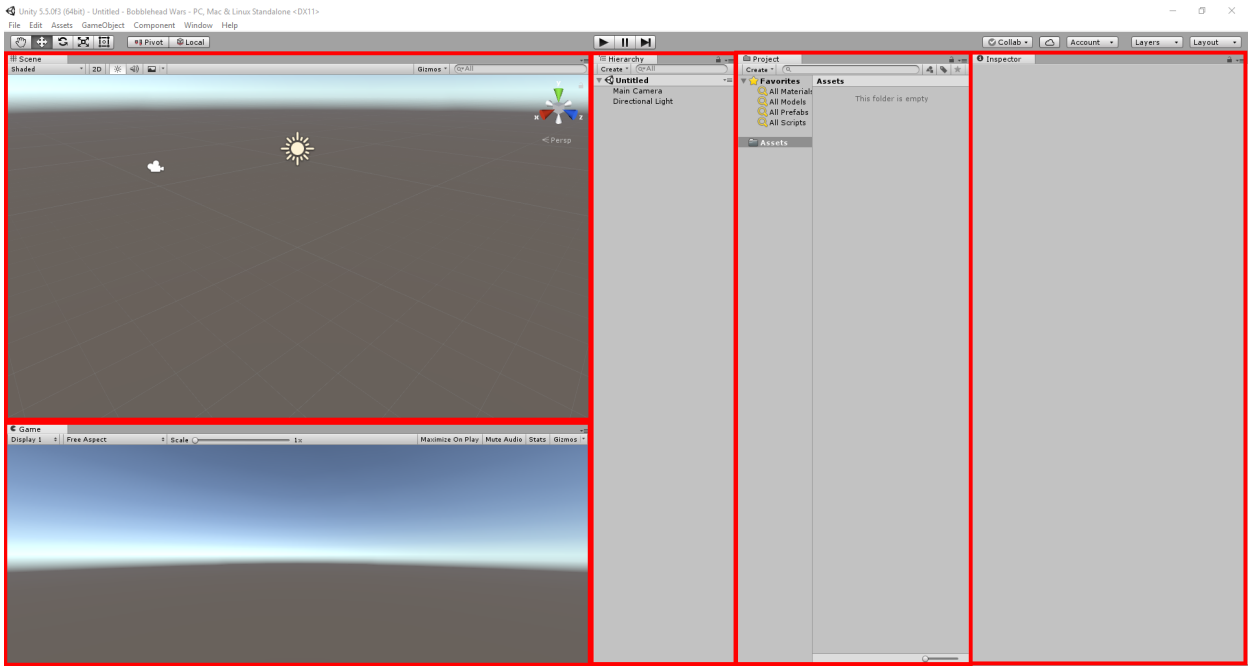


If not, click the Layout button in the top-right and select **2 by 3** from the dropdown.



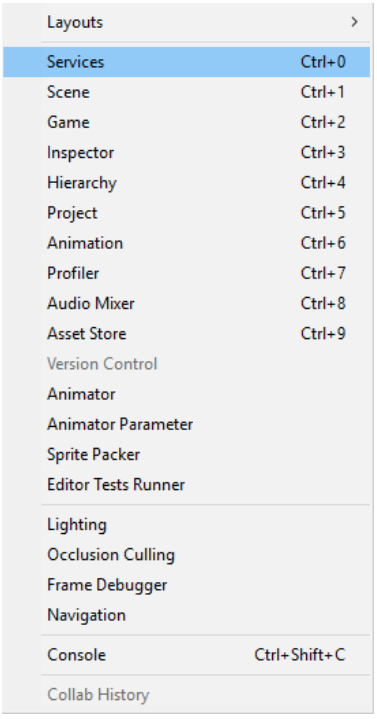
Each layout is composed of several different views. A **view** is simply a panel of information that you use to manipulate the engine. For instance, there's a view made for placing objects in your world. There's another view for you to play the game.

Here's what the interface looks like when broken down into individual views:



Each red rectangle outlines a view that has its own purpose, interface and ways that you interact with it. Throughout this book, you'll learn about many of these views.

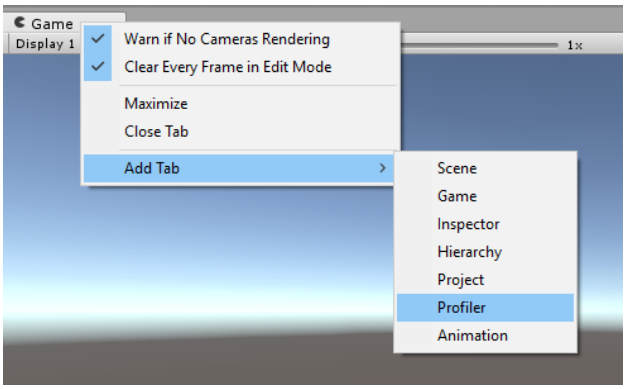
To see a list of all views, click the **Window** option on the menu bar.



The Unity user interface is completely customizable so you can add, remove, and rearrange views as you see fit.

When working with Unity, you'll typically want to rearrange views into a **Layout** that's ideal for a given task. Unity allows you to save layouts for future use.

In the Editor, look for the **Game tab** (the view to the lower left) and **right-click** it. From the drop-down, select **Add Tab** then choose **Profiler**.

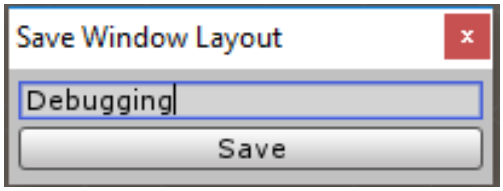


The Profiler view lets you analyze your game while it's running. Unfortunately, the profiler is also blocking the Game view, so you won't be able to play the game while you profile it — not so helpful.

Click and hold the **Profiler tab** and **drag it** to the **Scene tab** above.

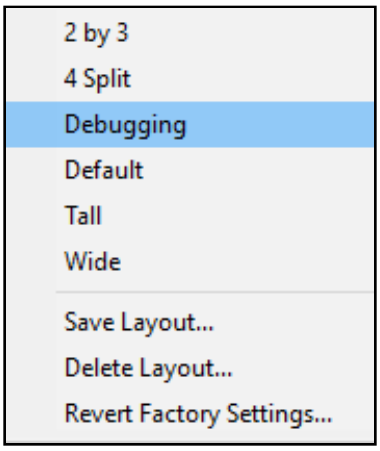
As you see, views can be moved, docked and arranged. They can also exist outside the editor as floating windows.

To save the layout, select **Window\Layouts\Save Layout...** and name it **Debugging**.



Whenever you need to access this particular layout, you can select the Layout button and choose Debugging.

When clicked, you'll see a listing of all your views.



You can also delete layouts. If you ever accidentally trash a stock layout, you can restore the default layouts.

Organizing your assets

Beginners to Unity might imagine that you develop your game from start to finish in Unity, including writing code, creating 3D models and textures, and so on.

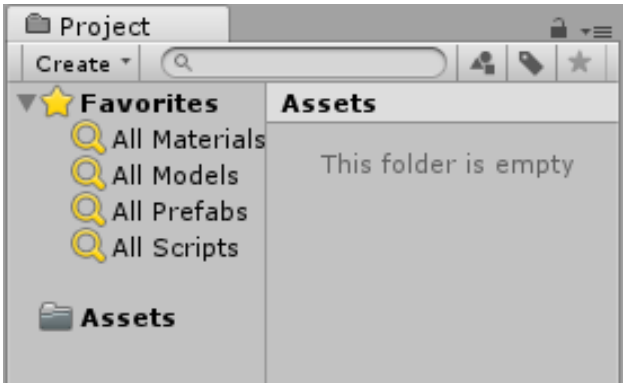
In reality, a better way of thinking about Unity is as an integration tool. Typically you will write code or create 3D models or textures in a separate program, and use Unity to wire everything together.

For Bobblehead Wars, we've created some 3D models for you, because learning how to model things in Blender would take an entire book on it's own!

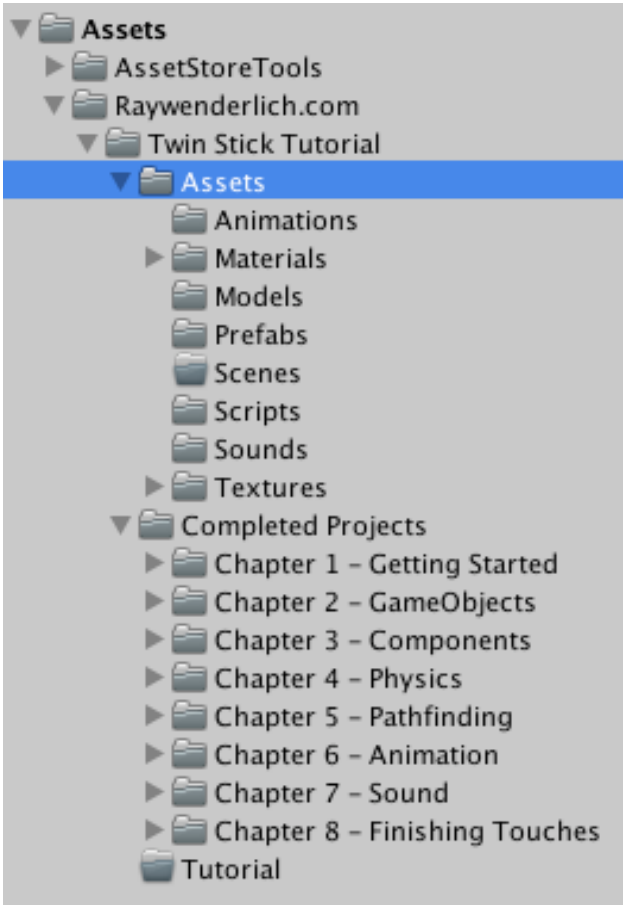
In this chapter, you will learn how to import models into your game.

But before you do, it pays to be organized. In this game, you're going to have a lot of assets, so it's critical to organize them in way that makes them easy to find.

The view where you import and organize assets is called the **Project Browser**. It mimics the organization of your file system. Here is the Project Browser for a fresh install.



If you are using the same project that you imported that assets into, your Project Browser may look like this:



If this is the case, make sure to save all your work into the **Assets** folder. For the sake of consistencty, all the following screenshots of the Project Browser are with an empty project.

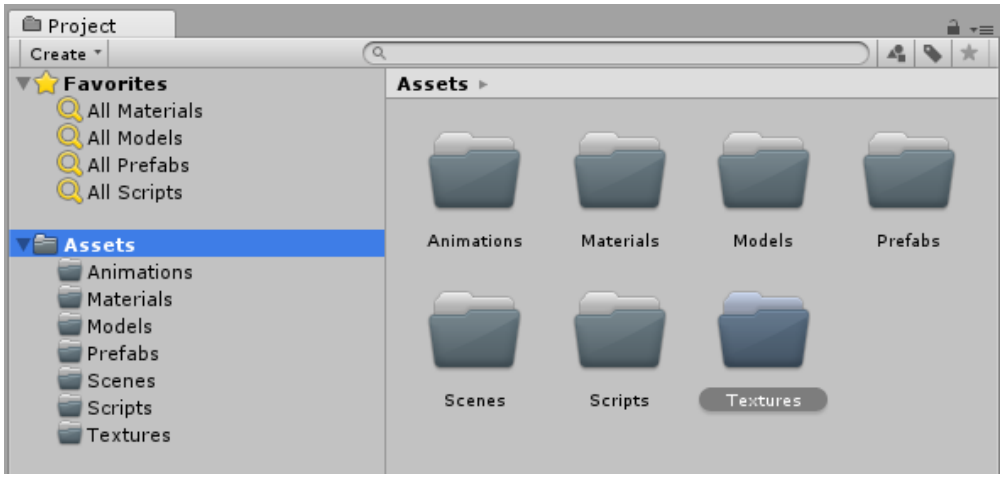
In the Project Browser, select the **Assets** folder and click the **Create** button. Select **Folder** from the drop-down and name it **Models**.

This will be home to all your models. You may feel tempted to create folders and manipulate files in your file system instead of the Project Browser. That's a bad idea — do not do that, Sam I Am!

Unity creates metadata for each asset. Creating, altering or deleting assets on the file system can break this metadata and your game.

Create the following folders: **Animations**, **Materials**, **Prefabs**, **Scenes**, **Scripts** and **Textures**.

Your Project Browser should look like this:



Personally, I find large folder icons to be distracting. If you also have a preference, you can increase or decrease the size by using the slider at the bottom of the Project Browser.

Note: All the screenshots in this book will show the smallest setting.

Importing Assets

Now that you've organized your folders, you're ready to import the assets for the game. If you imported the assets from the Unity Asset Store, you can skip to the section labeled, "*Add models to the Scene view*".

First, you'll import the star of the show: the space marine.

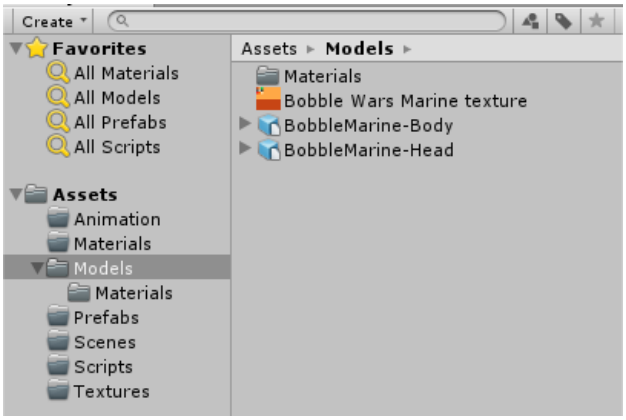
In the downloaded starter files for this chapter, open the **resources** folder and look for three files:

1. **BobbleMarine-Head.fbx**
2. **BobbleMarine-Body.fbx**
3. **Bobble Wars Marine texture.psd**

Drag these three files into the Models folder. Don't copy BobbleWars.unitypackage: that comes later.

What is an FBX file? FBX files typically contain 3D models, but they can also include textures and animations. 3D programs, such as Maya and Blender, allow you to export your models for import into programs such as Unity using this file format.

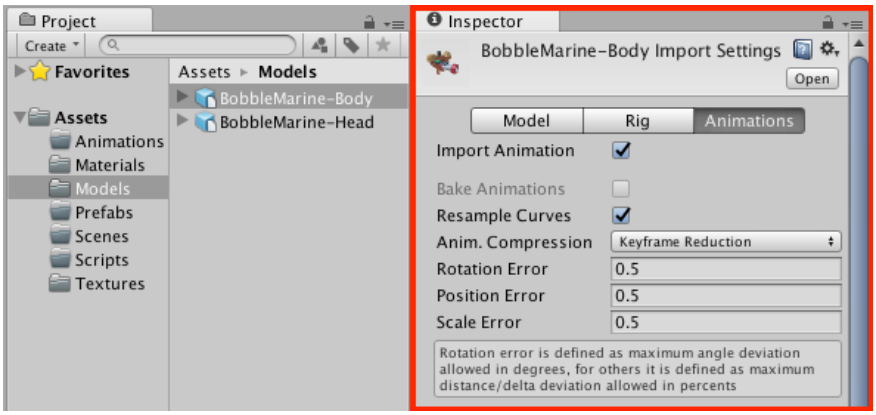
Select the **Models** folder and you'll see that you have a bunch of new files. Unity imported and configured the models for you and created a folder named Materials.



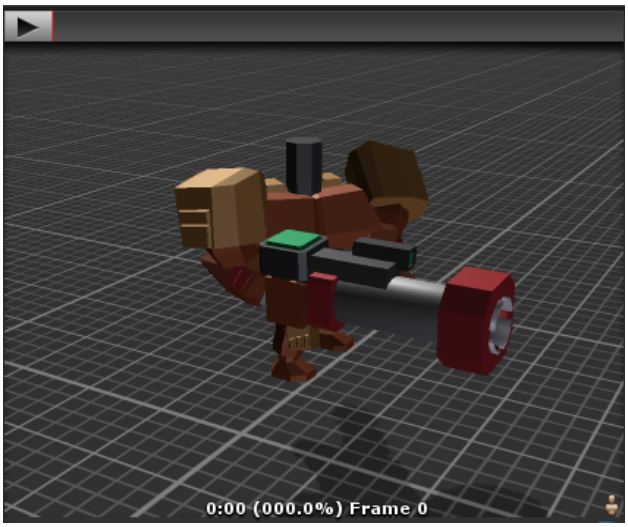
To keep things tidy, move **Bobble Wars Marine texture** from the Models folder to the **Textures** folder. Also move the **contents** of the newly generated **Materials** folder (in the Models folder) into the **parent-level Materials** folder, and then **delete** that new **Materials** folder by pressing **delete** (or command-delete on the Mac).

What are materials? Materials provide your models with color and texture based upon lighting conditions. Materials use what are known as shaders that ultimately determines what appears on screen. Shaders are small programs written in a specific shader language which is far beyond the score of this intro book. You can learn more about materials through Unity's included documentation.

Switch back to the Models folder and select **BobbleMarine-Body**. The Inspector view will now display information specific to that model as well as a preview.



If you see no preview, then its window is closed. At the bottom of the Inspector, find a **gray bar** then **drag it upwards** with your mouse to expand the preview.



The Inspector allows you to make changes to the model's configuration, and it allows changes to any selected object's properties. Since objects can be vastly different from one another, the Inspector will change context based on the object selected.

Installing Blender

At this point, you've imported the models and texture for the space marine. The models were in FBX format, and the texture was in PSD format.

We supplied the space marine models to you in the FBX format, as this is a popular format for artists to deliver assets. But there's another popular format you should understand how to use as well: Blender files.

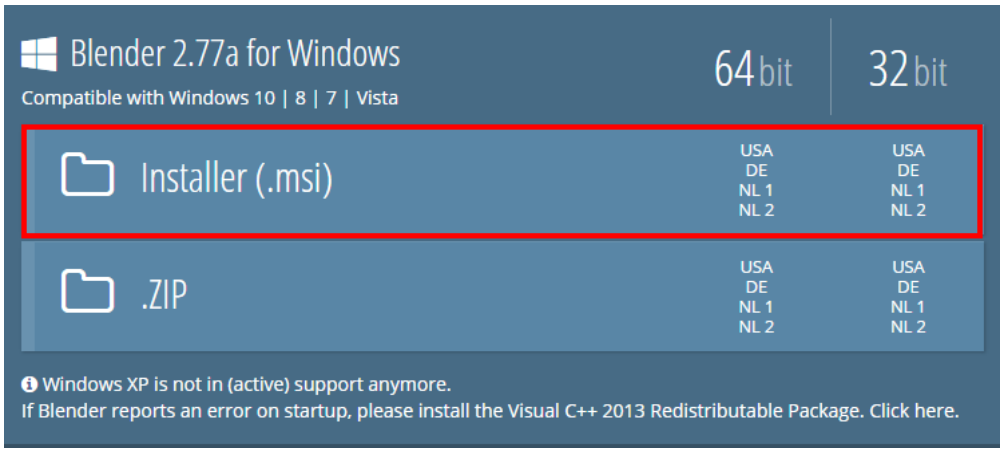
Unlike FBX, Blender files contain the source model data. This means you can actually edit these files inside Blender, and the changes will immediately take effect in Unity, unlike an FBX file.

With an FBX, you'd need to export and re-import the model into Unity *every time you change it*.

There is a small tradeoff for all this delicious functionality. For Unity to work with Blender files, you need Blender installed on your computer. Blender is free, and you'll be happy to know that you'll use it to make your own models in a few chapters.

> **Note:** Blender is not required if you have imported this tutorial from the Unity Asset Store.

Download and install Blender at the following URL: https://www.blender.org/download/



> **Note**: Blender evolves at a rapid pace, so the version you see on your desktop will probably be different than this screenshot.

After you install Blender, run the app and then quit. That's it - you can now use Blender files with Unity.

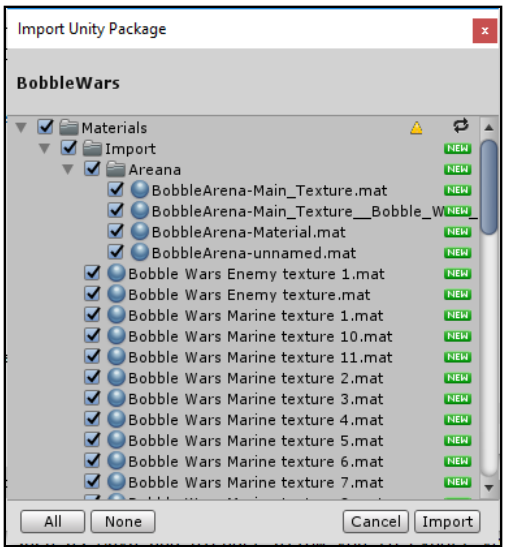
Importing Packages

Now that you've installed Blender, you can now import the rest of the assets.

The rest of the assets are combined into a single bundle called a **Unity package**. This is a common way that artists deliver assets for Unity, especially when you purchase them from the Unity store.

Let's try importing a package. Select **Assets\Import Package\Custom Package...**, navigate to your resources folder and select **BobbleheadWars.unitypackage** then click **Open**.

You'll be presented with a list of assets included in that package, all of which are selected by default. Note that some of these are Blender files, but there are also other files like textures and sounds as well. Click the **Import** button to import them into Unity.



The import will add a bunch of additional assets to your project. If you get a warning, just dismiss it.

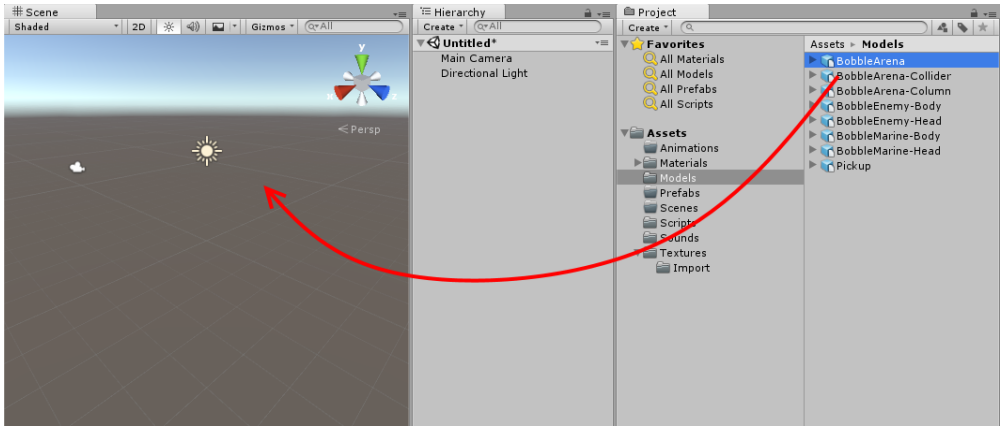
As you did before, to keep things tidy, **single click** the newly generated **Materials** folder (in the Models folder) and rename it to **Models**. Drag this new folder into the **parent-level Materials** folder.

Add models to the Scene view

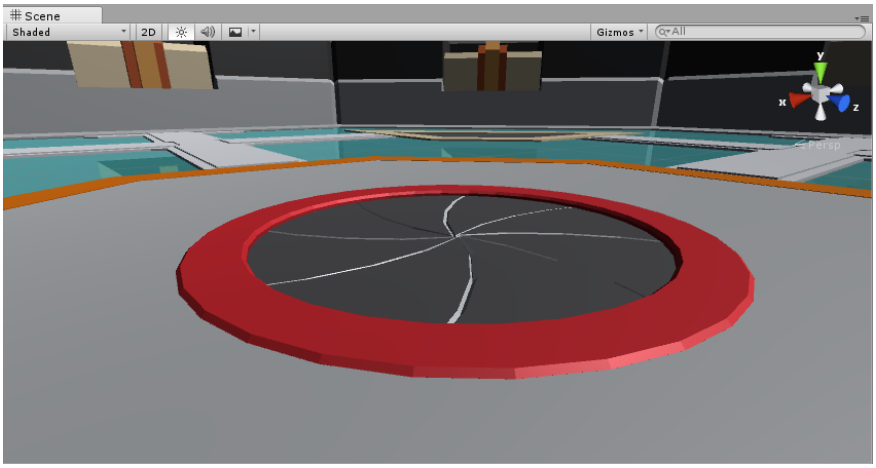
At this point, you have imported everything into Unity. It's time to start putting your game together, and you'll kick it off by adding your models to the Scene view.

The Scene view is where game creation happens. It's a 3D window where you'll place, move, scale and rotate objects.

First, make sure to select the **Scene** view tab. Then, in the Project Browser, select **BobbleArena** from the **Models** subfolder and drag it into the **Scene** view.



Check out the arena in the Scene view:



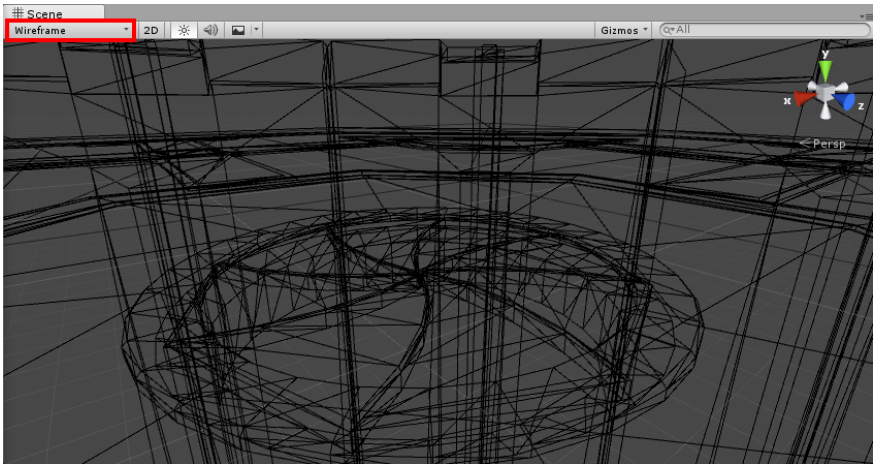
Pretty cool, eh?

The Scene view gives you a way to navigate your game in 3D space:

- **Right-click** and **rotate your mouse** to look around.
- Hold down the **right mouse button** and use the **WASD keys** to actually move through the scene.
- Moving too slow? Give it some juice by holding down the **Shift** key.
- **Scroll** with your **mouse wheel** to zoom.
- **Press** your **mouse wheel** and **move** your mouse to pan.

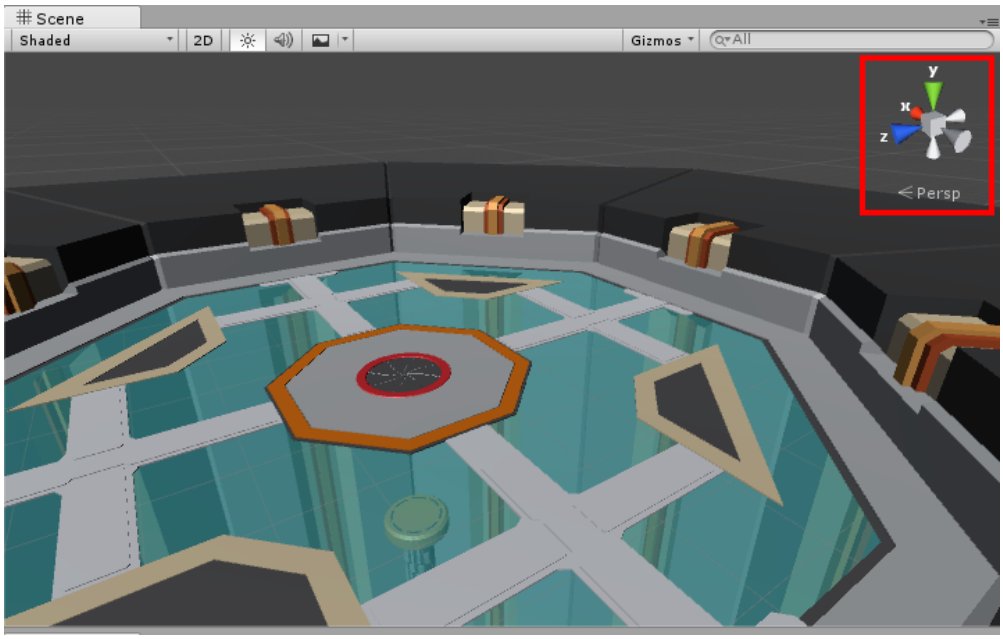
By default, the view displays everything with textures in a shaded mode. You can switch to other viewing modes such as wireframes or shaded wireframe.

Let's try this out. Just underneath the Scene tab, click the **Shaded dropdown** and select **Wireframe**. Now you'll see all your meshes without any textures, which is useful when you're placing meshes by eye.



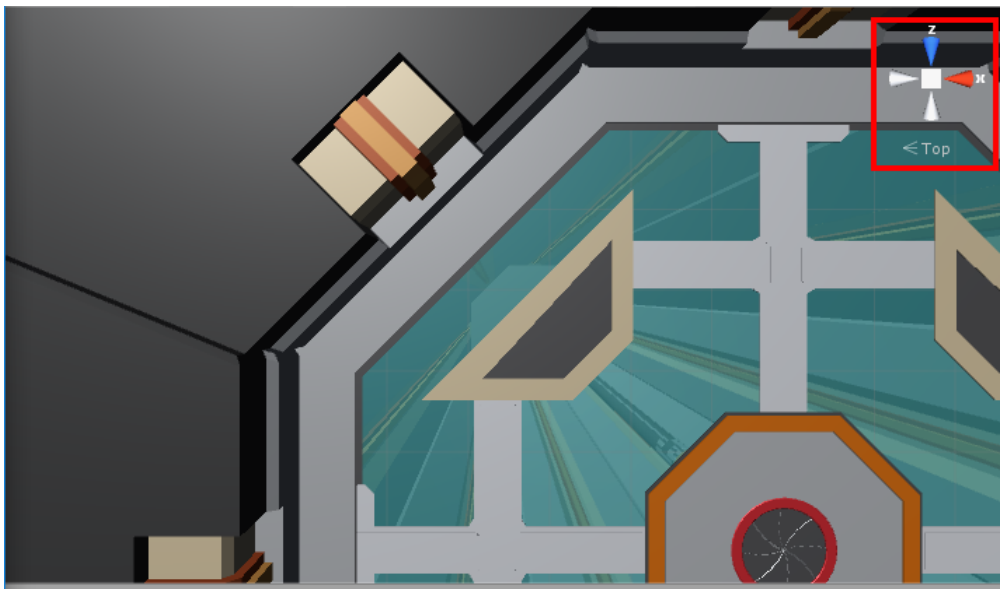
Switch the Scene view back to Shaded textures.

In the Scene view, you'll notice a gizmo in the right-hand corner with the word **Persp** underneath it. This means the Scene view is in perspective mode; objects that are closer to you appear larger than those that are farther away.

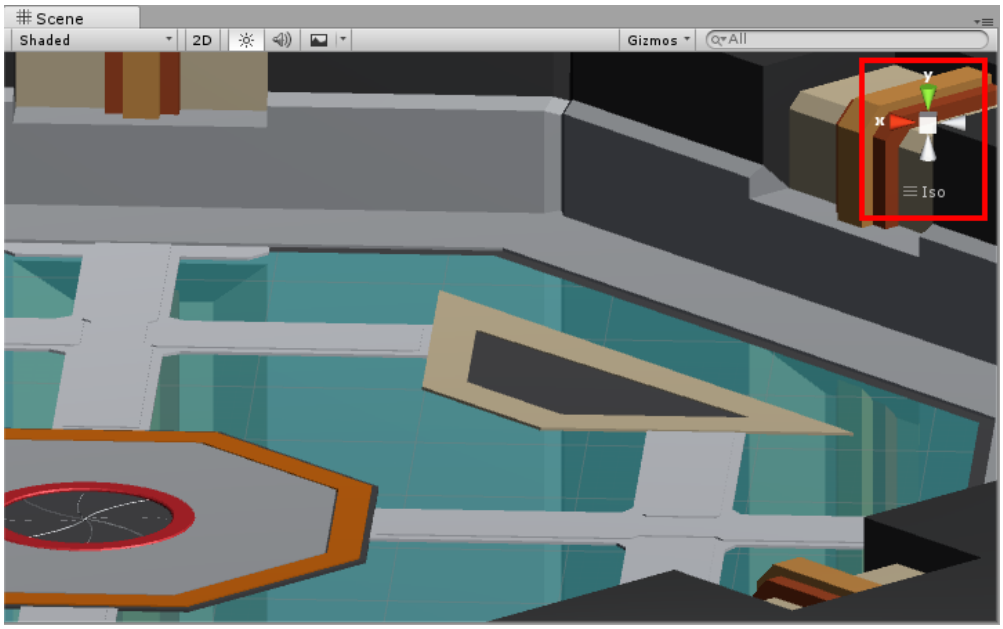


Clicking on a colored axis will change your perspective of the scene. For example, click the green axis and the Scene view will look down from the Y-axis. In this case, the Persp will read **Top** because you're looking at the world from that perspective.

How's it feel to be on top of the world? :]



Clicking the center box will switch the view into **Isometric mode** aka Orthographic mode. Essentially, objects are the same size regardless of their proximity to you.

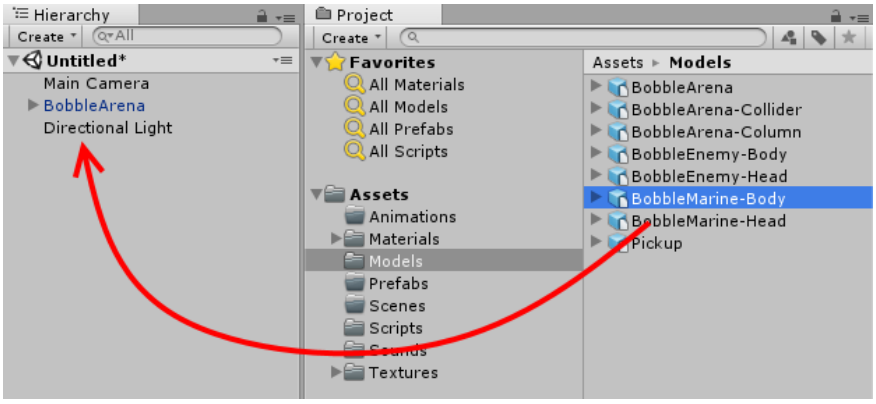


To return to Perspective mode, simply click the center box again. You'll learn more about Isometric mode later in this book.

Adding the hero

At this point, you have the arena set up, but it's missing the guest of honor!

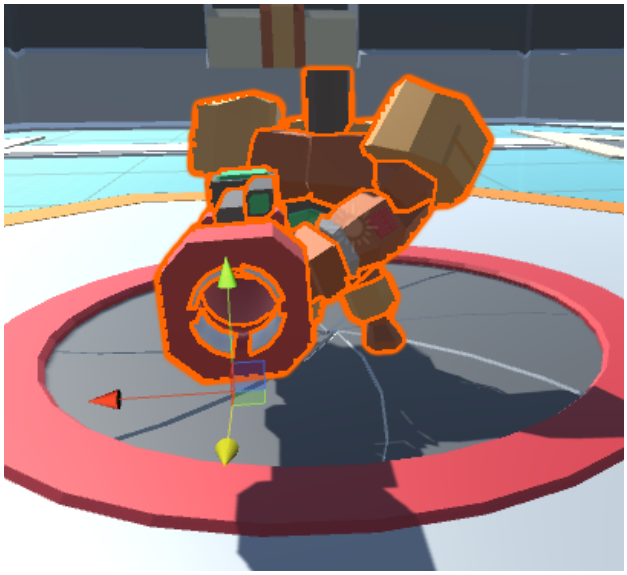
To fix this, in the Project Browser find and open the Models folder then drag **BobbleMarine-Body** into the Hierarchy view.



In Unity, games are organized by scenes. For now, you can think of a scene as a level of your game. The Hierarchy view is a list of all objects that are currently present in the scene.

Note that your scene already contains several objects. At this point, it contains your arena, the space marine, and two default objects: the main camera and a directional light.

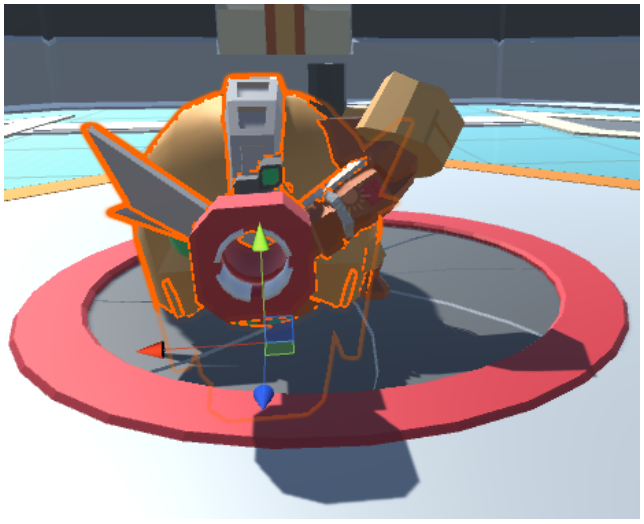
With the marine body still selected, **hover the mouse** over the Scene view and press the **F** key to zoom to the marine. This shortcut is useful when you have many objects in your scene, and you need to quickly get to one of them.



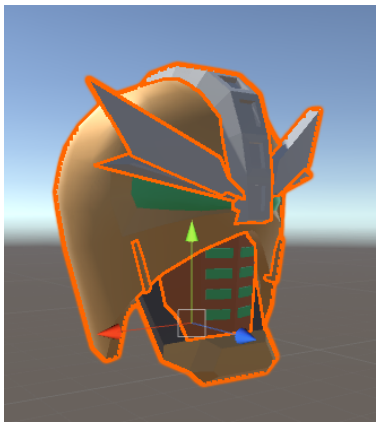
Don't worry if your space marine isn't placed at this exact position. You'll fix that later.

While the space marine's dismembered body is pretty intimidating, he's going to have a hard time seeing and bobbling without a head!

Drag the **BobbleMarine-Head** from the Project Browser into the Hierarchy. Chances are, the head will not go exactly where you'd hope.



Select the head in the **Hierarchy** to see its navigation gizmos. By selecting and dragging the colored arrows, you can move the head in various directions.



The colored faces of the cube in the middle of the head allow you to move the object along two axes at the same time. For example, selecting and dragging the red face — the x-axis — allows you to move the head along the y- and z-axes.

You can use the toolbar to make other adjustments to an object. The first item is the hand tool. This allows you to pan the Scene and is equivalent to holding down the middle mouse button.

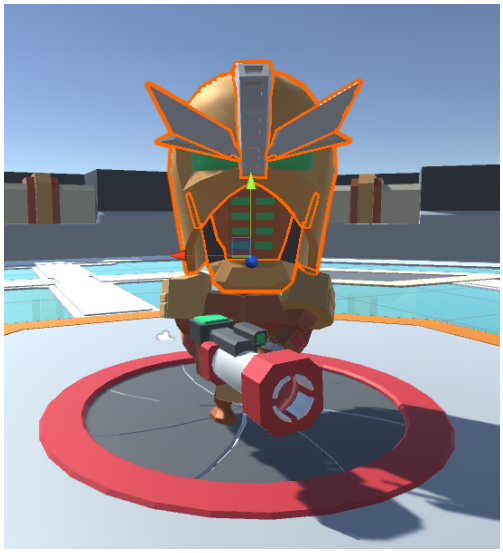
Select the position tool to see the position gizmo that lets you reposition the selected object.

Use the rotate tool to rotate the selected object.

The scale tool allows you to increase and decrease the size of an object.

The rect tool lets you rotate, scale and reposition sprites. You'll be using this when working with the user interface and Unity 2D.

Using the aforementioned tools, tweak the position of the helmet so it sits on the neck. Also, see if you can move the space marine so he's centered in the arena. In the next chapter, you'll position these precisely with the Inspector. When done, the marine should like this:



After positioning the space marine's head, select **File\Save Scene**. Unity will present you with a save dialog.

Name the scene **Main**, and after you finish creating it drag the file to the **Scenes** folder.

> **Note:** Unity does not autosave and unfortunately can be a little bit "crashy" at times. Make sure to save early and often. Otherwise, you will lose work (and possibly sanity).

Where to go from here?

Congratulations! You now have a space marine at the ready to kill all the alien monsters, and you've learned a lot about the Unity user interface and importing assets along the way.

Specifically, in this chapter you've learned:

- **How to configure the Unity layout**, including customizing it for specific tasks.
- **How to import assets** and organize them within Unity.
- **How to add assets to a scene** and position them manually.

In the next chapter, the aliens will finally catch up with the space marine - and you'll learn about GameObjects and Prefabs along the way!

Chapter 2: GameObjects

By Brian Moakley

In the last chapter, our brave hero was left alone with fantasies of blasting aliens running through his barely attached bobblehead.

In this chapter, you'll make his dreams come true. But first, you must understand one crucial concept: **GameObjects**.

> **Note**: This chapter's project continues from the previous chapter. If you didn't follow along or want to start fresh, please use the starter project from this chapter.

Introducing GameObjects

In Unity, game scenes contain objects with a name you'll never guess: GameObjects. :] There are GameObjects for your player, the aliens, bullets on the screen, the actual level geometry — basically, everything in the game itself.

Just like the Project Browser contains all assets, the Hierarchy contains a list of GameObjects in your scene.

To see this, you'll need to have your project open, so if it isn't open already, do so now.

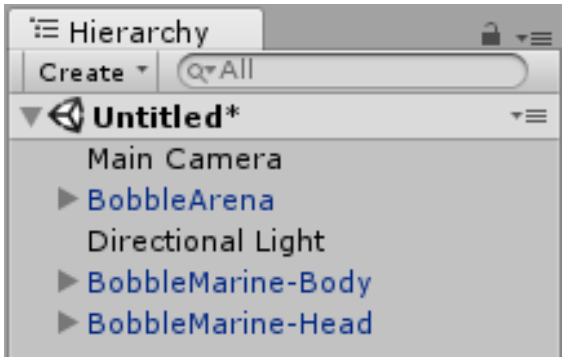
Note: Unity's way of opening project files is a bit strange.

You can navigate your file system and look for a scene file. Double-click the scene then Unity will open your project with the current scene selected.

Or, if you start Unity or click **File\Open Project**, you'll see a project list. Select the desired project and Unity will take care of the rest.

If your project isn't listed, click the **Open** button to bring up a system dialog box. Instead of searching for a particular file, try navigating to the top-level directory of your Unity project and click **Select Folder**. The engine will detect the Unity project within the folder and open it.

Once your project is open, take a look at your Hierarchy and count the GameObjects.

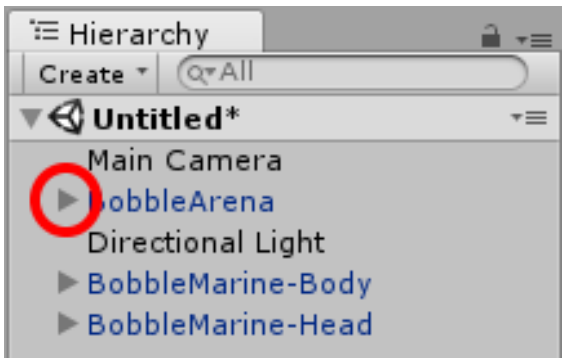


Your first thought may be *three* because you added three GameObjects in the last chapter: arena, space marine body and space marine head.

However, there are two other GameObjects: **Main Camera** and **Directional Light**. Remember how Unity creates these by default? Yes, these are also GameObjects.

Yet, there are even more GameObjects. You'll notice that there are disclosure triangles to the left of the GameObjects you imported.

Holding down the **Alt button** on PC or Option on Mac, **click** each **disclosure triangle**.



As you can see, you have so many GameObjects:

Three important points to remember:

- **GameObjects can contain other GameObjects**. On a base level, this useful behavior allows organizing and parenting of GameObjects that are related to each other. More importantly, changes to parent GameObjects may affect their children — more on this in just a moment.
- **Models are converted into GameObjects**. Unity creates GameObjects for the various pieces of your model that you can alter like any other GameObject.
- **Everything contained in the Hierarchy is a GameObject**. Even things such as cameras and lights are GameObjects. If it's in the Hierarchy, it's a GameObject that's subject to your command.

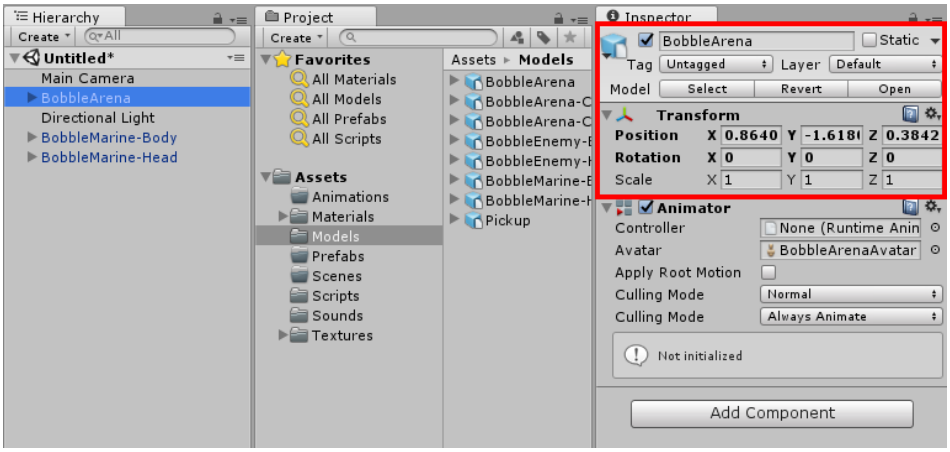
Our hero is so bored that he's picking his nose with his gun. You need to get him moving, but first, you need to reposition your GameObjects.

Moving GameObjects

Before starting, **collapse** all the GameObject trees by clicking the disclosure triangles.

Select **BobbleArena** in the Hierarchy and take a moment to observe the **Inspector**, which provides information about the selected GameObject.

GameObjects contain a number of components, which you can think of as small units of functionality. There is one component that all GameObjects contain: the **Transform** component.

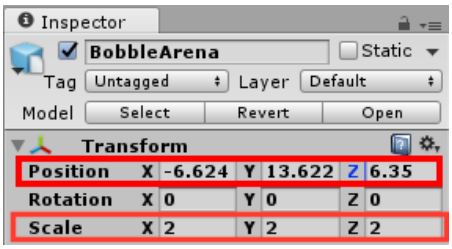


The Transform component contains the position, rotation, and scale of the GameObject. Using the inspector, you can set these to specific numbers instead of having to rely upon your eyeballs. When hovering the mouse over the axis name, you'll see arrows appear next to the pointer.

Press the **left mouse button** and **drag** the mouse either left or right to adjust those numbers. This trick is an easy way to adjust the values by small increments.

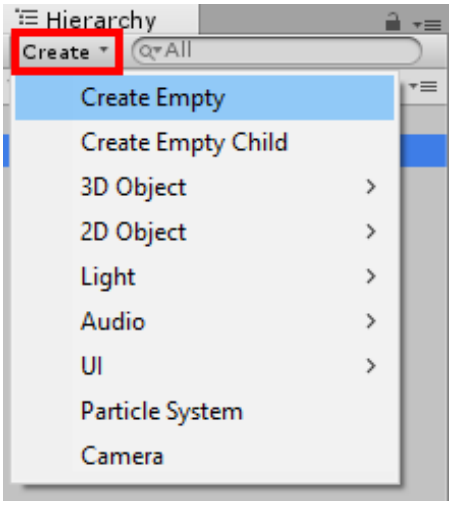
With **BobbleArena** selected, set **Position** to (**-6.624**, **13.622**, **6.35**). As I developed this game, the arena ended up in this position. You could just as well place it in the center of the game of the game world.

Set the **Scale** to (**2.0**, **2.0**, **2.0**). This gives the player more room to navigate the arena.



If you zoom out of the Scene view, you'll probably notice the space marine is suspended in the void; mostly likely, he's questioning his assumptions about gravity. You could move his head and then his body, but your life will be much easier if you group his body parts into one GameObject.

In the Hierarchy, click the **Create** button and select **Create Empty**.



An **empty** is a GameObject that only has only the one required component that all GameObjects have - the Transform component, as you learned earlier.

> **Note**: You can also create an empty GameObject by clicking **GameObject\Create Empty**. This goes for other things such as components. There is no "preferred" way to do things — go with whatever works best for your workflow.

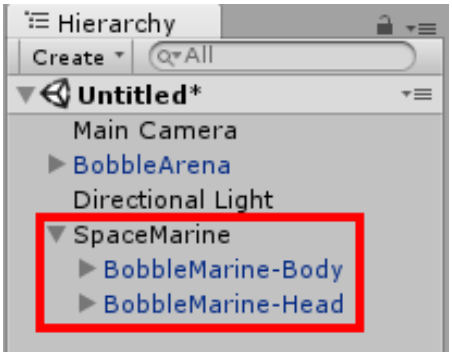
Parenting the space marine

In the Hierarchy, you'll see your new GameObject creatively named: **GameObject**. **Single-click** the **GameObject** and name it **SpaceMarine**.

You can insert spaces in GameObjects' names, e.g., *Space Marine*. However, for the sake of consistency, you'll use camel casing for names in this book.

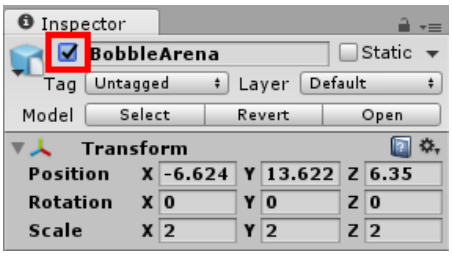
Drag **BobbleMarine-Body** and **BobbleMarine-Head** into the **SpaceMarine** GameObject.

A few things happen when you parent GameObjects. In particular, the position values for the children change even though the GameObjects don't move. This modification happens because GameObject positions are always relative to the parent GameObject.



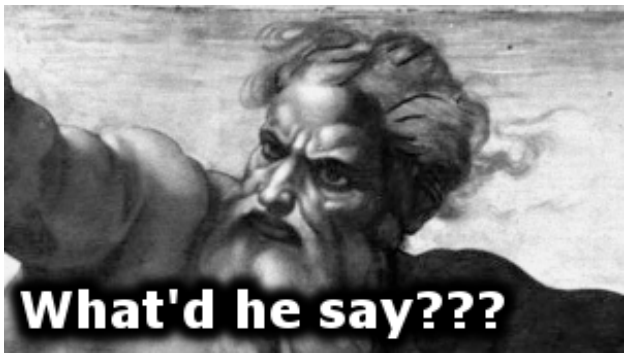
Select the **SpaceMarine** in the Hierarchy. Go to the **Scene** view and press **F** to focus on it. Chances are, the arena is blocking your view.

Thankfully, you don't need to get Dumbledore on speed dial. You can make it disappear! Select **BobbleArena** in the Hierarchy, and in the **Inspector**, **uncheck** the box to the left of the GameObject's name. This will make the arena disappear.



You only should see the hero now. Select the **SpaceMarine** GameObject. In the Inspector, **mouse over** the **X position label** until you see the scrubber arrows. Hold the **left mouse button** and move your mouse **left or right**. Notice how all the GameObjects move relative to the parent.

As you can see, having parents does have its advantages. No offense to any parentless deities out there.



When you parent a GameObject in another, the position of the child GameObject won't change. The difference is that of the child GameObject is now positioned relative to the parent. That is, setting the child to (0, 0, 0) will move the child to the center of the parent versus the center of the game world.

You'll do this now to assemble your marine.

Select **BobbleMarine-Body**, and in the **Inspector**, set **Position** to (**0**, **0**, **0**). Go select **BobbleMarine-Head** and set **Position** to `(**-1.38**, **6.16**, **1.05**) in the Inspector.

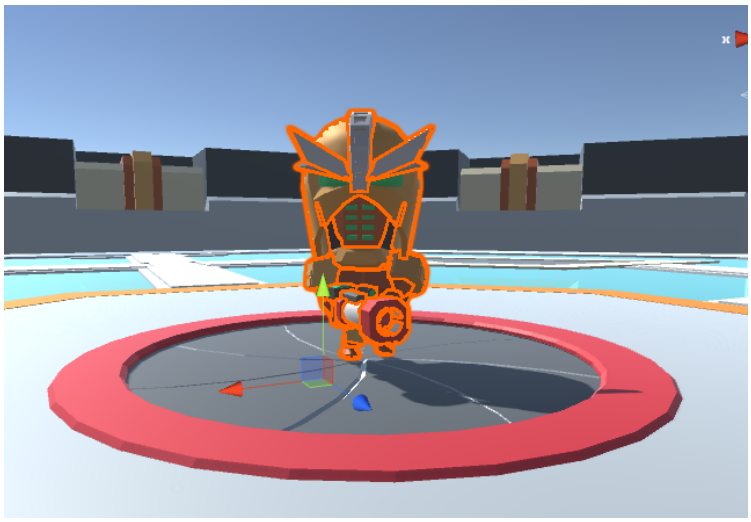
Congratulations! Your hero is assembled.

Positioning the marine

Now to it's time to place the space marine in his proper starting position. Select **BobbleArena**, and in the Inspector, **check** the box next to the name to re-enable it.

Select **SpaceMarine**, and in the Inspector, set its **position** to (**-4.9**, **12.54**, **5.87**). Also, set the **rotation** to (**0**, **0**, **0**). Your marine should end up directly over the hatch. If this isn't the case, then feel free to tweak the values until that is the case.

Once the hero is in place, press **F** in the Scene view so you can see him standing proud.



The hero should now be positioned precisely over the elevator, ready to rock. Unfortunately for him, his grandiose rock party will soon degrade into a bug hunt.

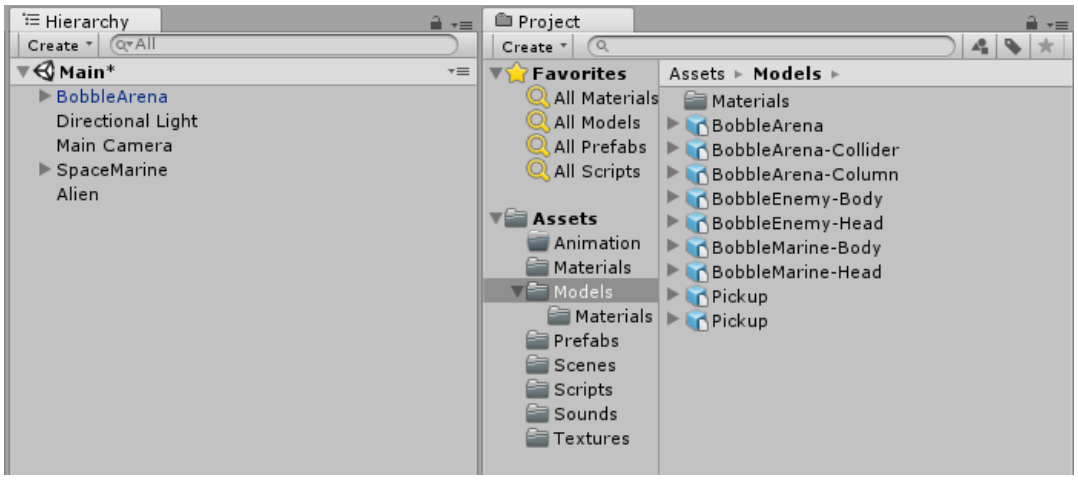
Creating a prefab

This game features creepy crawly bugs, and like the hero, they’re composed of many pieces. Some assembly is required.

In the Hierarchy, click the **Create** button and select **Create Empty** from the drop-down. **Single-click** the GameObject to name it **Alien**.

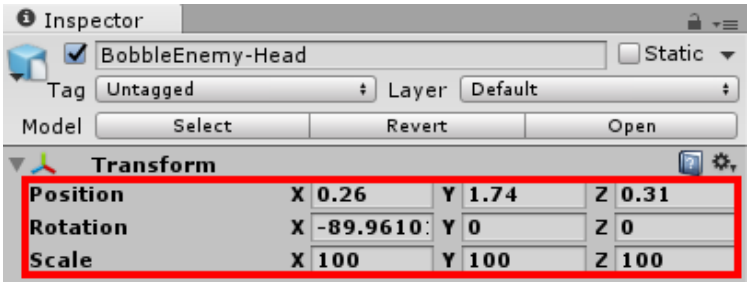
Select **Alien** in the Hierarchy, and in the Inspector, set the **position** to: (**2.9**, **13.6**, **8.41**).

From the Project Browser, drag **BobbleEnemy-Body** from the **Models** folder into the **Alien GameObject**.

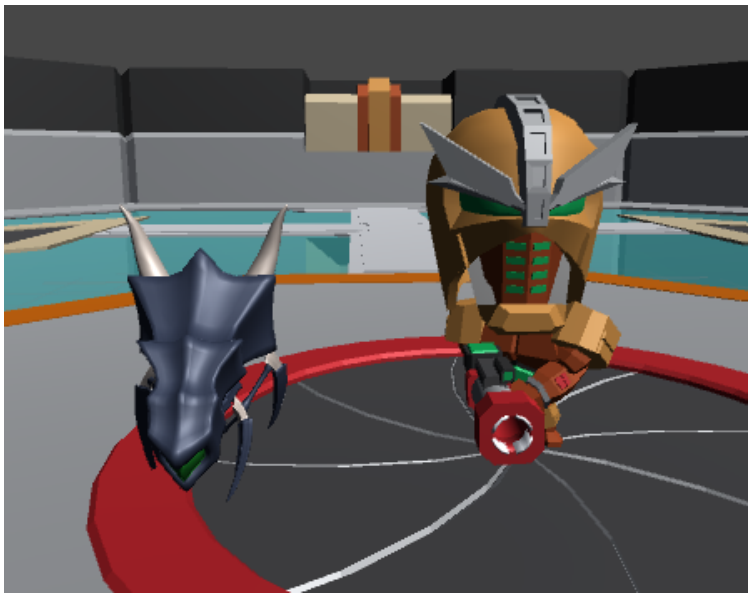


Set **BobbleEnemy-Body Position** to (**0**, **0**, **0**). Now the alien and hero should be side by side in the arena.

As creepy as the alien is without a head, the hero needs more to shoot at than that spindly little frame. From the Project Browser, drag **BobbleEnemy-Head** into the **Alien GameObject**. Set **Position** to (**0.26**, **1.74**, **0.31**), **Rotation** to (**-89.96**, **0**, **0**) and **Scale** to (**100**, **100**, **100**).



That's one fierce little bug. They go together so well that you could mistake them for the next superstar crime-fighting duo.

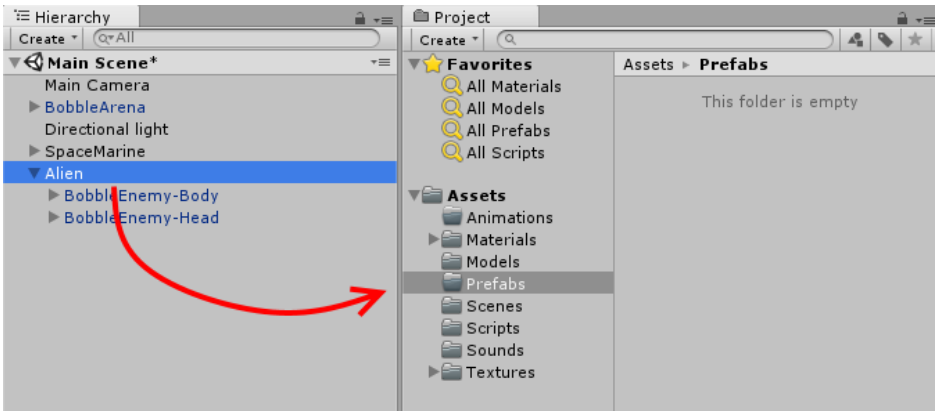


At this point, you have one parent GameObject for the hero and another for the alien. For the hero, this works great because you need only one. For the alien, you're going to need many — so, so many.

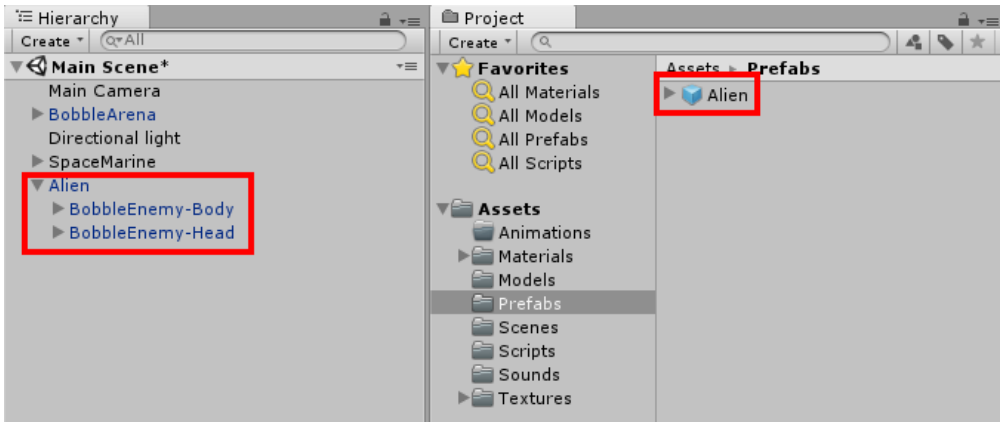
You could copy and paste the alien to make clones, but they'd all be individuals. If you needed to make a change to the alien's behavior, you'd have to change each instance.

For this situation, it's best to use a **prefab**, which is a master copy that you use to make as many individual copies as you want. Prefabs are your friend because when you change anything about them, you can apply the same to the rest of the instances.

Making a prefab is simple. Select the **Alien GameObject** and **drag** it into the **Prefabs folder** in the Project Browser.

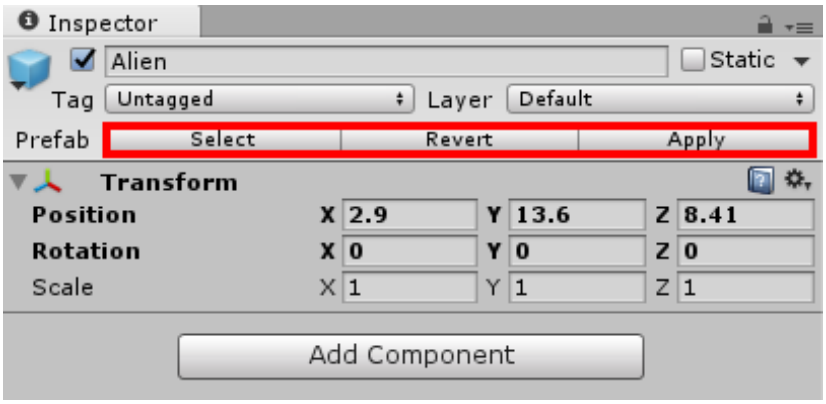


A few things have changed. There's a new entry in your Prefabs folder with an icon beside it. You'll also note the name of the GameObject in the Hierarchy is now blue:



Note: You don't have to drag your model into that specific folder to create a prefab — all that's required is dragging a GameObject into *any* folder in the Project Browser. Having a Prefabs folder is simply good housekeeping.

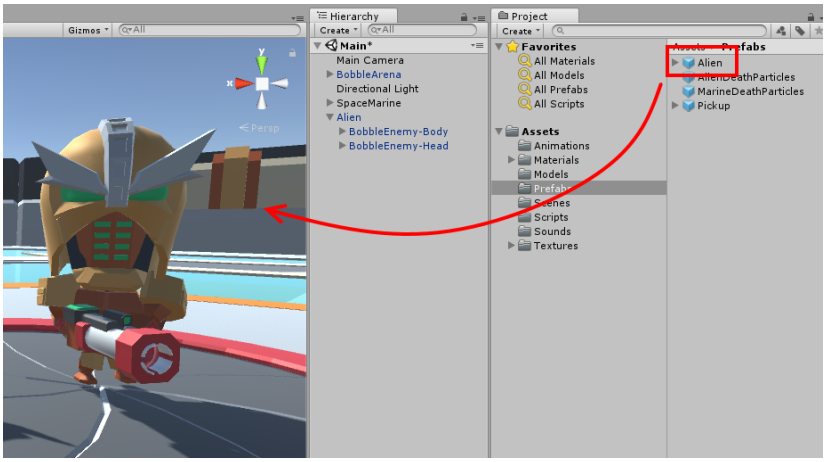
The blue indicates the GameObject has been either instanced from a prefab or a model, such as the BobbleArena. Select the **Alien** GameObject in the Hierarchy and look at the Inspector; you'll notice some additional buttons:



Here's the breakdown of these new buttons:

- **Select** will select the prefab inside the Project Browser. This is useful when you have lots of files and want easy access to the prefab to make changes.
- **Revert** will undo changes you've made to your instance. For example, you might play around with size or color but end up with something horrible, like a viciously pink spider. You'd click the Revert button to restore sanity.
- **Apply** will apply any changes you made to that instance to its prefab. All instances of that prefab will be updated as well.

Creating a prefab instance is quite easy. Select the **Alien** prefab in the Project Browser and **drag it** next to your other Alien in the Scene view.

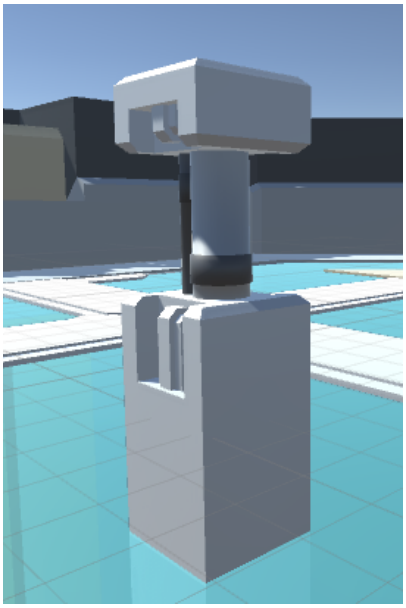


You can also drag an instance to the Hierarchy. As you can see, creating more aliens is as easy as dragging the `Alien prefab` from the Project Browser. But you don't need droves of aliens yet, so **delete** all the Aliens from the Hierarchy. You delete a GameObject by selecting it in the Hierarchy, and pressing Delete on your keyboard, (Command–Delete on a Mac), or you can right-click it and select **Delete**.

Fixing the models

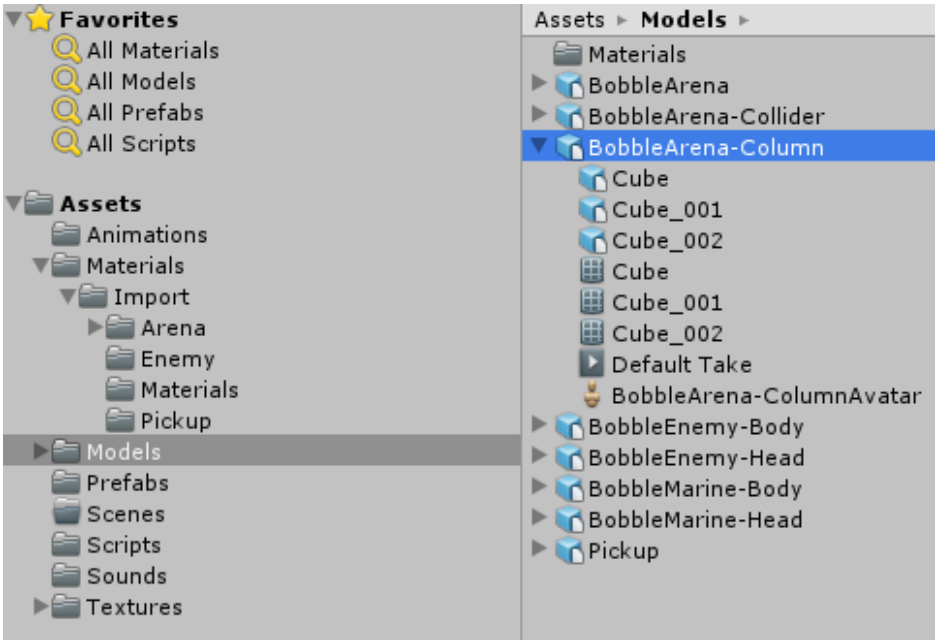
The next to-do is fixing some of your models. In this case, Unity imported your models but lost references to the textures.

In the **Models** folder of your Project Browser, drag a **BobbleArena-Column** into the Scene view. You'll find a dull white material.

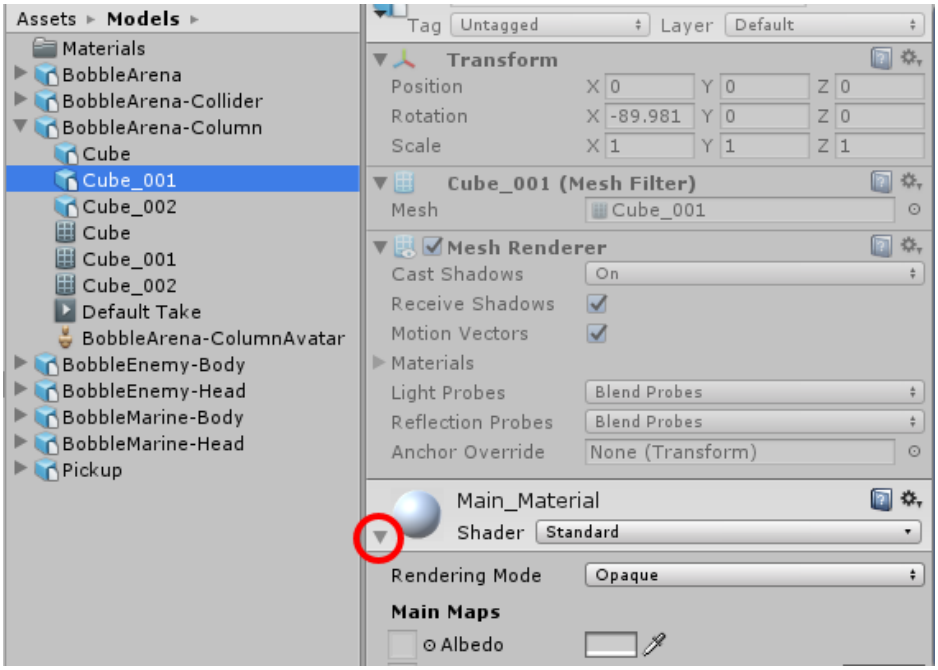


If a material name or texture name changes in the source package, Unity will lose connection to that material. It tries to fix this by creating a new material for you but with no textures attached to it.

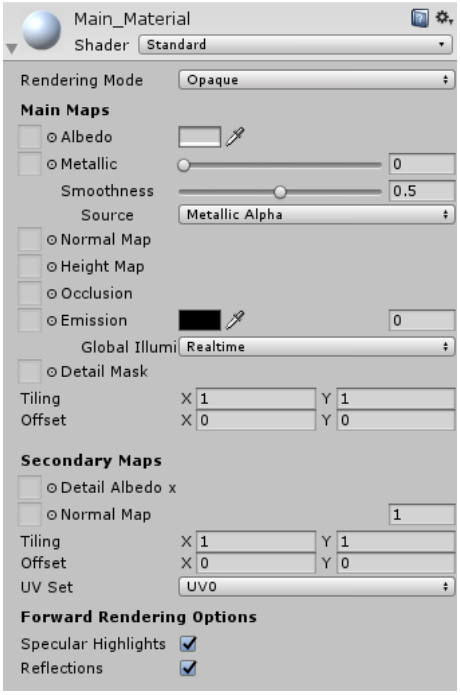
To fix this, you have to assign a new texture to the material. In the Project Browser, select the **Models** subfolder and then, expand the **BobbleArena-Column** to see all the child objects.



Next, select **Cube_001** in the Project Browser, and in the Inspector, **click the disclosure triangle** for the **Main_Material** shader.

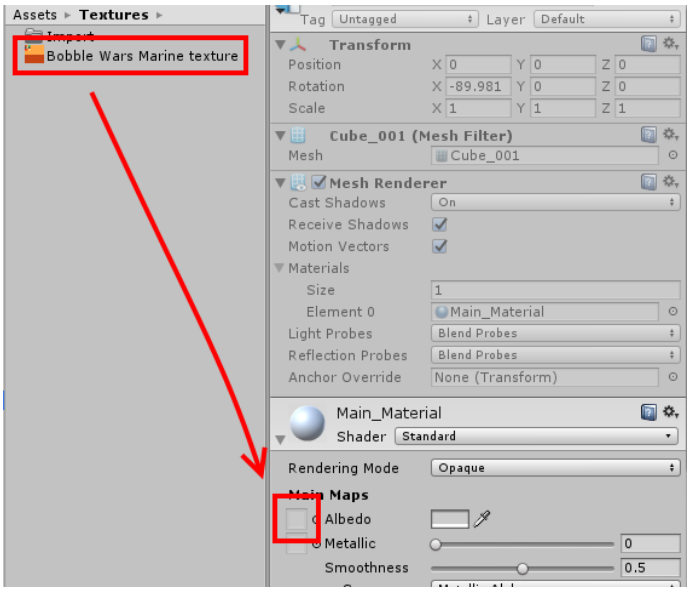


You'll see that there are a lot of options! These options configure how this material will look.

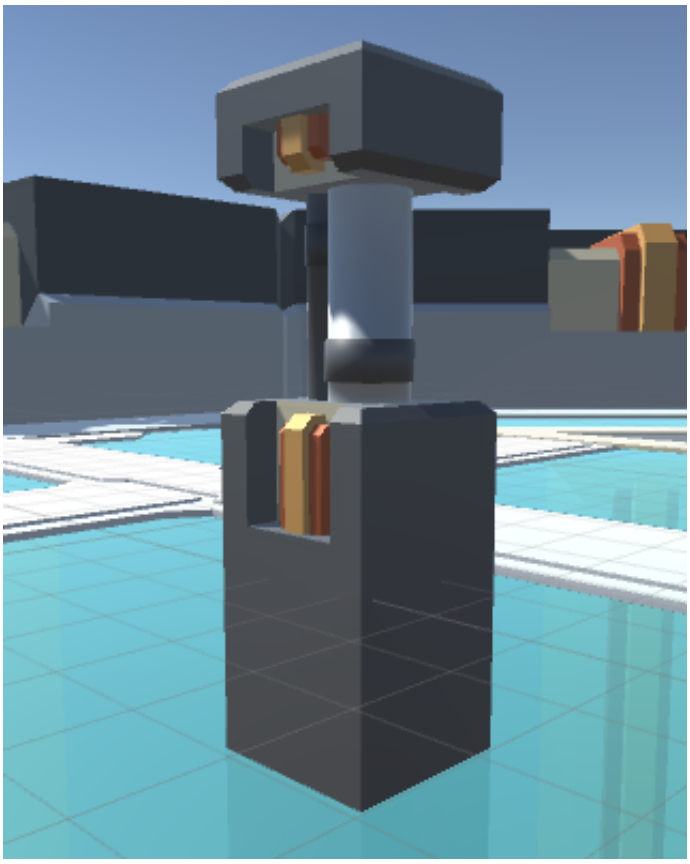


For example, the Metallic slider determines the metal-like quality of the material. A high value metallic value means the texture will reflect light much like metal. You'll notice a grey box to the left of most of the properties. These boxes are meant for textures.

In this case, all you want is a an image on your model. In the Project Browser, select the Textures folder. **Click and drag** the **Bobble Wars Marine texture** to the **Albedo** property box located in the shader properties.

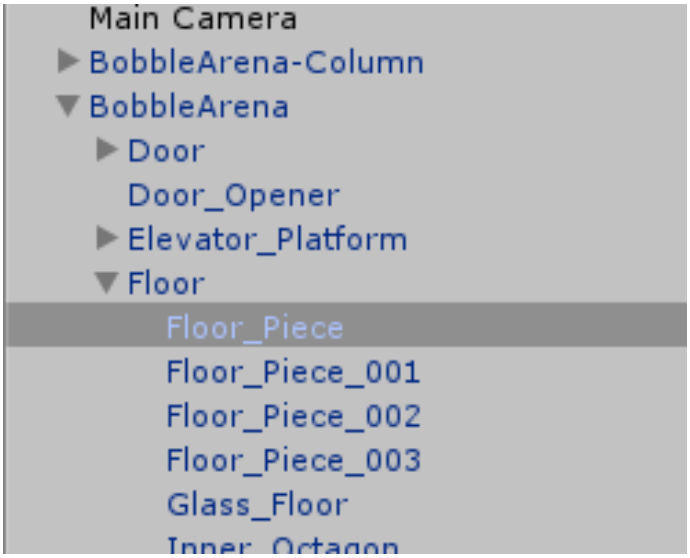


The Albedo property is for textures, but you can put colors there are well. Once you do this, your columns will now have a texture.

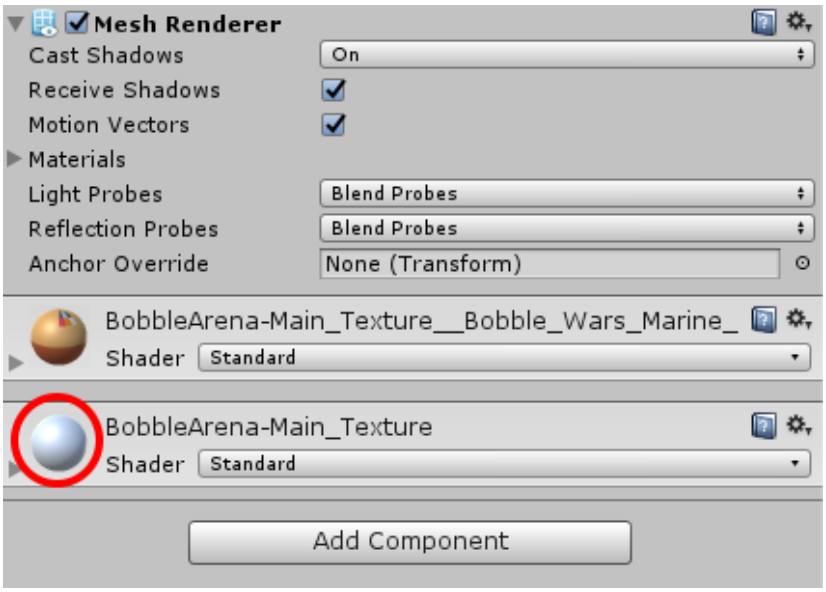


The arena also suffered a texture issue.

In the Hierarchy, expand the **BobbleArena** and then expand **Floor**. Select the **Floor Piece**.

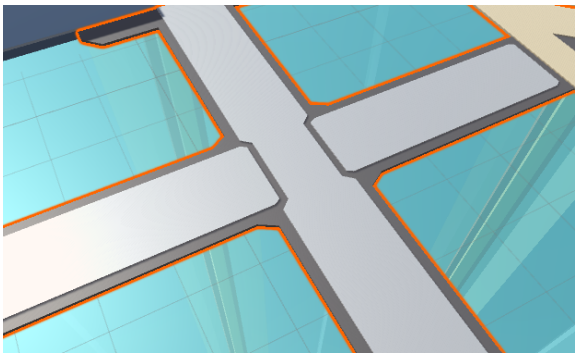


In the Inspector, you'll notice two materials attached to the floor piece. The **BobbleArena-Main_Texture** material does not have a texture assigned to it. You can tell because the material preview is all white.



Like you did for the column, select the **Textures** folder in the Project Browser. **Click and drag** the **Bobble Wars Marine texture** to the **Albedo** property box located in the shader properties.

Your floor will now acquire borders. How stylish!



You'll also notice that not just one, but all the floor sections acquired borders. This is because they all use the same material.

Adding obstacles

Now, that you have your models fixed, it's time add a bunch of columns to the arena. You'll make a total of seven columns, and you'll use prefabs for this task.

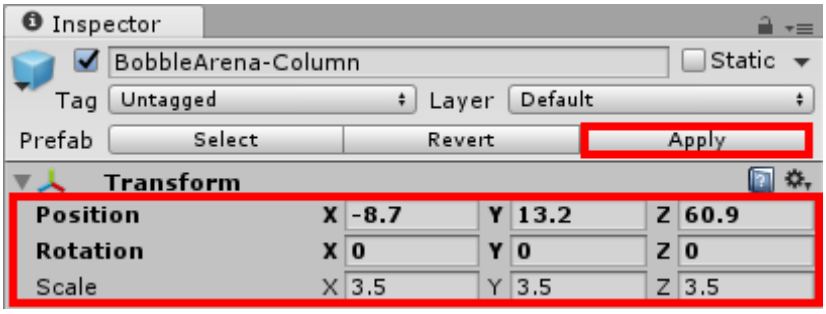
> **Note:** Whenever it seems like you should make a duplicate of a GameObject, use a prefab instead — it's another best practice. Some Unity developers insist

on using prefabs for everything, even unique objects.

The thinking is that it's much easier to create a prefab and make duplicates than it is to turn a group of existing GameObjects into prefab instance. The former method requires minimal work whereas the latter requires you to extract the commonalities into a prefab while maintaining any unique changes for each instance. This results a lot of work.

In the Hierarchy, drag the **BobbleArena-Column** into your **Prefabs** folder to turn it into a prefab.

With **BobbleArena-Column** instance still selected in the Hierarchy View, go to the Inspector and set **position** to (**1.66**, **12.83**, **-54.48**). Set **scale** to (**3.5**, **3.5**, **3.5**). You *do* want all the prefabs to be the same scale, so click the **Apply** button.



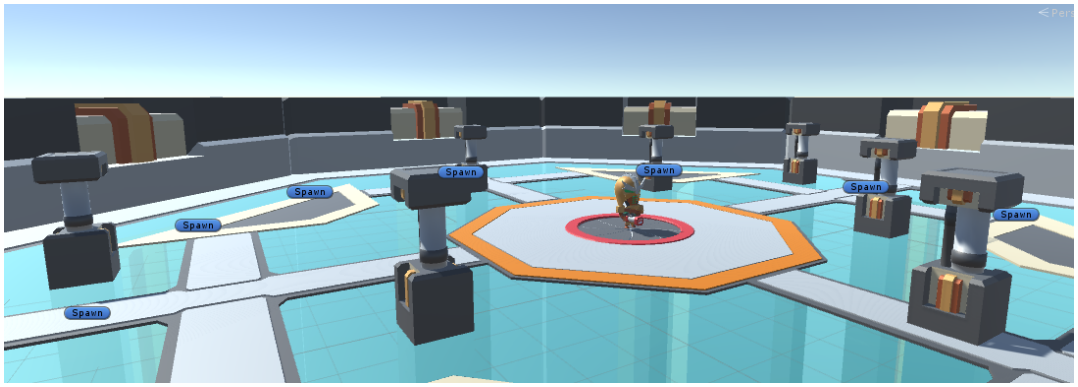
Now it's time for you to make the rest of the columns.

Dragging one column at a time from project to project in the Scene view can be a little tedious, especially when there are several instances. Duplicate a column by selecting one in the Hierarchy and pressing **Ctrl–D** on PC or **Command–D** on Mac.

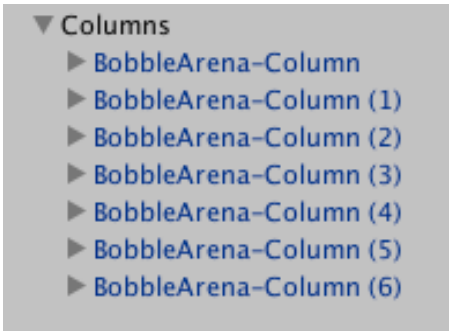
Create a total of six duplicates and give them following positions:

- **Column 1**: (**44.69**, **12.83**, **-28.25**)
- **Column 2**: (**42.10**, **12.83**, **30.14**)
- **Column 3**: (**-8.29**, **12.83**, **63.04**)
- **Column 4**: (**80.40**, **12.83**, **-13.65**)
- **Column 5**: (**-91.79**, **12.83**, **-13.65**)
- **Column 6**: (**-48.69**, **12.83**, **33.74**)

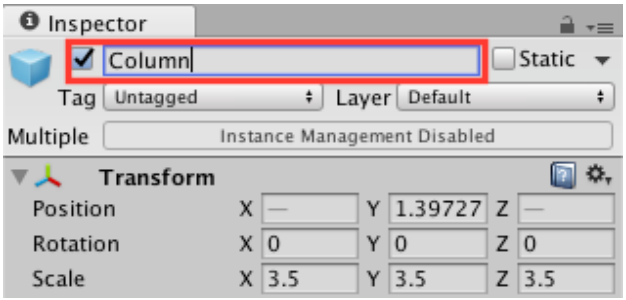
You should now have seven columns in the arena.



The arena looks good, but the Hierarchy is a bit messy. Tidy it up by clicking the **Create button** and select **Create Empty**. Rename the GameObject to **Columns** and drag each column into it.



You'll notice the columns have a similar name with unique numbers appended to them. Since they essentially act as one entity, it's fine for them to share a name. Hold the **Shift** key and **select all the columns** in the Hierarchy. In the Inspector, **change the name** to **Column**.



Note: As you can see, it's possible to change a common property for a bunch of GameObjects at once.

Creating spawn points

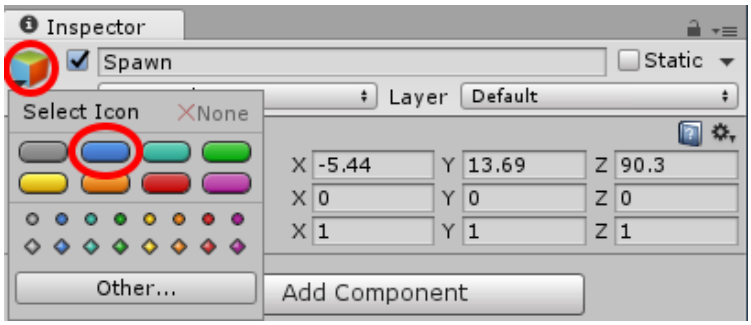
What good are bloodthirsty aliens unless they spawn in mass quantities? In this section, you'll set up spawn points to produce enemies to keep our hero on his toes.

So far, you've assembled the game with GameObjects that you want the player to see. When it comes to spawn points, you don't want anybody to see them. Yet, it's important that *you* know where they lie.

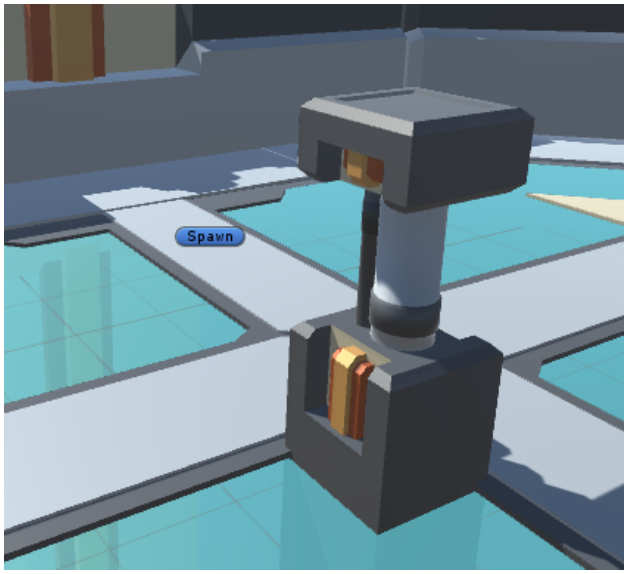
You could create a 3D cube and place it in your scene to represent a spawn point, then remove it when the game starts, but that's a clunky approach. Unity provides a simpler mechanism called **labels**, which are GameObjects visible in the Scene view, but invisible during gameplay. To see them in action, you'll create a bunch of different spawn points similar to how you created the columns.

Click the **Create** button in the Hierarchy and select **Create Empty**. Give it the name **Spawn** and set **position** to (**-5.44**, **13.69**, **90.30**).

In the Inspector, **click** the **colored cube** next to the checkmark. A flyout with all the different label styles will appear. Click **blue capsule**.

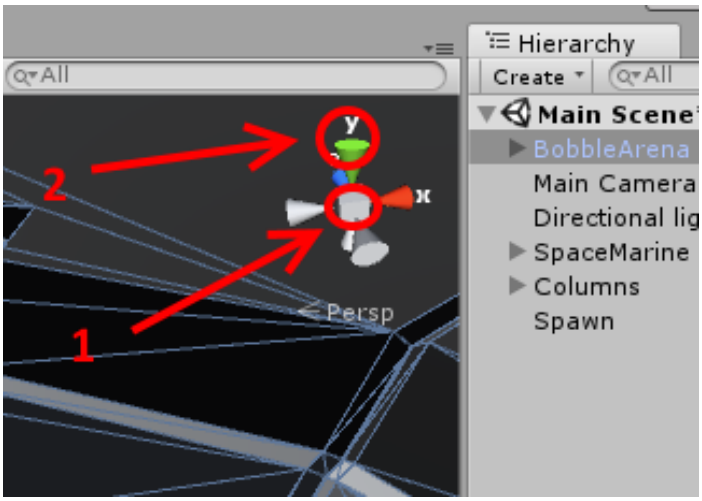


Look at your Scene view; you'll see it's been annotated with the Spawn point.



You need to create ten more spawn points. Make placement easier by doing the following:

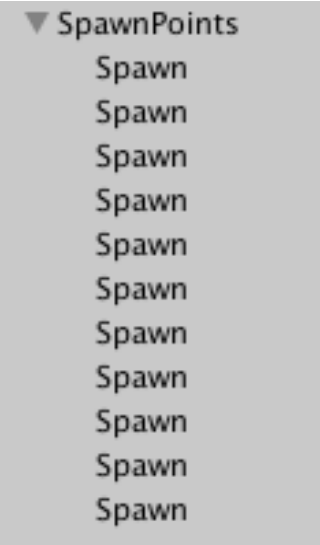
1. In the Scene view, **click** on the **center cube** of the **scene gizmo** to switch the Scene view to Isometric mode.
2. Click the **green y-axis arrow** so that you look down on the scene.



Now go ahead with duplicating and repositioning ten more spawn points, according to the following image:

Don't worry if you don't get them exactly the same - game design is more of an art than a science!

Once you're done, click the **Create** button in the Hierarchy, select **Create Empty** and name it **SpawnPoints**. Drag all the spawn points into it. **Batch rename** them like you did with the columns to **Spawn**.



Congratulations! Your game is now properly set up. Make sure to **save**!

Where to go from here?

At this point, you have your hero, his enemy the alien, and the arena in which they will battle to the death. You've even created spawn points for the little buggers. As you set things up, you learned about:

- **GameObjects** and why they are so important when working with Unity.
- **Prefabs** for when you want to create many instances of a single GameObject.
- **Labels** to help you annotate game development without interfering with the game.

There's still not any action in your game, but that's fine. You're ready to learn how to give your game the breath of life and take it away (via the space marine's magnificent machine gun).

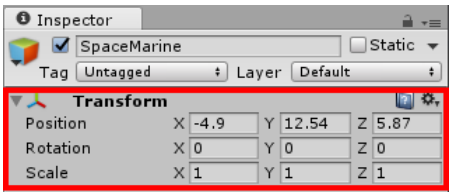
Chapter 3: Components

By Brian Moakley

At this point, you've accomplished your goal of having all the main actors ready, but it's essentially an empty stage. Sure, you might win some avant-garde awards for a poignant play on the lack of free will, but that's not going to pay the bills.

In this chapter, you'll add some interactivity to your game through the use of **Components**, which are fundamental to Unity game development. If you were to think of a GameObject as a noun, then a component would be a verb; a component does something on behalf of a GameObject.

You've learned a bit about components already. In previous chapters, you learned how each GameObject has one required component: the **Transform** component, which stores the GameObject's position, rotation, and scale.



But Unity comes with far more components than that. For instance, there's a **light** component that will illuminate anything near it. There's an **audio source** component that will produce sound. There's even a **particle system** component that will generate some impressive visual effects. And of course, if there isn't a component that does what you want, you can write your own.

In this chapter, you'll learn about several types of components, including the **Rigidbody** component, the **script** component, and more. By the end of this chapter, your marine will be running and gunning!

Getting started

If you completed the last chapter, **open** your current **Bobblehead Wars** project to pick up where you left off. If you got stuck or skipped ahead, open the **Bobblehead Wars** starter project from this chapter's resources.

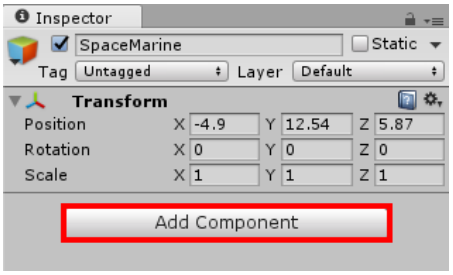
Currently, the space marine only knows how to stand like a statue. He needs to move around if he's going to avoid being some alien's snack.

The first thing you'll add is a **Rigidbody** component, which opts the GameObject into the physics engine. By adding it, you'll enable the GameObject to collide with other GameObjects.

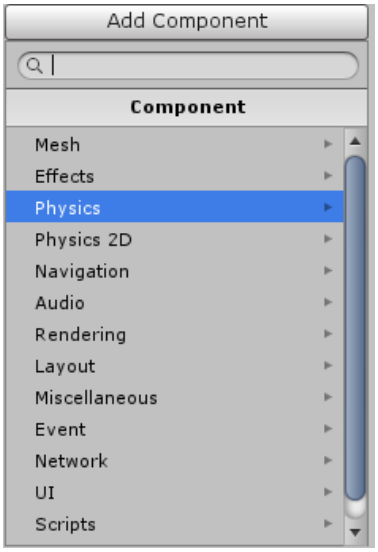
You'll learn much more about rigidbodies and collisions when you reach the chapter on physics. For now, just take it at face value and move forward.

Adding the rigidbody component

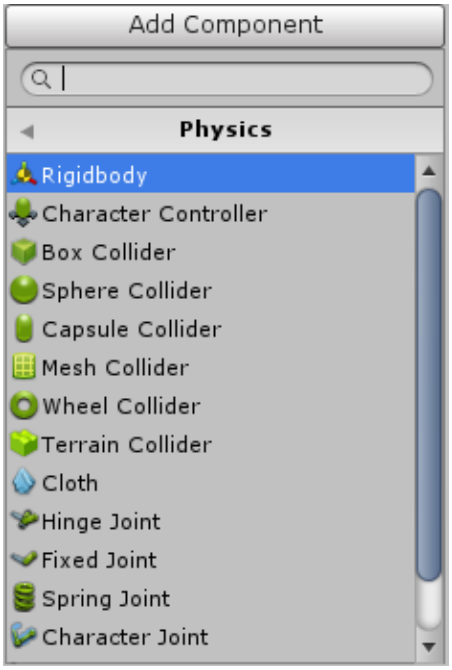
In the Hierarchy, select the **SpaceMarine** GameObject and click the **Add Component** button in the Inspector.



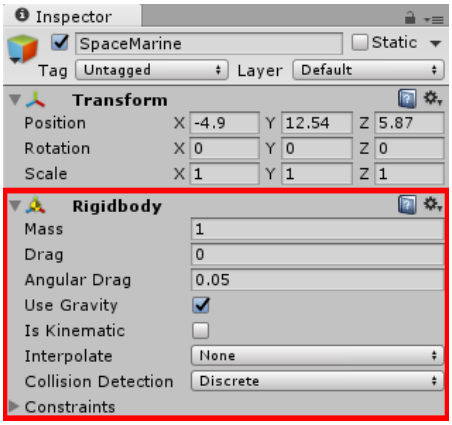
You'll see many different categories. When you know which component you need, simply search by name. Otherwise, select one of the categories and pick the best option.



Click the **Physics category** then select **Rigidbody**.

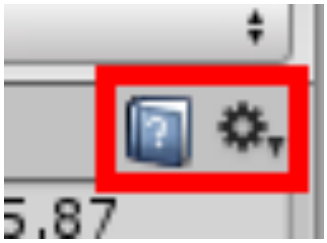


You'll see that a Rigidbody component was attached to the GameObject. Congratulations! You've added your first component.

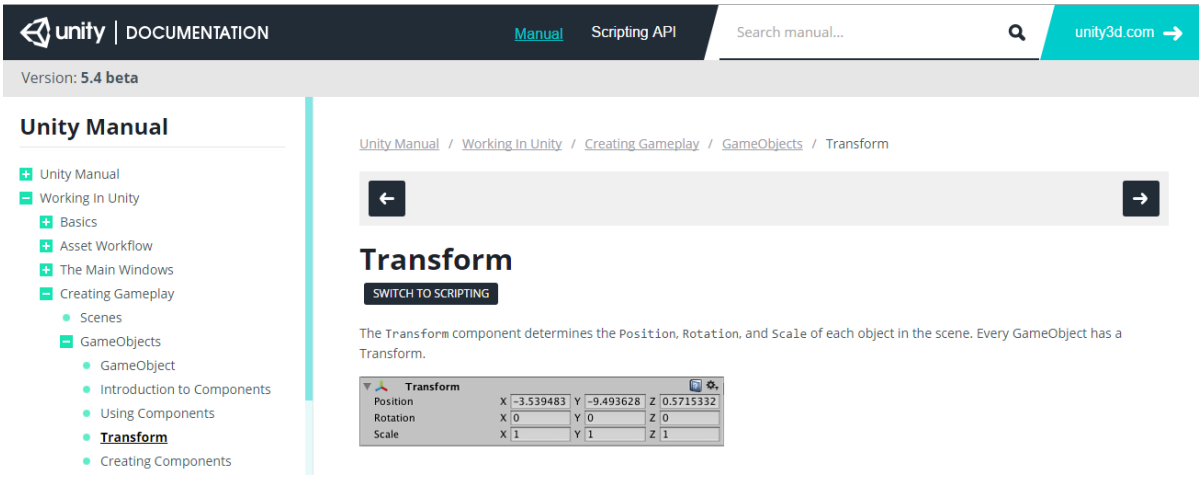


Each component has its own set of properties, values and so forth. These properties can be changed in the Inspector or in code. Components also have their own icon to make it easy to determine their type at a glance.

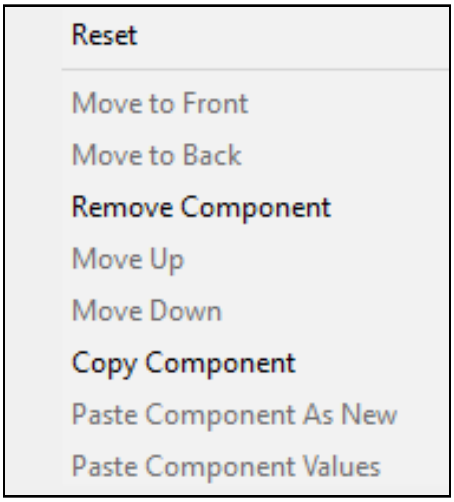
You'll notice a couple of icons in the right-hand corner of each component, like this:



The first icon is the **Reference** icon. **Click it**. It'll open another window with the documentation page for that component. If you installed the documentation, this page is on your computer, and yes, the search bar works.



You'll also see a gear icon. **Click it**. This dialog will appear:



Here are the most important options listed here:

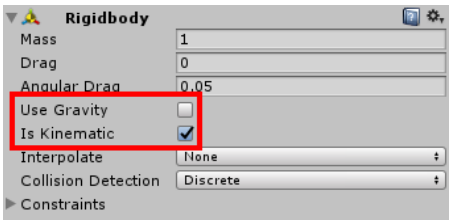
- **Reset** will reset the component to its default values.
- **Move to Front** and **Move to Back** adjust the ordering of sprites in 2D games.

- **Remove Component** will delete the component from the GameObject — you can undo this action.
- **Copy Component** allows you to copy a component from one GameObject and paste it onto another.
- **Paste Component as New** will paste a copied component to a GameObject.
- **Paste Component Values** allows you to overwrite the values of the current component from a copied component.

Specifically, you can copy the values of a component while your game is being played. When you stop the game, you can paste those values onto another component. This is quite useful because sometimes it's useful to tweak things as you play to see what works in practice.

By adding the Rigidbody component to the space marine, you've made it so that he now can respond to collision events and respond to gravity. However, you don't want him bouncing off enemies or walls. You definitely want to know about those collisions events, but you don't want the hero to fly out of control just because he bumped into a column.

In the **Rigidbody** component, check the **IsKinematic** checkbox. Also, **uncheck** the **Use Gravity** option.



The `isKinematic` property tells the physics engine that you're manually controlling the marine, rather than letting the physics engine move the marine for you. But the physics engine is still aware of where the marine is and when it collides with something. This is helpful so that when the marine collides with an alien, the physics engine will notify you so you can make something happen - like the space marine getting chomped!

By unchecking the `Use Gravity` option, the space marine won't be affected by gravity. Again - you don't need this because you'll be moving the marine manually and are only using the Rigidbody for collision detection.

Now comes the fun part: it's time to make that marine dance! Or move. That's fine too.

Unity provides some built-in components to make movement easy, and you'll use them in the next chapter. But for learning purposes, in this chapter you'll do everything manually through the power of scripting.

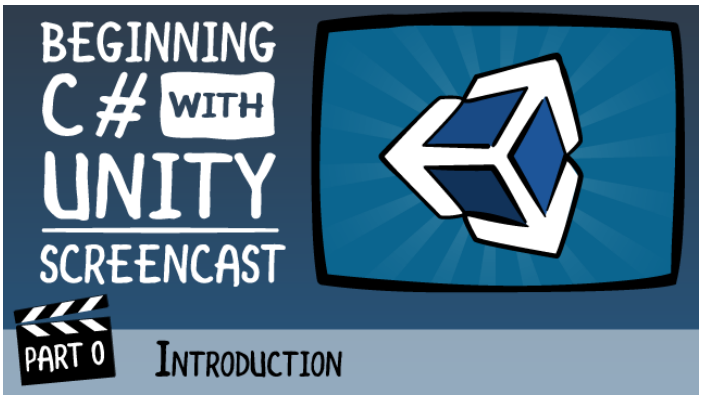
Introducing scripting

For the non-programmer, at best scripting is a small barrier to entry. At worse, scripting is the devil that never sleeps.

Coming from an arts background, I can say that scripting is not as hard as you might imagine. Like anything, it just requires some time and patience to understand how it works.

In this book, you'll write scripts in C#, which is a popular language developed by Microsoft that works for both mobile and desktop apps. It's feature-rich and fully versatile. Unfortunately, I can't teach it all within the covers of this book.

Thankfully, you can learn C# at raywenderlich.com where there's a free course that teaches the language from the ground up, for complete beginners.



It's designed for non-programmers and taught within the context of Unity.

If you're a beginner to programming, definitely check that out first. If you're an experienced developer, or a particularly brave beginner feel free to continue along, since everything in this book is step-by-step; just understand that there may be some gaps in your understanding without C# knowledge.

Why not Javascript?

Believe it or not, Unity supports other programming languages such as Javascript and Boo. Some people consider Javascript to be "easy" to learn, so you may wonder why we elected to use C#.

The truth of the matter is that Unity's implementation of Javascript IS NOT JavaScript. It's actually called UnityScript and considerably different than the Javascript widely used on the web. While you can certainly be productive with it in Unity, there are no viable use cases for it outside of the engine.

C#, on the other hand, is completely compatible with the outside world. Skills that you learn in Unity can easily be applied outside of game development. For instance, if you find yourself disliking game development (or having a hard time making a

living) but enjoying the language, you can transition those skills into a C# development job, creating desktop or mobile apps, or even developing backend server apps.

Finally, most professional Unity developers write in C#, so by taking the time to learn it, you'll be in a better position to take advantage of game development opportunities as they come up.

The way coding works in Unity is that you create **scripts**. Scripts are simply another type of component that you attach to GameObjects, that you get to write the code for.

A script derives from a class called **MonoBehaviour**, and you can override several methods to get notified upon certain events:

- **Update()**: This event occurs at every single frame. If your game runs at sixty frames per second, `Update()` is called sixty times. Needless to say, you don't want to do any heavy processing in this method.
- **OnEnable()**: This is called when a GameObject is enabled, and also when an inactive GameObject suddenly reactivates. Typically, you deactivate GameObjects when you don't need them for a while but will have a need at a later point in time.
- **Start()**: This is called once in the script's lifetime and before Update() is called. It's a good place to do set up and initalization.
- **Destroy()**: This is called right before the object goes to the GameObject afterlife. It's is a good place to do clean up such as shutting down network connections.

There are many other events that you'll discover throughout this book. To see a complete listing, head to the `MonoBehavior` reference on Unity's site:

- https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

Creating your first script

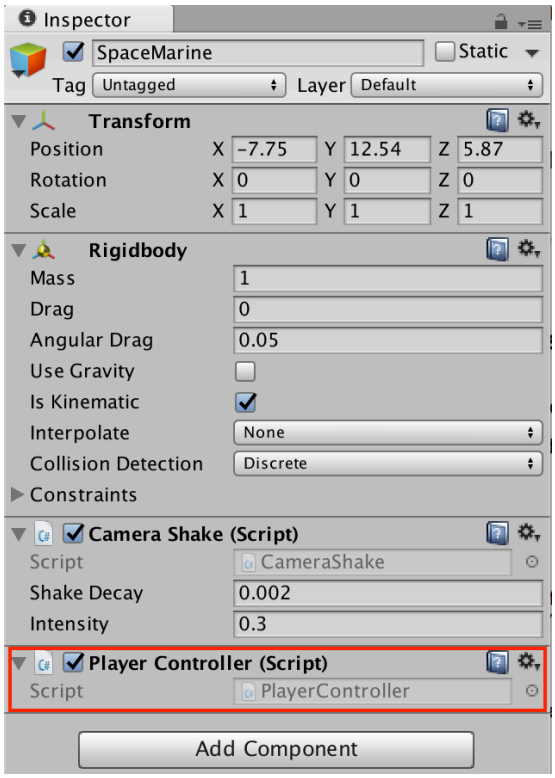
It's showtime!

You have many options for creating a script. You could click the Add Component button and then select New Script.

But I'd like you to try it this way: select the **Scripts** folder in the **Project Browser**, and then click the **Create** button. Select **C# Script** from the drop-down and name it **PlayerController**.

You'll see your new script in the Scripts folder. **Drag it** from the **Scripts** folder onto the **SpaceMarine GameObject**.

You should now see the script listed as one of the components on the Space Marine:



You've added your first custom component! Granted, it doesn't do anything...yet

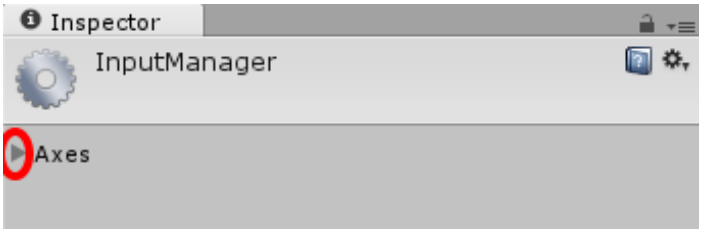
You'll change that in just a moment, but before you do, you need to learn about the Input Manager.

Managing Input

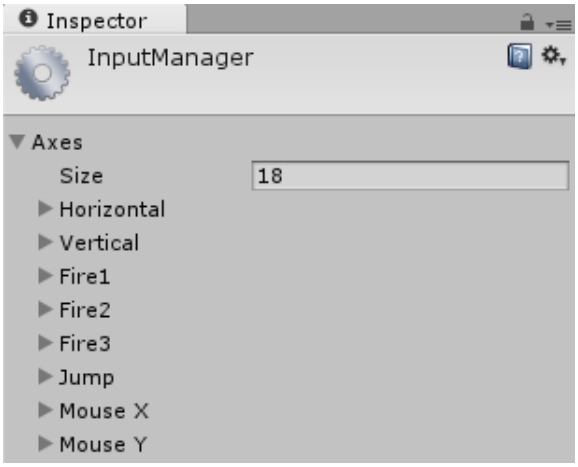
Inputs are the game's controls, and if you're developing a game for a desktop computer, your users will expect the ability to rebind keys. Unity's **Input Manager** makes this easy, and is the preferred way for your game to deal with user input.

To get access to the Input Manager, click **Edit\Project Settings\Input**.

The Inspector will look pretty empty. Click the **disclosure triangle** next to the word **Axes**.

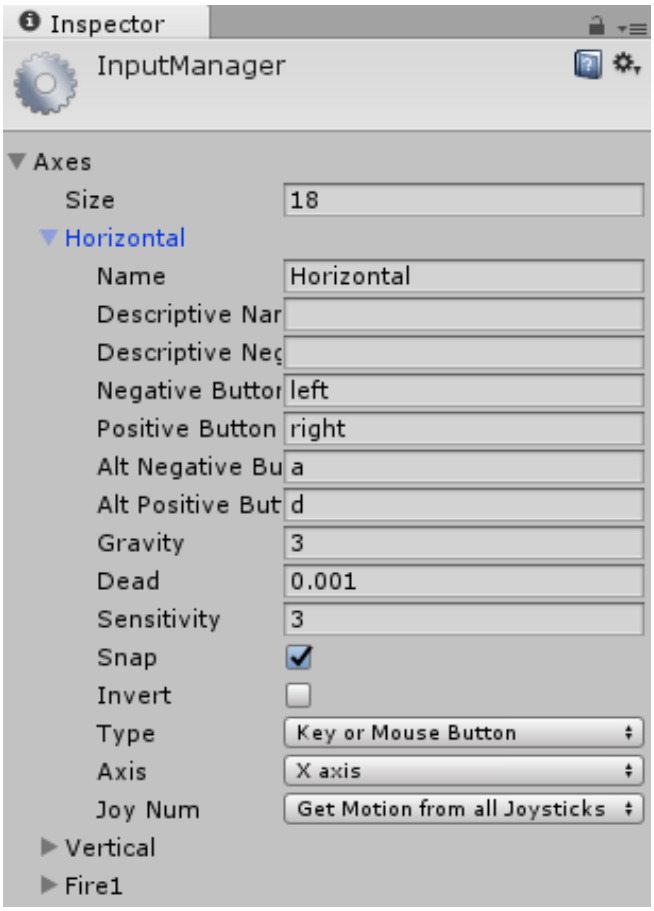


Once expanded, you'll see all the pre-configured inputs that are available to you.



The first property is **Size**, and it's the number of inputs your game uses. You can decrease the number to decrease the amount and increase it if you want more inputs. The current amount of inputs is more than enough for this game.

Click the **disclosure triangle** next to **Horizontal**. Here you can configure the input for the horizontal axis, i.e., left or right.



Here's a breakdown of the key fields:

- **Horizontal** is the name Unity gives the input, but you can name it anything. This is also the name you reference in code, which you'll see in a moment.
- **Descriptive Name and Negative Name** are the names presented to the user in the Unity game launcher if they want to remap the keys. You can disable the Unity game launch and provide your own key mapping interface if you'd like, so these aren't required properties.
- **Negative and Positive Buttons** are the actual keys being used. Unity allows buttons to have negative or opposite keys. For instance, the `right` arrow key is positive while the `left` arrow key is negative. You don't need to provide a negative for all keys — it wouldn't make sense to provide a negative key for a use action.
- **Alt Negative and Alt Positive Buttons** are alternative keys. In this case, instead of left and right arrow keys, you enable the `a` and `d` keys.

The other fields mostly relate to the functionality of analog sticks. For simplicity, this game will only use keyboard input. If you wanted to make the game a bonafide twin stick shooter, these are the options you'd tweak to create a tight control scheme.

Accessing input from code

Now to actually implement the control scheme. In the Project Browser, **double-click** the **PlayerController** script to launch the editor.

> **Note**: If you've never set up or used a code editor before, prepare to see MonoDevelop launch. MonoDevelop is part of Unity's default installation and allows you to edit code.
>
> However, you can use other editors. For instance, uf you're running Unity on Windows, you can opt to use Visual Studio instead. To learn more about Visual Studio and MonoDevelop, check out the Appendix chapter, "Code Editors".

When the code editor opens, you'll see your first script. Every new script contains empty implementations for `Start()` and `Update()`.

Look for a blank line below the first `{` (aka "curly bracket") — that's the class definition. Add the following code there:

```
public float moveSpeed = 50.0f;
```

> **Note**: If you are new to programming languages, it's critical that you copy everything exactly as it is written. Any deviation will produce errors.

> Programming languages are very precise and become grumpy when you use the incorrect case or syntax.
>
> If your script throws an error, carefully review the code to make sure you didn't miss, forget or mess up something.

`moveSpeed` is a variable that determines how fast the hero moves around in the arena. You set it to a default value of `50` that you can change later in Unity's interface.

Now, to write the actual moving code. In `Update()`, add the following:

```
Vector3 pos = transform.position;
```

This bit of code simply gets the current position of the current GameObject — in this case the space marine since that is what this script is attached to — and stores it in a variable named `pos`. In Unity, a `Vector3` encapsulates the x, y and z of a GameObject, i.e., the object's point in 3D space.

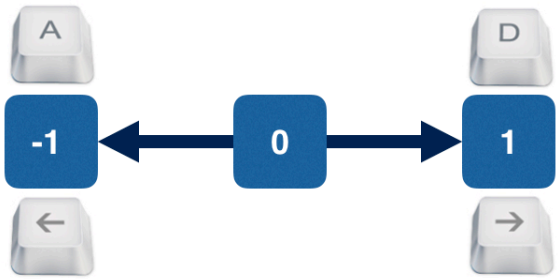
Now comes the tricky part. Add the following after the previous line:

```
pos.x += moveSpeed * Input.GetAxis("Horizontal")
  * Time.deltaTime;
pos.z += moveSpeed * Input.GetAxis("Vertical")
  * Time.deltaTime;
```

When you move the object, it'll only be on z- and x-axes because y represents up and down. Because there are two different input sources ("Horizontal" for left and right, and "Vertical" for up and down), you need to calculate values for each axis separately.

`Input.GetAxis("Horizontal")` retrieves a value from the Horizontal component of the Input Manager. The value returned is either `1` or `–1`, with `1` indicating that a positive button in the Input Manager is being pressed.

According to the settings you saw defined earlier, this is either the `right` arrow or `d` keys. Similarly, a value of `–1` indicates that a negative button is being pressed, meaning it was either the `left` arrow or `a` key.



Whatever the returned value may be, it's then multiplied by the `moveSpeed` and added to the current `x` position of the GameObject, effectively moving it in the desired direction.

The same thing happens with `Input.GetAxis("Vertical")`, except it retrieves a value from the vertical component of the Input Manager (indicating the `s`, `down`, `w` or `up` keys), multiplies this (`1` or `–1`) value by the `moveSpeed` and adds it to the `z` position of the GameObject.

So what's with `Time.deltaTime`? That value indicates how much time has passed since the last `Update()`. Remember, `Update()` is called with every frame, so that time difference must be taken into account or the space marine would move too fast to be seen.

TL/DR: `Time.deltaTime` ensures movement is in sync with the frame rate.

> **What do these numbers mean?**
>
> By default, Unity considers 1 point to be equal to 1 meter, but you don't have to follow this logic. For instance, you may be making a game about planets, and that scale would be way too small. For the purposes of simplicity in this book, we have sized our models to follow Unity's default of one point per meter.

Now that you've altered the location, you have to apply it to the SpaceMarine. Add the following after the previous line:

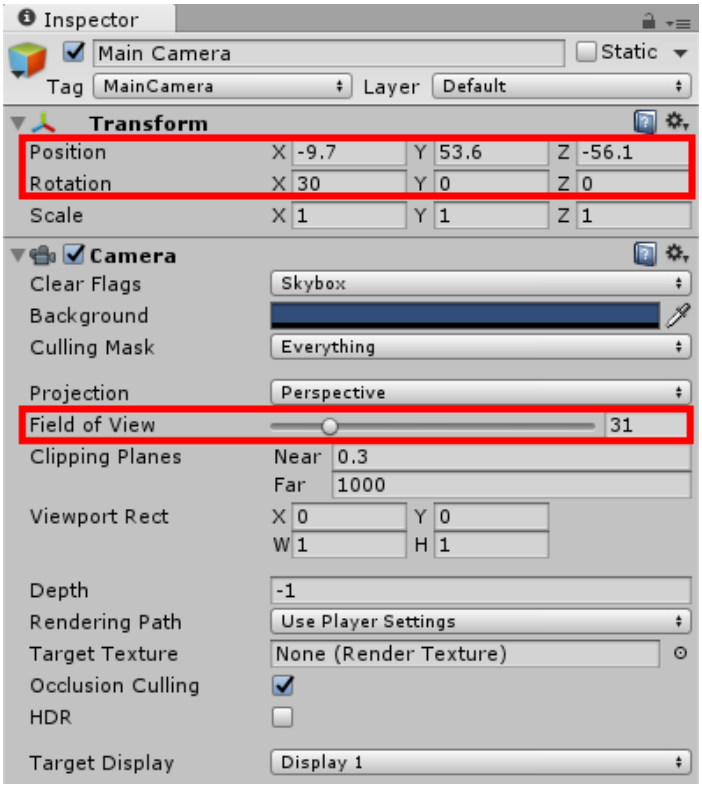
```
transform.position = pos;
```

This updates the SpaceMarine's position with the new position.

Save the script and switch back to Unity. You may be tempted to run your game, but there's a slight problem. The camera is not positioned correctly.

In the Hierarchy, select the **Main Camera**, and in the Inspector, set **Position** to (**-9.7**, **53.6**, **-56.1**) and **Rotation** to (**30**, **0**, **0**). I got these values by moving the camera around manually and looking at the Camera preview in the lower right until I was happy with the result.

In the **Camera** component, set **Field of View** to **31**. This effectively "zooms in" the view a bit.



Now it's time to give your game a test run. Look for the play controls at the center-top of the editor. Click the **play** button.

> **Note**: You'll notice the controls give you two more options: **pause** and **pause stepper**. Pause allows you to, well, pause your game in motion. The stepper allows you to step through the animation one frame at a time and is especially useful for debugging animation issues.

Now, look at the Game window and move your character by pressing the arrow keys or WASD keys. Behold...life!

The Game Window

The Game window is where you actually play the game. There are two life-and-death details to keep in mind as you play.

First, when you start playing, the interface becomes darker to give a visual queue that you're in play mode.

Second, when you play your game you can change anything about it (including changing values on components in the inspector), but when you stop playing the game, **ALL YOUR CHANGES WILL BE LOST**. This is both a blessing and a curse. The blessing is that you have the ability to tweak the game without consequence. The curse is that sometimes you forget you're in play mode, continue working on your game, then for some reason the game stops. *Poof! Buh-bye changes!*

Thankfully, you can (and should) make play mode really obvious. Select **Edit\Preferences** on PC or **Unity\Preferences** on Mac to bring up a list of options.

Select the **Colors** section. Here you can change the colors used throughout the editor. Look for **Playmode tint**. Click the **color box** next to it, and then give it an unmistakable color — I prefer red. Now play your game to see if it's conspicuous enough.

Camera movement

There's only one problem with the space marine's movement: he will slip off screen. You want the camera to follow the hero around the arena, so he doesn't get away from you.

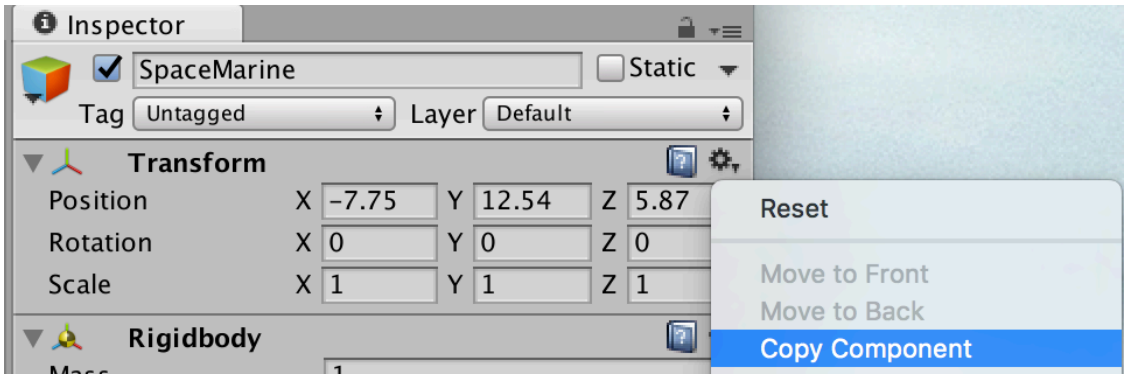
With a little scripting, you can keep the marine in focus.

In the Hierarchy, click the Create button and select **Create Empty**. Name it **CameraMount**.

The basic idea is you want CameraMount to represent the position the camera should focus on, and have the camera be relative to this position.

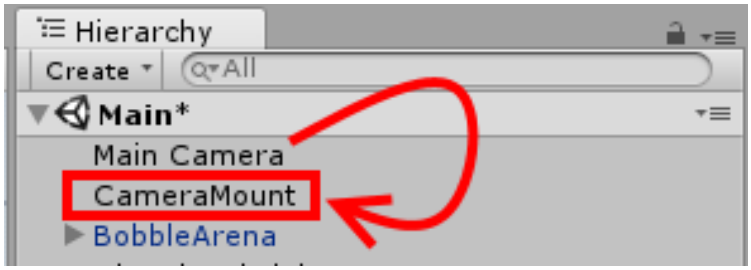
Initially you want the camera to focus where the space marine is, so let's configure the CameraMount to be at the exact same position as the space marine.

To do this, select the **space marine**, click on the gear button to the upper right of the Transform component, and select **Copy Component**:



Then select the **CameraMount**, click on the gear button to the upper right of the Transform component, and select **Paste Component Values**:

Next, drag the **Main Camera** GameObject into the **CameraMount** GameObject.



Great! Now as you move the player around, you can move the CameraMount to move with the player, and the camera will track the player. You just need to write a script to do this.

With the CameraMount selected, click the **Add Component** button in the Inspector and select **New Script**. Call it **CameraMovement** and set the language to **C Sharp**.

> **Note**: When you make a new script by clicking the Add Component button, Unity will create the script in the top level of your assets folder. Get into the habit of moving assets into their respective folders the moment you make or see them.

Drag your new file from the top level of the assets folder into the **Scripts** folder.

Double-click the **CameraMovement** script to open it in your code editor. Underneath the class definition, add the following variables:

```
public GameObject followTarget;
public float moveSpeed;
```

`followTarget` is what you want the camera to follow and `moveSpeed` is the speed at which it should move. By creating these as public variables, Unity will allow you to set these within the Unity editor itself, so you can set the `followTarget` to the space marine and fiddle with the `moveSpeed` to your heart's content, as you'll see shortly.

Now add the following to `Update()`:

```
if (followTarget != null) {
  transform.position = Vector3.Lerp(transform.position,
    followTarget.transform.position, Time.deltaTime * moveSpeed);
}
```

This code checks to see if there is a target available. If not, the camera doesn't follow.

Next, `Vector3.Lerp()` is called to calculate the required position of the CameraMount.

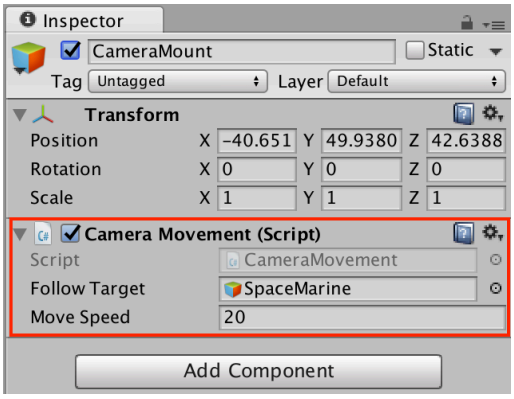
`Lerp()` takes three parameters: a start position in 3D space, an end position in 3D space, and a value between 0 and 1 that represents a point between the starting and ending positions. `Lerp()` returns a point in 3D space between the start and end positions that's determined by the last value.

For example, if the last value is set to `0` then `Lerp()` will return the start position. If the last value is `1`, it returns the end position. If the last value is `0.5`, then it returns a point half-way between the start and end positions.

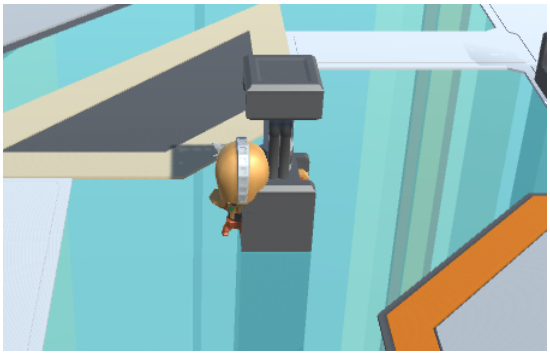
In this case, you will supply the camera mount position as the start and the player position as the end. Finally, you multiply the time since the last frame rate by a speed multiplier to get a suitable value for the last parameter. Effectively, this makes the camera mount position smoothly move to where the player is over time.

Save your code and return to Unity. If you look in the Inspector now, you'll see two new fields named **Follow Target** and **Move Speed**. As mentioned earlier, these were automatically derived by Unity from the `public` variables you just added to the script. These variables need some values.

With the CameraMount still selected in the Hierarchy, drag **SpaceMarine** to the **Follow Target** field and set the **Move Speed** to **20**.



Play your game to see what's changed.



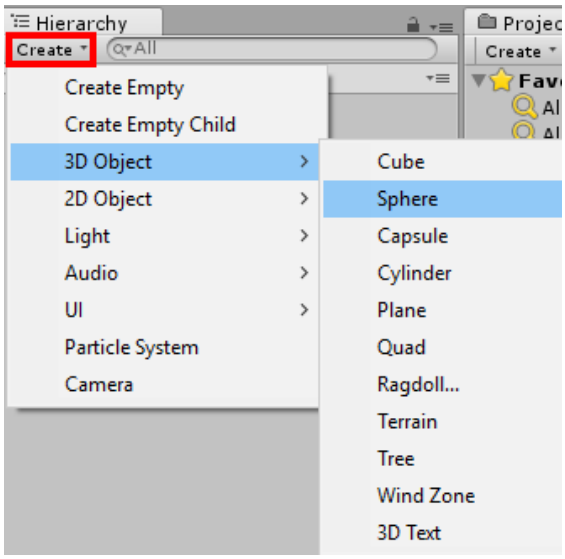
The marine is a real superstar now, complete with a personal camera crew. Granted, he can't turn, and he walks right through objects just like Kitty Pryde, but these are easily solvable issues that you'll tackle in the next chapter.

> **Note**: The bigger the move speed, the faster the camera mount will move to the player. The smaller, the more the camera will "lag" behind the player's position, letting the player "jump ahead" of the camera. Try changing the move speed to a smaller value like 2 and see what happens for yourself!

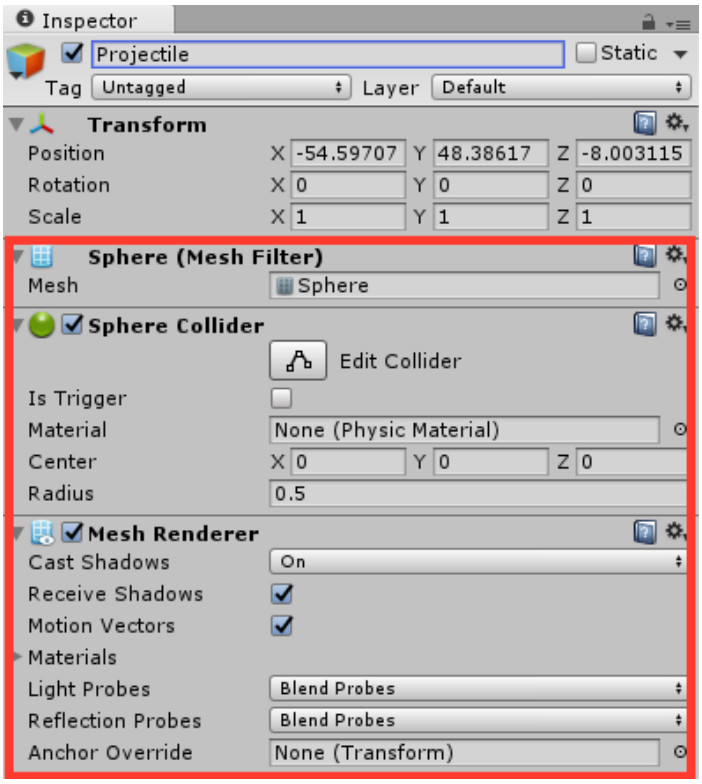
Adding Gunplay

Unfortunately, the finer parts of diplomacy are lost on the flesh-eating antagonists of this game. It's best you give the hero some firepower so he can protect himself on his terribly relaxing (terrible?) vacation.

First, you need to create a bullet. In the Hierarchy, click the **Create** button. From the drop-down, select **3D Object\Sphere** to create a sphere in the Scene view.



Give it the name **Projectile**. With it still selected, check out the Inspector. You'll notice a bunch of new components.



The three new components are:

1. The **Mesh Filter** is a component that contains data about your model's mesh and passes it to a renderer.
2. The **Mesh Renderer** displays the mesh. It contains a lot of information about lighting, such as casting and receiving shadows.
3. Finally, you'll notice the sphere contains a **Sphere Collider**. This component serves as the GameObject's boundaries. You'll learn cover colliders in the next chapter.

Since you want the bullet to participate in Unity's physics, it needs a rigidbody.

Luckily, you've done this before. Click the **Add Component** button, and select **Rigidbody** from the **Physics** category. Make sure to **uncheck** Use Gravity.

Since the marine will burn through lots of projectiles, **drag it** from the Hierarchy to the **Prefabs** folder in the **Project Browser**. **Delete** the **projectile** from the **Hierarchy** because you don't need it now that it's gone on to be a prefab.

At this point, you need to create a script to launch the projectile. In the Project Browser, select the **Scripts** folder then click the **Create** button. Choose **C# Script** and name it **Gun**. Double-click the file to launch the code editor.

This file needs a few properties underneath the class definition. Add the following:

```
public GameObject bulletPrefab;
public Transform launchPosition;
```

Again when you create a public variable on a script, Unity exposes these variables in the editor. You will set the `bulletPrefab` to the bullet prefab you just created, and you will set the `launchPosition` to the position of the barrel of the Space Marine's gun.

Next, add the following method:

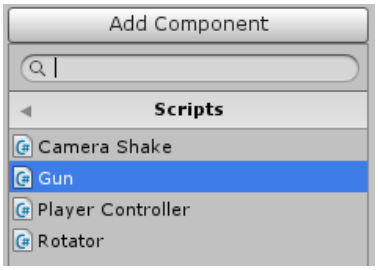
```
void fireBullet() {
  // 1
  GameObject bullet = Instantiate(bulletPrefab) as GameObject;
  // 2
  bullet.transform.position = launchPosition.position;
  // 3
  bullet.GetComponent<Rigidbody>().velocity =
    transform.parent.forward * 100;
}
```

Let's review this section by section:

1. `Instantiate()` is a built-in method that creates a GameObject instance for a particular prefab. In this case, this will create a bullet based on the bullet prefab. Since `Instantiate()` returns a type of `Object`, the result must be cast into a `GameObject`.
2. The bullet's position is set to the launcher's position — you'll set the launcher as the barrel of the gun in just a moment.
3. Being that the bullet has a rigidbody attached to it, you can specify its velocity to make the bullet move at a constant rate. Direction is determined by the transform of the object to which this script is attached — you'll soon attach it to the body of the space marine, thus ensuring the bullet travels in same the direction as the marine is facing.

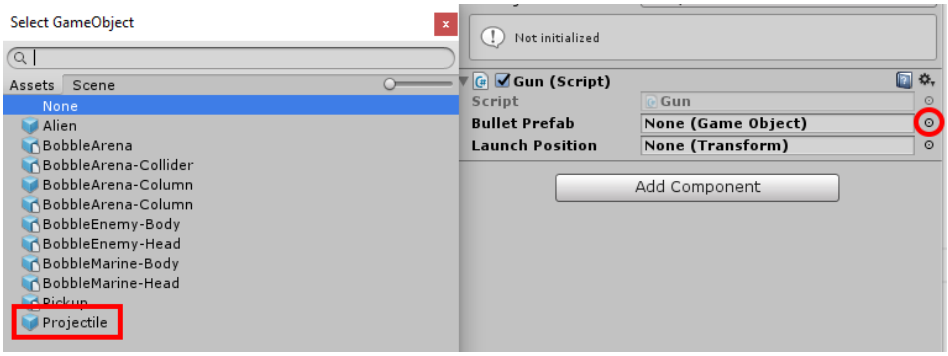
Save and switch back to Unity. In the Hierarchy, expand the SpaceMarine GameObject and **select** the **BobbleMarine-Body** GameObject.

In the Inspector, click the **Add Component** button and near the bottom of the list of components, select **Scripts**. From the list of scripts, choose **Gun**.



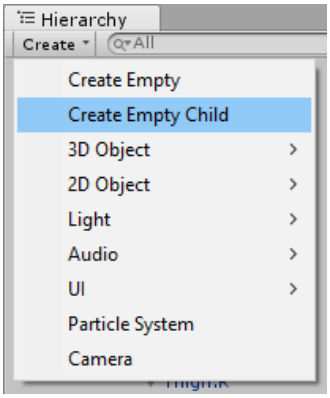
You'll see that your Gun script component has been added to the body of the marine. You'll also notice there are two new fields: **Bullet Prefab** and **Launch Position**. Do those sound familiar?

Click the circle next to **Bullet Prefab**. Select **Projectile** from the resulting asset list. Now you have loaded the bullet and just need to set the launch position.



In the Hierarchy, hold the **Alt** key on PC or **Option** on Mac and click the **disclosure triangle** next to the **BobbleMarine-Body**. You'll see a large list of child GameObjects. Look for **Gun**.

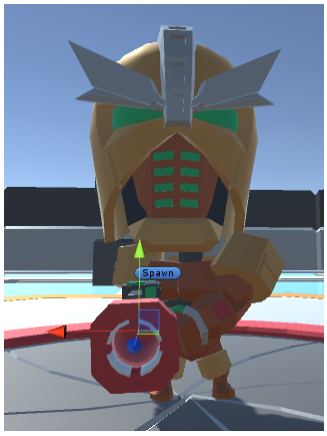
Select that GameObject and click the **Create** button. Choose **Create Empty Child** and rename it to **Launcher**. This new GameObject lives in the center of the gun's barrel and represents where bullets will spawn from - feel free to move it around in the scene editor if you'd like to tweak the spawn position.



Keep all the GameObjects expanded and select **BobbleMarine-Body** so that the Inspector shows all the components. Drag the new **Launcher** GameObject into the `Gun` component's **Launch Position** field.

Notice that when you add the GameObject to a transform field, Unity finds and references the attached transform.

It's official! The marine's gun is locked and loaded. All that's left is the firing code. Thankfully, it's pretty easy.



Switch back to your code editor and open **Gun.cs**.

The gun should fire when the user presses the mouse button and stop when the user releases it.

You could simply check to see if the button is pressed in `Update()` and call `fireBullet()` if so, but since `Update()` is called every frame, that would mean your space marine would shoot up to 60 times per second! Our space marine can shoot fast, but not *that* fast.

What you need is a slight delay between when you shoot bullets. To do this, add the following to `Update()`:

```
if (Input.GetMouseButtonDown(0)) {
  if (!IsInvoking("fireBullet")) {
    InvokeRepeating("fireBullet", 0f, 0.1f);
  }
}
```

First, you check with the Input Manager to see if the left mouse button is held down.

> **Note**: If you wanted to check the right mouse button, you'd pass in `1`, and for the middle mouse button, you'd pass in `2`.

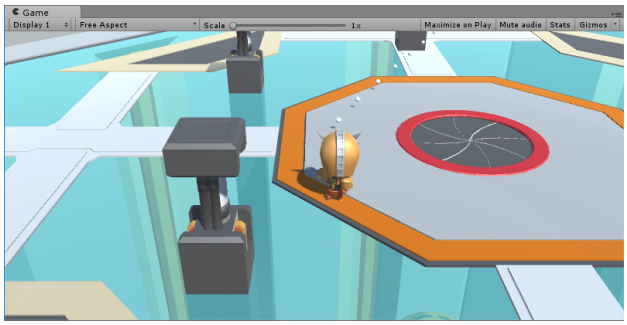
If the mouse is being held down, you check if `fireBullet()` is being invoked. If not, you call `InvokeRepeating()`, which repeatedly calls a method until you call `CancelInvoke()`.

`InvokeRepeating()` needs a method name, a time to start and the repeat rate. `InvokeRepeating()` is a method of `MonoBehaviour`.

After that bit of code, add the following:

```
if (Input.GetMouseButtonUp(0)) {
  CancelInvoke("fireBullet");
}
```

This code makes it so the gun stops firing once the user releases the mouse button. **Save** your work and return to Unity, then **play** the game.



Hold down the mouse button. You have bullets for days!

Where to go from here?

At this point, you should be feeling more comfortable with Unity. You have a walking space marine with a functioning weapon. You've learned the following:

- **Components** and how they give your GameObjects behavior.
- **Scripting** and how to use scripts to create custom behavior.
- The **Input Manager** and how to access it from code.
- The **Game window** and how to test your games.

This was a thick, heavy chapter and you made it to the end. Congratulations! You've come a long way.

There's still a lot more to do. Your poor marine is a sitting duck because he can't turn around to see what's sneaking up behind him. At the same time, he has nothing to worry about because there are no aliens hordes attacking him.

You'll solve these issues and more in the next chapter when you learn about the laws of game physics and how you can bend them to your own will!

Chapter 4: Physics

By Brian Moakley

Physics is one of those subjects that triggers nightmare flashbacks of my high school days when rolling toy cars down slopes was the peak of intellectual pursuit. If you're like me, then the notion of learning physics is about as exciting as watching a download progress meter.

Don't worry: Unity makes physics great again. Its built in physics engine allows you to easily create games with explosions, guns, and bodies smashing into beautiful walls - *without* having to study a boring textbook.

In this chapter, you'll learn how Unity's physics engine works, and use it to add collision detection, raycasts, and even bobbleheads into the game. Let's get started!

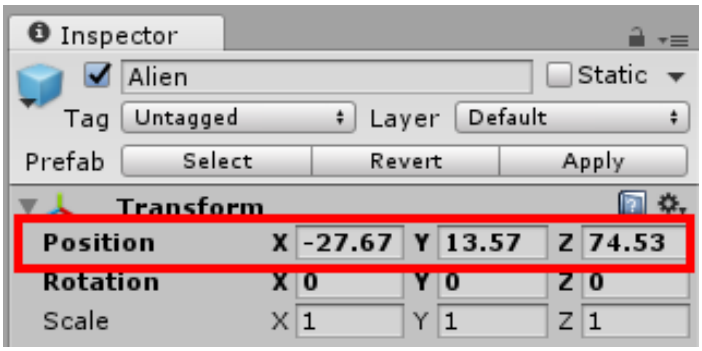
Getting started

Open your existing project, or you can open the starter project for this chapter.

Currently, the space marine can run around the arena and shoot things. However, with no targets and the no ability to turn around, he's bored and prone to sneak attack. He also has the superhero ability of walking through walls. You're going to solve these problems with physics!

Giving the marine something to shoot

First, you need to add a target for the marine. In the Project Browser, drag an **Alien** from the **Prefabs** folder and place it in front of the marine. In the Inspector, set the Alien's **position** to (**-27.67**, **13.57**, **74.53**).



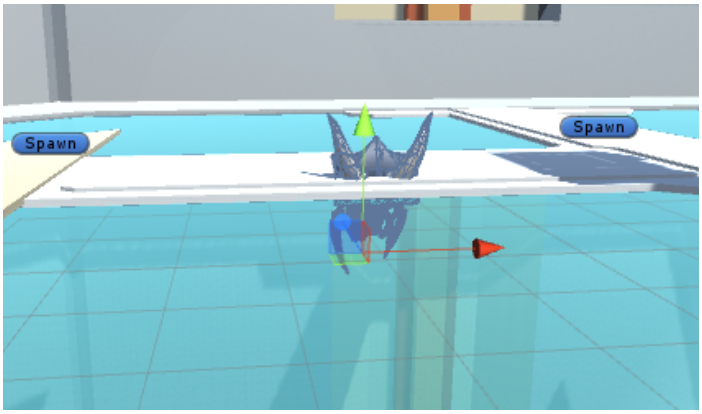
Play the game and shoot the alien.

Haha! The bullets pass right through without inflicting any harm, which is the default behavior for all GameObjects. You have to opt your GameObjects into receiving collisions. Sound familiar? Can you guess what component you'll use?

That's right; you need another Rigidbody! In the previous chapters, you used this component to opt objects into the physics engine. With the Rigidbody in place, not only can the GameObject respond to forces like gravity, but it can get a notification when another object, such as a bullet, collides with it. Then you can define what happens after such a collision, e.g., the alien explodes or vanishes.

Your alien has no Rigidbody, which is why it's impervious to our hero's attack. Select the **Alien** in the Hierarchy. In the Inspector, click the **Add Component** button. Select the **Physics** category and select **Rigidbody**.

Play your game.



Um, ok, the alien falls through the floor. Funny, but not ideal. To fix this, you can either turn off the gravity for the alien or use a collider.

Adding colliders

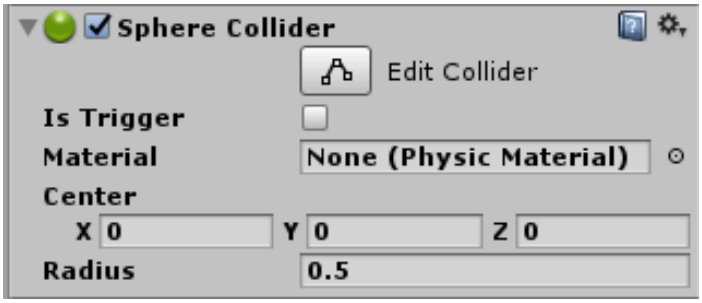
Colliders perform a variety of functions. They allow you to define the bounds of a GameObject for the purpose of collisions and are also useful for keeping your GameObjects from falling through the floor.

Usually for colliders, you want to use primitive shapes such as spheres and boxes. For example, you'll make a sphere that's about the same size of the alien. It won't match the shape of the alien exactly, but this is usually "good enough" for collision detection purposes, and results in a much faster game. If you need more precise collision detection, you can use a mesh collider instead - more on that in just a minute.

First, let's stop that alien from dropping through the floor.

Select the **Alien** GameObject in the Hierarchy and click the **Add Component** button in the Inspector. Select the **Physics** category, and from the components listing, select **Sphere collider**.

You'll see the collider, complete with many options, appear in the Inspector.



Here's the breakdown of the options:

- The **Edit Collider** button allows you to move and resize a collider visually in the Unity scene editor. Right now if you zoom in, you'll notice that the sphere is much smaller than the alien, and this option would allow you to move and resize it to better represent the alien's shape.

  Alternatively, you can set the center and radius properties manually, as you will do shortly.

- The **Is Trigger** checkbox means that the collider can pass through other colliders. Although the collider doesn't act like a hard surface, you'll still get collision notifications.

  A good use case for a trigger is to open a door when a player steps on a certain area.

- The **Material** option allows you to apply something known as a **Physics Material**, which are materials that have physical properties. For instance, you could create a surface that causes players to slide and assign it to ice models to make a skating rink.

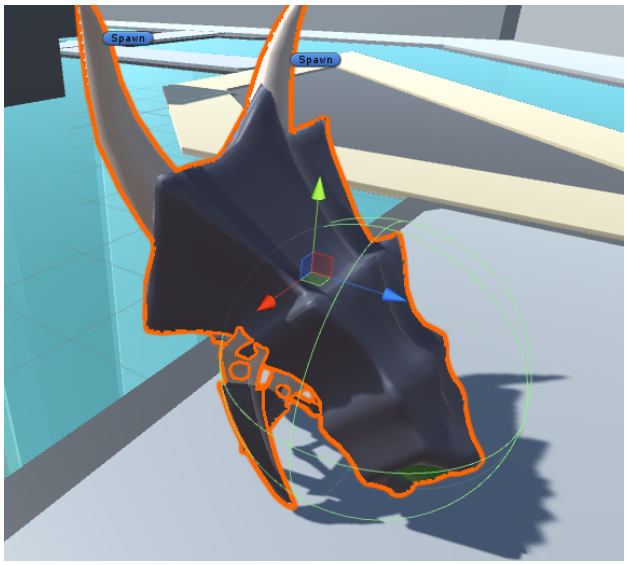
- The **Center** is a sphere collider's center point.

- The **Radius** determines the size of the collider.

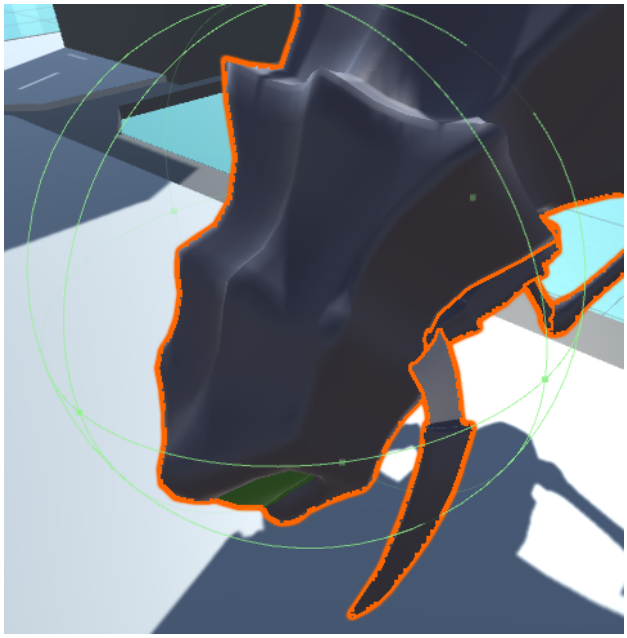
With all that mind, you can fix the collider for the alien to make it behave.

If the alien isn't already selected in the Hierarchy, select it now. In the Inspector for the sphere collider, set the **Center** to (**0.34**, **2.54**, **0.57**) and the **Radius** to **2.52**. Check out the Scene view now.

You'll see a green outline around the alien — it's the radius of the collider.

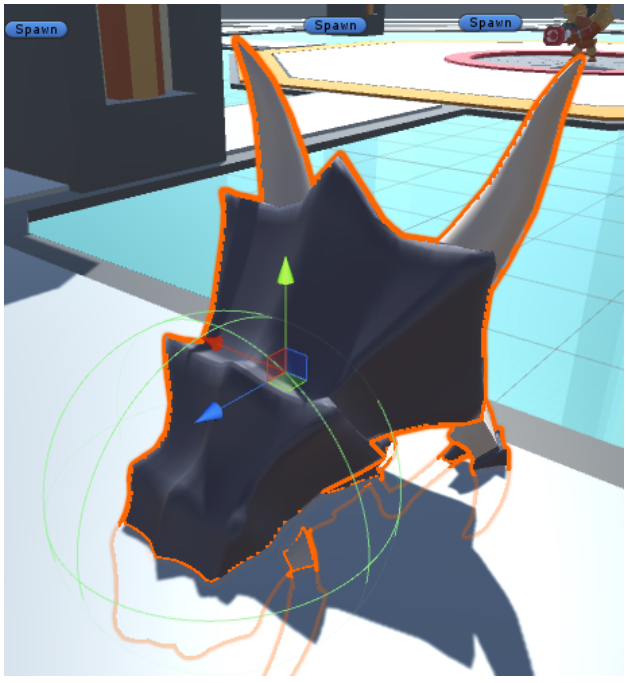


Click the **Edit Collider** button to see control points along the collider.



You can select those points and drag to either increase or decrease the size of the collider. This is how I came up with the radius and position values you entered a second ago originally - just sizing it until it was an approximation of the alien's size. Feel free to play around with the placement of the collider, and then undo to get back to the original size.

After you're done, play the game to see what happens. Make sure to **Apply** your changes to the prefab.



Unfortunately, the alien still falls through the floor. This is because there's no collider for the floor - you need to create one for that as well.

As mentioned, there are many colliders at your disposal and you're not restricted to just one. Any GameObject may have multiple colliders, giving you the ability to create complex shapes and reactions to collisions.

You could add a bunch of colliders to the arena to approximate its shape, but the arena is quite complicated and that would take a lot of work. So in this case, you'll use a mesh collider instead.

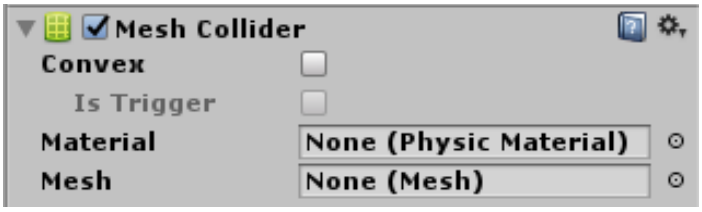
A **mesh collider** creates the collider boundaries from an actual mesh object, i.e., the geometry used to create the model. It's a great option when you have a complicated model. The problem with mesh colliders is they can be "expensive" — it takes a lot of resources to maintain the collider and determine if it's involved in collisions. In such cases, it helps to have a very low-poly collider.

To create a low-poly collider, you just take the original model and strip it of extraneous detail until you're down to the shape. Mike Berg, the artist for Bobblehead Wars, has done that for you.

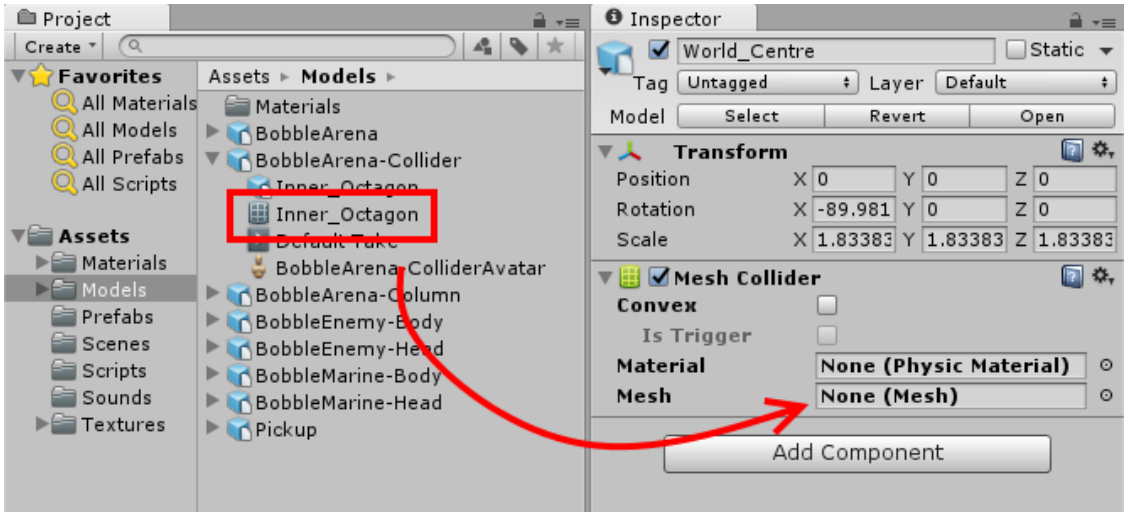
In the Hierarchy, expand the **BobbleArena** GameObject and select the **World_Center** GameObject.

> **Note:** If you don't see the **World_Center** GameObject, it wasn't imported with the model. Thankfully, you can make it yourself. Create an **Empty GameObject** and name it **World_Center**. Drag it into **BobbleArena**. Set the **position** to (**0**, **0**, **0**), the **rotation** to (**-90**, **0**, **0**), and the **scale** to (**1.83**, **1.83**, **1.83**) and the

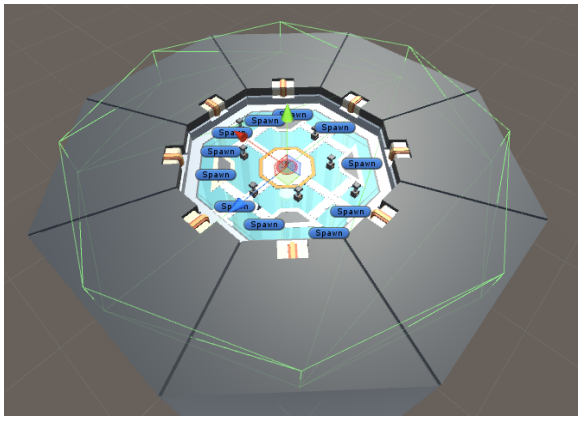
In the Inspector, click the **Add Component** button. Select **Physics**, and in the component listing, click **Mesh Collider**.



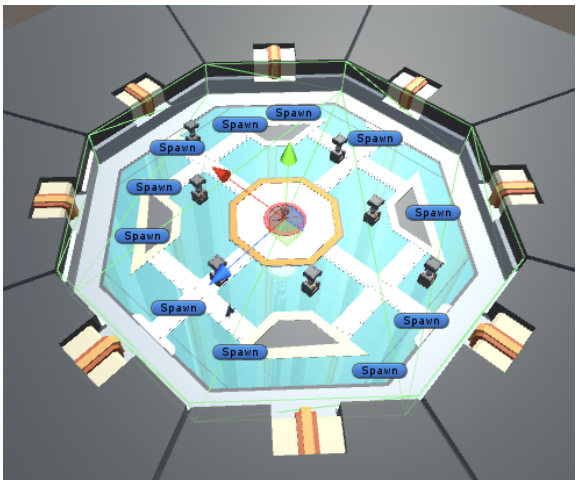
Find the **BobbleArena-Collider** in the **Models** folder, expand it by clicking the disclosure triangle, and then drag the **Inner_Octagon** mesh to the collider's **Mesh** property.



Initially, the mesh is a little too large and misaligned.



In the transform component, set the **Rotation** to (**-89.98**, **0**, **-22.26**) and the **Scale** to (**0.949**, **0.949**, **0.949**). The collider will now match the arena.



Play your game! Unfortunately for the hero, the bug no longer falls through the floor.

Blast that bugger away!

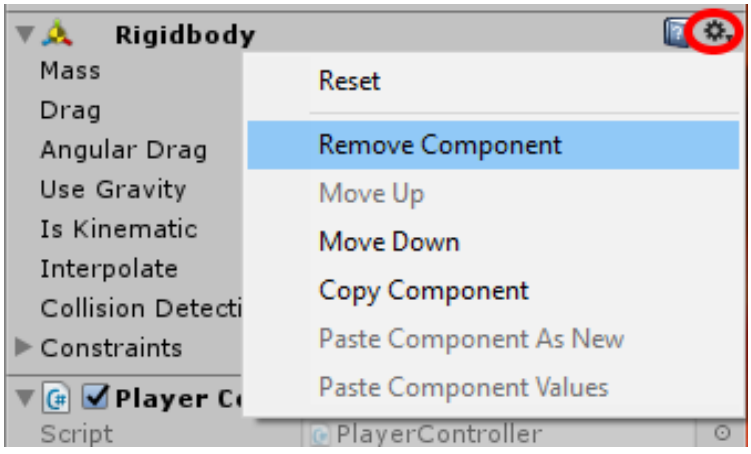
Introducing character controllers

The hero, not to be forgotten, needs some upgrades.

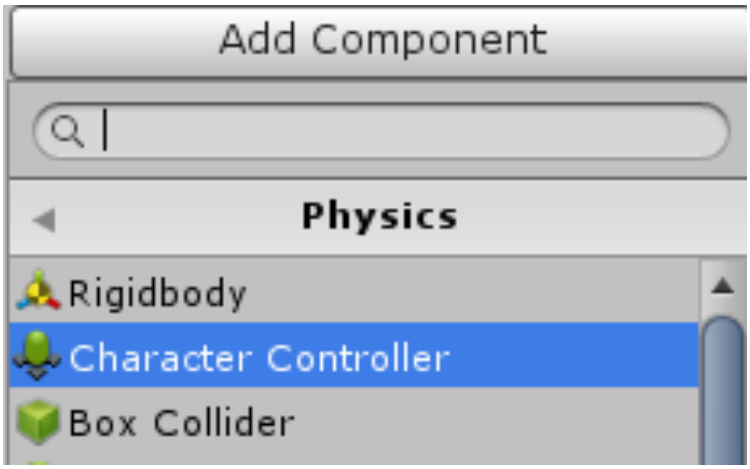
In the last chapter, you learned how to make the character move by adding a Rigidbody, adding colliders, and writing a script to make your character move. This was great for learning purposes, but there's a simpler way: the character controller.

A **character controller** is a built-in component that makes it easier to make characters in your game move based on user input. It combines all of the components that you added manually last chapter into one, and provides a simple API to use it.

Sorry to do this to you after all that work in the last chapter, but you need to strip the marine of his Rigidbody. Select the SpaceMarine in the Hierarchy, and in the Inspector, click the Rigidbody's **gear icon** and select **Remove Component**.

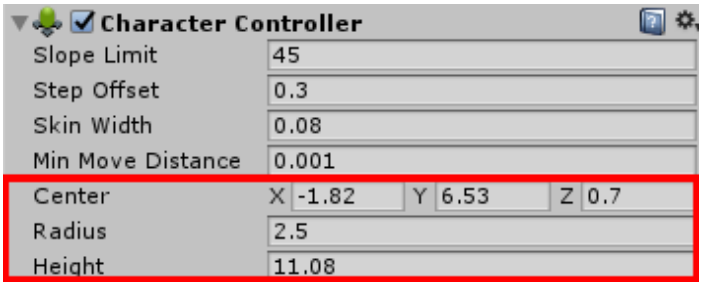


Click the **Add Component** button, go to the **Physics** category and choose **Character Controller**.



You'll notice a bunch of fields that influence the character's movement.

Set **Center** to (**-1.82**, **6.53**, **0.7**), **Radius** to **2.5** and **Height** to **11.08**. These properties affect the size of the collider.



In the Scene view, you'll see a fine green capsule around the marine.

Now that you've added a character controller, you need to make a few tweaks to your PlayerController script. Start by double-clicking the **PlayerController** script to open it in your editor.

First, you need to get a reference to `CharacterController`. You could get this in `Update()`, but it's best to store it in an instance variable during `Start()`. That way, the game doesn't spend unnecessary time fetching components.

Add the following beneath the `moveSpeed` variable declaration:

```
private CharacterController characterController;
```

This creates an instance variable to store the `CharacterController`.

In `Start()`, add the following to get a reference to the component:

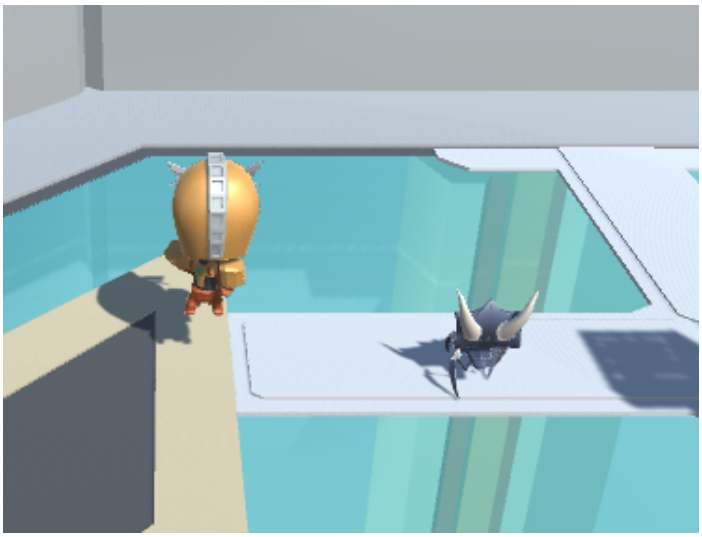
```
characterController = GetComponent<CharacterController>();
```

`GetComponent()` gets a reference to current component passed into the script. Replace all the code in `Update()` with the following:

```
Vector3 moveDirection = new Vector3(Input.GetAxis("Horizontal"),
    0, Input.GetAxis("Vertical"));
characterController.SimpleMove(moveDirection * moveSpeed);
```

First, the above code creates a new `Vector3` to store the movement direction then it calls `SimpleMove()` and passes in `moveDirection` multiplied by `moveSpeed`. `SimpleMove()` is a built-in method that automatically moves the character in the given direction, but not allowing the character to move through obstacles.

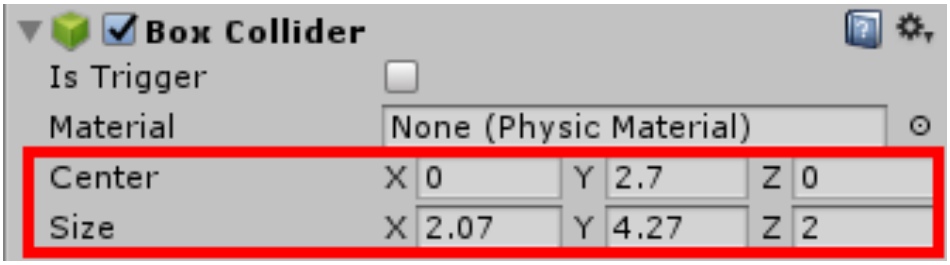
Save your changes and play the game.



You'll see the hero move much like before but with simpler code. In addition, your hero stops when he collides with the wall of the arena.

Unfortunately, you still have the problem of a character that runs through columns, which is easy to fix with colliders. Go ahead and make the fix now.

Stop your game. In the Project Browser, select the **BobbleArena-Column** in the **Prefabs** folder then click the **Add Component** button. In the **Physics** category, add a **Box Collider**. In the Inspector, set **Center** to (**0**, **2.7**, **0**) and **Size** to (**2.07**, **4.27**, **2**).

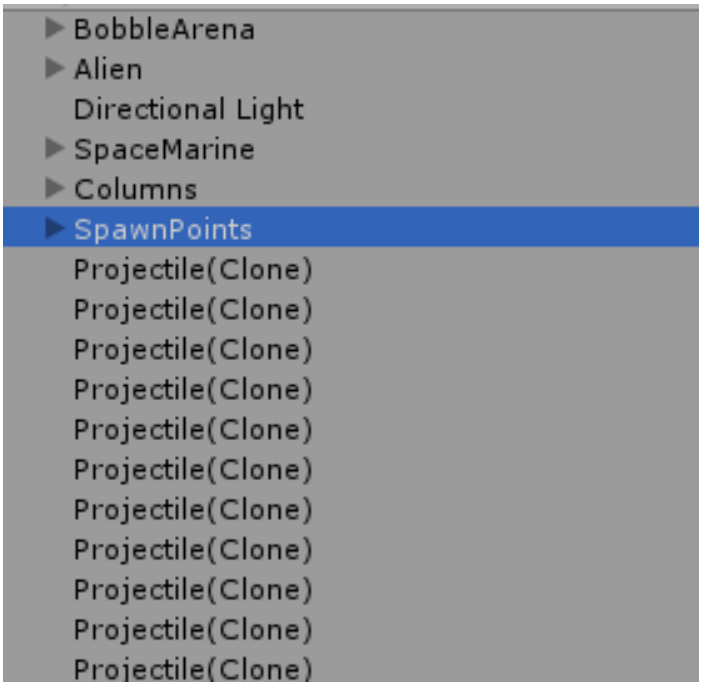


Once you make changes to the column, your next thought is probably that you need to apply those changes to the rest of the prefabs. You don't have to because you've modified the source prefab. Those changes are added automatically to all the instances!

Now play your game and charge at a column. You're now blocked!

Take a moment to shoot at the alien.

Two things will happen: The alien will get pushed around, and the arena floor will be littered with bullets. In fact, if you look at the Hierarchy, you'll see a boatload of bullets.



The best way to handle these issues is with collisions and layers.

Destroying old objects

As you can see, you create a lot of GameObjects when firing bullets. They're novel, but if you don't take action, your game will have serious performance issues.

You need to delete unused bullets as soon as they are out of play to keep the Hierarchy clean and the game performing.

What's the best way to manage used-up bullets?

Here's the problem: the game creates a new GameObjects every time the gun fires and over time. Since the player can fire continually, this will result in more and more bullets, until eventually your game runs out of memory and crashes. Creating unbounded objects by mistake is a common problem in game design.

You can delete the bullets, or you can reuse them by using **pools**. For example, after the GameObject exits the play area, it's put into a pool of objects to use in the future, rather than deleting it.

When the player fires again, the game checks if there are any available bullets in the pool. If not, the game creates a new bullet. Otherwise, it reuses an existing one. After a while, the game reuses bullets instead of squandering processing time on the creation and deletion of bullets.

In this chapter, for simplicity's sake you will delete the object, but you may want to investigate pools if performance becomes an issue.

In the Project Browser, open the **Prefabs** folder and select the **Projectile** prefab. Click the **Add Component** button and select **New Script** at the bottom. Give it the name **Projectile**, choose **C Sharp** as the language then click **Create and Add**.

Double-click the script to open it in the editor and add these two methods:

```
void OnBecameInvisible() {
  Destroy(gameObject);
}

void OnCollisionEnter(Collision collision) {
  Destroy(gameObject);
}
```

`OnBecameInvisible()` is a useful method that is called when the object is no longer visible by any camera. In this case, this is called when the bullet goes off the screen. This is a perfect time to destroy the bullet, as it's no longer needed.

Note: There are two important things to understand about this.

First, for this to work the GameObject must have a renderer component attached to it. Otherwise, Unity has no idea if the GameObject is visible or not.

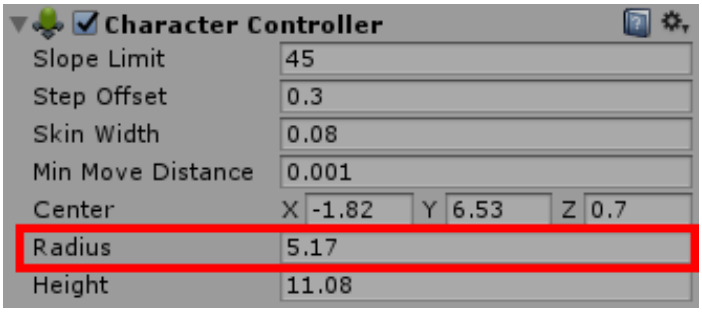
Second, Unity considers a GameObject invisible if it's invisible from *all* cameras. For instance, the GameObject may not be visible in the Game window but if it's visible in your **Scene view**, then it's still considered visible.

Weird? Yes, but that's how Unity rolls. :]

`OnCollisionEnter()` is called during a collision event. The `Collision` object contains information about the actual collision as well as the target object. For example, this will be called when a bullet hits an alien or a wall. When this occurs, you simply destroy the bullet.

Now play your game and fire. Your bullets disappear as they collide into objects. How about another test?

With the game still running, select the **SpaceMarine**. In the Character Controller component, set **Radius** to `5.17`.



Now, fire your gun. Believe it or not, the gun is firing, but the bullets are hitting the marine's collider, and then being instantly destroyed, so it looks as though nothing is happening.

You can solve this problem with selective collisions that you set up with layers.

Collisions and layers

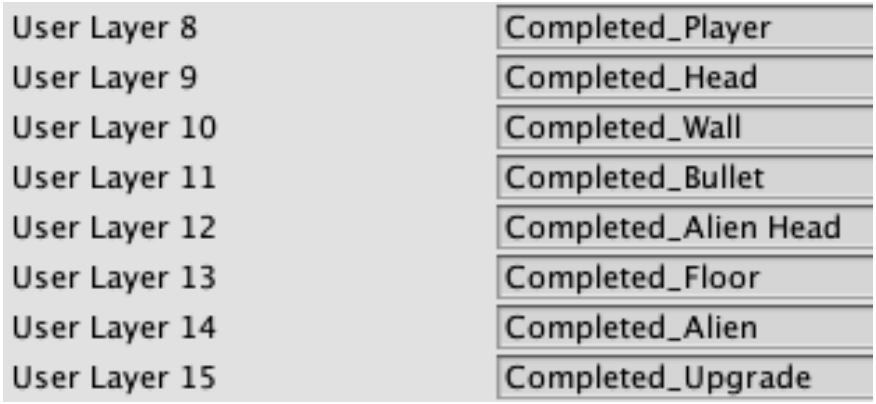
By default, all GameObjects exist on one physics layer. This means that all GameObjects with Rigidbodies will collide with one another.

As you can see from how the bullets collide with the space marine, the default isn't ideal. You need the bullets to ignore the marine — and to do this, you need to create a new layer.

First, **stop** the game and navigate to **Edit\Project Settings\Tags and Layers**. The Inspector will show the Tags and Layers view with the layers section expanded. The first eight layers are controlled by Unity but the rest are wide open.

Label User Layer 8 and onwards as follows: **Player**, **Head**, **Wall**, **Bullet**, **Alien Head**, **Floor**, **Alien** and **Upgrade**.

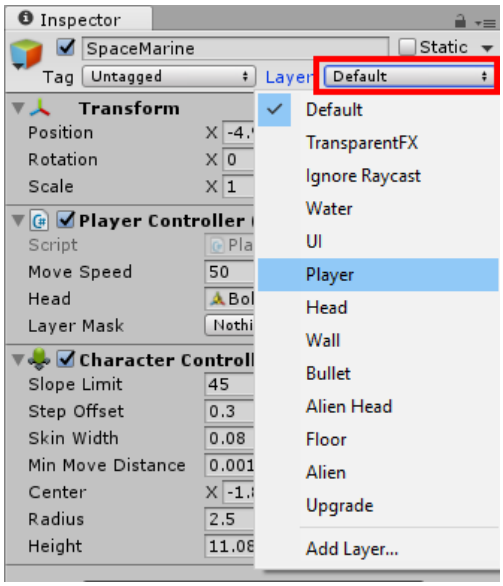
If you are doing this tutorial in the same project as the downloaded assets, you'll note that the layers have already been included. This is so you can run the sample games. If so, it will look like the following:



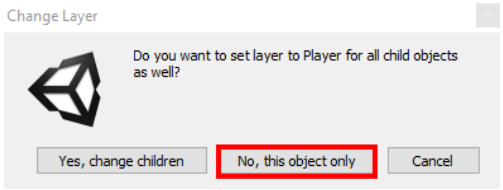
Just add the layers after the included layers. For the sake of clarity, the tutorial will only show user the added layers.

Yes, you'll use all those physics layers in your game. Notice how they correspond with the GameObjects.

In the Hierarchy, select the **SpaceMarine**. In the Inspector set the **Layer** drop-down to **Player**.

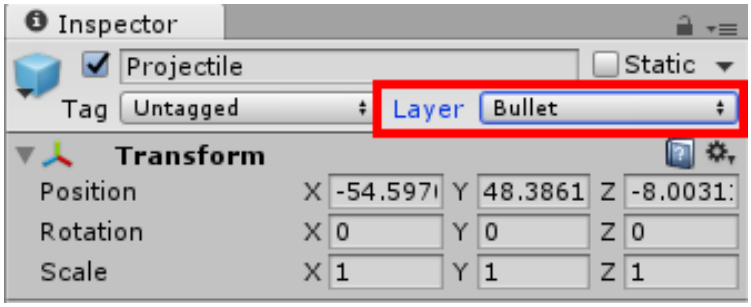


You'll see a prompt that asks if you want to set the layer for all the child objects. Click **No, this object only**.

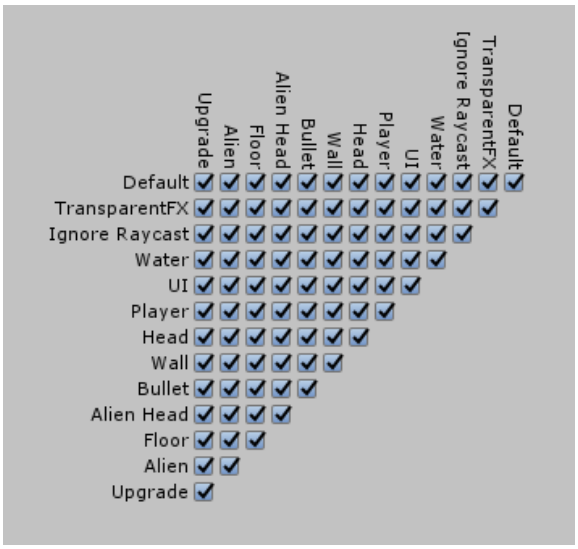


Note: GameObjects can only belong to one layer, but their children can belong to different layers, as you'll see later in this book.

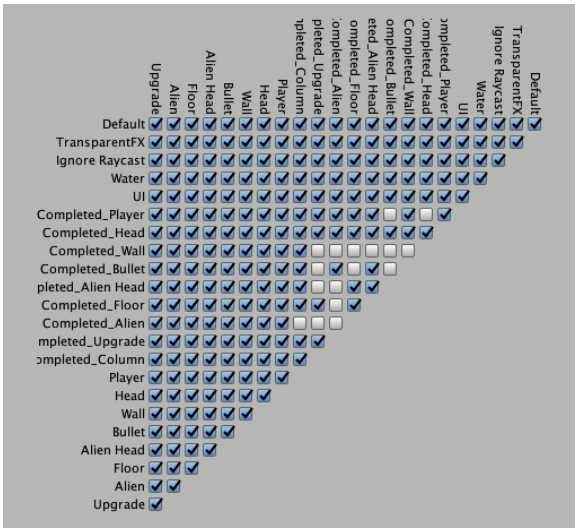
Next, in the **Prefabs** folder, select the **Projectile** prefab, and in the layer drop-down, select **Bullet**.



By default, all layers interact with each other, and you have to opt them out of collisions. To do this, click **Edit\Project Settings\Physics**, scroll to the bottom and look for a matrix of layers.



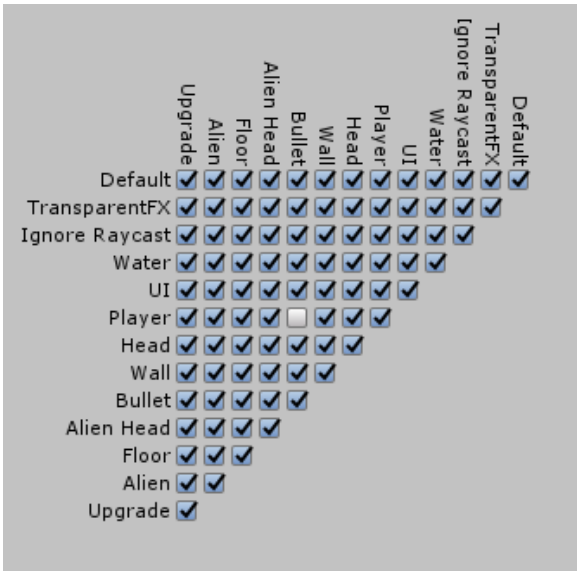
If you are doing this tutorial with the downloaded assets, you'll notice that the layer matrix is already completed. This is so you can run the completed games. The matrix will look like the following:



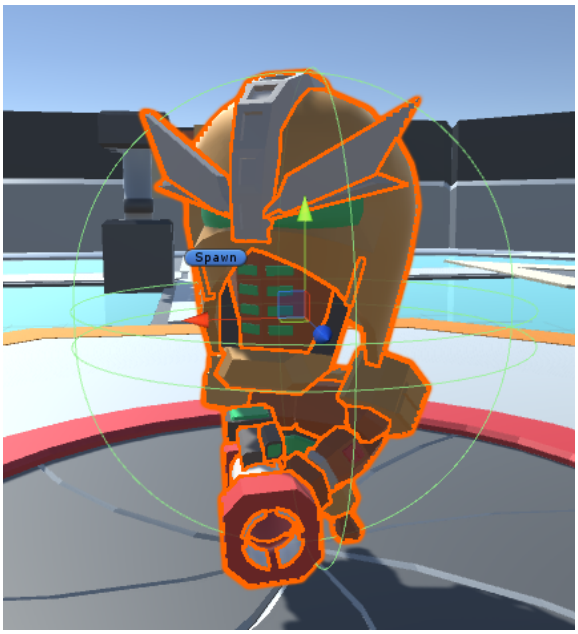
For the sake of clarity, this tutorial will only show the layers that you add versus the prebuilt layers. This to make it easier to read the matrix.

The y-axis shows the subject layers, and the x-axis shows the collision target layers. The checkmark indicates whether the subject layer will interact with the target layer.

In the **Player row**, **uncheck** the **Bullet column**. Now, the Player layer will ignore all interactions with the Bullet layer.



Play your game and run that same test while it's live — select the **SpaceMarine** and set the **Radius** to `5.17` in the Inspector. Fire your gun. This time, you get bullets galore!



Joints

When you first started making this game, you called it `Bobblehead Wars` but so far, it's been all wars and no bobble. Thankfully, Unity physics comes to the rescue with **joints**.

These aren't the kinds of joints that would require you to be in Amsterdam, Oregon, Colorado or Washington, nor are they the kinds of joints made by the filmaker Spike Lee. These are *physics* joints with a more conventional purpose: connecting two GameObjects together in a relationship defined by the physics engine.

Think about a wrecking ball for a moment, specifically, the chain. Each link would be a GameObject, and you'd need to join them together with joints to allow each link to move individually but affect its neighbors, and in turn, the whole group is influenced by the ball.

There are many joints available to you:

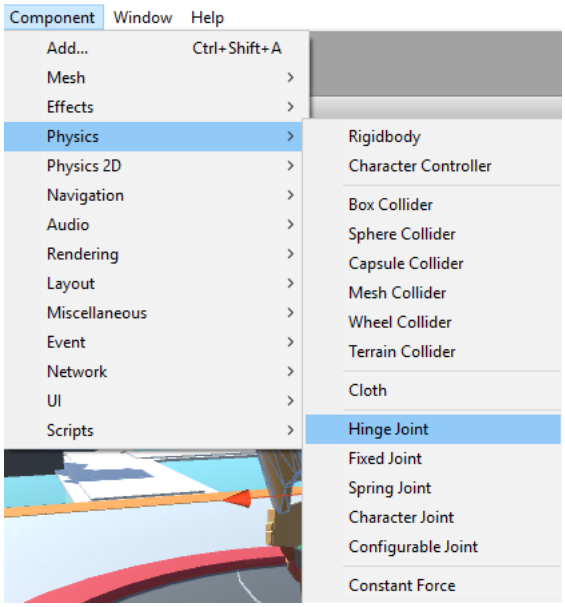
- A **Hinge Joint** ties the movement of one GameObject to another — for instance, a door hinge.
- A **Spring Joint** acts like an invisible spring between two GameObjects.
- A **Fixed Joint** restricts a GameObject's movement with an unrelated GameObject. A good example (from the documentation, even) is a sticky grenade.
- A **Character Joint** allows you to constrain motion to create rag doll effects.
- A **Configurable Joint** incorporates the features of the other joints and more. In essence, you use it to modify existing joints or create your own.

You'll use a **Hinge Joint** to make the head bobble back and forth.

Playing off the example above, the space marine's body is the doorframe, and his head is the door. Every time the marine moves, you'll add force to his head to make it bounce back and forth.

Adding joints

In the Hierarchy, expand **SpaceMarine** and select the **BobbleMarine-Head**. Then from the main menu, click **Component\Physics\Hinge Joint**. (As you can see, there's more than one way to add a component.)



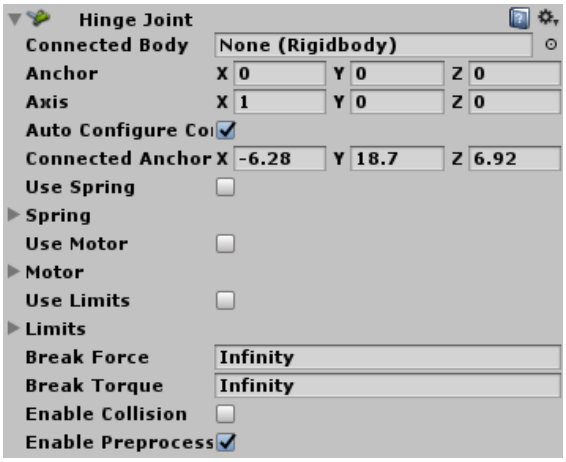
Check the Inspector. You'll see the hinge joint in there now.

For the hinge joint to work, the connected GameObjects must have Rigidbodies attached to them; Unity automatically adds these for you when you create a hinge joint.

In the Rigidbody component, **uncheck** the **Use Gravity** property.

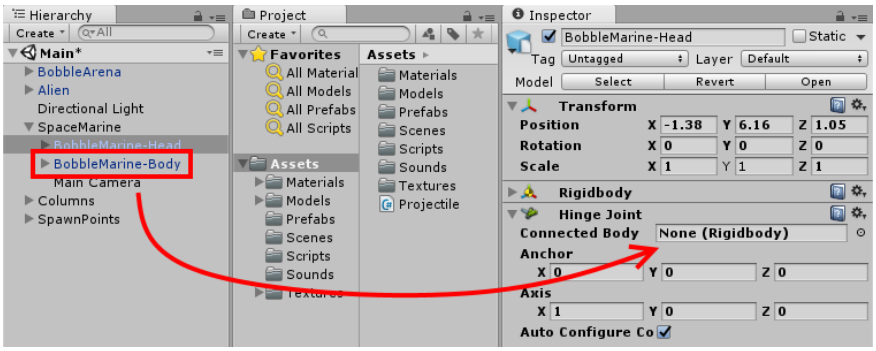
In the Hierarchy, select **BobbleMarine-Body** and in the Inspector, click the **Add Component** button. From the component listings, select **Physics** and then **Rigidbody**. **Uncheck** the **Use Gravity** property and **check** the **Is Kinematic** property.

Now, select **BobbleMarine-Head** in the Hierarchy, and look at the hinge joint in the Inspector. You'll see many options!



The first option is the **Connected Body**, and in this case, it's the other GameObject that the head will be "hinged" to.

In the Hierarchy, drag the **BobbleMarine-Body** to the **Connected Body** property. Now, the GameObjects are connected by physics.



The **Anchor** indicates where the end point of the joint attaches to the GameObject as specified in the object's **local coordinate space**. (0, 0, 0) within the head's local coordinate space represents the neck area, so leaving the value at (0, 0, 0) is fine.

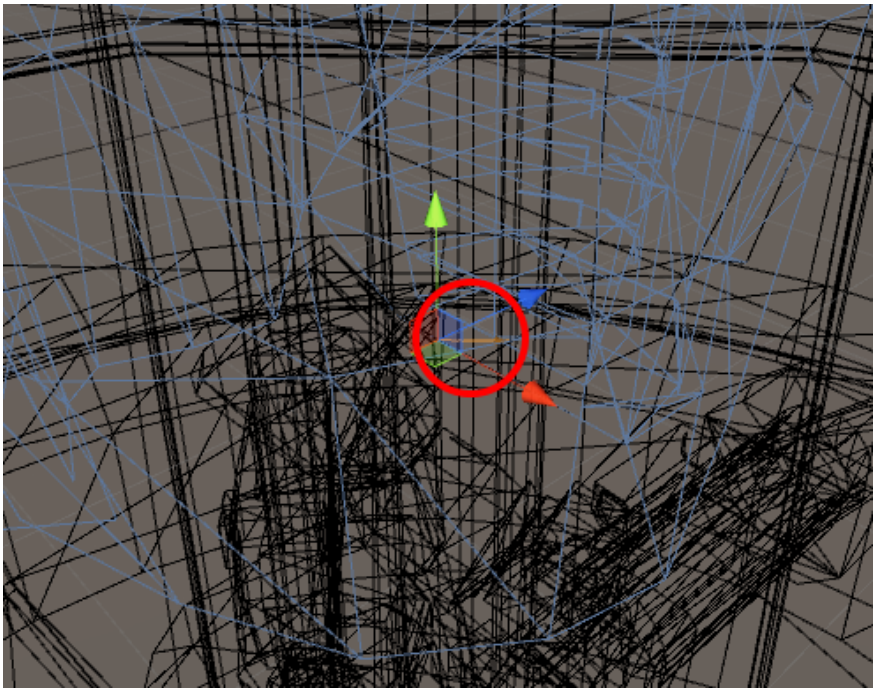
The **Axis** determines what direction the body will swing. You want this to move on the z-axis, i.e., up and down. Set the **Axis** to (**0**, **0**, **1**).

The **Connected Anchor** parameter specifies the anchor point of the other end of the joint. If the Connected Rigid Body field is empty, this value is in the scene's coordinate system. However, when Connected Rigid Body is set, as it is now to the Space Marine's body, the Connected Anchor coordinates refer to the connected rigid body's local coordinate space.

By default, Unity sets the **Connected Anchor** point to the same position as the **Anchor** point. For this game, you want these to be slightly different, so **Uncheck** the **Auto Configure Connected Anchor** checkbox.

Figuring out the connected anchor's best settings tends to involve some trial and error. Thankfully, Unity shows you the anchor point's location that you can use as a starting point.

Switch the Scene view to **Wireframe** mode to show a small orange line. This represents the anchor's position, and it'll update as you alter the Connect Anchor values.



Set the **Connected Anchor** to (**-2.06**, **6.16**, **0.16**) to pin the anchor to the marine's neck.

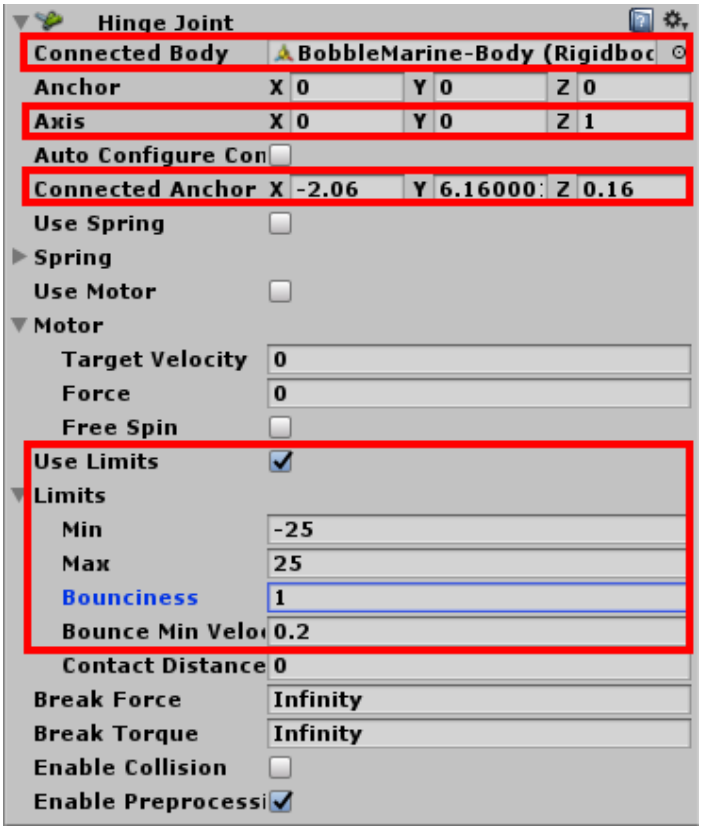
The spring properties will add "springy" behavior to the joint and the motor property will move the joint without any external force. To illustrate motor values, think of a door mysteriously opening on its own.

The **Limits** properties control how far the joint should swing.

Check the **Use Limits** checkbox and **expand** the **Limits** property. In the **Min** property, put –25 and set the **Max** to 25 - this represents that the head should bounce at most 25 degrees in either way. To make it bounce like a proper bobble head, set **Bounciness** to 1 — that's the max value.

Keep the rest of the values as they are, but take note of **Break Force** and **Break Torque**. You'd use these if you wanted the hinge to snap under a certain threshold, e.g., create the effect of a door being blown off its hinges.

Here's the configured joint:



The hinge needs a collider to work.

Click the **Add Component** button, and from the **Physics** category, select **Sphere Collider**. Set **Center** to (**0.41**, **2.7**, **-0.21**) and **Radius** to **2.36**. This closely matches the width of the head.

The last thing this effect needs is some force to make the head bob.

Open **PlayerController.cs** in your code editor. You need a reference to the connected head Rigidbody, so add the following instance variable:

```
  public Rigidbody head;
```

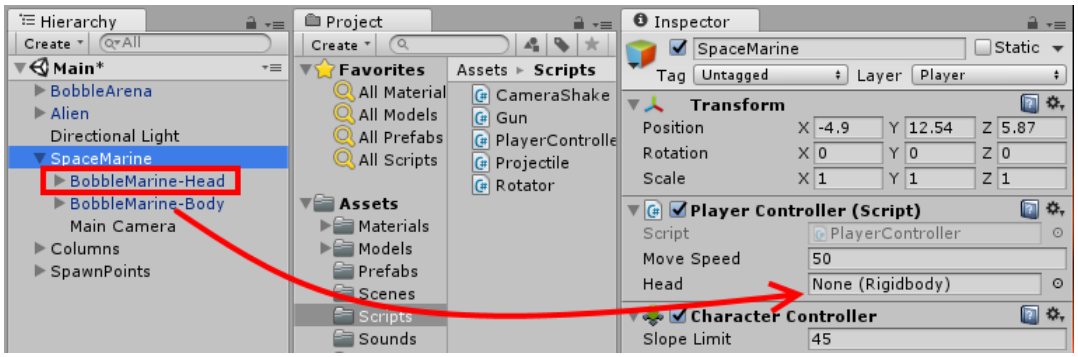
You might ask why you're using a Rigidbody instead of a GameObject.

Remember, a Rigidbody deals with physics. You're adding the head to apply force to it, and there's no way to apply force to a regular GameObject. Hence, you need a Rigidbody.

So how do you assign a Rigidbody to a public field?

Just like you did with the transform. Save your change and switch back to Unity.

Select the **SpaceMarine** in the Hierarchy. In the Inspector, look for a property named **Head** that can accept a rigidbody. **Drag** the **BobbleMarine-Head** to it.



Much like it does with the transform, Unity uses the Rigidbody component attached to the GameObject to assign to the head field.

Switch back to **PlayerController.cs**. Add the following underneath `Update()`:

```
void FixedUpdate() {

}
```

As you know, `Update()` runs with every frame and it's called as often as the frame changes, so 40 fps means it's called 40 times per second.

`FixedUpdate()` is an entirely different beast. Because it handles physics, it's called at consistent intervals and not subject to frame rate. If you were to check the `deltaTime` between each call to `FixedUpdate()`, you'd see it doesn't change.

Anything that affects a Rigidbody should be updated in `FixedUpdate()`. Given that you are applying force to a rigidbody to move the Bobblehead, you need to use `FixedUpdate()` instead of `Update()`.

Add the following to `FixedUpdate()`:

```
Vector3 moveDirection = new Vector3(Input.GetAxis("Horizontal"),
  0, Input.GetAxis("Vertical"));
if (moveDirection == Vector3.zero) {
  // TODO
} else {
  head.AddForce(transform.right * 150, ForceMode.Acceleration);
}
```

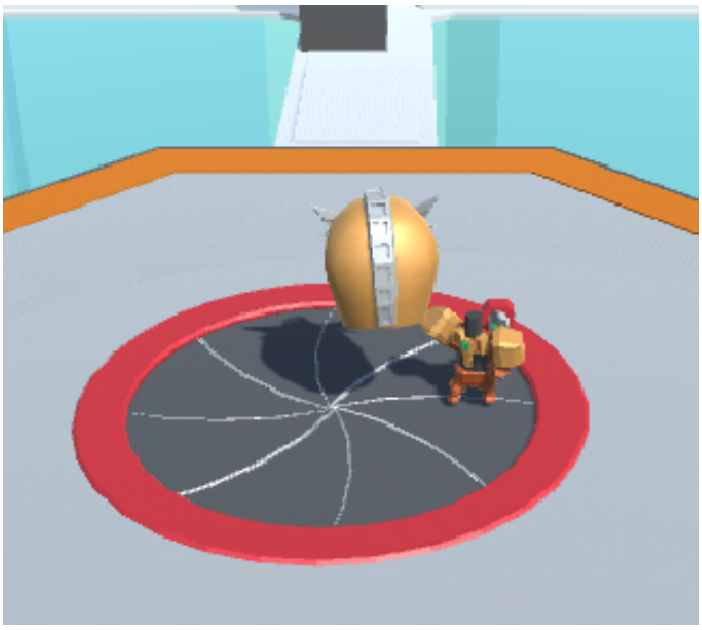
This code moves the head when the marine moves. First, you calculate the movement direction. If the value equals `Vector3.zero`, then the marine is standing still.

`AddForce()` is how you get the head to move. You provide a direction then multiply it by the force amount.

There are different force types, and you're using `ForceMode.Acceleration`, which gives a continuous amount of force that ignores the mass.

> **Note**: If you wanted to use the mass, you'd use `ForceMode.Force`. For more detail about the other options, search the documentation for `ForceMode`.

Now play your game and move. Uh-oh, by the looks of the marine's head, he needs an exorcist. Based on everything you've learned so far, can you guess what's going wrong?

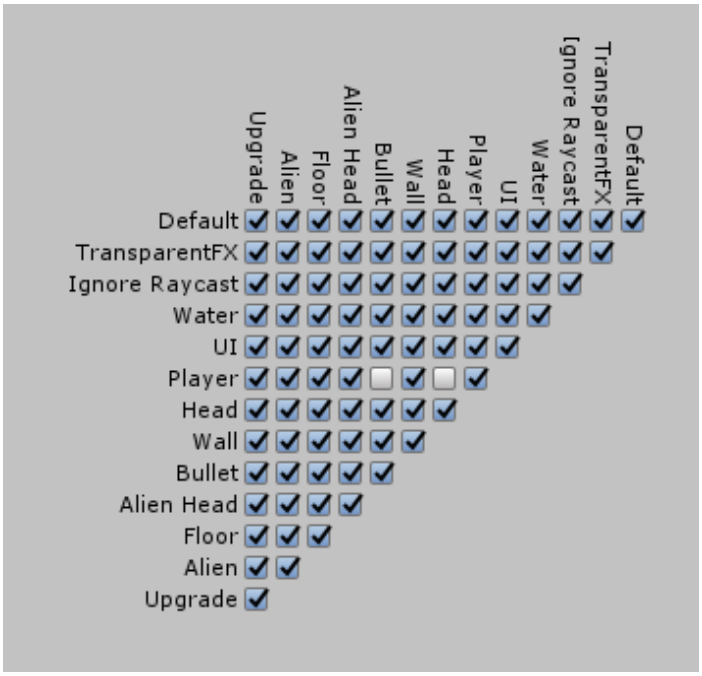


You added a sphere collider to the head, but the character controller added another collider, and there's interference between them. Layers to the rescue!

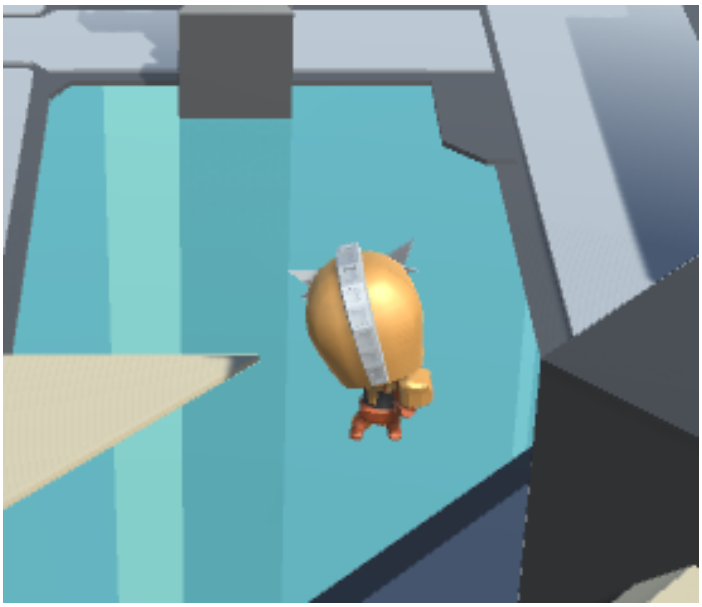
> **Note**: If the head is not moving, make sure that the head's `Rigidbody` component has `IsKinematic` unchecked.

Select **BobbleMarine-Head** in the Hierarchy. In the Inspector, assign it the **Head** layer and select the **Yes, change children** option in the confirmation dialog.

Finally, click **Edit\Project Settings\Physics**, and in the physics matrix, uncheck the **Head column** in the **Player row**.



Play your game. Begun, these Bobblehead wars have.



Raycasting

At this point, you can fire a weapon and push the alien around. Unfortunately, our hero still can't turn. Considering that these aliens love to chew on back hair (they truly are alien), the space marine needs to duck, spin and weave.

During gameplay, you also want the marine to shoot in the direction of the mouse pointer. Seems simple enough, right?

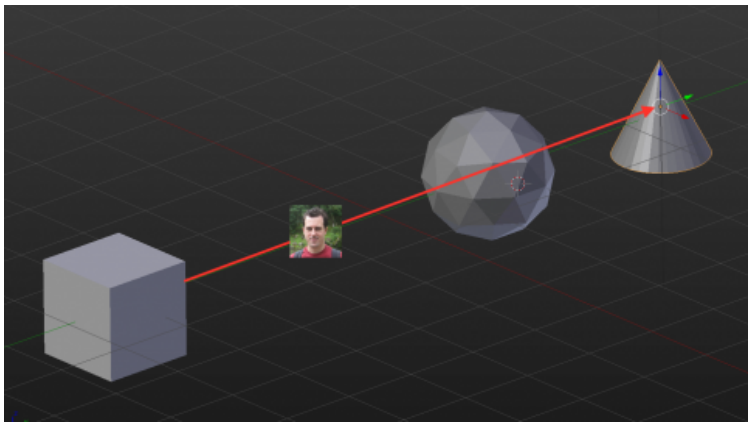
Fundamentally, it's a tricky problem; the game exists in 3D space but the mouse exists in 2D.

The solution is to convert the mouse pointer to 3D space, then determine where it's pointing. It sounds like a complicated process, but it's quite easy with **raycasting**.

Raycasting shoots an invisible ray from a target to a destination, and it sends a notification when it hits a GameObject. You'll likely encounter a lot of GameObjects that intersect with a ray. Using a mask lets you filter out unwanted objects.

Raycasts are incredibly useful and serve a variety of purposes. They're commonly used to test if another player was struck by a projectile, but they can help you test if there's geometry underneath a mouse pointer, as you'll soon see.

The following image shows a raycast from a cube to a cone. Since the ray has a cone mask on it, it ignores the iconsphere layer and reports a hit to the cone.



You'll cast a ray from the camera to the mouse pointer. Then the space marine will turn to wherever the ray hits the floor, essentially following the mouse.

Open **PlayerController.cs** and add the following instance variables:

```
public LayerMask layerMask;
private Vector3 currentLookTarget = Vector3.zero;
```

The `LayerMask` lets you indicate what layers the ray should hit. `currentLookTarget` is where you want the marine to stare. Since you don't know where to look at the start of the game, you set the value to zero.

> **Note**: As you add instance variables, you'll see that public and private variables become jumbled together. It's for the sake of brevity. A best practice is to group them by access modifier and keep public and private variables together.

Next, add the following to `FixedUpdate()` underneath the existing code:

```
RaycastHit hit;
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
```

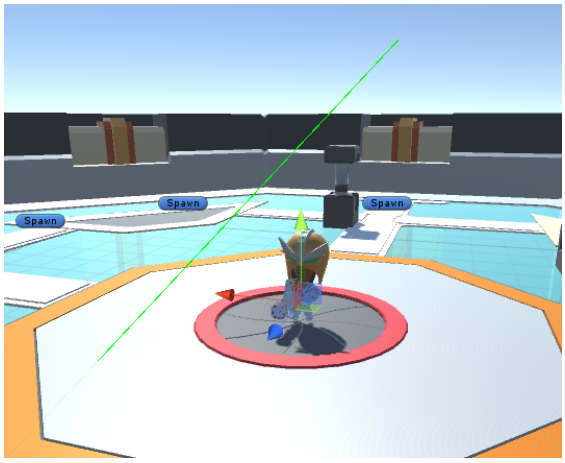
First, this creates an empty `RaycastHit`. If you get a hit, it'll be populated with an object. Second, you actually cast the ray from the main camera to the mouse position.

Rays are invisible, so they are hard to debug. You might *think* they will hit an object, but you won't *know* until you actually see it hit something. Add the following code:

```
Debug.DrawRay(ray.origin, ray.direction * 1000, Color.green);
```

This will draw a ray in the Scene view while you're playing the game.

Return to Unity and run your game. Look in the Scene view while moving your mouse in the Game window. You should see the ray as a green line.



Return to PlayerController.cs, and add this code where you left off in `FixedUpdate()`:

```
if (Physics.Raycast(ray, out hit, 1000, layerMask,
    QueryTriggerInteraction.Ignore)) {
}
```

These two lines of code do quite a bit:

- `Physics.Raycast` actually casts the ray.
- First you pass in the `ray` that you generated along with the `hit`. Since the `hit` variable is marked as `out`, it can be populated by `Physics.Raycast()`.
- `1000` indicates the length of the ray. In this case, it's a thousand meters.
- The `layerMask` lets the cast know what you are trying to hit.
- `QueryTriggerInteraction.Ignore` tells the physics engine not to activate triggers.

Add the following code inside the braces:

```
if (hit.point != currentLookTarget) {
  currentLookTarget = hit.point;
}
```

The `hit.point` comprises the coordinates of the raycast hit — it's the point where the hero should look. If it's different, perhaps because the player moved the mouse, then `currentLookTarget` will be updated.

Add the following after the previous bit of code:

```
// 1
Vector3 targetPosition = new Vector3(hit.point.x,
  transform.position.y, hit.point.z);
// 2
Quaternion rotation = Quaternion.LookRotation(targetPosition -
  transform.position);
// 3
transform.rotation = Quaternion.Lerp(transform.rotation,
  rotation, Time.deltaTime * 10.0f);
```

This is where the actual rotation comes into play.

1. You get the target position. Notice that the `y` axis is derived from the marine's position, because you want the marine to look straight ahead instead of the floor. If you used the floor, then the marine would rotate downwards.
2. You calculate the current `Quaterion`, which is used to determine rotation. (You can learn all about Quaterions in the Unity API appendix chapter.) To find the target quaternion, you subtract the `targetPosition` from the current position. Then you call `LookRotation()`, which returns the quaterion for where the marine should turn.
3. Finally, you do the actual turn by using `Lerp()`. Remember, `Lerp()` is used to change a value (such as rotation in this place) smoothly over time. To learn more about `Lerp()`, check out the Unity API appendix chapter.

Save your changes and go back to Unity. In the Hierarchy, select the **SpaceMarine**. Look for the Player Controller script in the Inspector, and select **Floor** in the drop-down.

The last thing to do is assign the `floor` physics layer to the, er, floor. In the Hierarchy, expand the **BobbleArena** and select **World_Center**. In the layer dropdown, select **floor**.

Play your game like you mean it.

Now the marine follows your mouse cursor and turns on a dime. Now you're playing with power! :]

Where to go from here?

Wow! That was quite a chapter! As you can see, the physics engine has a stockpile of features and does a lot of cool stuff. In this chapter, you learned about:

- **Character Controllers** and how they help simplify your code.
- **Collisions and Layers** for when you want to collide with another object or opt out of collisions entirely.
- **Joints** and how to put them in practice.
- **Raycasting** and how to use rays to increase interactivity.

There's a lot more to do in Bobblehead Wars. So far, you have just one alien, and he's little more than a creepy-crawly couch potato.

In the next chapter, you'll make a horde of aliens and unleash them upon the marine. He better stay sharp and hope that raycast is working right!

Chapter 5: Managers and Pathfinding

By Brian Moakley

Bobblehead Wars is coming along handsomely! At this point, you have a feisty space marine who can turn, shoot, and gun down aliens. Also, through the use of a hinge joint, his head can wobble back and forth so he can do so in style.

Nevertheless, Bobblehead Wars isn't ready for release. First, there is just one alien when the space marine was expecting a horde. And second, the alien currently isn't that scary - he just stands around!

To fix this, you'll learn how to spawn enemies over time and make them chase after the space marine, through the power of the Game Manager and pathfinding. Let the battle begin!

Introducing the GameManager

Unity refers to an object that handles a specific subsystem as a **manager**. Given your experience in the workplace, you might think this means they do nothing and leave all the work for their underlings, but in Unity managers actually lend a hand. ;]

For instance, an input manager is an object that lets you adjust all the controls for your game. There's also a settings manager that handles all your project settings, and a network manager for managing the state of a networked game.

In this chapter, you'll create your own kind of manager. Specifically, you'll create a game manager so that you can easily tweak your game setting while it's being played. This will make it much easier to balance your game during play testing.

Imagine that you're playing and there are just too many enemies. The space marine has no chance. You could make an adjustment to your game manager to dial back the aliens hordes in real time.

Remember, any changes you make while your game is played will be lost, right?

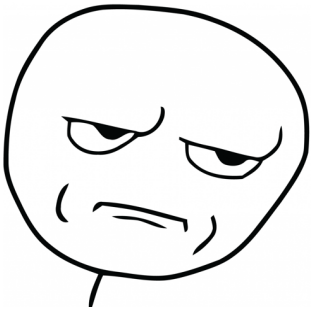
There's an exception to this rule that you'll try out in this chapter.

Creating the game manager

Open your project from Chapter 4. If you need to start fresh, please find the starter project that's included with the resources for this chapter.

In the Hierarchy, click the **Create** button, select **Create Empty** from the drop-down and name it **GameManager**.

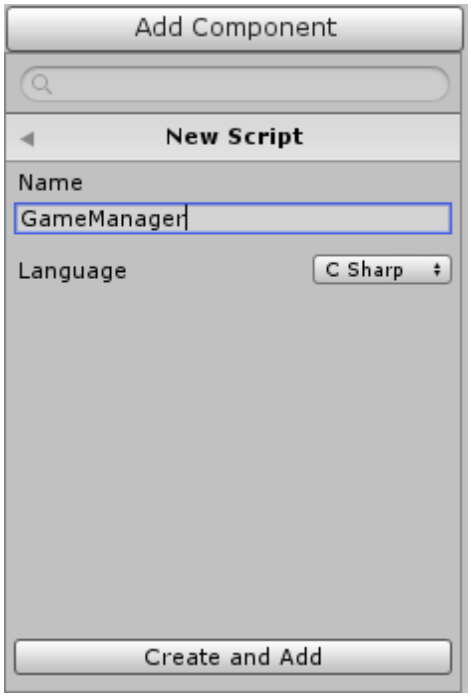
Congratulations! You've made your game manager. That's it for this chapter.



Ok, ok, you're right. That wasn't very nice. You have *much* more to do. :]

Unity doesn't provide a "stock" game manager component — it would be silly because each game is unique. You need to write your own.

Select the **GameManager** in the Hierarchy and click the **Add Component** button in the Inspector. Pick **New Script** and name it **GameManager**. Set the language to **C sharp** then click **Create and Add**.



Don’t forget to move the script into your scripts folder!

Double-click the **GameManager script** to open it in your code editor. Get ready to spend some time in here; this script will manage all aspects of your game, and you have many variables to add.

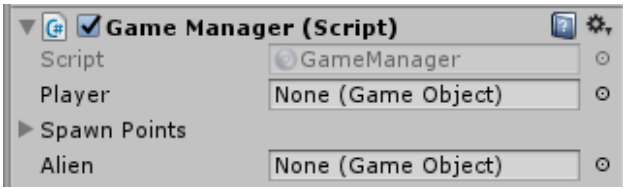
Add this code underneath the opening brace:

```
public GameObject player;
public GameObject[] spawnPoints;
public GameObject alien;
```

These three variables are critical to the manger’s function. Here’s a breakdown:

- `player` represents the hero’s GameObject. `GameManager` will use this to determine its current location.
- `spawnPoints` are locations from which aliens will spawn in the arena. You declare it as an array because you have multiple locations — remember those capsules you laid out across the arena floor?
- `alien` represents the prefab for the alien. The GameManger will create an instance of this object when it’s time to spawn.

Save your script and switch back to Unity. Select the **GameManager** in the Hierarchy to see three new fields.

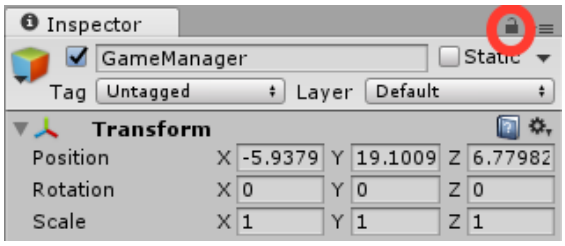


Notice that **Spawn Points** has an adjacent disclosure triangle, indicating that it takes multiple values.

First, you’ll populate the **player** field. **Select** the **SpaceMarine** in the Hierarchy. You’ll notice the Inspector switches from the **GameManager** to the **SpaceMarine**, which isn’t what you actually want the engine to do. There are a three ways to work around this behavior:

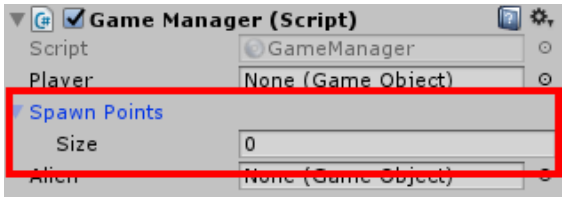
1. As you've done in previous chapters, you could click and drag the object to the property (rather than clicking without dragging). When you click and drag, the Inspector doesn’t select the given object.
2. A second option is to click the round circle to the right of a property to bring up a popup to select the approrpiate object.
3. A third option is to lock the GameManager to the Inspector. This prevents you from accidentally selecting another GameObject.

Locking the GameManager is the best option of the two since you've got quite a few changes to make. To do this, select the **GameManager** in the hierarchy, and click the **lock icon** at the top of the Inspector.

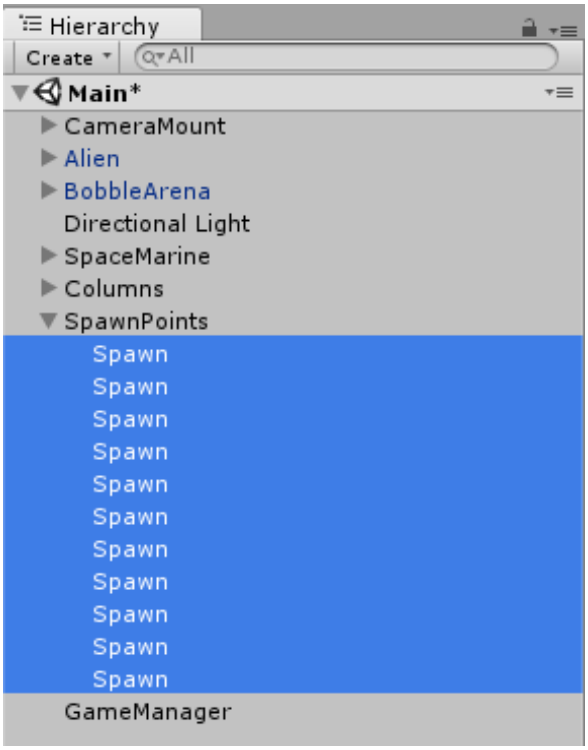


Select the **SpaceMarine** in the Hierarchy; the Inspector doesn't change anymore. In fact, the Inspector will remain locked to the GameManager until you click the lock again.

Drag the **SpaceMarine** to the **Player** field. **Expand** the **SpawnPoints** GameObject in the Hierarchy.



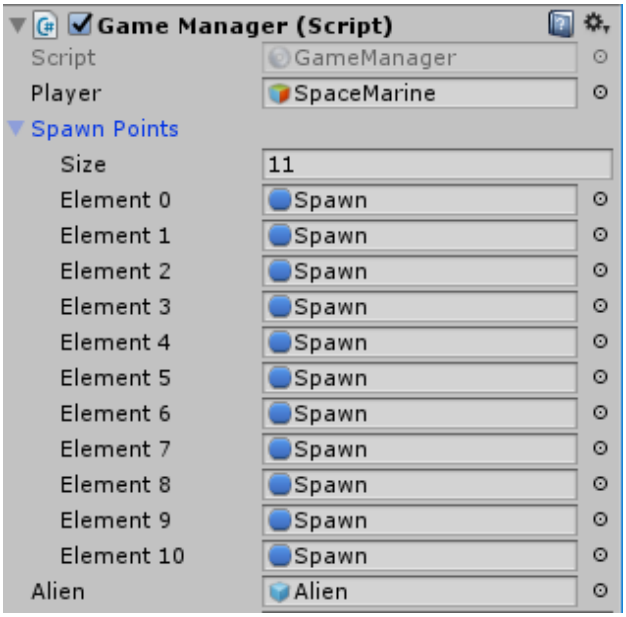
Select all the spawn points by pressing **Shift** and **selecting** the top and bottom **Spawn** objects.



Drag all the spawn points to the **Spawn Points** property in the GameManager. You'll know you're in the right spot when a green plus icon appears.

Next, **Drag** the **Alien prefab** from the Prefabs folder (**not** the Hierarchy) to the **Alien** field.

The GameManager should now look like this:



Unlock the Inspector and switch back to your code editor.

Add the following beneath the previous variables:

```
public int maxAliensOnScreen;
public int totalAliens;
public float minSpawnTime;
public float maxSpawnTime;
public int aliensPerSpawn;
```

These are some variables that you will use to configure gameplay for Bobblehead Wars:

- `maxAliensOnScreen`: will determine how many aliens appear on the screen at once.
- `totalAliens`: will represent the total number of aliens the player must vanquish to claim victory.
- `minSpawnTime` and `maxSpawnTime`: will control the rate at which aliens appear.
- `aliensPerSpawn`: will determine how many aliens appear during a spawning event.

Next you need to add a few private variables that the game manager will need for recordkeeping. Place the following underneath the public variables:

```
private int aliensOnScreen = 0;
private float generatedSpawnTime = 0;
private float currentSpawnTime = 0;
```

Here's what these will manage:

- `aliensOnScreen`: will track the total number of aliens currently displayed. You'll use it to tell the game manager whether to spawn or wait.
- `generatedSpawnTime`: will track the time between spawn events. You'll randomize this to keep the player on their toes.
- `currentSpawnTime`: will track the milliseconds since the last spawn.

Finally, this script needs some C# generics. Add the following at the top of the script underneath all the other `using` statements:

```
using System.Collections.Generic;
```

This allows the use of generic arrays, which you'll use in just a moment.

Spawning the aliens

Next up is writing the spawning code. With the `GameManager` script still open, add the following to `Update()`:

```
currentSpawnTime += Time.deltaTime;
```

`currentSpawnTime` accumulates the amount of time that's passed between each frame update. Next, add the following:

```
if (currentSpawnTime > generatedSpawnTime) {

}
```

The entirety of the spawning code exists between the parentheses. Once `currentSpawnTime` exceeds `generatedSpawnTime`, you get new aliens.

Between the braces, add the following:

```
currentSpawnTime = 0;
```

This bit of code resets the timer after a spawn occurs — no reset means no more enemies.

Next, add the following:

```
generatedSpawnTime = Random.Range(minSpawnTime, maxSpawnTime);
```

This is your spawn time randomizer. It creates a time between `minSpawnTime` and `maxSpawnTime`. You'll see it come into play during the next change to`Update()`.

Now, add the following:

```
if (aliensPerSpawn > 0 && aliensOnScreen < totalAliens) {

}
```

Here's the logic that determines whether to spawn. First, `aliensPerSpawn` should be greater than zero, and `aliensOnScreen` can't be higher than `totalAliens`. This code is a preventative measure that stops spawning when the maximum number of aliens are present.

Between the braces, add the following:

```
List<int> previousSpawnLocations = new List<int>();
```

This creates an array you will use to keep track of where you spawn aliens each wave. This will be handy so you make sure not to spawn more than one alien from the same spot each wave.

Next, add the following:

```
if (aliensPerSpawn > spawnPoints.Length) {
  aliensPerSpawn = spawnPoints.Length - 1;
}
```

This limits the number of aliens you can spawn by the number of spawn points.

Add the following:

```
aliensPerSpawn = (aliensPerSpawn > totalAliens) ? aliensPerSpawn -
totalAliens : aliensPerSpawn;
```

This is another chunk of preventive code. If `aliensPerSpawn` exceeds the maximum, then the amount of spawns will reduce. This means a spawning event will never create more aliens than the maximum amount that you've configured.

Now comes the actual spawning code. Add the following:

```
for (int i = 0; i < aliensPerSpawn; i++) {

}
```

This loop iterates once for each spawned alien. Add the following between the braces:

```
if (aliensOnScreen < maxAliensOnScreen) {
  aliensOnScreen += 1;
  // code goes here
}
```

This code checks if `aliensOnScreen` is less than the maximum, and then it increments the total screen amount.

After the comment, add the following:

```
// 1
int spawnPoint = -1;
// 2
while (spawnPoint == -1) {
  // 3
  int randomNumber = Random.Range(0, spawnPoints.Length - 1);
  // 4
  if (!previousSpawnLocations.Contains(randomNumber)) {
    previousSpawnLocations.Add(randomNumber);
    spawnPoint = randomNumber;
  }
}
```

This code is responsible for finding a spawn point.

1. `spawnPoint` is the generated spawn point number. Because it references an array index, it's set to `-1` to indicate that a spawn point hasn't been selected yet.
2. This loop runs until it finds a spawn point or the spawn point is no longer `-1`.
3. This line produces a random number as a possible spawn point.
4. Next, it checks the `previousSpawnLocations` array to see if that random number is an active spawn point. If there's no match, then you have your spawn point. The number is added to the array and the `spawnPoint` is set, breaking the loop. If it finds a match, the loop iterates again with a new random number.

Now that you have the spawn point index, you need to get the actual spawn point GameObject. Hang in there for a few more steps! You're close to unlocking an achievement.

Add the following after the closing brace of the `while` statement:

```
GameObject spawnLocation = spawnPoints[spawnPoint];
```

Since there are already assigned spawn points in the Inspector, this grabs the spawn point based on the index that you generated in the last code.

Now, to spawn the actual alien. Add the following:

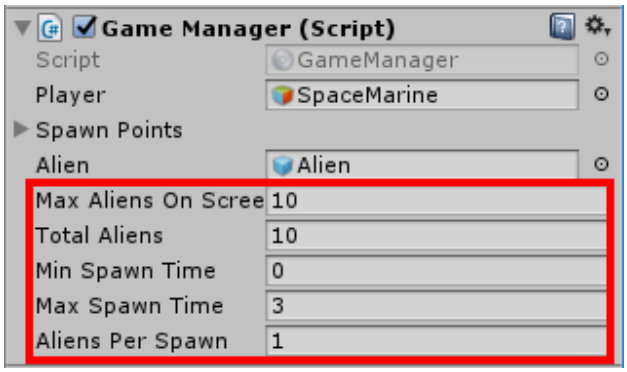
```
GameObject newAlien = Instantiate(alien) as GameObject;
```

`Instantiate()` will create an instance of any prefab passed into it. It'll create an object that is the type `Object`, so you must cast it into a `GameObject`. Now add the following:

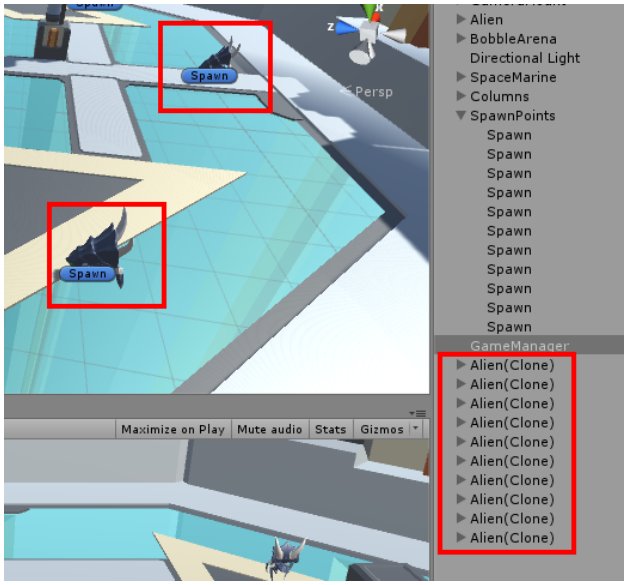
```
newAlien.transform.position = spawnLocation.transform.position;
```

This positions the alien at the spawn point. Save your code and switch back to Unity.

Select the GameManager and go to the Inspector. Set the **Max Aliens on Screen** to `10`, **Total Aliens** to `10`, **Min Spawn Time** to `0`, **Max Spawn Time** to `3`, and **Aliens Per Spawn** to `1`.



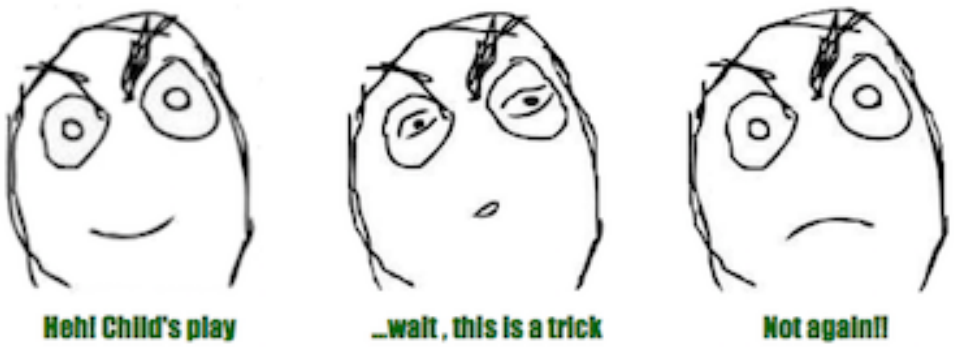
Play your game and watch the Scene view. Glorious! Look at all those creepy critters. Achievement unlocked!



Pathfinding in Unity

The aliens have arrived, but they're little more than decorations at this point. They need to get moving!

Seems like an easy enough task, yes? You could set a target for the aliens, and in each frame move them towards the target. Problem solved. End of chapter!



Of course it's not that easy. First, you'd have to rotate the alien constantly so it faces the space marine. Second, a real problem occurs when an alien runs into a column.

The aliens will keep walking into the column until something causes them to move in a different direction. It's pretty ridiculous when a single column can block an entire mob.

Perhaps you could get away with this behavior 20 years ago, but modern gamers expect better than this.

So what do you do?

Your first inclination may be to write the pathfinding. For instance, you could write code that makes the aliens turn and move until they are free of the column, then redirect them towards the hero.

It's not that it wouldn't work, but you'd be treating a fundamental problem like an edge case. What would happen if you added another floor to the mix? You'd have to apply similar fixes, and then your code quickly becomes an unreadable pile of spaghetti.

Thankfully, there's a better way.

Unity's navigation system

It turns out Unity has pathfinding solutions you can use right out of the box, through its navigation system. Four objects comprise the navigation system:

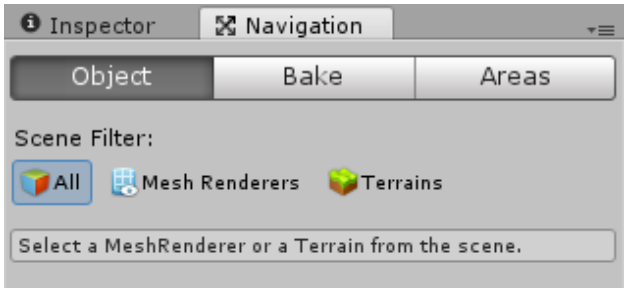
1. **NavMesh**: Represents walkable areas. This works on flat and dimensional terrain such as stairs.
2. **NavMesh Agent**: The actor or agent using the NavMesh. You give your agent a goal and it will find its way using the NavMesh.
3. **Off-Mesh Link**: A shortcut that can't be represented on the map, such as a gap that characters can actually cross.

4. **NavMesh Obstacle**: Obstacles that the agent should avoid. Your agents will use pathfinding to figure a way around them.

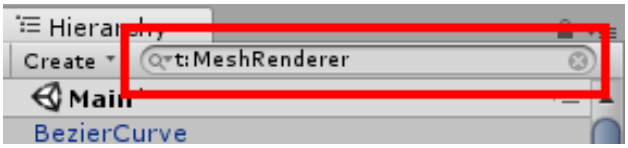
To get the aliens to chase the hero, you first need a NavMesh over the entire floor to designate it as a walkable area. To get started, click **Window\Navigation**.

Unity will display the Navigation window (it appears in the right panel where the Inspector usually is by default). Take note of the tabs:

- **Object**: Lists the GameObjects you've added to the NavMesh.
- **Bake**: Configures the NavMesh's properties.
- **Areas**: Defines an area mask, much like a physics layer.



Select the **Object** tab and in it, select the **Mesh Renderers** icon to apply a filter to the Hierarchy. Now it'll only display GameObjects that have renderers attached to them. You can see it in the search field:



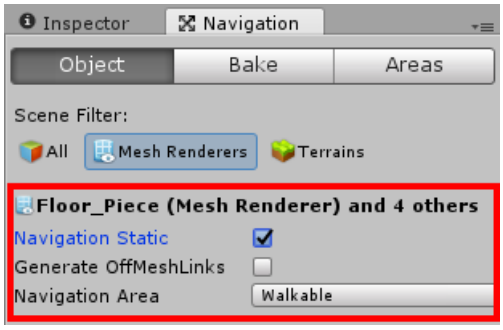
You can filter the Hierarchy by other types. To remove the filter, you can simply click the small clear button (but don't do that right now).

Press **Control** on PC or **Command** on Mac and select **Floor_Piece, Floor_Piece_001, Floor_Piece_002, Floor_Piece_003, Glass_Floor**.

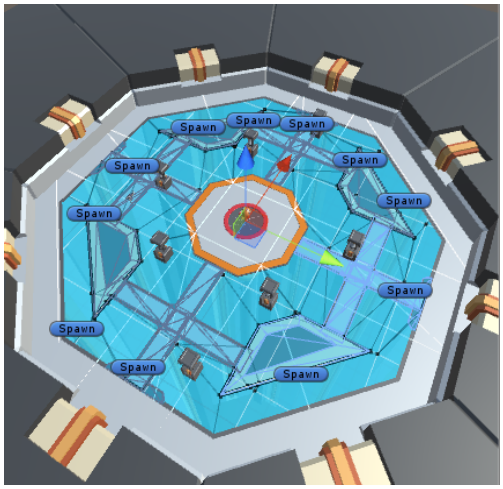
Some options will appear in the Navigation window. **Check** the **Navigation Static** checkbox to tell your selected meshes not to move.

Leave the **Generate OffMeshLinks** option **unchecked**. Otherwise, you'll get links between selected GameObjects.

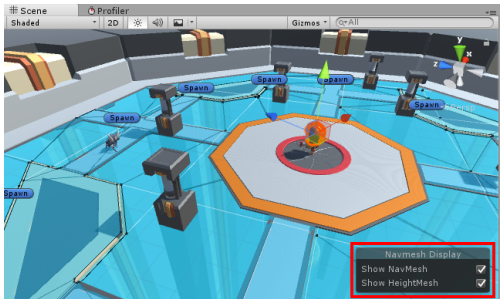
Finally, you'll see a Navigation Area option — this is the navigation mesh you're currently building. Right now, you're working on the **Walkable mask** but can switch to **Not Walkable** or **Jump**. Your custom masks also appear in this drop-down as you create them in the Areas tab.



Click **Bake** at the bottom of the Navigation view to save the mesh. Now you have a place for aliens to walk! After clicking the Bake button, the arena will turn a lighter blue. This indicates you've got a walkable NavMesh mask.

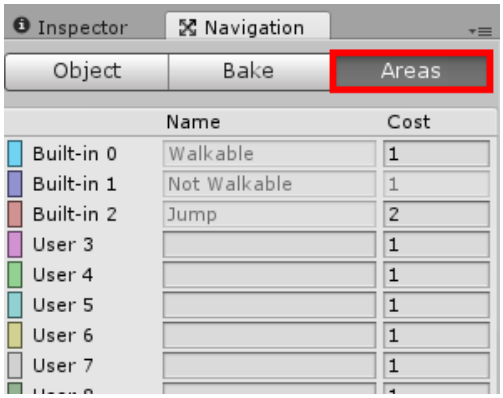


If you don't see the blue nav mesh, check the **Show NavMesh** checkbox in the Scene view.



If you selected a different navigation area, such as Jumpable, then the color would have been orange.

Click the Areas tab to see all the areas and colors that are available to you:



The pretty colors will "vanish" when you switch from the Navigation window, and you'll learn about the Bake tab in just a bit.

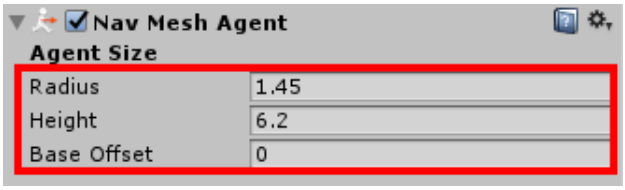
Clear the search in the Hierarchy at this point, as you are done with it.

Creating agents

You have the mesh, so now you need agents. Select the **Alien** in the Hierarchy (you should clear the Hierarchy filter to easily find this), and in the Inspector tab click the **Apply** button at the top. As previously mentioned, this is how you push changes to the GameObject back up to the corresponding prefab.

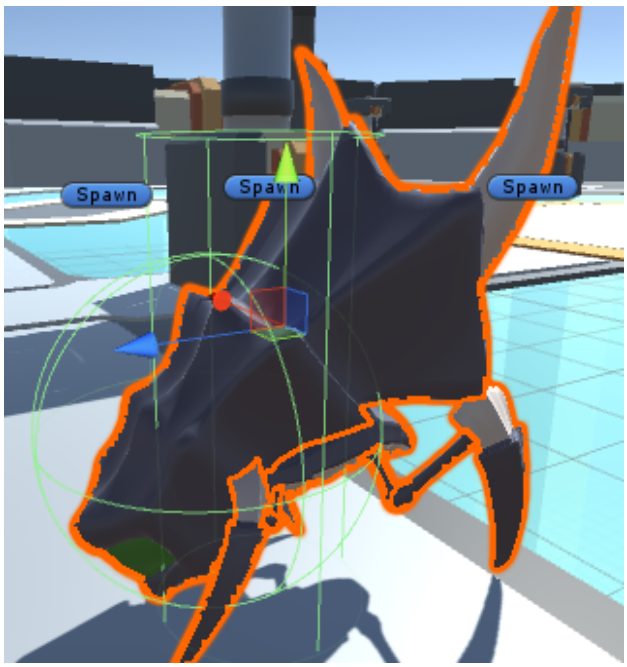
Select the **Alien** prefab in the Prefabs folder. Click **Add Component** then click **Nav Mesh Agent** in the **Navigation** category. You have many options here:

Start by configuring the size of the agent. In the Agent Size section, set **Radius** to `1.45`, **Height** to `6.2` and **Base Offset** to `0`.



Radius determines the collision distance between other agents. **Height** is the clearance to pass below an overhang, and **Base offset** is the difference between the center of the agent and the center of the GameObject.

You can see visual representations of these figures in the Scene view where a green cylinder defines the agent's bounds.



At this point, you've configured the NavMesh Agent, and this new component will drive your GameObject. However, there's a rigidbody attached. Since the rigidbody won't be controlled by physics, **check** the **Is Kinematic** checkbox.

Now you need to set the agent's goal. Click the **Add Component** button then click **New Script** and name it **Alien**. Open it in your code editor.

First you need access to the `NavMeshAgent` classes. Underneath the `using` statements, add the following:

```
using UnityEngine.AI;
```

Add the following variables underneath the opening class brace.

```
public Transform target;
private NavMeshAgent agent;
```

The `target` is where the alien should go — notice the variable name doesn't reference the hero. You don't always want the aliens to chase the hero down. For instance, the hero might throw a noise-maker to attract aliens to a certain area and then mow them down, or maybe he'll get a powerup that makes the aliens turn on each other.

By making the enemies' goal a target instead a certain character, you leave yourself open to more game design opportunities.

Next, add the following to `Start()`:

```
agent = GetComponent<NavMeshAgent>();
```

This gets a reference to the `NavMeshAgent` so you can access it in code. Now, in `Update()`, add the following code:

```
agent.destination = target.position;
```

Save the file and switch over to **GameManager.cs** to set the target for the alien.

Return to where you left off in `Update()`, and add the following code just after the line that sets `newAlien.transform.position`:

```
Alien alienScript = newAlien.GetComponent<Alien>();
```

This gets a reference to the `Alien` script. Now, add:

```
alienScript.target = player.transform;
```

This sets the target to the player's current position.

Finally, the alien should rotate towards the hero right after being spawned. Otherwise, it'll squander time by rotating instead of attacking. Add this:

```
Vector3 targetRotation = new Vector3(player.transform.position.x,
  newAlien.transform.position.y, player.transform.position.z);
newAlien.transform.LookAt(targetRotation);
```

The code rotates the alien towards the hero using the alien's y-position so that it doesn't look upwards and stares straight ahead.

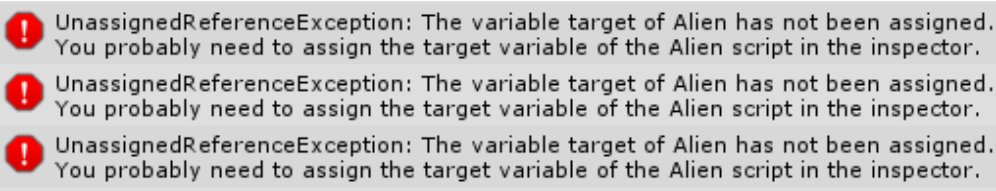
Save and play your game.

You'll notice a bunch of things, none of which will bring you immediate joy:

1. There are a ton of errors in the console.

2. The aliens move crazy slow.
3. The aliens walk through the columns.

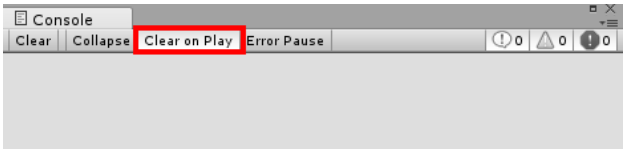
Start with the first issue. You should see the following in the console:



It states that no variable target was assigned to the alien, i.e. one of the aliens doesn't have a reference to the space marine. This is an easy fix. Open **Alien.cs** in your code editor, and then change `Update()` to match the following:

```
if (target != null) {
  agent.destination = target.position;
}
```

Now run your game and check if the console is clear. If not, make sure to click the **Clear On Play** option.



You successfully resolved the console errors, but you merely addressed the symptoms. Why did you get console errors in the first place?

Play the game again and watch the aliens carefully. You'll see all but one of them milling about. You dragged this non-moving alien into the Scene a while back — it wasn't created by the GameManager. Hence, it has no target and the GameManager doesn't know about it.

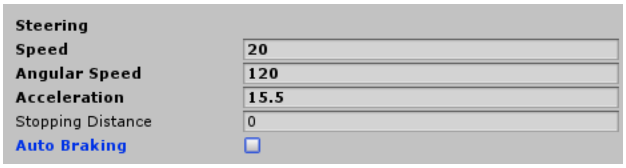
This seemingly small issue has potential to break the balance of your game. To fix this, find the alien in the hierarchy and delete it. Buh-bye little orphan alien!

Move on to the second issue: slow-moving aliens. Thankfully, this is an easy fix. It's part of the `NavMeshAgent` properties.

In the Project browser, select the **Alien** from the **Prefabs** folder. You'll see that the NavMeshAgent component has a section called **Steering**. Here's the breakdown:

- **Speed**: how fast the agent moves in world units per second. **Set** this to **20**.
- **Angular Speed**: determines rotation speed in angles per second. **Set** this to **120** (the default value).
- **Acceleration**: how fast the agent increases speed per second. **Set** this to **15.5**.

- **Stopping Distance**: how close the agent stops near its goal. **Set** this to **0** (the default value).
- **Auto Braking**: slows the agent as it approaches its goal. **Uncheck** this.



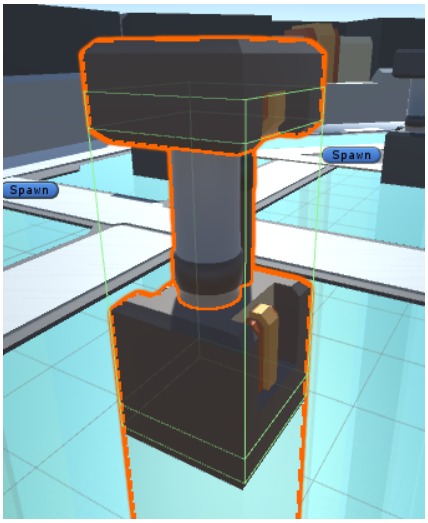
Underneath that section, you'll notice an **Obstacle Avoidance** section. **Quality** determines how the path will be defined. At the highest quality, the path will be determined according to obstacles and other agents. At the lowest quality — none — the path will ignore agents and merely try to avoid immediate collisions.

Since there are many agents in play, set the **Quality** to **Low Quality** to reduce CPU time for calculating the paths.

Play your game. Now the aliens will move with purpose.

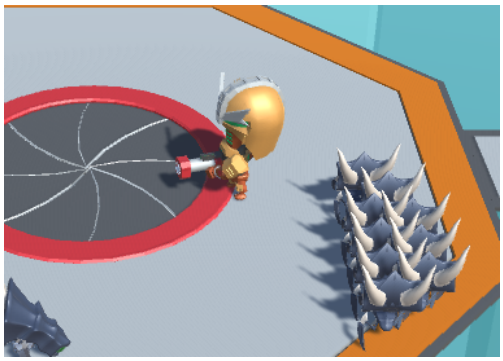
Now for the last problem: the little buggers can't just walk through the columns. In the Project browser, select the **BobbleArena-Column** in the Prefabs folder. Click the **Add Component** button. Select **Nav Mesh Obstacle** in the **Navigation** category.

This works just like a plain old collider. Set the **Shape** to **Box** then **Center** to (**0**, **2.3**, **0**) and **Size** to (**2.07**, **4.27**, **2**).



You'll notice your columns gain a green wireframe that indicates the `NavMeshObstacle`.

Play your game, but don't move this time.



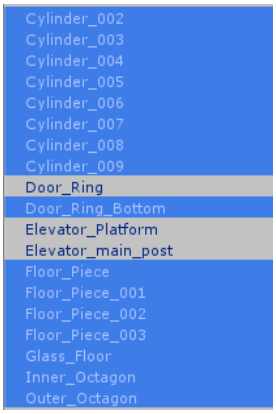
Although you resolved the last three problems, now you have a new one: a cluster of aliens in the wrong part of the map.

The root cause is that they're blocked by other renderers. You added only a few pieces of geometry whereas the bugs are getting stuck on the rest of them.

Open the **Navigation** window, and at the bottom of the window, click the **Clear** button.

This will reset the Walkable NavMesh. Add the following renderers to it:

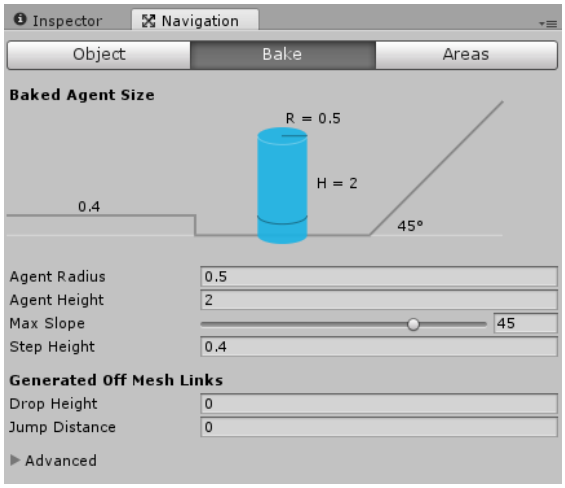
- **Cylinder_xxx**
- **Door_Ring_Bottom**
- **Floor_Piece_xxx**
- **Glass Floor**
- **Inner_Octagon**
- **Outer_Octagon**



Check the **Navigation Static** checkbox and click the **Bake** button — at the bottom, not in the Bake tab — and play your game.

Thankfully, the aliens can make it to the space marine, although some may get stuck. You're getting closer to perfection but not there yet!

The problem now is that the NavMesh's default properties expect smaller agents. In the Navigation window, click the **Bake tab** — not the Bake button.



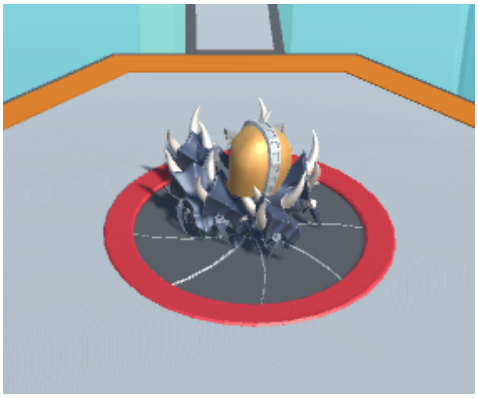
The Bake tab is where you configure the NavMesh. Here's a rundown of the most important properties and how to set them:

- **Agent Radius**: determines how close an agent can get to an obstacle. **Set** it to `1.45`.
- **Agent Height**: determines minimum height of a ceiling for the agent to pass beneath. **Set** it to `6.2`.
- **Max Slope**: determines the angle at which a slope becomes impassable for the agent. Leave as-is.

- **Step Height**: determines the minimum height at which a step becomes impassable for the agent. **Set** to `0.73`.

Where did you get those magic numbers? They are from your NavMesh agent's settings, i.e., your alien. Unity recommends you set the values to the **smallest** agent that's in use.

Click the **Bake** button. Play your game once it's fully baked. The space marine will promptly find that he's the center of attention.



Tuning the pathfinding

Although the pathfinding appears to work without any issue, you actually have a significant problem to solve. Every time you assign a destination to a NavMeshAgent, Unity goes through the process of a calculating a route to that destination.

It wouldn't be a big deal if it happened here and there, but consider that the game will spawn many aliens and perform this calculation each time. Review your implementation with that fact in mind. You'll see that you're calculating the pathfinding once per frame.

This isn't scalable — there will be more pathfinding calculations when aliens spawn. Eventually, your frame rate will tank and it'll kickoff a tragic chain of events for your player.

Open **Alien.cs** in your code editor. Under the public variables, add the following:

```
public float navigationUpdate;
private float navigationTime = 0;
```

`navigationUpdate` is the amount of time, in milliseconds, for when the alien should update its path. `navigationTime` is a private variable that tracks how much time has passed since the previous update.

In `Update()`, replace `agent.destination = target.position` with the following:

```
navigationTime += Time.deltaTime;
if (navigationTime > navigationUpdate) {
  agent.destination = target.position;
  navigationTime = 0;
}
```

This code checks to see if a certain amount of time has passed then updates the path.

Switch back to Unity. In the Project view, select the **Alien** prefab in the **Prefabs** folder. In the Alien script, set the **Navigation Update** field to `0.5`.

Now play your game. You'll notice the aliens are a little slower to dial in on the marine, but for the most part, they will get to him.

At this point, you'd want to play around with the variables that affect play and performance to find that balance point. Testing and profiling your game is the only way to find the right configuration!

Final touches

You have now hordes of aliens chasing a marine. There's just one thing left to be done — you need something to happen when the marine blows them to bits.

Switch back to **Alien.cs** in your code editor and add the following code before the last closing brace:

```
void OnTriggerEnter(Collider other) {
  Destroy(gameObject);
}
```

This code works like a typical collision code, except that it makes the collision event into a trigger.

Why a trigger? Remember, you set the Alien's rigidbody to Is Kinematic, so the rigidbody won't respond to collision events because the Navigation system is in control.

That said, you can still be informed when a rigidbody crosses a collider through trigger events. That way, you can still respond to them.

In this case, you call `Destroy()` and pass in the `gameObject`, which will delete the alien instance. Save the file and switch back to Unity.

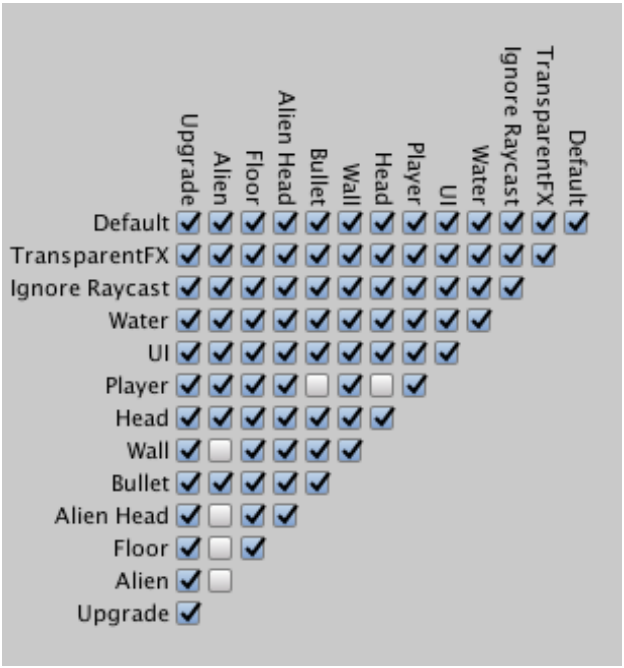
Select the **Alien** in the Prefabs folder, and in the Inspector, look for the **Sphere Collider**. **Check** the **Is Trigger** checkbox.

At this point, anything that collides with the alien will kill it — walls, columns, the marine, etc. You need to opt out of some collisions.

With the Alien prefab still selected, assign it to the **Alien layer**. When prompted to apply the layer to all child objects as well, select **No, this object only**. Next, in the Project browser, select the **BobbleArena-Column** in the Prefabs folder.

In the Layer drop-down, assign it to the **Wall** layer and, again, select **No, this object only** in the subsequent prompt dialog.

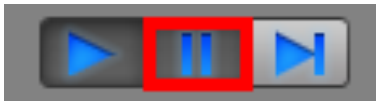
Next, click **Edit\Project Settings\Physics**. In the collision matrix, uncheck the following in the Alien column: **Wall, Alien Head, Floor, Alien.**



Play your game.

Now you can shoot Aliens to your heart's content, and when you collide with them, they'll despawn. In a later chapter, you'll give the aliens a more fitting death. But as it is now, you can get a good feel for your game. Take this moment to do a little more tuning.

Restart your game, and pause right as it starts.



Select the **GameManager** in the Hierarchy, and in the Inspector, set the following values:

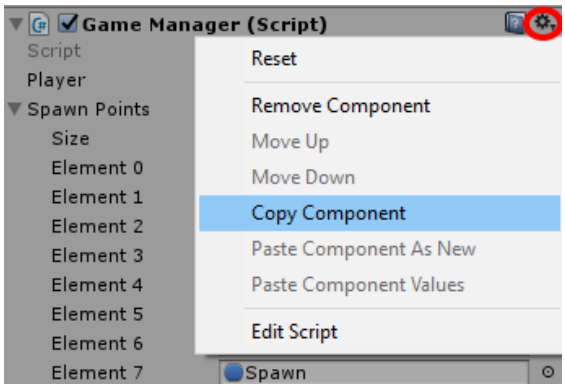
- **Max Aliens On Screen**: `50`

- **Total Aliens**: `50`
- **Min Spawn Time**: `0`
- **Max Spawn Time**: `3`
- **Aliens Per Spawn**: `3`

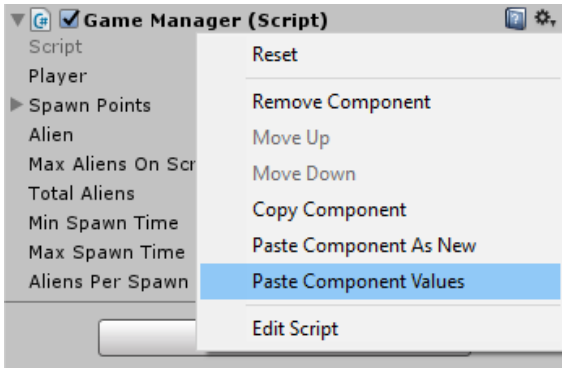
Now **unpause** the game, and prepare to be confronted by a lot of aliens. If you experience a slowdown, dial back the numbers until you find what works best for your system.

Once you find the right amount, **pause** the game one more time. **Do not stop the game**. You need to copy the GameManager values. You could write them down, but there's a better way.

Select **GameManager** in the Hierarchy (if it isn't already), and in the Inspector, **click the gear icon** over the Game Manager script component. From the drop-down, select **Copy Component**.



Now **stop** your game. Pay no attention to the fact that the Game Manager reverts the values. **Click the gear icon** again and select **Paste Component Values**.



The component will update with all the values you added during run time. Pretty awesome, eh? The downside is that you can only do this with one component at a time.

Where to go from here?

It's hard to believe that one little chapter on pathfinding could bring a relatively static game to life. As you worked through the process of pathfinding, you learned about:

- **Creating Managers**, such as a Game Manager to manage the gameplay and allow you to tweak settings in real time.
- **NavMeshes** and how they define movable areas for GameObjects.
- **NavAgents** and how to build them and give them targets.
- **NavObstacles** to have your agents avoid various parts of your level geometry.

Not surprisingly, there's much more to learn. Have you noticed that everybody moves without moving their legs? Curious. In the next chapter, you'll give your models a little spring in their step and learn all about Unity's animation system in the process.

Chapter 6: Animation

By Brian Moakley

At this point, you've learned how to move GameObjects in Unity in a variety of ways, including:

1. **Writing scripts to manually change the position**. You learned this in Chapter 3, "Components", when you wrote scripts to move the player, camera, and bullets.
2. **Using the Character Controller**. You learned this in Chapter 4, "Physics", when you replaced the player movement script with the built-in Character Controller, which has powerful functionality and is often easier than writing movement scripts on your own.
3. **Letting the Unity physics engine move objects for you**. You learned this in Chapter 4, "Physics" when configured the space marine's bobblehead by attaching it to the body with a joint and periodically applying a force to make it bobble.
4. **Employing the Navigation System to move objects on your behalf**. You learned this in Chapter 5, "Pathfinding", when you configured the aliens as a NavMesh agent and gave it a target so they could pathfind their way to the space marine.

There's another important way to move GameObjects in Unity that you haven't explored yet: the animation system, otherwise known as Mecanim.

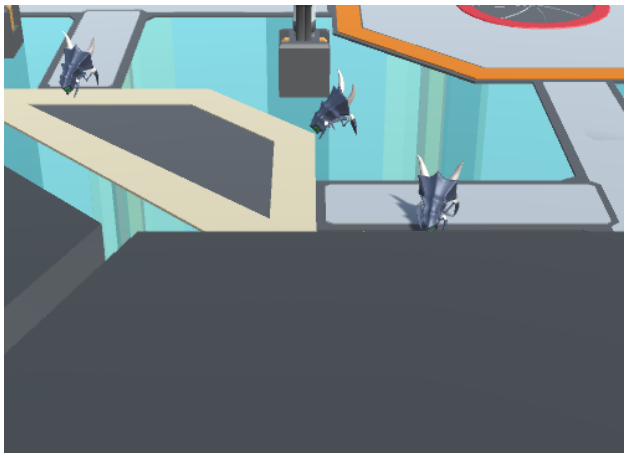
Mecanim is a powerful system that allows you to animate models as well as *every single exposed property* in Unity.

That's right, if you can set a value in the Inspector, you can animate that value over time! Powerful stuff.

Getting started

The first animation that you will create is a simple animation to make the arena walls go up and down.

This is because currently there's an issue. As the space marine moves towards the camera, the wall blocks your view of him. This is *bad* mojo; not can you not see what you're shooting, but the spawning aliens can attack instantly.



Causing people to lose because their view was blocked is no way to win gamers' hearts and minds!

One technique to solve the problem is to make the wall transparent, but that doesn't solve the problem of the partially obscured play area. For slow-paced games, it wouldn't be a serious problem, but fast reaction times are required to win Bobblehead Wars.

To solve the problem, you'll lower the walls of the arena when the player gets close and raise them when the player walks away.

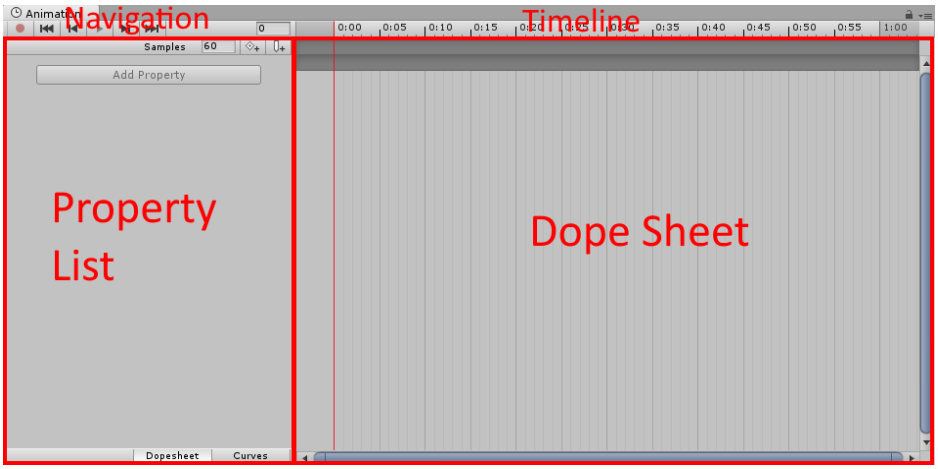
The animation window

Open your project from the last chapter, or if you're reading this chapter out of order, feel free to open the starter project that's located in the resources for this chapter.

You're going to animate various child GameObjects of the BobbleArena. You could animate them individually, but this technique will have you grouping child GameObjects together, which is far more efficient.

In the Hierarchy, select the **BobbleArena** GameObject. Click **Window\Animation** to open the Animation window.

The window has several sections, and you'll spend quite a bit of time here in this chapter.



On the left, you have a **Property List** where you select the properties you intend to animate. On the right, you have the **Dope Sheet** where you create **keyframes** - more on this in a moment.

Above the dope sheet is the **Timeline** where you scrub back and forth on the animation to preview how it looks. Finally, above the property list, you'll see **Frame Navigation** where you control playback and add animation events.

Animation events allow you to run code at certain frames in an animation.

Introducing keyframe animations

Unity animates objects through the use of **keyframe animation**.

In keyframe animation, you set up a series of (time, value) pairs to indicate how you want an object to animate over time. You don't need to specify every single second; the beauty is the game engine will automatically calculate the values in-between your **key frames**.

For example, say have a cube at `(0, 0, 0)` that you want to move to `(0, 10, 0)` two seconds later. Then you would set up two keyframes:

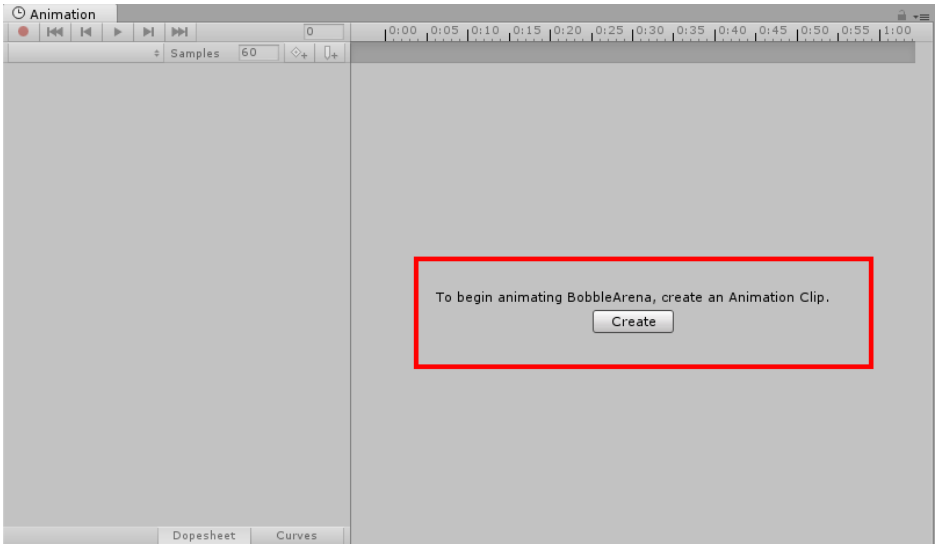
- Time 0 seconds: Position `(0, 0, 0)`
- Time 2 seconds: Posttion `(0, 10, 0)`

Where would the cube be at time 1 second? Unity would automatically calculate this for you through interpolation - for example `(0, 5, 0)`.

Keyframe animation makes creating animations easy. You just have to specify the important points - again, the key frames, and Unity takes care of the rest!

Your first animation

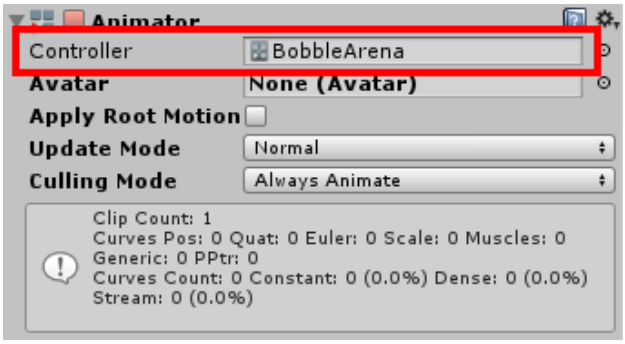
In the dope sheet, click the **Create** button.



Give it the name **Walls.anim** and save it in the Animations folder. While you were busy saving, Unity was busy behind the scenes:

- Created a new animation clip
- Created a new animation controller

In the Hierarchy, select **BobbleArena** and see it for yourself in the Inspector.

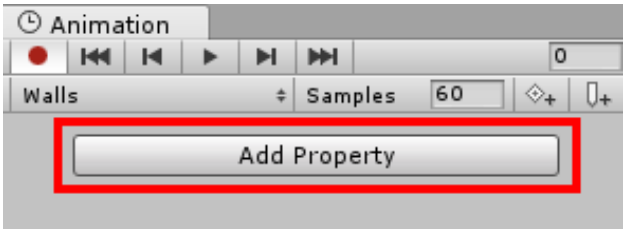


That new animator controller was assigned to the `Controller` property. Note that Unity only does these things when you create a *new animation* on a GameObject *without an animator controller*.

Unity attaches an animator controller on the GameObject but it also creates an animator controller file in the same folder as your animation clips. This file keeps track of all your states which you'll learn in just a bit.

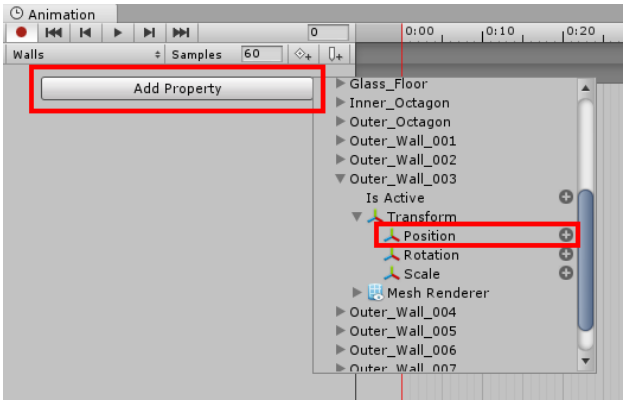
Now to the part where you actually make the animation!

Click the **Add Property** button in the Animation window.

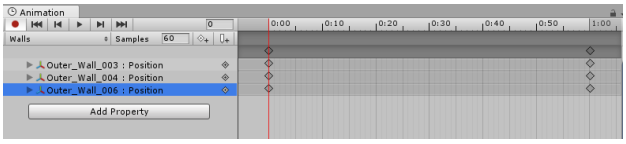


You're in the place where you select the GameObject or property that you wish to animate. You'll see each component listed as well as the child GameObjects.

Expand the **Floor** disclosure triangle then expand the **Outer_Wall_003** GameObject. Next, expand the **Transform** component and click the **plus sign** next to the **Position** property.

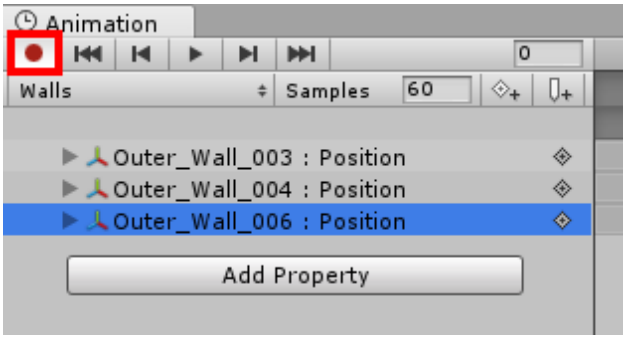


Do the same thing for **Outer_Wall_004** and **Outer_Wall_006**.



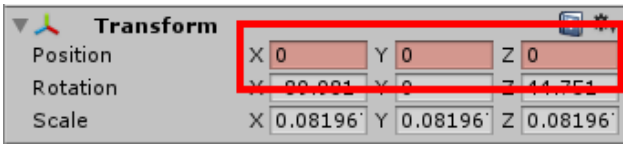
At this point, you're technically ready to start recording the animation. Do you feel ready? Huh? Do you?

Good! Click the little red record button in the Animation window with authority.



By pressing the button, you've also caused the Unity's player controls to have turned red. It's go time now!

If you select one of the GameObjects that you're animating, you'll also notice that its position properties turn red.

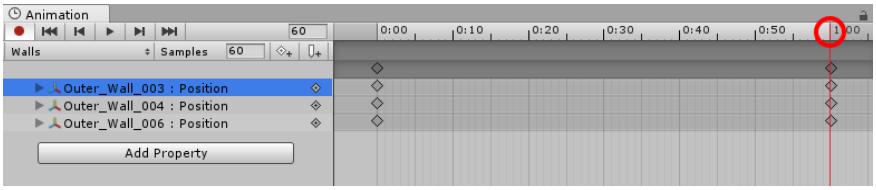


These are all visual reminders that you're currently animating. It's surprisingly easy to forget that you're in animation mode.

Many developers suffer from this acute condition, aptly named *animanesia*. It's what happens when you're working in animation mode but start making changes to your game; you get deep into it then realize — in one terrible flash — that you've just made a bunch of useless animation clips.

The unpleasant discovery usually triggers a storm of curses to flow from your mouth, which is then followed by begrudgingly redoing all your work.

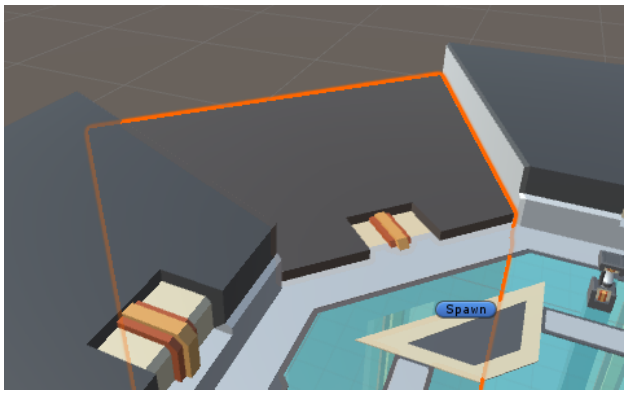
Go to the Dope Sheet and **Click the timeline** above the keyframes at the 1:00 mark.



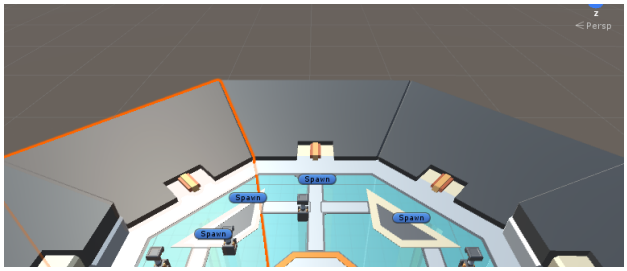
Since you are in animation mode, you can think of this as "time traveling" to what the animation will look like at the 1:00 mark. You can see what it will look like in the Unity editor, and modify the values.

Note that if you don't move the scrubber exactly over the keyframes at the 1:00 mark, Unity will create new keyframes at the scrubber's current position.

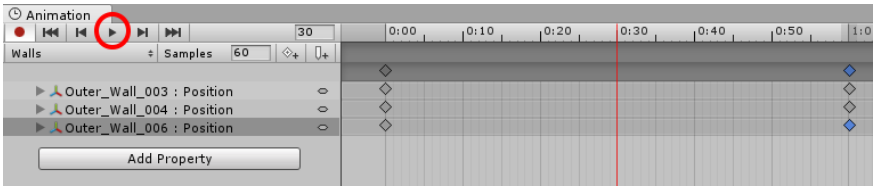
Make sure that just **Outer_Wall_003** is selected, and in the Inspector, set the **y position** to `-10.78`. In the Scene view, you'll see the wall drop down.



Do the same for **Outer_Wall_004** and **Outer_Wall_006**. The walls should look as follows:



Now, **click** the animation **play** button (in the animation window, not the normal play button). Just like that, you have moving walls!



Hit the animation **play** button again to stop the looping animation. Note you can also move the scrubber on the timeline to see the interpolated values at any point.

When you finish, **click** the **recording button** to stop recording mode.

Animation states

As you start creating animations, you'll accrue many clips. Some GameObjects are going to have many animations, so you'll need a way to control which animations run when. It's possible to do this via through code, but why would you go to the hassle when Unity provides a tool that makes this easy?

Meet the **Animator**.

You can think of the animator as the director in a movie. It runs around giving orders of which animations to play when.

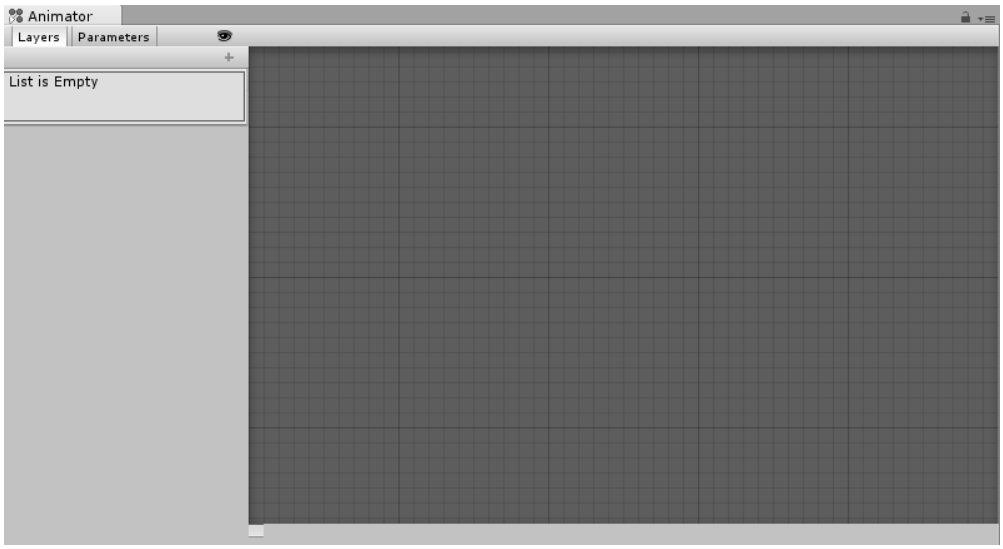
For instance, when a character stands idle, the animator could order an animation to move the head and portray breathing and fidgeting. When a character's horizontal speed increases, the animator could order a walking animation; once speed surpasses a defined threshold, the animator could order a running animation.

With the animator, you define the conditions and their animation relationships. Then, in essence, the game takes care of itself.

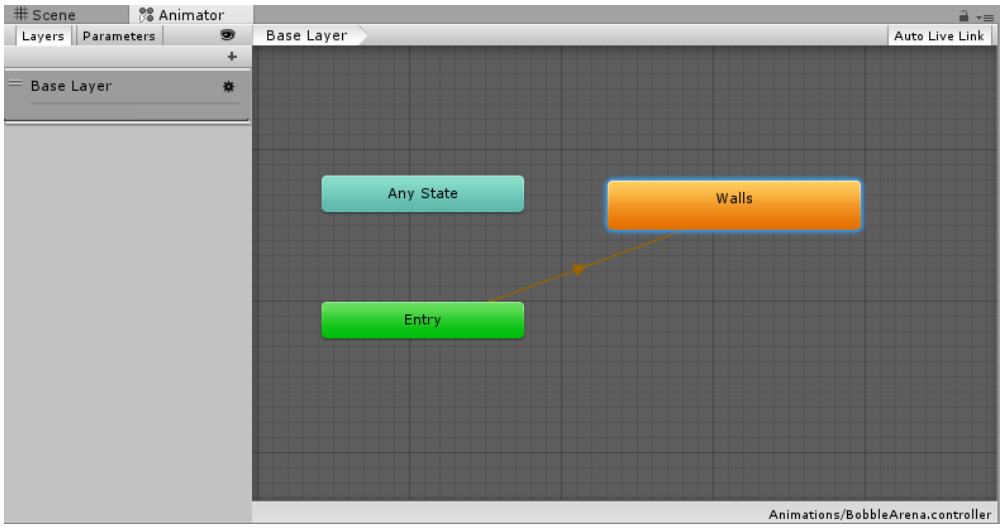
Click **Window\Animator** to open the Animator window.

> **Note:** When you first start working with Unity, you may confuse the *Animation* and *Animator* windows.
>
> Try a little memory trick: the anima-*tion* window is where you do crea-*tion*. The animat-*or* window is where you're an organiz-*er*.

You'll probably see an empty window because you didn't select a GameObject that has an animator component attached to it.



In the Hierarchy, click **BobbleArena** and check if you see the following in the Animator:



You'll see the `Walls` animation is already present in the controller. Any new animation clips are added the current animator controller automatically.

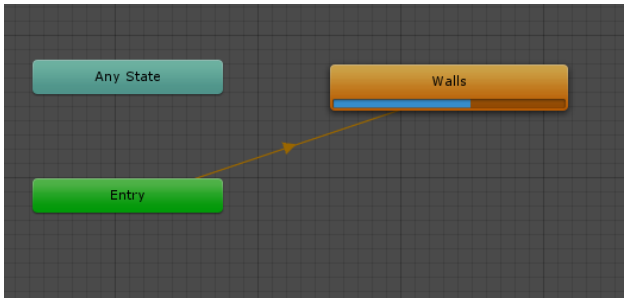
Each rectangle indicates an animation state:

- The green rectangle is the entry point of the animator.
- The orange rectangle indicates the default state; it's always played first.
- The arrows indicate transitions between states.

Play your game and move the hero until you can see the lower walls. The animation loops endlessly:

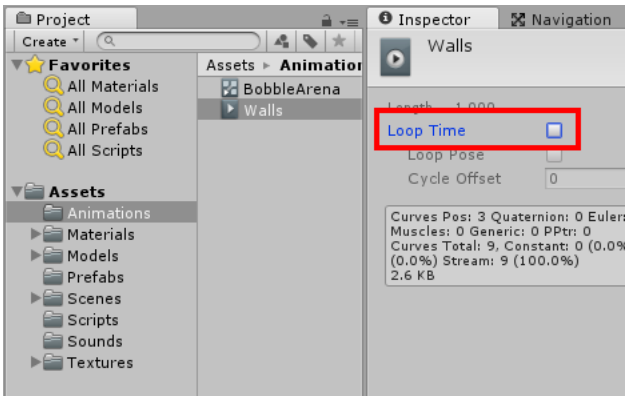
The `Walls` animation plays because it's the designated default state. It loops because that's the default behavior.

Watch the animator as it plays to see the animation's progress.



In the Project browser, click the **Animations** folder and **select** the **Walls** animation. In the Inspector, **uncheck Loop Time**.

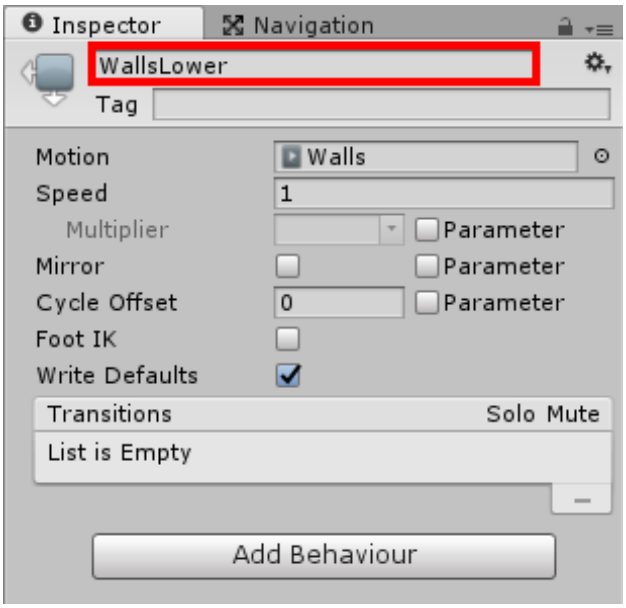
That's how you override the default — no loop for you!



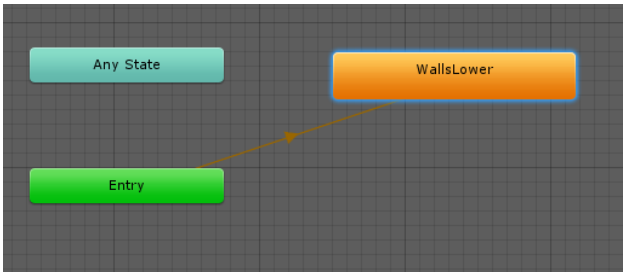
Unfortunately, "Walls" isn't a very descriptive name. Click the **Walls** state in the Animator and move your eyes to the Inspector to see all the available properties.

You'll learn and change a bunch of these throughout this book.

For now, change the name from **Walls** to **WallsLower**. That's it in here.

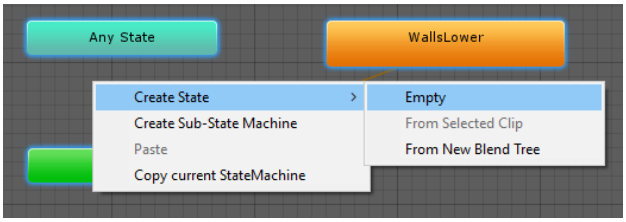


Notice that the state reflects the updated name.

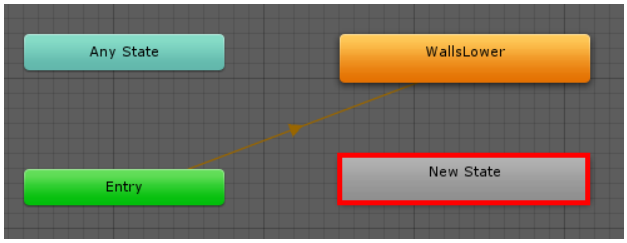


Being that you have a `WallsLower` state, you need a `WallsRaise` state. You could create a new animation, but since this animation is very similar to the `WallsLower` animation (just reversed), I'll show you a shortcut.

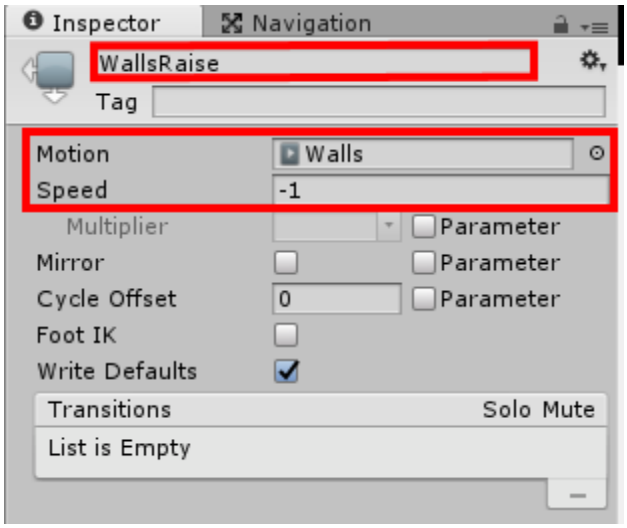
Right-click in the Animator, select **Create State** and then **Empty**.



You'll now have a new empty state.



Select this new state and name it **WallsRaise** in the Inspector. Next, drag the **Walls** animation from the Animations folder to the **Motion** property. Finally, set the **speed** to –1.

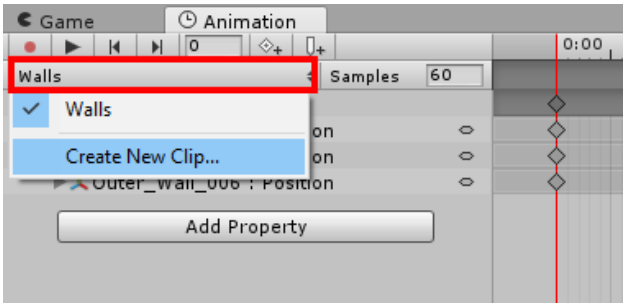


The `Motion` property references the actual animation clip. In this case, you're using the same animation clip that you used in `WallsLower`. The `Speed` property lets you designate how fast the clip will play. Setting the speed to `1` indicates that it will play at normal speed. Higher numbers make the clip run slower, whereas lower numbers make it run faster.

Why -1? Negative numbers play the clip in reverse. In this case, you're using the same clip, but playing it in reverse at normal speed.

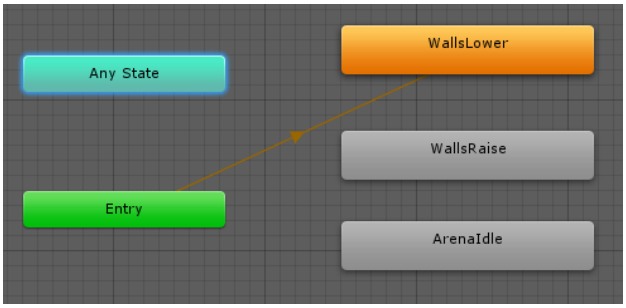
You need one more state: an idle state. Open the **Animation** window, making sure the **BobbleArena** is selected in the Hierarchy. You'll see the name of the currently displayed animation in the property list, which should be walls.

Click **Walls** and select **Create New Clip...** from the drop-down.



Give it the name **ArenaIdle.anim** and make sure to **stop recording**. That's right — your animation clip comprises an empty animation. Although you can create a state with no associated animation, Unity will do weird things that might ruin your plans. At times, it helps to play it safe and make an animation clip for the idle state, even if it's empty.

Your animator should look something like this:



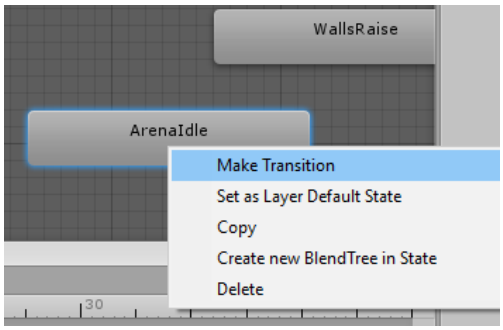
You need to organize it, and you'll do that through transitions.

Animation state transitions

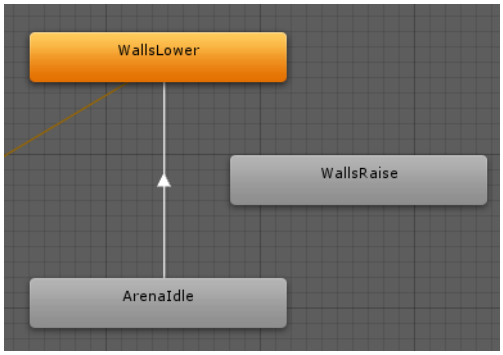
Transitions perform a variety of functions. They let you know the flow of animations, but more importantly, you can gate them according to certain conditions.

The first part of a transition is a condition, so that's the next thing you'll set up.

In the Animator window, **right-click** the **ArenaIdle** state and select **Make Transition**.

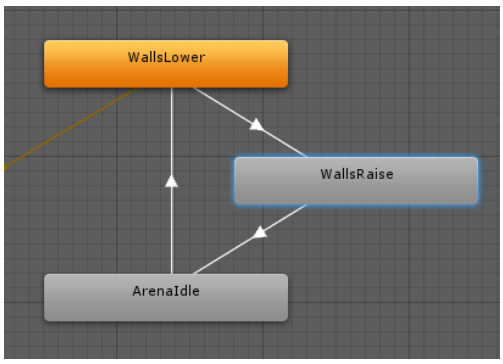


A line will now follow your mouse cursor. Click the **WallsLower** state to create that **new transition**.



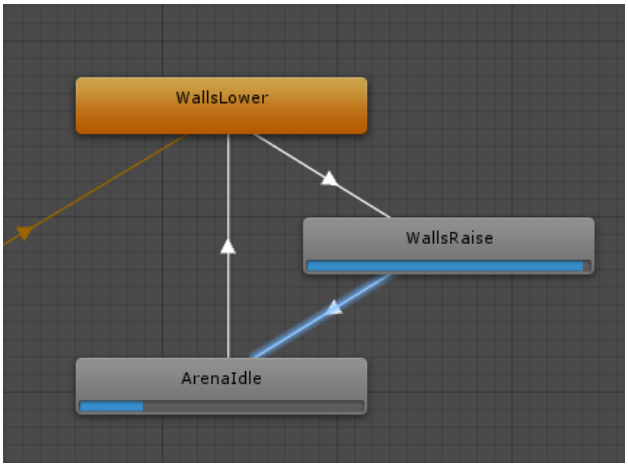
> **Note**: You might find that a state is out of view. By holding down the middle mouse button, you can pan around inside the animator and find it.

Create a **new transition** from **WallsLower** to **WallsRaise**, and then another from **WallsRaise** to **ArenaIdle**. Your states should look like this:



This depicts the entire animation flow.

Play your game while watching the animator.



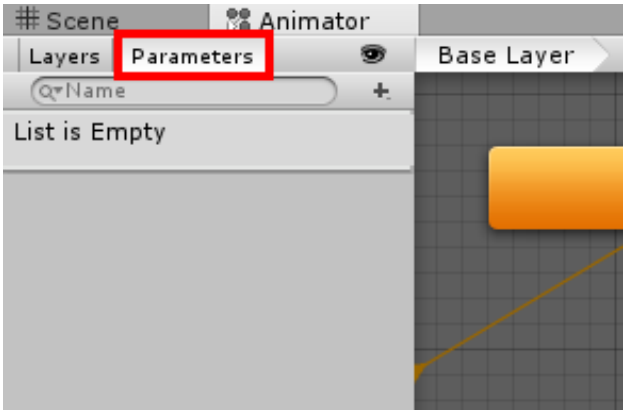
Select BobbleArena in the Hierarchy if you don't see any motion.

You'll notice the animation moves in an endless loop and that some states start before others finish; the Animator is blending the states.

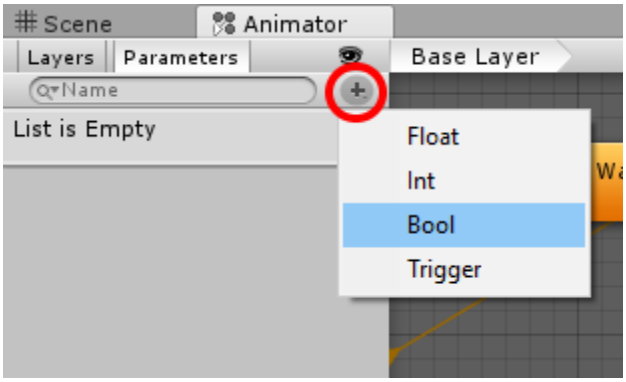
Animation state transition conditions

Looping isn't ideal in this case; you need a condition to prevent it. Also, blending is helpful, but you might want to make it more or less subtle. In some cases, you may not want it at all.

Go to the animator window and **click** the **Parameters** tab. It will currently read *List is Empty* because there are no parameters.



Click the **plus sign** and select **Bool** from the drop-down.



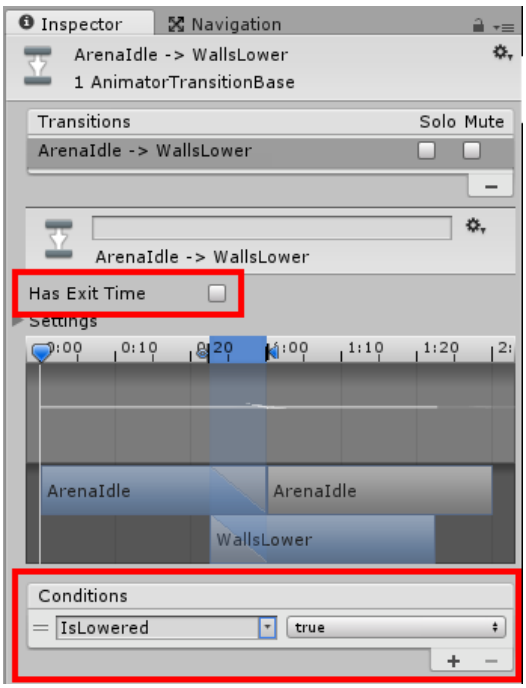
Give it the name **IsLowered**.

Although you chose **Bool** in this case, note that the menu has 4 options to choose from: **Float**, **Int**, **Bool**, and **Trigger**. Except for the Trigger, all of them correspond to the types used in C# . The `trigger` is a distinctive Unity condition type that behaves like a `bool`, except it resets itself after an animation.

Select the **transition arrow** between `ArenaIdle` and `WallsLower` to see your blending options in the Inspector.

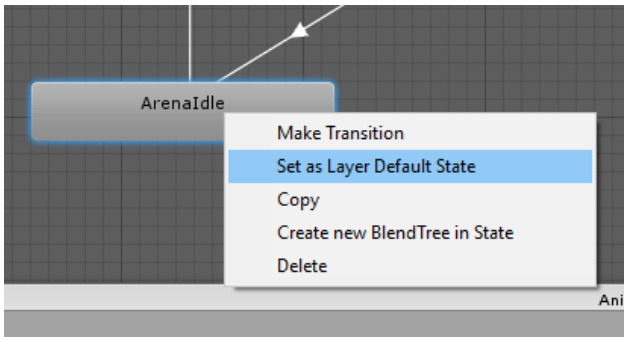
Uncheck Has Exit Time to tell Unity you want the animation to conclude immediately instead of waiting until the end of the animation. Otherwise, Unity will take additional time to blend the animation with the next animation.

In the Conditions section, **click** the **plus sign**. Since you have only one parameter, the `IsLowered` property will populate the field; set its condition to **true**.

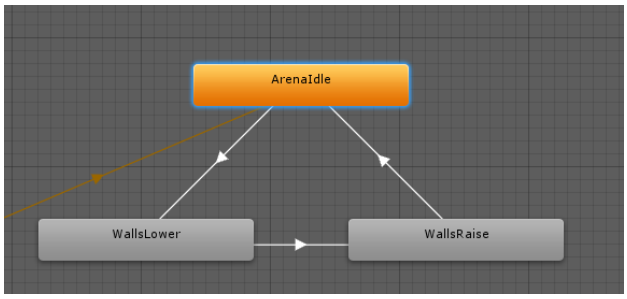


The reason you did this is because you will set the `IsLowered` boolean in code when you want the animation to trigger. By setting the condition that `IsLowered` must be true for this animation to run, once you set it in code the animation will play.

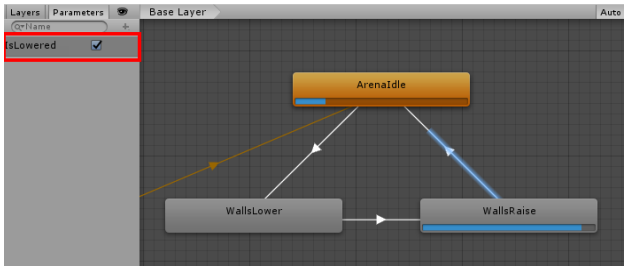
You need to make one more small adjustment before playing the game. In the Animator window, **right-click ArenaIdle** and elect **Set as Layer Default State** from the options.



This change indicates that `ArenaIdle` is the entry state for the animator controller. To make the relationships easier to understand at a glance, organize them like so:



Play your game again while keeping the Animator open. You'll see the wall doesn't drop because the condition hasn't been met. With the game still playing, **check IsLowered** in the Animator.



Now the wall lowers, raises and loops unless you uncheck IsLowered.

You need one more condition.

Stop your game and select the **transition arrow** from **WallsLower** to **WallsRaise**. In the Inspector, **uncheck Has Exit Time**. Click the **plus sign** in the Conditions section and set **IsLowered** to **false**.

Play it again and **click** the **IsLowered** checkbox. This time, the walls will lower, but they stop there due to the new condition. **Uncheck IsLowered** to raise the walls again.

You now have animations in place. Time to call them in code!

Triggering animations in code

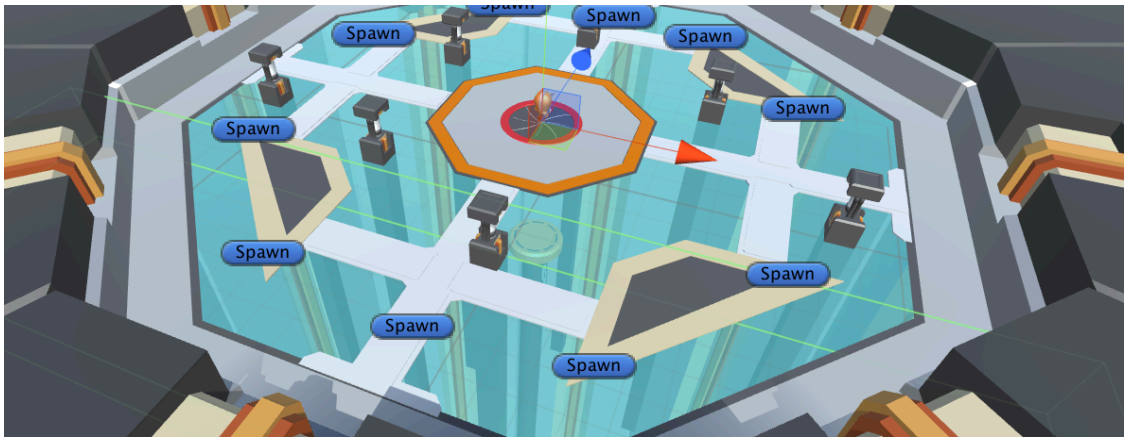
At this point, you have animation clips, animation states and conditions in place to activate your animations.

Next, you'll use a collider trigger to put everything together in code.

Select **BobbleArena** in the Hierarchy then click the **Create** button. From the drop-down, select **Create Empty Child** and name it **Wall Trigger**. Set **Position** to (**0**, **0**, **0**) and **Scale** to (**0.5**, **0.5**, **0.5**). Next, click the **Add Component button** and from the **Physics** category, select **Box Collider**.

In the Box Collider component, **check** the **Is Trigger** checkbox, set Center to (**0**, **6.72**, **-90.6**) and **Size** to (**250.55**, **17.14**, **66.7**).

You should see a large trigger in the Scene view toward the back walls. When the space marine walks into this area, you want to lower the walls.



To do this, select the **WallTrigger** then click the **Add Component** button and select **New Script**. Name it **ArenaWall**, make sure **language** is **C Sharp** and click the **Create and Add** button.

Double-click the new script to edit it, and add this variable inside the class:

```
  private Animator arenaAnimator;
```

You could make it a public variable so you can set this in the inspector, but this isn't necessary.

This is because `WallTrigger` is a child of the `BobbleArena`, so you can get a reference to the animator by accessing the `WallTrigger`'s `parent` property, and gettng the appropriate component.

To do this, add this to `Start()`:

```
GameObject arena = transform.parent.gameObject;
arenaAnimator = arena.GetComponent<Animator>();
```

First, this gets the parent GameObject by accessing the `parent` property on the transform. Once it has a reference to the `arena` GameObject, it calls `GetComponent()` for a reference to the animator.

Again, the above script works because it assumes the caller is a child of `BobbleArena`. It would fail if the caller were moved to a different GameObject.

Next add the following:

```
void OnTriggerEnter(Collider other) {
  arenaAnimator.SetBool("IsLowered", true);
}
```

When the trigger is activated, the code sets `IsLowered` to `true`. You do this by calling `SetBool()`.

> **Note:** There are similar methods for other types. For example, for a `float`, you'd use `SetFloat()`.

Now you need logic to raise the walls. Add the following:

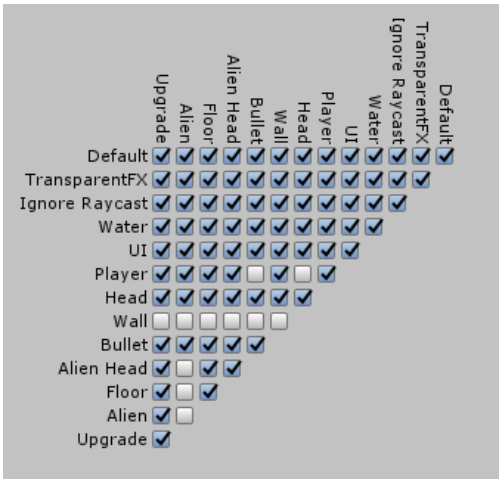
```
void OnTriggerExit(Collider other) {
  arenaAnimator.SetBool("IsLowered", false);
}
```

When the hero leaves the trigger, this code tells the animator to set `IsLowered` to `false` to raise the walls.

There's one outstanding detail for this script. Currently, any GameObject that enters the trigger lowers the walls. You need a filter so it only applies to the space marine.

Save the file and go back to Unity. With **WallTrigger** still selected, assign it the **Wall** layer. Navigate to **Edit\Project Settings\Physics**.

Uncheck all the checkboxes in the **Wall** row.



Play your game. Just as you intended, the walls lower when the marine walks down the screen, allowing you to see the action. How's that for convenience?

Animating models

At some point, you may want to add animations for your models. Unity provides this power; unfortunately, it's not always the right tool for the job. As mentioned in Chapter 1, "Hello, Unity", Unity is more of an integration tool than it is a creation tool.

Although Unity shares many features with popular 3D modeling and animation programs, it's not well-equipped to handle advanced tasks. Many developers prefer to create animations in an external program then import them into Unity.

That said, making simple animations is fine. After all, animating a custom model is no different from what you just did with the walls.

Take the alien for an example. Like the space marine, it has a large head that's begging to bobble. For learning purposes, rather than create this effect with physics, you'll use animation to create the effect.

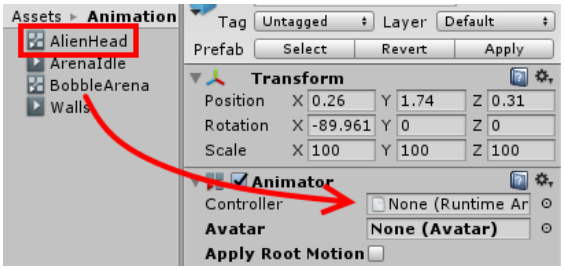
So you can easily preview the animation as you work on it, select the **Alien** in the Hierarchy, or drag an instance of the **Alien** into the Scene view if it's not there.

Last time you created an animation, Unity made the animator controller. This time, you'll do that for yourself.

Expand the Alien in the Hierarchy to reveal its child GameObjects. Select **BobbleEnemy-Head** and click the **Add Component** button. Select **Animator** from the **Miscellaneous** section.

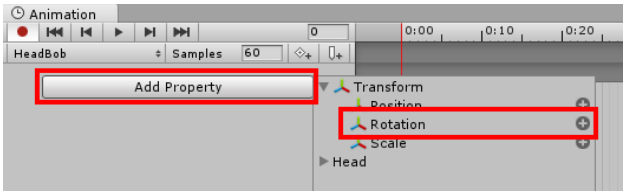
In the Project view, select the **Animations** folder and click the **Create** button. From the drop-down, select the **Animator Controller** and name it **AlienHead**.

Drag the **AlienHead** animator controller to the **Controller** property of the animator component.

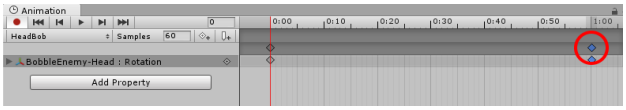


Open the animations window, make sure you've selected BobbleEnemy-Head and click the **Create** button. Call it **HeadBob.anim** and save it in the **Animations** folder.

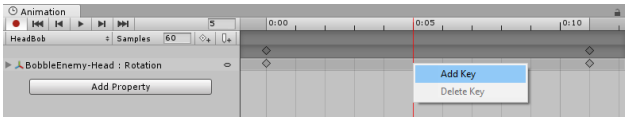
Click the **Add Property** button. From the flyout menu, expand the Transform component and click the **plus sign** next to **Rotation**.



You'll use three keyframes in this animation. But first, you need to make the animation smaller. Select the **last keyframe** and drag it to **frame 0:10**.



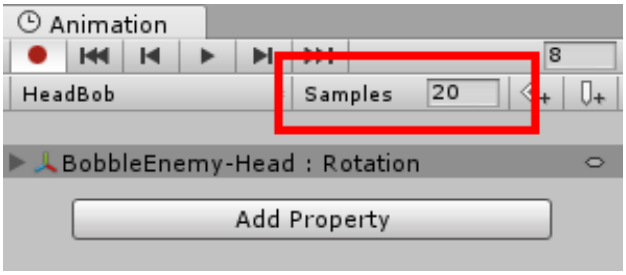
Next, move the **red scrubber** to **frame 0:5** then **right-click** and select **Add Key**.



With the middle keyframe still selected, go to the Inspector. Set **Rotation** to (**-61**, **0**, **0**) — now play the animation.

Watch your alien do a little head banging. You'll notice that changing one middle keyframe value resulted in a difference that's causing the head to move up and down.

You'll notice that the animation runs somewhat fast. In the **Samples** field, set the value to `20`. For perspective, keep in mind that the last animation you made ran at the default 60 frames per second — this one will run slower.



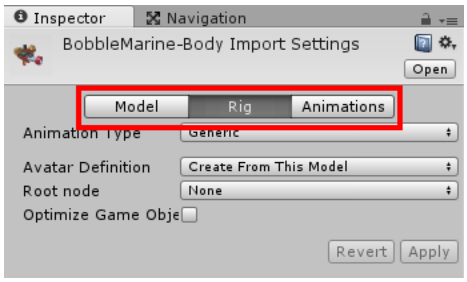
Click the **red record button** in the animation window to stop recording mode. In the Inspector, **apply the changes** to the prefab so all the instances bobble, and delete the temporary alien from the hierarchy.

Play the game again to see if the aliens all belong to headbanging club.

Imported animations

Unity provides a number of tools for animating GameObjects, and as you know, you can import animations with models. In fact, you've already imported some with the space marine and alien! All you need to do is incorporate them into your game, which is pretty easy.

In the Project view, select the **BobbleEnemy-Body** GameObject from the Models folder. You'll see three tabs in the Inspector: **Model**, **Rig** and **Animations**.



The **Model** and **Rig** tabs deal with the actual model structure.

Avatars and Rigs

Typically, models are **rigged**, which means Unity creates a system that acts as a virtual skeleton to helps your models move naturally.

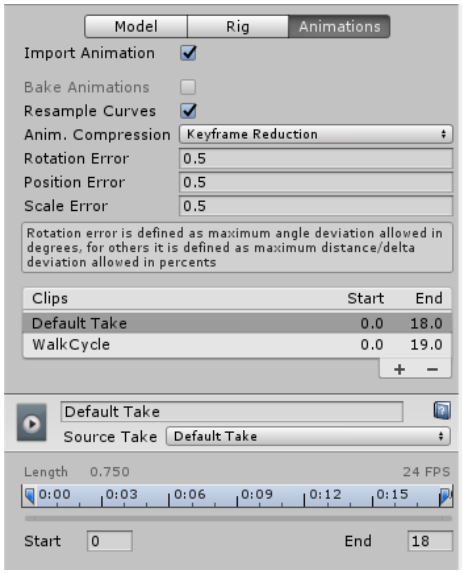
For instance, when you move an arm of a model, all the child GameObjects that comprise the arm will move in accordance with the rig's limitations as well as any that you dictate.

Rigs in Unity are called **avatars** and they can share animations. Unity provides a library of free motion capture animations, and with a proper rig, you can incorporate these motions with minimal work.

The **Animations** tab is where you configure your imported animations. You're about to get a crash course in all this...starting now!

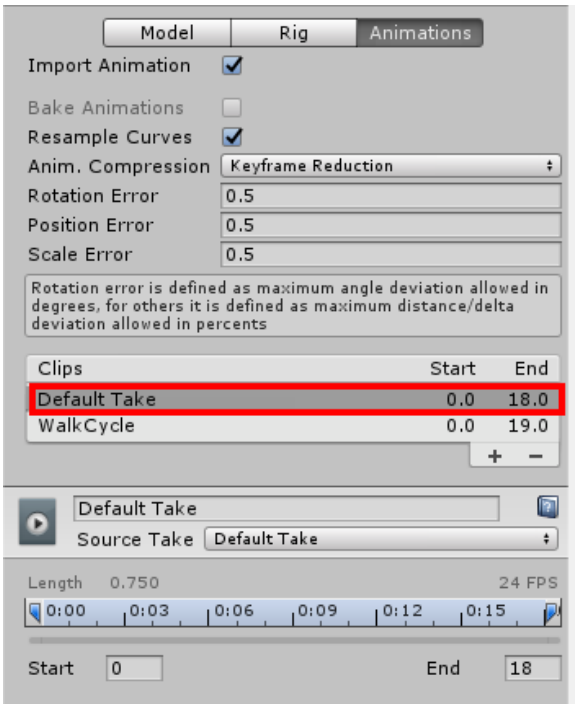
Animating the Alien

Click the **Animations** tab. Here's the animations you imported with the model.



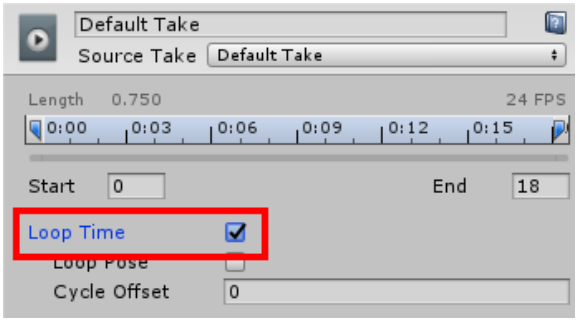
You have many options. The first deals with compressing the animation and addressing any resulting errors. Under the Clips section are your imported animation clips.

Select the **WalkCycle** animation.

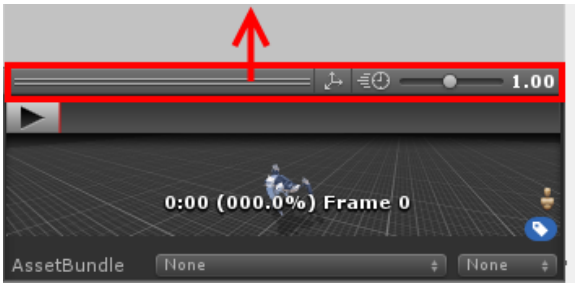


Beneath the animation are all tools for tweaking it. The timing bar allows you to edit the length of animation; a common use case is making a seamless loop effect.

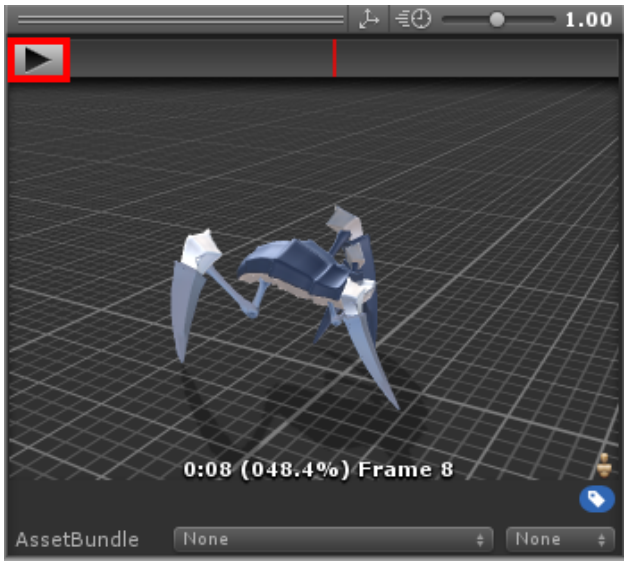
Check the box for **Loop Time**.



Look a little further down to find the preview pane. If you don't see a preview, look for a black bar at the bottom and drag it upwards.



Click the **play** button and the model will show the selected animation.



Once you're satisfied with the preview you'll need to save it. At the bottom of the Inspector, just above the preview window, click the **Apply** button.

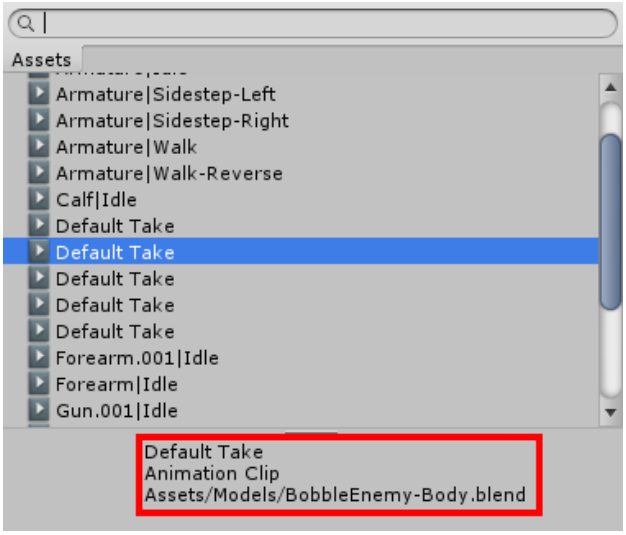
In the Project browser, select the **Animations** folder and click the **Create** button. From the drop-down, select **Animator Controller** and call it **AlienBody**. With the animation controller selected in the Project browser, open the **Animator window**.

Next, **right-click** in the Animator window and select **Create State\Empty**.

Select the new empty and name it **Walking** in the Inspector. Next, **click** the circle beside the **Motion** property.

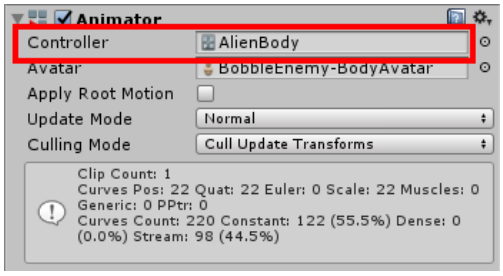
You're looking for **Default Take** but, in the list of animations, there are many clones. Select the **top-most Default Take**. Check the footer to see which model owns the animation.

Click through the Default Takes to find the one that belongs to **Assets/Models/BobbleEnemy-Body.blend**. Double-click it to select it.

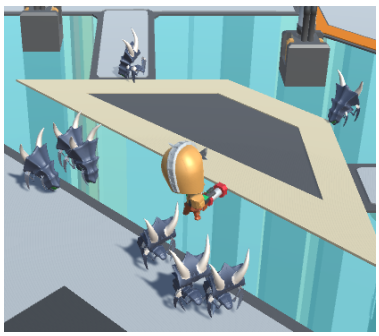


Go to the Prefabs folder in the Project browser and expand **Alien's** disclosure triangle. Select **BobbleEnemy-Body**. You'll see that it already has an animator component attached.

In the Project browser, drag the **AlienBody** animation controller to the **Controller** property in the Inspector.



Play your game. Congrats! You officially have creepy-crawly aliens!



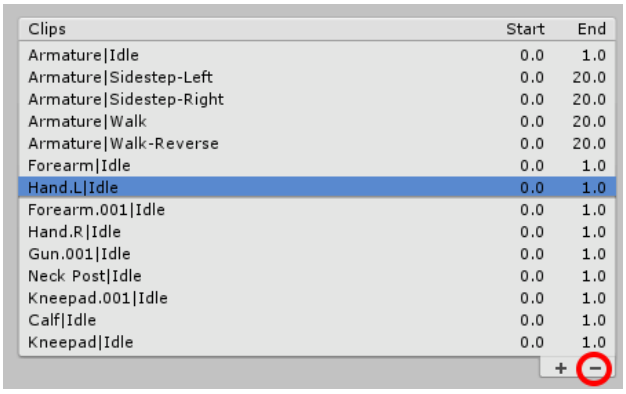
Animating the space marine

Now that the aliens mobilized, it's time to summon the space marine.

In the **Models** folder in the Project view, select the **BobbleMarine-Body** and click the **Animations** tab in the Inspector.

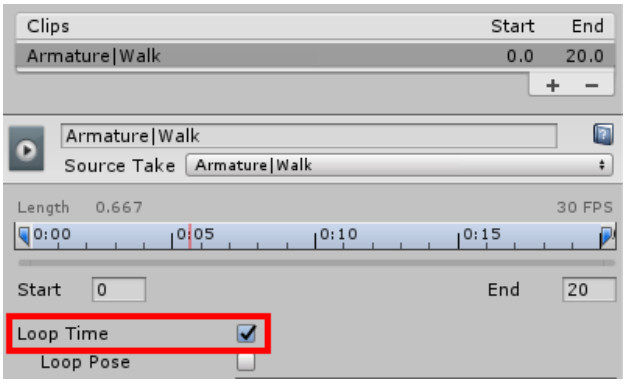
You'll see many unused animations that were imported with the model. This may have been the result of the export to FBX, but in future projects you may have animations you don't need.

Delete all of them EXCEPT **Armature|Walk animation** and **Armature|Idle** by selecting and clicking the **minus sign**. Unfortunately you must do this one at a time.



If you mistakenly delete the wrong animation, just click the **Revert** button at the bottom of the Inspector.

Breathe a sigh of relief now that you have a tidy workspace. Select the **Armature|Walk** animation and check the **Loop Time** box beneath to loop the animation indefinitely.



Click Apply at the bottom to save changes.

You'll need to create a transition between the two different states. In the Project browser, select the **Animations** folder and click the **Create** button. From the drop-down, select **Animator Controller**. Give it the name **SpaceMarine**.

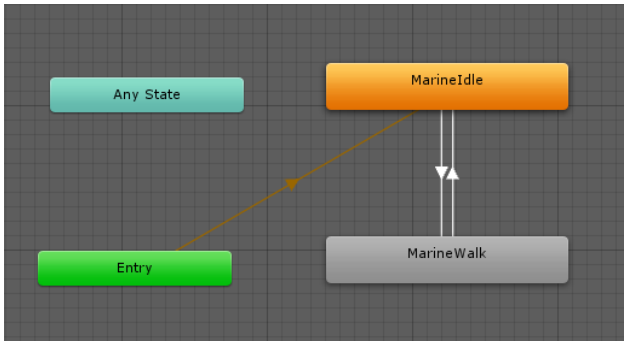
In the Hierarchy, expand the SpaceMarine and select **BobbleMarine-Body**. There is already an animator attached to it — you can see it in the Inspector

Drag the **SpaceMarine animation controller** from the Project browser to the **Controller** property on the Animator.

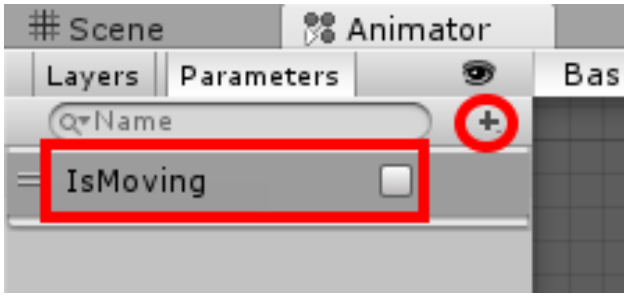
With the BobbleMarine-Body still selected in the Hierarchy, switch to the Animator. Create two different states: **MarineIdle** and **MarineWalk**. Assign **Armature|Idle** to the **MarineIdle** and **Armature|Walk** animation to the **MarineWalk** state.

Set **MarineIdle** as the default state.

Now, you need to add some transitions. Right-click **MarineIdle** and select **Make Transition**. Drag it to **MarineWalk**. Do the same from **MarineWalk** to **MarineIdle**. Your animator controller should look like the following:

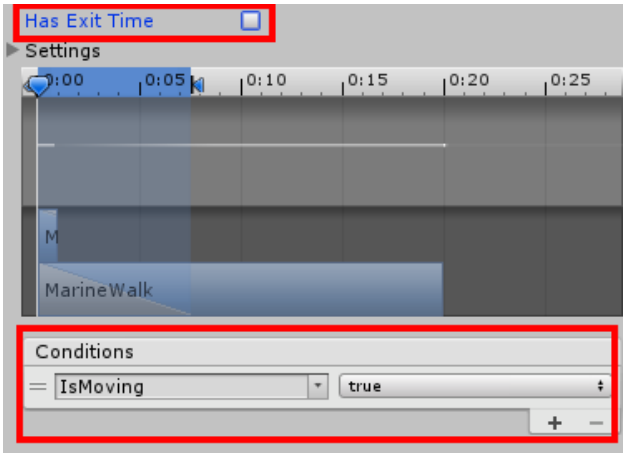


Click the **Parameters** tab then click the **plus sign**. Select **Bool** from the drop-down and name it: **IsMoving**.



Now click the transition from **MarineIdle** to **MarineWalk**, and in the Inspector, **uncheck Has Exit Time**.

Also, click the **plus sign** under Conditions, and set the value to **true**.



Do the same for the transition from **MarineWalk** to **MarineIdle**, but set the Condition to **false**.

When `IsMoving` is true, the walking animation will play. Otherwise, the marine will stand still.

To make this work, you need to write a little code.

Open the **PlayerController** script in your code editor and add the following instance variable underneath the others:

```
public Animator bodyAnimator;
```

Next, in `FixedUpdate()`, find the `\\ TODO` comment (inside the if statement that checks if `moveDirection == Vector3.zero`). Replace it with the following:

```
bodyAnimator.SetBool("IsMoving", false);
```

In the `else` branch, add the following before the closing brace:

```
bodyAnimator.SetBool("IsMoving", true);
```

This simply sets the appropriate values on the Animator.

Save the code and switch back to Unity.

In the Hierarchy, drag the **BobbleMarine-Body** into the **Body Animator** property of the **SpaceMarine**.

Finally, play your game. Now your SpaceMarine can do the moonwalk! :]

Where to go from here?

Bobblehead Wars is really coming along. You have bobbling heads and moving characters. In the process of making all these animations, you thoroughly explored Unity's animation engine, Mecanim. In the process you learned:

- **Keyframe animation**: The basics of creating an animation clip.
- **Animator**: How to use it and manage animations within it.
- **Triggering animations**: How to do this in code (like a boss).
- **Outside animations**: How to use animations created outside of Unity.

Although the game is coming together, it feels like there's something missing. How can you have a game without sound?!

You'll fix that in the next chapter, where you'll bring Bobblehead Wars to life with some kickass sound effects and groovy tunes.

Chapter 7: Sound

By Brian Moakley

Graphics are always top of your mind when you're creating games, largely because we're visual creatures. That said, graphics can only take a game so far.

Sound gives depth to the visuals and can trigger powerful emotions. For example, graphics can adeptly depict a large cavernous structure, but echoing footsteps and the sound of dripping water make it an immersive experience.

In this chapter, you'll take Bobblehead Wars from good to great, throught the power of Unity's audio engine!

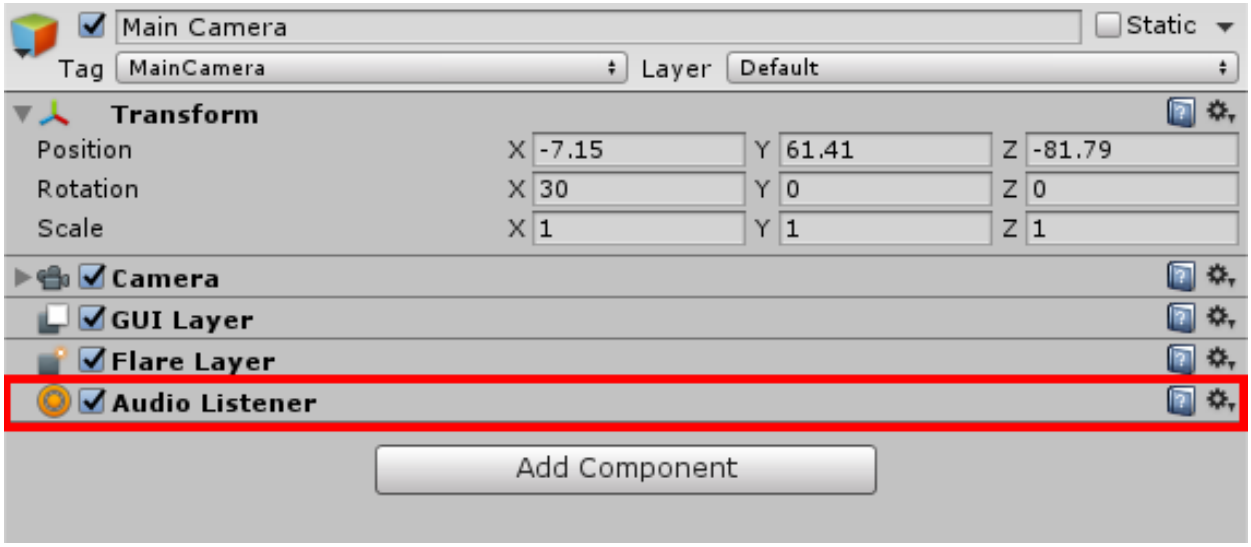
Getting started

Open your Bobblehead Wars project that's in progress, or start fresh with the starter project from the resources for this chapter.

The first thing you need to hear sounds in Unity is an **Audio Listener**. This is a component that you attach to a GameObject so the engine knows where the listener is. This is important because some audio depends on location - for example if you're near a humming generator, the noise will be loud, but if you move far away you might not even be able to hear it.

By default, every new scene includes an audio listener that's attached to the Main Camera.

Click on the camera to see it for yourself:



Note: This default listener can trigger an error when you import a first person character controller, like you will do later in this book. You'll receive a message that states:

"There are 2 audio listeners in the scene. Please ensure there is always exactly one listener in the scene."

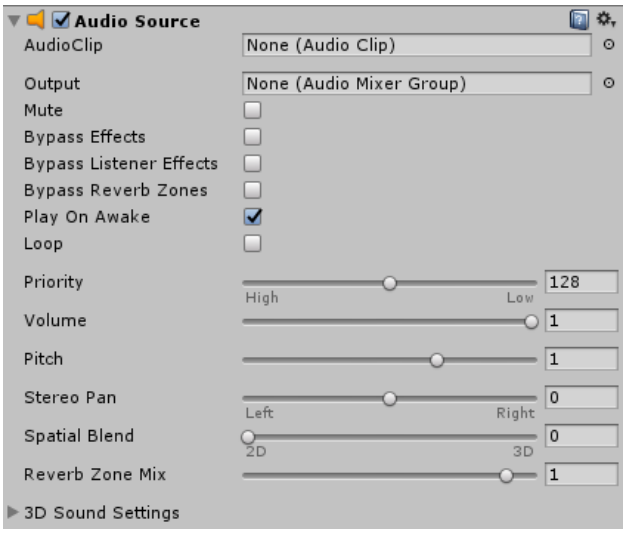
Unity's a bit like the Highlander when it comes to listeners: "There can be only one". Add the AudioListener to the single GameObject that most accurately represents the listener.

The second thing you need to play sounds in Unity is an **Audio Source**. As you might guess, this is an object that emits a sound, such as a sound effect or background music.

You already have an Audio Listener set up, so let's add your first Audio Source - one for background music in the game. As Corinne Bailey Rae would say, it's time to put your records on!

Playing background music

Select **Main Camera** in the Hierarchy. Click the **Add Component** button and select **Audio Source** in the Audio category.



You're adding the audio source for the background music to the camera, since that's the same place the audio listener is. This way, the volume of the background music will always stay the same; think of it like carrying around your own boom box!

As you can see, you have an array of options. Here's the breakdown:

- **AudioClip**: The actual sound file. You create a clip by dragging a sound file into Unity, and then the engine converts it to an audio clip. In your case, you imported the sounds when you imported the models in Chapter 1, "Hello, Unity".

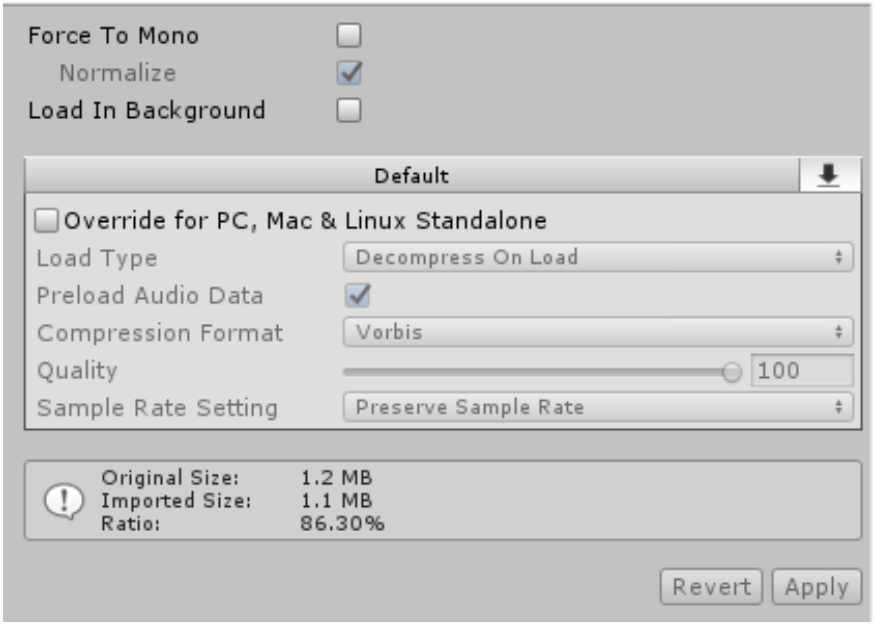
> **Pro tip**: Avoid formats such as mp3 because Unity sometimes re-encodes compressed files to a lower quality when it exports the game. To avoid these, we have provided the audio as ogg files.

- **Output**: This represents the Audio Mixer to use. You'll learn what this means later in this chapter.
- **Mute**: Check this if you'd like to mute the audio. This can be useful to programatically mute sounds mid-game, or to disable sounds while testing.
- **Bypass Effects/Listener Effects/Reverb Zones**: Turns off any additional effects you might add to a sound. Effects can add depth and increase realism. For instance, consider the difference between footsteps in that cavernous structure with and without an echo. By the way, you used to have to buy a paid Unity license to access these features — but they are free for all to use now.

- **Play on Awake**: Plays the sound as soon as the GameObject is activated. Note that this is checked, which is great: it means that the background music will start playing automatically as soon as the game begins.
- **Sound Sliders**: Adjusts the current sound's properties, such as volume and pan location.
- **3D Sound Settings**: Changes how a sound plays in 3D space. For instance, you can adjust the doppler level here. That can be used to create the pitch and frequency of the sound for a fast-moving object, such as a bullet whizzing past your player.

It's important to understand that Unity doesn't factor in level geometry when playing sounds. Here's a case where level geometry matters: imagine that two players, each with a large ray gun, stand on opposite sides of a 2-meter thick concrete wall. If player one fires, in Unity player two would hear the shot as if player 1 was right next to him (rather than it being muffled as you might expect).

Let's look at the audio clip for the background music. In the Project browser, open the **Sounds** folder and select **bw_bg**. The Inspector shows you all the properties for this audio clip.



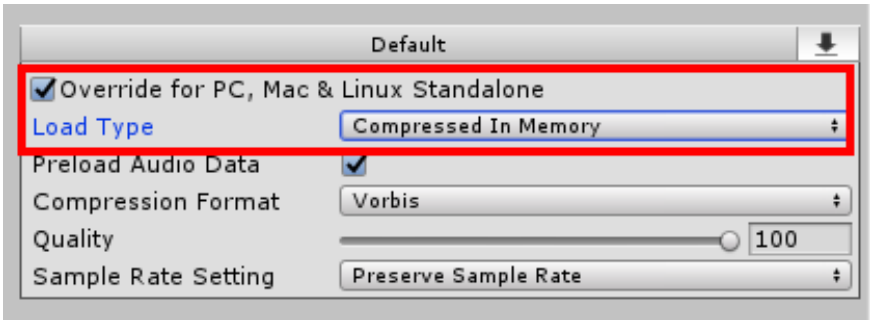
In here, you can decide whether to go with stereo or mono, load the clip in a background thread or normalize.

You also have the ability to change how the audio clip plays, which can have a ripple effect on performance. By default, audio clips are decompressed in memory when loaded.

Ogg files expand ten times when decompressed, so 1 MB file requires 10 MB when it's loaded. It's not ideal.

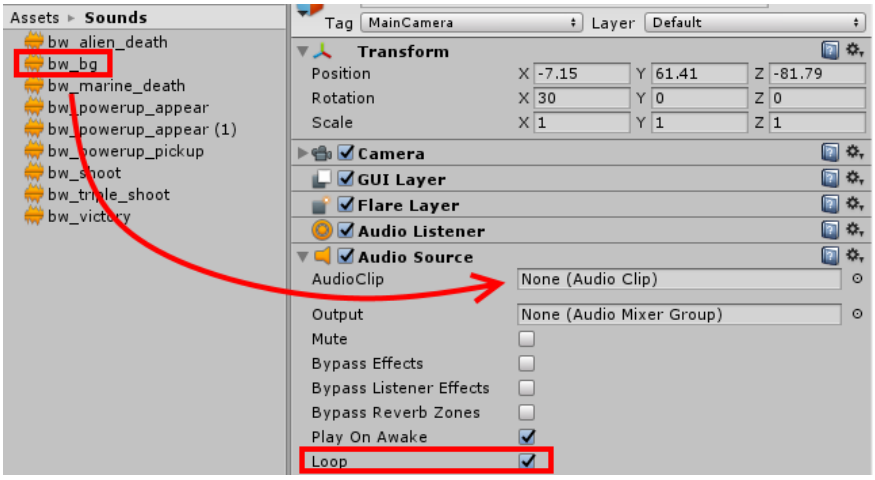
Check the **Override for PC, Mac, & Linux** standalone option. From the **Load Type** option, select **Compressed in Memory**. Instead of hogging memory space, the audio will decompress as it plays. This setting does increase CPU usage, but it's a fair trade-off in this case.

Note that you have the option to stream the file too, but you can try that out another time.



Click **Apply** to save changes.

Next, select the **Main Camera** and drag **bw_bg** from the Project browser to the **AudioClip** property on the Main Camera's audio source. **Check** the **Loop** checkbox to make it repeat continuously.



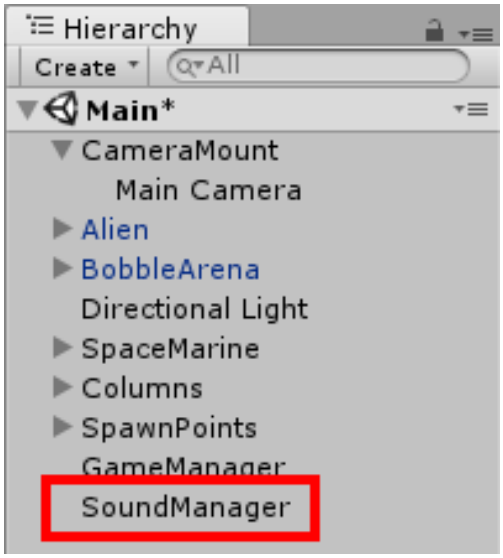
Play your game and listen to those sweet, sweet tunes. Don't worry about volume for the moment. You've got a sound manager to build.

Building a sound manager

Sounds stack up quickly when you're building a game. I bet you're already thinking about audio you'd like in Bobblehead Wars — like a sound effect for when the marine fires his gun, or perhaps a battle cry for the aliens.

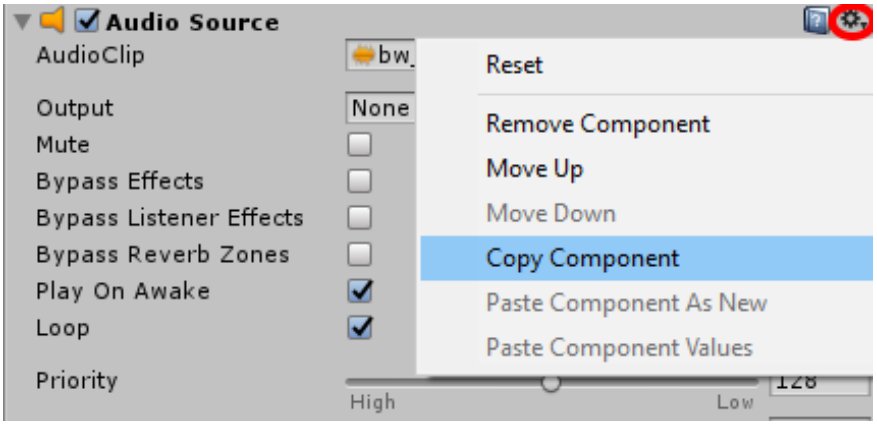
You could create AudioClip properties throughout your scripts, but it's hard to keep things organized. It's actually easier to build and use a sound manager as an audio library for your game.

In the Hierarchy, click the **Create** button and from the drop-down, select **Create Empty**. Give it the name **SoundManager**.



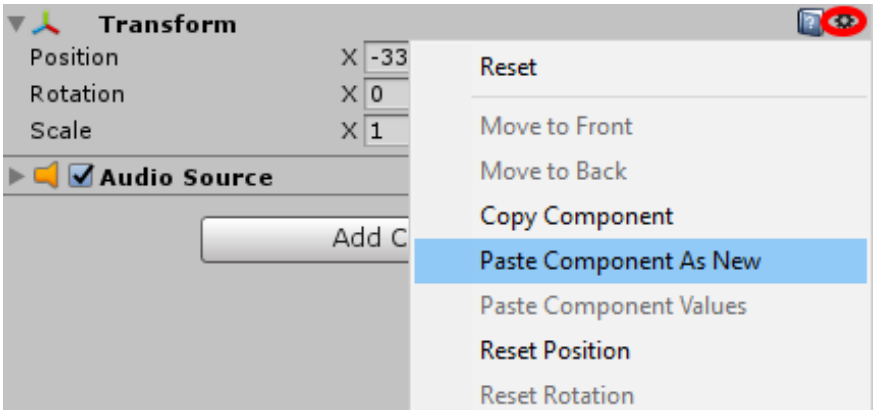
Click the **Add Component** button, and from the Audio category, select **Audio Source**. SoundManager will provide audio clips when asked most of the time, but sometimes it iwll play sounds directly too.

Now that you have a sound manager, you should move the background audio source to the sound manager, to keep all the audio in one place. To do this, select **Main Camera** and expand the Audio Source component in the Inspector. **Click** the **gear icon** and select **Copy Component** from the options.



Click the **gear icon** again, and this time, select **Remove Component**. Finally, select the **SoundManager** in the Hierarchy and find the Transform component in the Inspector.

Click the **gear icon** and choose **Paste Component as New**.



Now you have two audio sources. The first plays sound effects while the second plays the background music.

Click the **Add Component** button and select **New Script**. Name it **SoundManager**, make sure the language is **C Sharp**, and then click **Create and Add**. Open the script in your code editor.

You want to be able to access this script anywhere in your game. After the opening brace, add the following:

```
public static SoundManager Instance = null;
private AudioSource soundEffectAudio;
```

`Instance` will store a static reference to the single instance of `SoundManager`. This will be handy so you can quickly get a reference to the `SoundManager` anywhere in your scripts, without having to go through the normal process of making a public variable that you set in the Unity editor.

`soundEffectAudio` refers to the audio source you added to the `SoundManager` earlier that will be used to play sound effects. Since you won't need to adjust the second audio source that plays background music in code, there's no reason to create a reference for it.

Next, add the following to `Start()`:

```
if (Instance == null) {
  Instance = this;
} else if (Instance != this) {
  Destroy(gameObject);
}
```

This is a simple coding design pattern, known as a Singleton pattern, that ensures that there is always only one copy of this object in existence. If this is the first `SoundManager` created, it sets the static `Instance` variable to the current object. If for some reason a second `SoundManager` is created, it automatically destroys itself to make sure that one and only one `SoundManager` exists.

Next, add the following after the last bit of code:

```
AudioSource[] sources = GetComponents<AudioSource>();
foreach (AudioSource source in sources) {
  if (source.clip == null) {
    soundEffectAudio = source;
  }
}
```

You need to get a handle to the AudioSource you added to the SoundManager that you'll use to play sound, but currently you have two AudioSources on the GameManager. How can you find the correct one?

Well, one way is to search through all the AudioSource components for the one you're looking for. `GetComponents()` returns all of the components of a particular type. Although Unity tends to return components in the same order as they are the Inspector, you shouldn't count on that as that behavior may change. Instead you should look for something that distinguishes the one you're looking for. In this case, since you added music to one of the sources, this checks for the audio source that has no music before setting `soundEffectAudio`.

Now you're going to build out your sound effect library.

Add the following instance variables after the class definition:

```
public AudioClip gunFire;
public AudioClip upgradedGunFire;
public AudioClip hurt;
public AudioClip alienDeath;
public AudioClip marineDeath;
public AudioClip victory;
public AudioClip elevatorArrived;
public AudioClip powerUpPickup;
public AudioClip powerUpAppear;
```

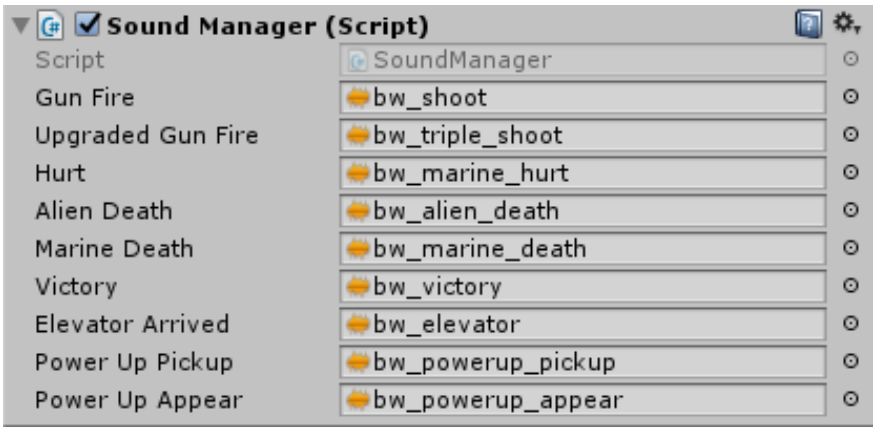
These variables represent the clips for each sound effect in the game. You've made them all public so you can set them to the correct clips in the Unity editor.

Save and return to Unity. Select **SoundManager** in the Hierarchy, open the **Sounds** folder, and then assign the following sounds to the SoundManager:

- `Gun Fire`: **bw_shoot**
- `Upgraded Gun Fire`: **bw_triple_shoot**
- `Hurt`: **bw_marine_hurt**
- `Alien Death`: **bw_alien_death**
- `Marine Death`: **bw_marine_death**
- `Victory`: **bw_victory**

- `Elevator Arrived`: **bw_elevator**
- `PowerUp Pickup`: **bw_powerup_pickup**
- `PowerUp Appear`: **bw_powerup_appear**

The component should look as follows:



Good work! You've got a sound library nicely organized in one spot, and are ready to integrate the rest of the audio into the game.

Playing sounds

In the Hierarchy, expand the **SpaceMarine** and select **BobbleMarine-Body**. In the Inspector, click the **Add Component** button, and select **Audio Source** from the Audio category.

Next, **double-click** the **Gun** script to edit it and add the following instance variable:

```
private AudioSource audioSource;
```

Add the following to `Start()`:

```
audioSource = GetComponent<AudioSource>();
```

This just gets a reference to the attached audio source — you'll be using this a lot. In `fireBullet()`, add the following before the closing brace:

```
audioSource.PlayOneShot(SoundManager.Instance.gunFire);
```

This code plays the shooting sound.

Note that there are two ways to play sound effects via code in Unity:

- `PlayOneShot()`, as you've used here, allows the same sound effect to play muliple times at once, effectively overlapping the sound. For example, the space marine could shoot one bullet and while the sound effect for that was still playing, the space marine could quickly fire another bullet, and you'd hear both sound effects at once. The space marine fires quickly, so this is what we want here.
- `Play()` only allows the sound effect to play a single time at once. If you play a sound effect, and then call `Play()` again, it will restart the sound effect, making the first effect sound like it was cut off. The advantage of `Play()` over `PlayOneShot()` is that `Play()` can be stopped in progress where `PlayOneShot()` cannot.

Save the file and go back to Unity. Play your game and fire that gun to hear the shooting sounds.

It's sounding awesome, but it is a bit disappointing that you don't hear the aliens die when you blast them away. So let's add the sound for that next.

Note that you can't make the alien play its own sound as you did for the bullet, because when an alien dies, you destroy the alien GameObject. When you destroy a GameObject, any sounds playing on an AudioSource stop instantly.

So instead, you'll make the GameManager play the death sounds. Although it's tempting to use the SoundManager for all the sounds, doing so would narrow your options. You'll learn more about this when you dive into the mixer.

Open the **SoundManager** script in your code editor, and add the following method:

```
public void PlayOneShot(AudioClip clip) {
    soundEffectAudio.PlayOneShot(clip);
}
```

This is a simple a wrapper call to `PlayOneShot()`.

Save and close the **SoundManager** script, then open the **Alien** script. In `OnTriggerEnter()`, add the following before the closing brace:

```
SoundManager.Instance.PlayOneShot(SoundManager.Instance.alienDeath);
```

Save the script and head back to Unity. Play the game and obliterate some aliens. Explosions abound!

Adding power-ups

You've made it pretty difficult for the poor space marine. In Chapter 4, "Physics", you added an excessive amount of aliens. In Chapter 5, "Pathfinding", you've taught the aliens how to chase after the space marine, out for blood.

Worst of all, you've been making him work without theme music. Who does that? That's cruel!

Now's a good time to give our hero a boost — something to level playing field. You'll create a powerup that'll sound really cool and temporarily triple the rate of fire.

First, you need to create the triple-shot gun. Open the **Gun** script, and add the following under the instance variables:

```
public bool isUpgraded;
public float upgradeTime = 10.0f;
private float currentTime;
```

- `isUpgraded` is a flag that lets the script know whether to fire one bullet or three.
- `upgradeTime` is how long the upgrade will last (in seconds).
- `currentTime` keeps track of how long it's been since the gun was upgraded.

Take a look at `fireBullet()`. Note how it creates each bullet on one line and sets its position on the following line. Instead of copying these lines multiple times, you'll refactor it into a method.

Add the following method:

```
private Rigidbody createBullet() {
  GameObject bullet = Instantiate(bulletPrefab) as GameObject;
  bullet.transform.position = launchPosition.position;
  return bullet.GetComponent<Rigidbody>();
}
```

As you can see, this method simply encapsulates the bullet creation process. It returns a `Rigidbody` after you create the bullet, all you need to do is set its velocity.

Next, replace `fireBullet()` with the following:

```
void fireBullet() {
  Rigidbody bullet = createBullet ();
  bullet.velocity =  transform.parent.forward * 100;
}
```

This code creates the bullet just like the previous version of the script. Now add the following to modify how the upgraded gun fires bullets (before the final curly brace):

```
if (isUpgraded) {
  Rigidbody bullet2 = createBullet();
  bullet2.velocity =
    (transform.right + transform.forward / 0.5f) * 100;
  Rigidbody bullet3 = createBullet();
  bullet3.velocity =
    ((transform.right * -1) + transform.forward / 0.5f) * 100;
}
```

This bit of code fires the next two bullets at angles. It calculates the angle by adding the forward direction to either the right- or left-hand direction and dividing in half. Since there is no explicit left property, you multiply the value of `right` by –1 to negate it.

Now to integrate the sound. Add the following before the closing brace:

```
if (isUpgraded) {
  audioSource.PlayOneShot(SoundManager.Instance.
    upgradedGunFire);
} else {
  audioSource.PlayOneShot(SoundManager.Instance.gunFire);
}
```

This provides the shooting sound. Now you can see how working with `SoundManager` makes playing sounds relatively easy.

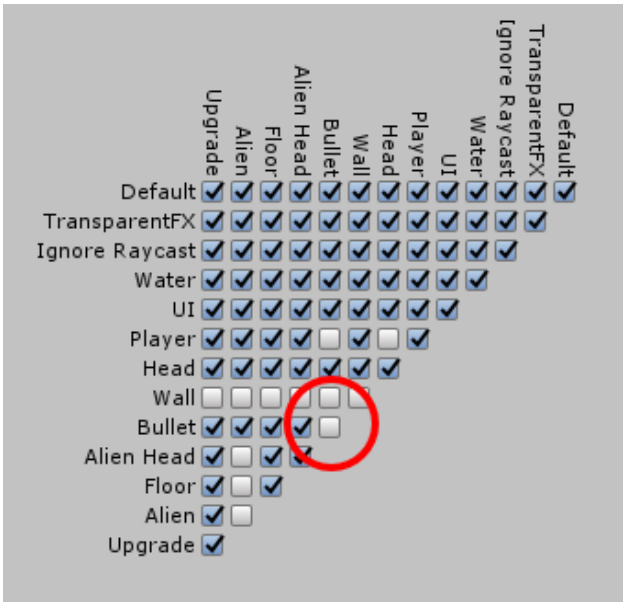
Next, you need a way to initiate the upgrade. Add the following new method:

```
public void UpgradeGun() {
 isUpgraded = true;
 currentTime = 0;
}
```

This method lets the gun know it's been upgraded and sets the counter timer to zero.

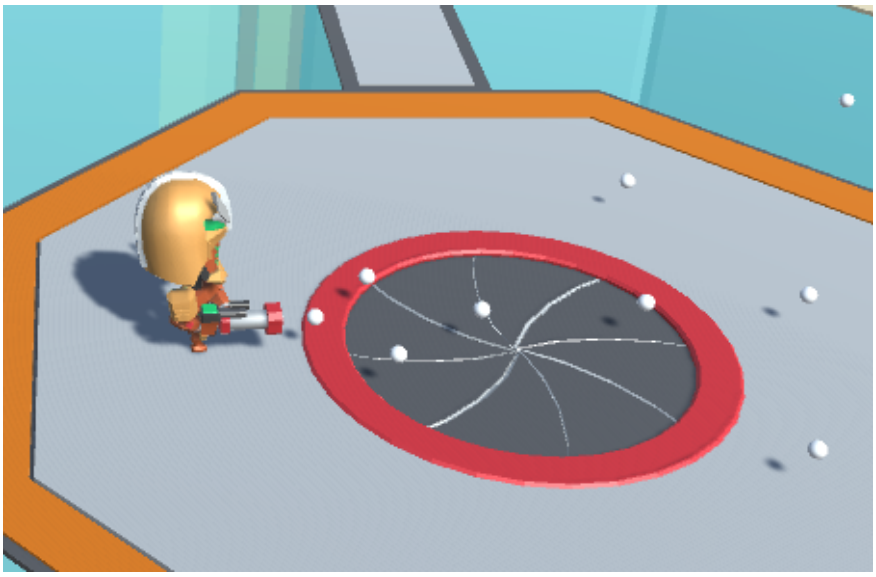
Save the file and switch back to Unity. Since you're firing three bullets at once, there's a chance they'll collide with each other, so you need to disable bullet collisions. Do you remember how to do this?

Click **Edit\Project Settings\Physics** and scroll down to the collision matrix. **Uncheck** the box where the Bullet column and Bullet row intersect.



Believe it or not, you can test your power-up already, thanks to the power of live-editing values in the Unity editor.

Play the game and select the **BobbleMarine-Body**. In the Inspector, **check** the **Is Upgraded** checkbox and blast away!



A power-up pick-up

It's great that you can test the power-up while debugging, but let's add a way for the space marine to get his awesome new gun without cheating.

But first, let's make it so the power-up expires. After all, our hero will get spoiled if you leave it like this!

To do this, switch back to the **Gun** script and add the following to `Update()`:

```
currentTime += Time.deltaTime;
if (currentTime > upgradeTime && isUpgraded == true) {
  isUpgraded = false;
}
```

This increments the time. If the time here is greater than the time the player has been upgraded, this code takes the upgrade away.

Excellent work! You've successfully coded the powerup and just need to add it to your game. **Save** the file and switch back to Unity.

In the Project view, open the **Prefabs** folder and select the **Pickup prefab**. You'll see it has some configuration already.

Click the **Add Component** button and select **New Script**. Name it **Upgrade**, set the language to **C Sharp** and click **Create and Add**.

Open the script in your code editor. Replace the contents of the class with the following:

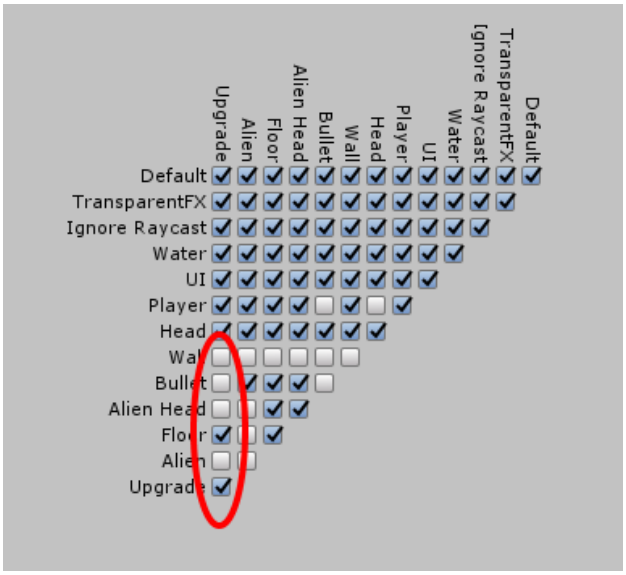
```
public Gun gun;

void OnTriggerEnter(Collider other) {
  gun.UpgradeGun();
  Destroy(gameObject);
  SoundManager.Instance.
    PlayOneShot(SoundManager.Instance.powerUpPickup);
}
```

When the player collides with the powerup, this sets the gun to upgrade mode, and then it will destroy itself (each pickup is good for one upgrade only). Also, a sound plays when the player picks up the upgrade.

Since the powerup should only be available to the space marine, you need to adjust the physics layers. The pickup is already set to the `Upgrade` layer. You just need to adjust the collision matrix. **Save** and switch back to Unity.

Click **Edit\Project Settings\Physics**. In the layers matrix, **uncheck** the following from the **Upgrade column**: **Bullet**, **Alien Head**, **Alien**.



With all that set up, you're clear to write the spawning code for the powerup — yes, it spawns like the aliens. Open the **GameManager** script in your code editor and add the following instance variables:

```
public GameObject upgradePrefab;
public Gun gun;
public float upgradeMaxTimeSpawn = 7.5f;

private bool spawnedUpgrade = false;
private float actualUpgradeTime = 0;
private float currentUpgradeTime = 0;
```

Here you've defined several new variables:

- `upgradePrefab` refers to the GameObject the player must collide with to get the update — you already imported this into your project.
- `gun` is a reference to Gun script because the gun needs to associate it with the upgrade.
- `upgradeMaxTimeSpawn` is the maximum time that will pass before the upgrade spawns.
- `spawnedUpgrade` tracks whether the upgrade has spawned or not since it can only spawn once.
- `actualUpgradeTime` and `currentUpgradeTime` track the current time until the upgrade spawns.

Now to determine the actual upgrade time. Add the following to `Start()`:

```
actualUpgradeTime = Random.Range(upgradeMaxTimeSpawn - 3.0f,
  upgradeMaxTimeSpawn);
actualUpgradeTime = Mathf.Abs(actualUpgradeTime);
```

The upgrade time is a random number generated from `Random.Range()`. The minimum value is the maximum value minus `3`, and `Mathf.Abs` makes sure it's a positive number.

Add the following to the start of `Update()`:

```
currentUpgradeTime += Time.deltaTime;
```

This adds the amount of time from the past frame. Now, add the following:

```
if (currentUpgradeTime > actualUpgradeTime) {
  if (!spawnedUpgrade) { // 1
    // 2
    int randomNumber = Random.Range(0, spawnPoints.Length - 1);
    GameObject spawnLocation = spawnPoints[randomNumber];
    // 3
    GameObject upgrade = Instantiate(upgradePrefab)
      as GameObject;
    Upgrade upgradeScript = upgrade.GetComponent<Upgrade>();
    upgradeScript.gun = gun;
    upgrade.transform.position =
      spawnLocation.transform.position;
    // 4
    spawnedUpgrade = true;
  }
}
```

Here's the breakdown:

1. After the random time period passes, this checks if the upgrade has already spawned.

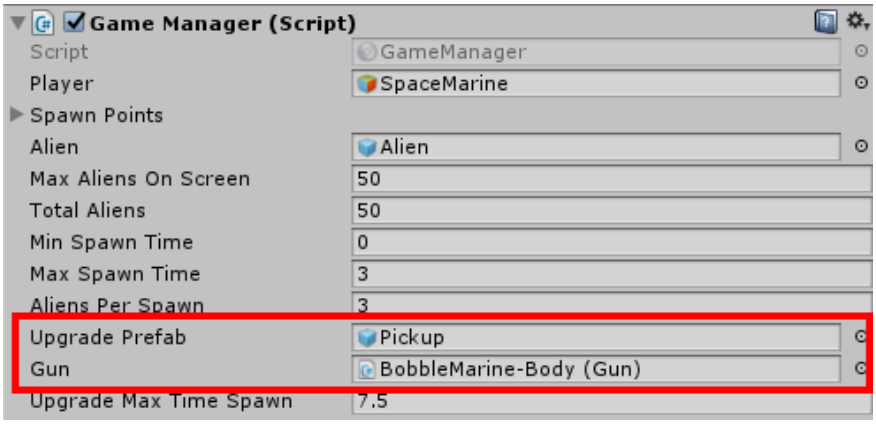
2. The upgrade will appear in one of the alien spawn points. This ups the risk and reward dynamic for the player because the sides of the arena are the most dangerous.
3. This section handles the business of spawning the upgrade and associating the gun with it.
4. This informs the code that an upgrade has been spawned.

Now back to the business of sound! The player needs an auditory queue when the powerup is available. Add the following after `spawnedUpgrade = true`:

```
SoundManager.Instance.PlayOneShot(SoundManager.Instance.
  powerUpAppear);
```

Save the game and switch to Unity. Select the **GameManager** in the Hierarchy. In the Inspector, drag the **Pickup** from the **Prefabs** folder to the **upgradePrefab** field. Also, drag **BobbleMarine-Body** from the Hierarchy to **Gun**.

Since the SpaceMarine has a Gun script attached, Unity will reference the Gun component like it has done for Transforms and Rigidbodies.



Save your work and play the game. Check out that working powerup!

Introducing the Audio Mixer

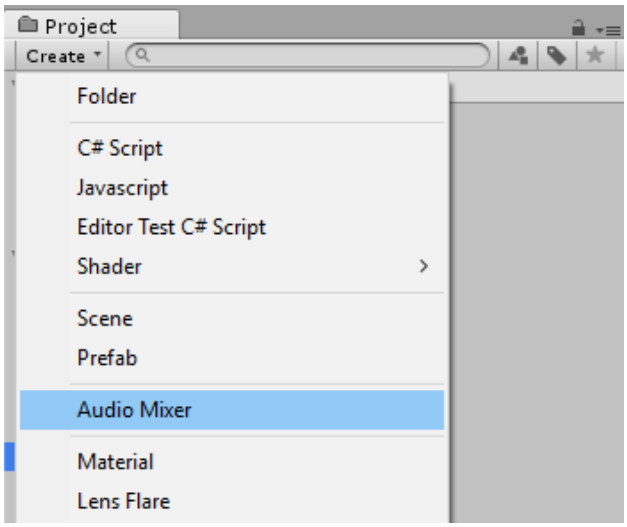
While playing your game, you might notice that the sounds are a little out of whack. For example, the music is a bit too loud, the death sounds are a tad too soft and the bullets are lacking pizzazz!

You could tweak each sound individually, but you have better things to do, such as killing aliens and designing an epic game.

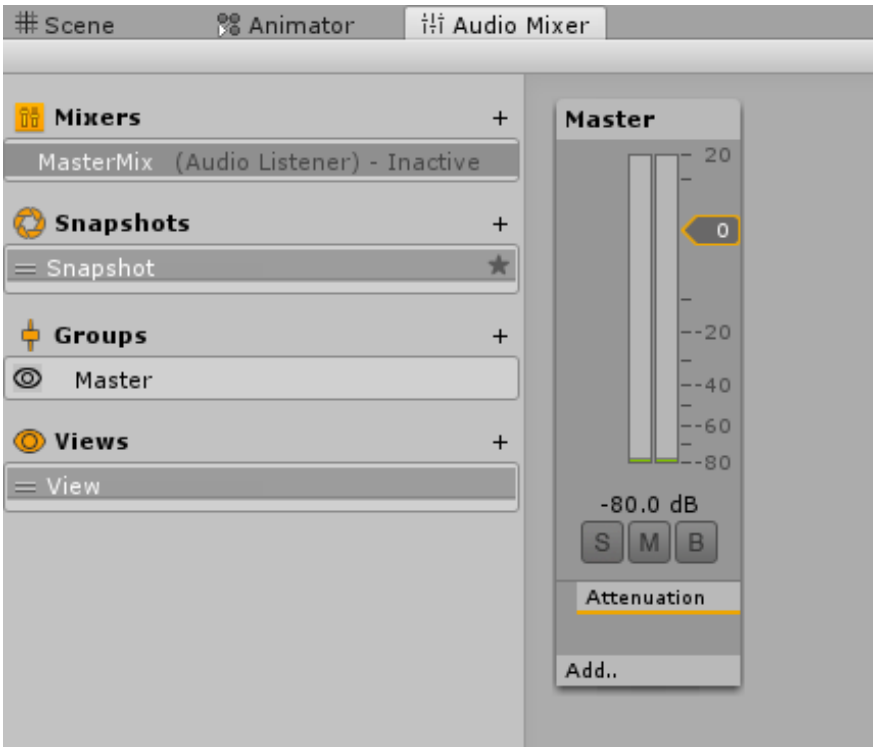
Think about it; you'd have to play the game and keep track of the individual sound levels then make adjustments. While you could copy some of the changes, most of the work would be done by hand.

Before Unity 5, the painful one-at-a-time process would have been your only option, but Unity 5 puts an audio mixer in your hands.

Go to the Project browser and select the **Sounds** folder. Click the **Create button** and select **Audio Mixer** in the drop-down. Name it **MasterMix**.

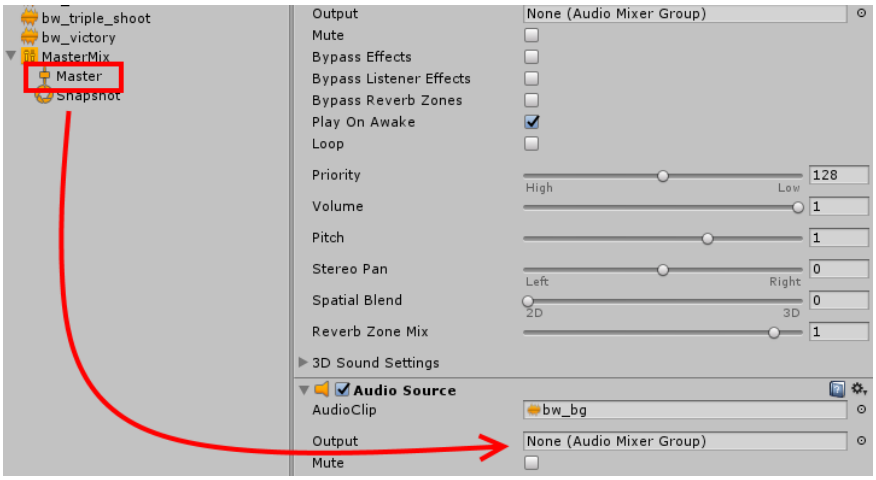


Click **Window\Audio Mixer** and drag the view to a place in the UI where it can expand horizontally instead of vertically.

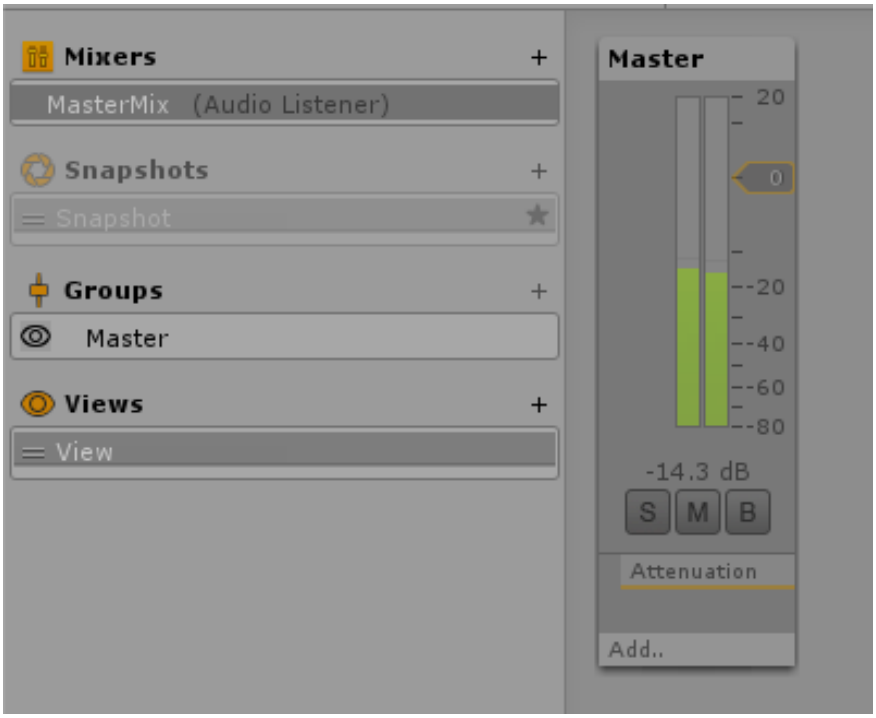


In the mixer, you'll see only one group named Master. This group will affect anything that's assigned to the mixer. Right now, you have assigned nothing, so the mixer is sitting idle.

Expand **MasterMix** in the Project browser, and you'll see the group **Master**. Select **SoundManager**, and in the Inspector, find the audio source that controls the background music. Drag the **Master group** into the **Output** property.

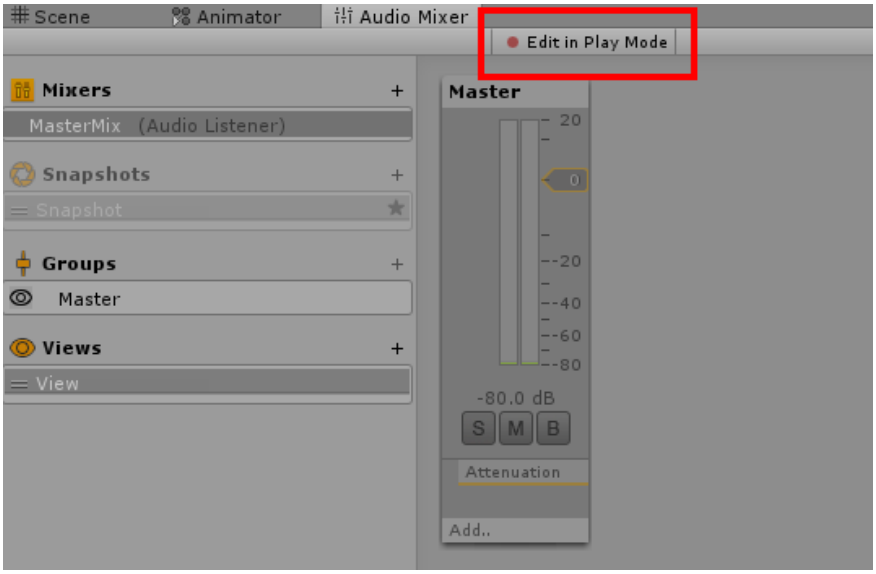


Play the game again.

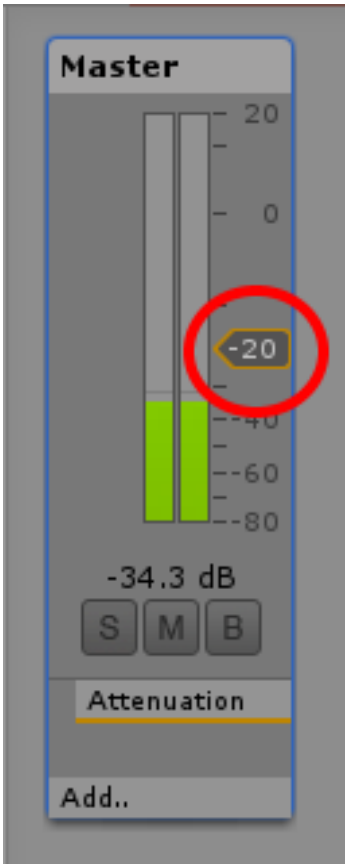


You'll see a cool bar chart showing the sound levels of the Master group. By default, you can't adjust the levels while your game is being played — but you can when you click the **Edit in Playmode** button, so do that now.

The **Edit in Playmode** button will appear when you play your game.



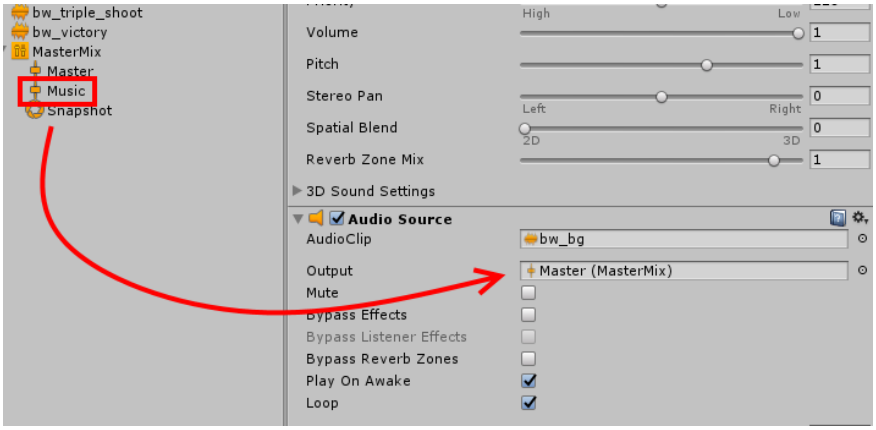
It'll turn red and allow you to adjust levels while you're in play mode. Lower the **volume** to **-20** and stop the game. You'll notice that the group stays at that level.



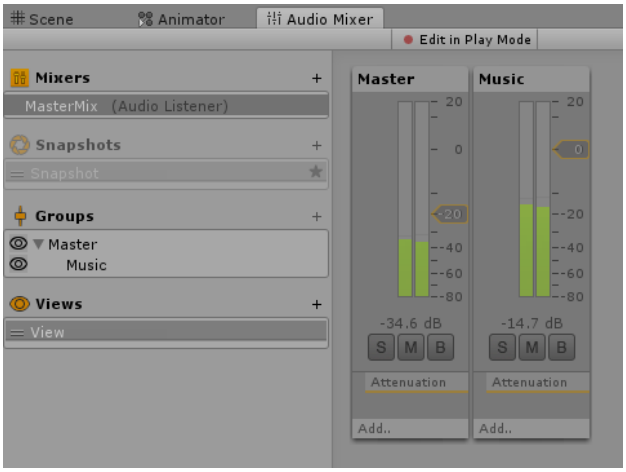
The mixer's capabilities shine when you create subgroups.

Stop the game. Click the **plus sign** in the **groups** section and name it **Music**.

In the Hierarchy, select the **Sound Manager**, and in the Inspector, drag the **Music** group from the MasterMix to the background music AudioSource's **Output field** (overwriting the existing Master setting).

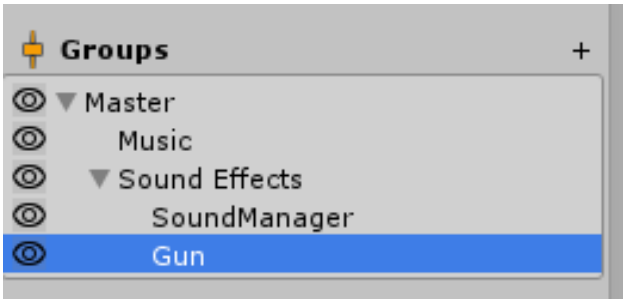


Play your game again. You'll see two levels in the mixer.

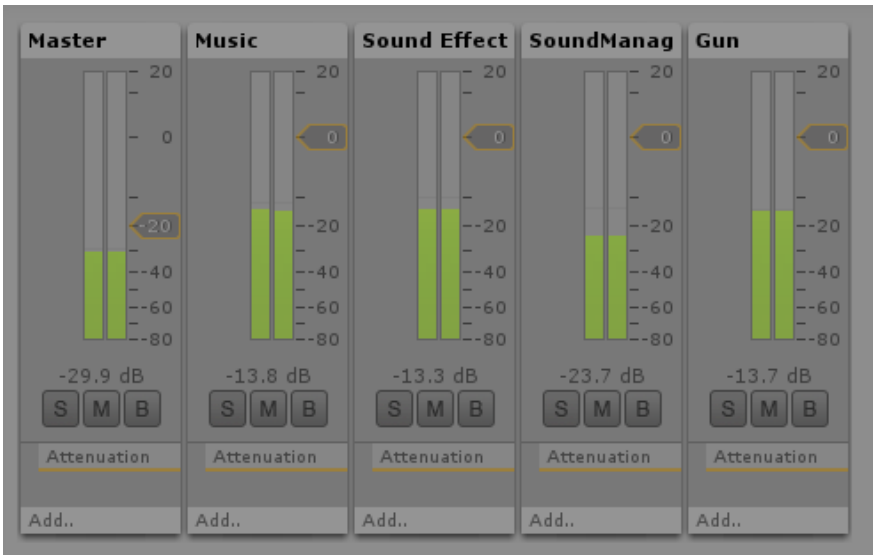


The Master group now controls overall volume, whereas the Music group controls background music.

In the Audio Mixer window, select the **Master** group and click the **plus sign** again. Call this new group **Sound Effects**. Under this new group, create the following groups: **Gun**, **SoundManager**.



Assign the **SoundManager** group to the non-background music AudioSource's **SoundManger output** properties. For the `Gun` group, select **BobbleMarine-Body** in the Hierarchy, and in the Inspector, set the Audio Source's **Output** property to the **Gun** mixer group. When you play the game, you'll see all the levels in action.



At this point, you could adjust the volume of each of the types of audio easily and independently.

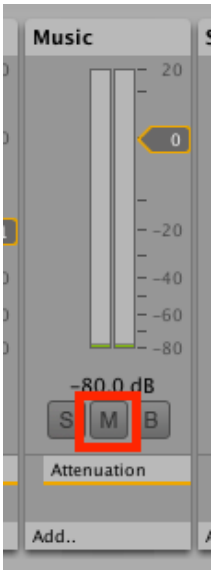
Isolating sounds

Sometimes, you want to only listen to certain sound effects while you're testing your game. For example, imagine you want to isolate the sound of the gun shooting so you can tweak that effect.

You could lower the volume for each, but that's a bad idea because it's easy to forget the original settings.

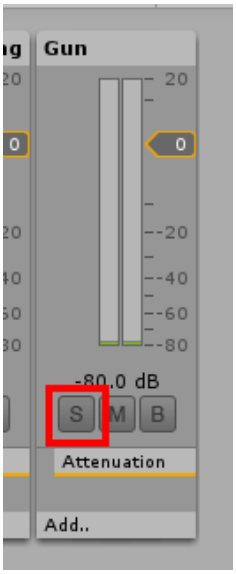
Notice that each group has three buttons on it. S is for **Solo** and it lets you isolate a certain group. M is for **Mute** and it works how you'd expect. B is for **Bypass** and it lets you bypass the effects in the group.

In the **Music** group, click the **M** button to mute the background music.



Play your game if you aren't already, and you'll notice that the background music is now muted, however you can still hear the other sounds.

To fix this, in the **Music** group click the **M** button again to unmute the background, and then in the **Gun** group, click the **S** button to hear the gunfire — and only the gunfire.

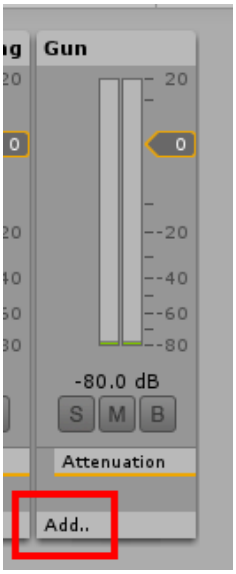


Ah, sweet beautiful gunfire - music to a space marine's ears!

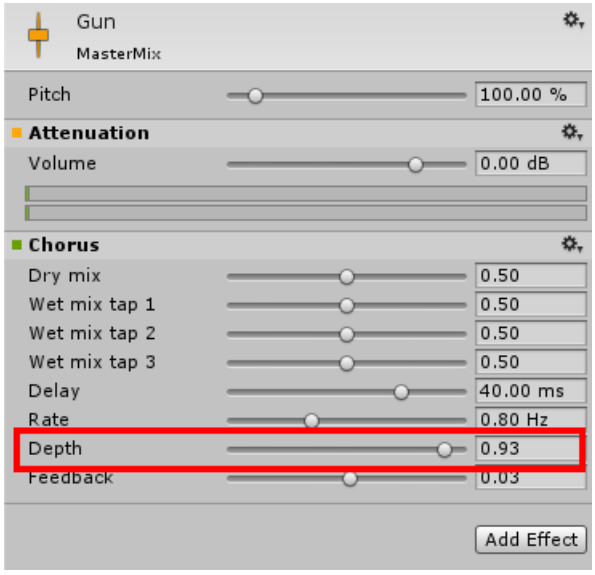
Adding audio effects

Another cool thing the Sound Manager can do is add special effects to audio files.

Click the **Gun** group to see the effect properties in the Inspector. Click the **Add Effect** button at the bottom:

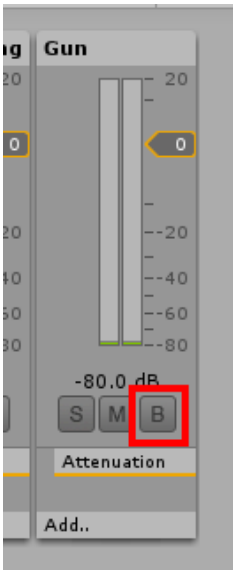


Then choose **Chorus**. Set the **Depth** to `0.93` and fire that gun.



The chorus makes it sounds like rapid fire. Feel free to play around with more effects — the mixer comes with plenty of them.

To hear the group without effects, **click** the **B** button.

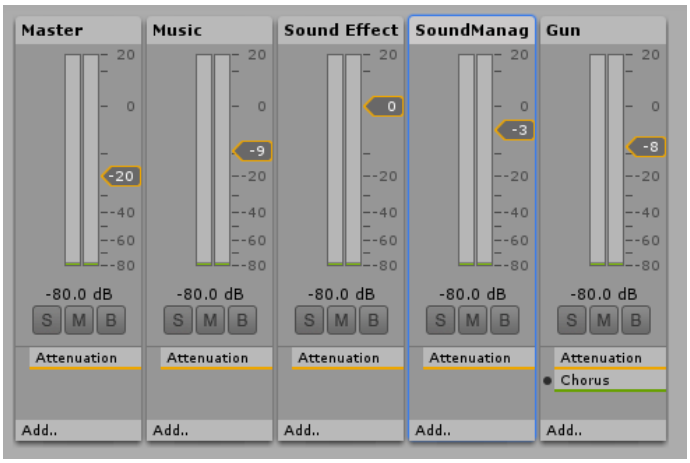


Click **B** again to put the effects back. **Click** the **S** button to re-enable the rest of the sounds.

At this point, hopefully you see that the using the Audio Mixer is much better than trying to adjust sounds manually through the SoundManager. If you'd gone that route, you'd lose the ability to easily set levels on individual sounds, group souns, add effects, and more. By creating a small subset of sounds, you can leverage the mixer while also enjoying the convenience of a single source.

Set the final values as follows:

- **Music** to –9
- **Sound Effects** to `0`
- **Sound Manager** to –3
- **Gun** to –8



So far, you've only scratched the surface of the possibilities with the audio mixer. You can also do things like hiding groups and saving them as views to aid your focus during various editing tasks. It's also possible to take Snapshots to save different groups of settings — either for the sake of comparison or just for different situations.

Where to go from here?

Wow! You've got sound now. With the addition of a few lines of code, your game should feel much more awesome.

In doing so, you've learned:

- The basics of Unity sound, such as the difference between **audio listeners** and **audio sources**
- The import settings for new audio files and how they can affect performance
- How to call your sounds in code
- How to balance all your sounds with the mixer
- **Bonus**: How to create a powerup, complete with sound effects!

What's left to do? You've got aliens. You've got lots of bullets. Sounds like you're all set!

What you're missing is an ending. In the next and final chapter for Bobblehead Wars, you'll put the finishing touches on the game, e.g., lots of exploding aliens and a marine who can't keep his head on straight. Lastly, you'll provide a winning condition.

Exciting things are ahead: stay tuned!

8 Chapter 8: Finishing Touches

By Brian Moakley

It’s funny to think that only seven chapters ago, you might have fired up Unity for the first time, and now you’re about to put final touches on your first game!

At this point, gameplay is pretty much set, sounds are properly mixed and the assets are pretty much launched.

What’s left to do? Ship it!

Actually, no. This game isn't early access! We're going to ship this completed. (Followed by a massive day one patch) ;]

Currently, Bobblehead Wars doesn’t provide a winning or losing condition. After a bit of time, the aliens stop spawning, leaving the space marine to ponder existence with nothing more than a loaded weapon in an empty arena.

In this chapter, you’ll provide a winning and losing condition and add some realism. In the process, you’ll learn about Unity Events, the Unity particle system, and some additional animation features.

Of course, before you even think about game endings, you need to squash a rather big bug that you might not have noticed.

Fixing the game manager

As the hero shoots aliens, the game takes them off the screen, but the GameManager knows nothing of their deaths.

Why is this important?

For one thing, the GameManager may want to send condolence cards to the families. :]

Secondly, the manager must know how many aliens are on the screen and when to end the game. Without that information, the space marine enjoys too much down time.

Thankfully, it's not an insurmountable problem. Instead of counting the alien population, you'll make it so that the GameManager gets notification of each death.

You'll do this by using Unity events. C# also has an event system, but Unity events are built right into the engine.

Open the **Alien** script in your code editor. Add the following underneath the existing `using` statements:

```
using UnityEngine.Events;
```

This allows you to access UnityEvents in code. Next, add the following instance variable inside the class:

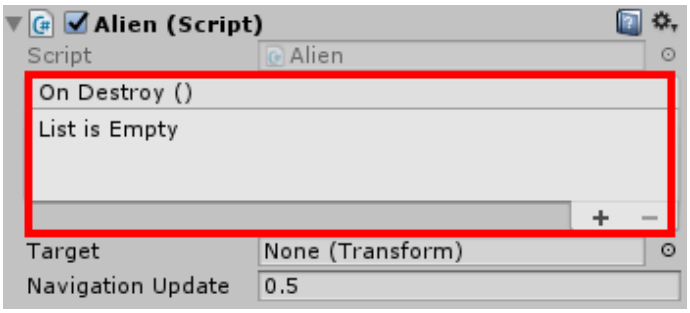
```
public UnityEvent OnDestroy;
```

The `UnityEvent` is a custom event type that you can configure in the Inspector. In this case, you're creating an `OnDestroy` event. The `OnDestroy` event will occur with each call to an alien.

Defining an event is only part of the equation. It's kind of like planning a party. No matter how well you plan, it's not a party if there are no guests.

Just as parties need guests, events need listeners, which do the job of taking notifications when something has happened. In the case of Bobblehead Wars, the GameManager needs to know when an alien dies. Hence, the GameManager needs to subscribe to the `OnDestroy` event.

Save and head back to Unity. In the Project browser, select the **Alien** in the **Prefabs** folder. The Inspector for the Alien component now displays the `OnDestroy` event.

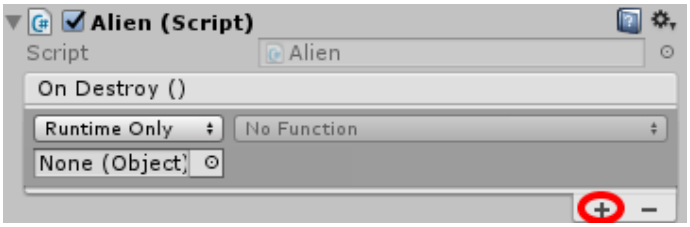


Before you can add GameManager to this list, you need to create a public method to subscribe to the event. Open the **GameManager** script and add the following at the bottom of the file, just before the closing brace:

```
public void AlienDestroyed() {
  Debug.Log("dead alien");
}
```

This prints a simple message to the console to let you know the GameManager got a notification. **Save** and switch to Unity.

In the Project browser, select the **Alien** in the **Prefabs** folder again. Find and select **OnDestroy** in the Inspector and click the **plus sign** to add a new event.



This is where you add event listeners. You'll notice a drop-down that reads **Runtime Only**, which means the events will run in play mode.

Click the circle next to where it says **None**, and a **Select Object** popup will appear. Scroll through, and you'll notice that the **GameManager** GameObject does not appear in the list (the script does, but not the GameManager object itself). What gives?

There's a logical explanation for this: A prefab can't contain a reference to a GameObject in the Hierarchy because that object may not be around when the game instantiates the prefab.

When you link the two together, you're introducing a potential error. For example, you could create another scene with aliens in it, but that scene may not have a GameManager in the other scene.

You can workaround this restriction with scripting.

Click the minus button to remove the entry in the `OnDestroy()` list you added a second ago, then open the **GameManager** script, and find `newAlien.transform.LookAt(targetRotation);` in **Update()**. Add the following after it:

```
alienScript.OnDestroy.AddListener(AlienDestroyed);
```

Here you call `AddListener` on the event and pass in the method to call whenever that event occurs — in this case, `AlienDestroyed`. In short, every time this event occurs, the GameManager gets a notification.

Now you need to generate the event. **Save** your changes and open the **Alien** script in the editor. Add the following method:

```
public void Die() {
    Destroy(gameObject);
}
```

This destroys the alien.

In `OnTriggerEnter()`, replace the line `Destroy(gameObject);` with:

```
Die();
```

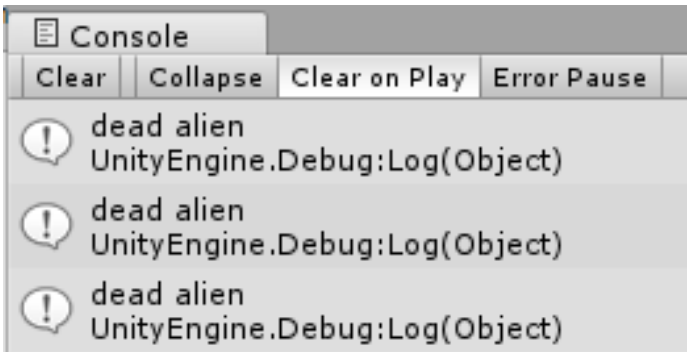
Here you've refactored the destroy code into another method. It allows other objects to trigger the death behavior.

In `Die()`, add the following before the `Destroy()` statement:

```
OnDestroy.Invoke();
```

This notifies all listeners, including the GameManager, of the alien's tragic death.

Save, play the game and shoot some aliens. You'll see a barrage of log messages.



When an object that has an event is destroyed, the best practice is to remove all listeners from that event. Otherwise, you may create a memory leak by having a reference cycle.

After `OnDestroy.Invoke();` add the following:

```
OnDestroy.RemoveAllListeners();
```

This removes any listeners that are listening to the event.

Now you need to update the GameManager. Open **GameManager** in your code editor, and replace `AlienDestroyed()` with the following:

```
public void AlienDestroyed() {
  aliensOnScreen -= 1;
  totalAliens -= 1;
}
```

This officially decreases the number of aliens on screen. **Save** your work and review how what you just did works:

1. The Alien script contains a UnityEvent called `OnDestroy`.
2. The GameManager script periodically spawns aliens. Each time it spawns an alien, it adds itself as a listener to the `OnDestroy` event.
3. When the alien collides with something and is about to die, it invokes its `OnDestroy` event. This has the effect of notifying the GameManager.
4. The GameManager updates the count of aliens on the screen, and the total aliens.
5. The alien removes all listeners and destroys itself.

Now take a breather — you just squashed that bug with a mega-size fly swatter. On to more important matters.

Killing the hero

So far life is a little bit too easy on the hero. The marine is impervious to death!

Let's put our Doctor Evil hats on, and talk about how we'll kill our hero (mwuahaha). Death in Bobble Wars will work in this fashion:

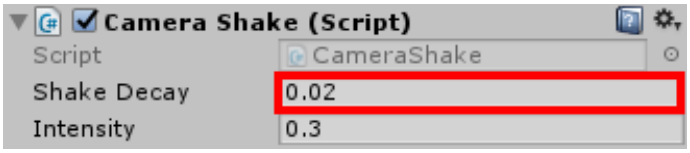
- Once the marine gets hurt, the camera will shake and you'll hear grunting to indicate the marine's agony.
- The more hits, the harder the shake.
- Finally, when the marine suffers that final blow, he'll emit a blood-curdling cry to signal his inglorious death and his head will pop off. I mean, why wouldn't it?

You'll start with the camera shake effect; this script is pre-included with the project for convenience.

In the Project view, select the **CameraShake** script and drag it to **Main Camera** in the Hierarchy.

This script works by providing a shake intensity then it calls `Shake()`.

Select **Main Camera** in the Hierarchy, and in the Inspector, change **Shake Decay** to `0.02` to set the shake's duration.

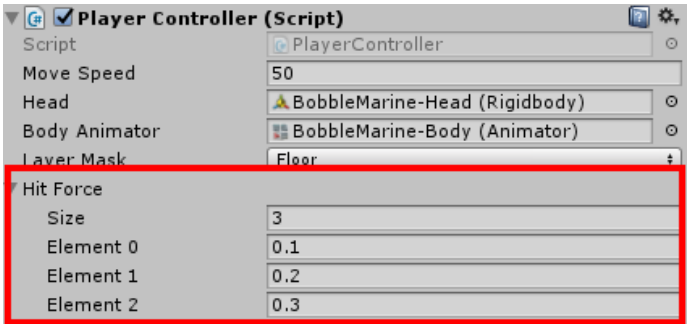


Next up is providing intensity values. Open **PlayerController** in your code editor and add the following instance variable:

```
public float[] hitForce;
```

This provides an array of force values for the camera.

Save and switch to Unity. Select **SpaceMarine** in the Hierarchy, and find the **Player Controller** component in the Inspector. Click the disclosure triangle next to **Hit Force** and set the **Size** to `3`. Add the following values for the elements: `0.1`, `0.2`, `0.3`:



At this point, you could call `CameraShake()` each time an alien collides with the hero, but it'll be a little problematic. When the hero is swarmed by aliens, he could die in a matter of seconds, and the game would cease to be fun.

To avoid this frustration, you'll add a delay between each hit to give the player time to react and escape a bad situation.

Open **PlayerController** in your code editor and add the following instance variables:

```
public float timeBetweenHits = 2.5f;
private bool isHit = false;
private float timeSinceHit = 0;
private int hitNumber = -1;
```

- `timeBetweenHits` is the grace period after the hero sustains damage.

- `isHit` is a flag that indicates the hero took a hit.
- The above variables mean the hero won't accrue damage for a period of time after taking a hit.
- `timeSinceHit` tracks of the amount of time in the grace period.
- `hitNumber` references the number of times the hero took a hit. It's also used to get the shake intensity for the camera shake.

Add the following method:

```
void OnTriggerEnter(Collider other) {
  Alien alien = other.gameObject.GetComponent<Alien>();
  if (alien != null) { // 1
    if (!isHit) {
      hitNumber += 1; // 2
      CameraShake cameraShake =
        Camera.main.GetComponent<CameraShake>();
      if (hitNumber < hitForce.Length) { // 3
        cameraShake.intensity = hitForce[hitNumber];
        cameraShake.Shake();
      } else {
        // death todo
      }
      isHit = true; // 4
      SoundManager.Instance
        .PlayOneShot(SoundManager.Instance.hurt);
    }
    alien.Die();
  }
}
```

Here's the breakdown:

1. First, you check if the colliding object has an `Alien` script attached to it. If it's an alien and the player hasn't been hit, then the player is officially considered hit.
2. The `hitNumber` increases by one, after which you get a reference to `CameraShake()`.
3. If the current `hitNumber` is less than the number of force values for the camera shake, then the hero is still alive. From there, you set force for the shaking effect and then shake the camera. (You'll come back to the death todo in a moment.)
4. This sets `isHit` to true, plays the grunt and kills the alien.

You're getting closer to making the hero's day, but not quite there yet. Once `isHit` is set to true, there's no unsetting it and the hero becomes invincible — the opposite of what you want to do here.

Add the following code to `Update()` to strip away those divine powers:

```
if (isHit) {
  timeSinceHit += Time.deltaTime;
  if (timeSinceHit > timeBetweenHits) {
    isHit = false;
    timeSinceHit = 0;
  }
}
```

This tabulates time since the last hit to the hero. If that time exceeds `timeBetweenHits`, the player can take more hits.

Play your game and charge those aliens. Resistance is futile.

The hero will now react to each hit until the maximum number of hits. At that point, nothing will happen - but you'll fix that next, in a dramatic way!

Removing the bobblehead

There's nothing more terrifying to a bobblehead then losing its precious head. Your space marine is no different. It's only fitting that the head rolls off the body to make the game's end spectacular and undeniable.

Time for a little more scripting. Open **PlayerController** in your editor and add the following instance variables:

```
public Rigidbody marineBody;
private bool isDead = false;
```

`marineBody` is, well, the marine's body. You're adding this because you'll make some physics alterations when the hero dies. And of course, `isDead` keeps track of the player's current death state.

Together, they let the script know if the space marine is dead.

Now you need a method that manages the death, so add the following:

```
public void Die() {
  bodyAnimator.SetBool("IsMoving", false);
  marineBody.transform.parent = null;
  marineBody.isKinematic = false;
  marineBody.useGravity = true;
  marineBody.gameObject
    .GetComponent<CapsuleCollider>().enabled = true;
  marineBody.gameObject.GetComponent<Gun>().enabled = false;
}
```

You set `IsMoving` to `false` since the marine is dead and you don't want a zombie running around. Next, you set the parent to `null` to remove the current GameObject from its parent. Then by enabling `Use Gravity` and disabling `IsKinematic`, the body will drop and roll, and you enable a collider to make this all work. Disabling the gun prevents the player from firing after death.

At this point, the head and body still share a joint. You need to destroy the hinge joint to enable the hero's inevitable disembodiment.

Add the following code beneath the previous block:

```
Destroy(head.gameObject.GetComponent<HingeJoint>());
head.transform.parent = null;
head.useGravity = true;
SoundManager.Instance
  .PlayOneShot(SoundManager.Instance.marineDeath);
Destroy(gameObject);
```

First, this destroys the joint to release the head from the body. Then, like the body, you remove the parent and enable gravity. Now that head can roll!

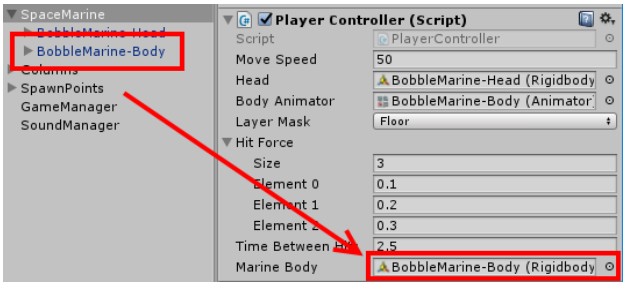
Finally, you destroy the current GameObject while playing the death sound.

Now to trigger the death. In `OnTriggerEnter()`, replace the `death todo` with the following:

```
Die();
```

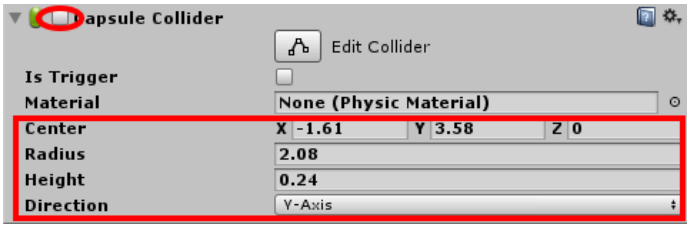
Save the file and go back to Unity.

Select **SpaceMarine** in the Hierarchy. In the Inspector, drag the **BobbleMarine-Body** to the new **Marine Body** property.



The space marine's body can fall now, but you need to attach a collider to it. Select **BobbleMarine-Body** and click the **Add Component** button. Select the **Physics** category and choose **Capsule Collider**.

Set **Center** to (**-1.61**, **3.58**, **0**), **Radius** to **2.08** and **Height** to **0.24**. Make sure the **Direction** is set to **Y-Axis**. Finally, **disable** the collider by unchecking the checkbox in the upper left of the component.



Currently, the GameManager may spawn more aliens after the space marine dies. There's no need to subject the player to an inevitable feeding frenzy!

Open **GameManager** in your code editor, and add the following before any of the existing code in `Update()`:

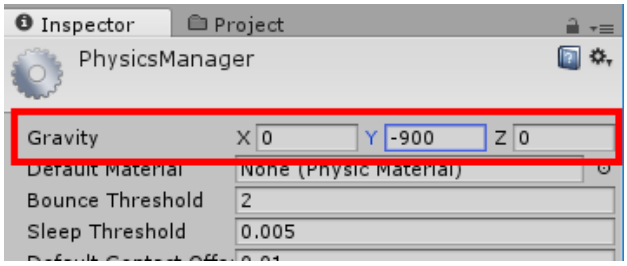
```
if (player == null) {
  return;
}
```

If there's no player, the GameManager won't spawn enemies. **Save** and switch to Unity.

Play your game and charge the aliens. After a bit of time, the marine successfully loses that head. Why is the marine's body still standing around?

This strange behavior is due to gravity and scaling. The typical gravity level is set to -9.81. If you scaled the models to one meter per unit, then they'd play well with gravity. However, your models are somewhat larger. You can fix this by scaling down the models or scaling up the gravity — you'll scale up the gravity in this book.

Click **Edit\Project Settings\Physics** and set **Gravity** to (**0**, **-900**, **0**).



Now the player's death makes both head and body fall like you'd see in a good episode of Game of Thrones.

Decapitating the alien

Aliens have bobbleheads too, so they should die the same way, but you'll have a little fun with the poor alien's head.

Open the **Alien** script and add the following instance variables:

```
public Rigidbody head;
public bool isAlive = true;
```

`head` will help you launch the head, and `isAlive` will track the alien's state.

Replace `Die()` with the following:

```
public void Die() {
  isAlive = false;
  head.GetComponent<Animator>().enabled = false;
  head.isKinematic = false;
  head.useGravity = true;
  head.GetComponent<SphereCollider>().enabled = true;
  head.gameObject.transform.parent = null;
  head.velocity = new Vector3(0, 26.0f, 3.0f);
}
```

This code works similarly to the marine's death, but you disable the animator for the alien. In addition, this code launches the head off the body.

Add this code before the closing brace:

```
OnDestroy.Invoke();
OnDestroy.RemoveAllListeners();
SoundManager.Instance.PlayOneShot(SoundManager.Instance.
  alienDeath);
Destroy(gameObject);
```

This block notifies the listeners, removes them, and then it deletes the GameObject.

You want to make sure that you don't call `Die()` more than once, so modify `OnTriggerEnter()` as follows:

```
void OnTriggerEnter(Collider other) {
  if (isAlive) {
    Die();
    SoundManager.Instance.PlayOneShot(SoundManager.Instance.
      alienDeath);
  }
}
```

Here you check `isAlive` first.

Finally, in `Update()`, surround the existing navigation code with a similar check to see if the alien is alive.

```
if (isAlive) {
  ...
}
```

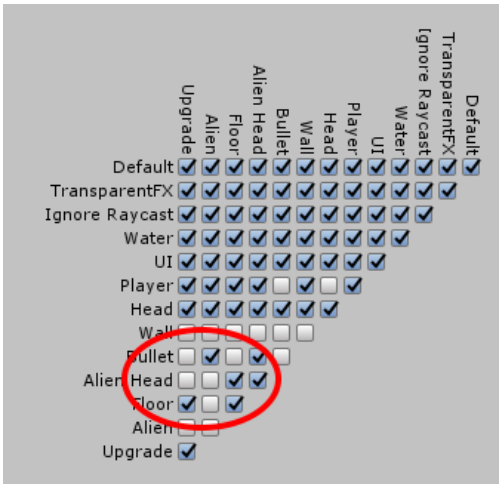
Save and switch to Unity. In the Project view, click the down arrow next to the Alien prefab to reveal is children, and select **BobbleEnemy-Head**. Click the **Add Component** button and select **Rigidbody** from the physics category. **Uncheck** the **Use Gravity** option and **check** the **Is Kinematic** option.

Click **Add Component** once again, and select **Sphere Collider** from the physics category. Set **Center** to (**0**, **0**, **0.02**) and **Radius** to **0.03**. Disable the component by **unchecking** the **component** checkbox.

Finally, assign it to the layer **Alien Head** and select **Yes, change children** when prompted.

Select the **Alien** prefab in the Project view. In the Inspector, drag the **BobbleEnemy-Head** to the **head** property.

Finally, select **Edit\Projects Settings\Physics**. In the **Alien Head** row, **uncheck the Alien** column if you haven't already.



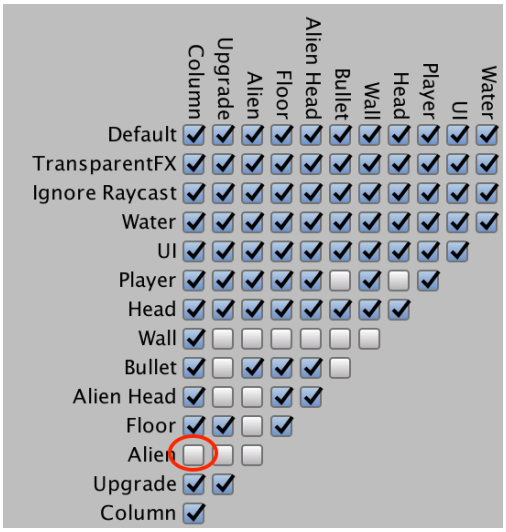
Play the game. It's getting interesting now!

You'll notice that sometimes the aliens are immune from bullets, and their heads can fly right through the columns. The reason is that you have colliders in conflict with one another.

In the Project browser, select the **Alien** prefab. Set the **Radius** to `2.63` in the **Sphere Collider** to give the alien plenty of collider space.

You'll need a new layer to resolve the issue of heads flying through columns. Select the **BobbleArena-Column** prefab in the Project browser. In the **Layer** drop-down, select **Add Layer**, name it **Column** and assign it to the **BobbleArena-Column** prefab. Select **Yes, change children** when prompted.

Finally, select **Edit\Projects Settings\Physics**. In the **Column** column, **uncheck** the **Alien** row.



Play your game.

It won't take long to accumulate an excess of disembodied alien heads. After a while, the space marine will struggle to differentiate the dead aliens from the living.

Stop your game and select the **BobbleEnemy-Head** prefab. In the Inspector, click the **Add Component** button and select **New Script**.

Name it **SelfDestruct** and click the **Create and Add** button. Open it in your editor. Inside the class, replace the contents of the script with the following:

```
public float destructTime = 3.0f;

public void Initiate() {
  Invoke("selfDestruct", destructTime);
}

private void selfDestruct() {
  Destroy(gameObject);
}
```

This deletes the attached GameObject after a given amount of time. You could have named it `AlienHead`, but its general name makes it available throughout the game.

Open **Alien** and add the following to `Die()` before the `Destroy(gameObject);` line:

```
head.GetComponent<SelfDestruct>().Initiate();
```

Now play the game and watch those heads disappear after three seconds.

Adding particles

Are you having a little trouble believing that the process of dismembering characters is so clean? Normally, beheading makes a big mess — this game should be no exception. You'll use particles to increase the, um, realism a bit.

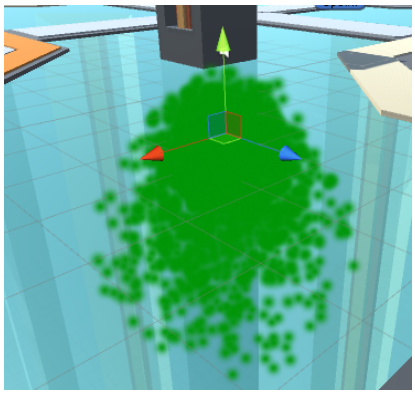
Particles are small bits of geometry created by a game engine that you can use to create effects such as fire, water and other chaotic systems.

Ever play a game that featured snow? In most cases, the effect was produced by a particle emitter directly over the character. It probably seemed like it was snowing in the entire game world, but the particle emitter was most likely following you around like a little dark cloud.

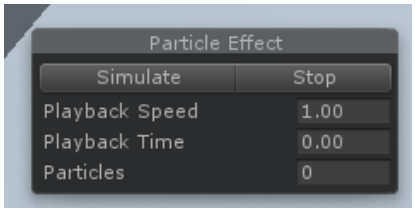
Particles are a deep topic that you should cover in more depth on your own. For convenience, you'll use pre-made particles.

> **Note:** If you'd like to explore Unity's particle system in depth, please check out our free detailed introduction at https://www.raywenderlich.com/introduction-unity-particle-systems.

In the Project browser, drag an instance of the **AlienDeathParticles** prefab into the scene. You'll see an instant splash of alien blood. Ew!



Select it in the Hierarchy; you'll see a dialog in the Scene view with a few options.

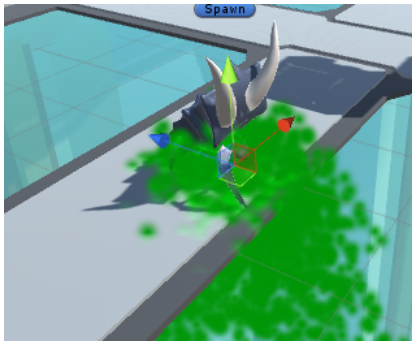


This control panel lets you tweak settings in real time when building particle effects. Press the **Simulate** button. You'll see the blood fly once again and notice that the button becomes a pause button. You also have a stop button to end the particle effect.

Temporarily drag an **Alien** prefab into the scene, and make **AlienDeathParticles** to make it a child of **Alien**. With **AlienDeathParticles** selected, set **Position** to (**0**, **1.07**, **-1.7**) and **Rotation** to (**-90**, **0**, **0**). Click the **Apply** button to apply the changes to the AlienDeathParticle prefab.

Also apply the changes to all the instances of the Alien prefab by selecting the **Alien** in the Hierarchy and **clicking** the **Apply** button in the Inspector.

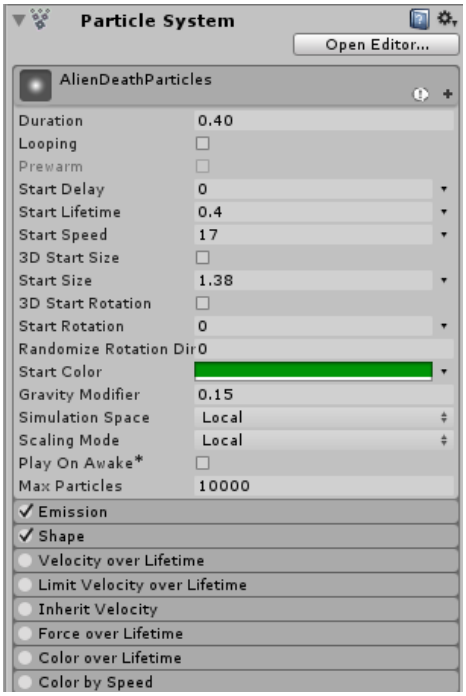
Now, click **AlienDeathParticles** in the Hierarchy and click on the **Simulate** button to see the alien bathed in green blood.



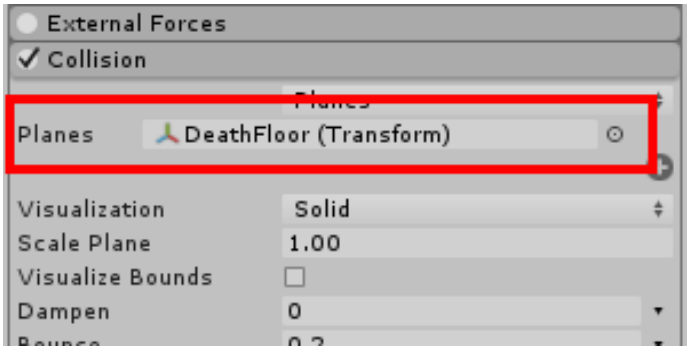
They're working, but they fall through the floor, so clearly you need special treatment for the particles. You could have the particles collide with the world, but in this case, a death floor makes more sense.

In the Hierarchy, select **BobbleArena**. Click the **Create** button, select **3D Object** and choose **Plane**. Name it **DeathFloor** and drag it to be a child of **BobbleArena**. Set **Position** to (**8.66**, **-0.16**, **12.15**) and **Scale** to (**-14.9**, **0.5**, **15.6**). **Uncheck** the **Mesh Collider** and **Mesh Renderer** to disable them.

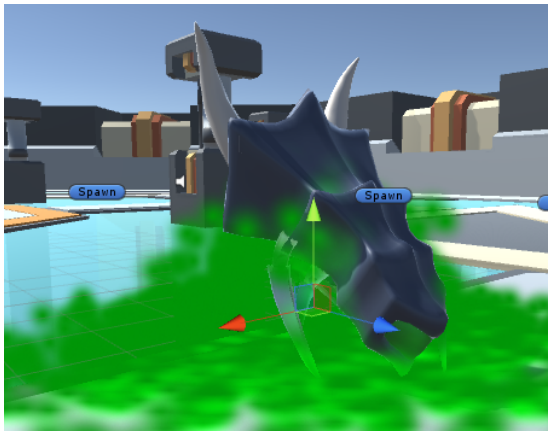
In the Hierarchy, select **AlienDeathParticles**. You'll see there are many options for particles.



Expand the **Collision** section by clicking on it — scroll down if you don't see it. Drag the **DeathFloor** to the **Planes** property.



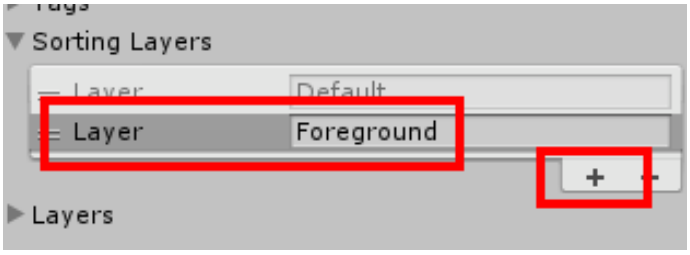
Click the **Simulate** button. The blood will pool on the new floor!



Now you have to address a small rendering issue. In this case, Unity renders the particles underneath the floor because it draws the floor *after* it draws the particles.

To avoid this, you need to assign the particles to a **Sorting Layer**, which will determine the order in which Unity renders objects. Typically, you use these with Unity 2D, but there are use cases for sorting layers in 3D games too.

Expand the **Renderer** section of the particle system. In the **Sorting Layer** property, select **Add Sorting Layer...**. The Inspector will change to **Tags & Layers**. Click the **plus** sign and name the sorting layer **Foreground**.



Return to the **Renderer** module, and in the **Sorting Layer**, select the **Foreground** property. Keep in mind you'll need to do the same thing for the space marine.

Now click the **Apply** button to apply the changes to the prefab.

You've unveiled a new problem: the death floor isn't added to the prefab because it's an instance in the scene and your alien prefabs don't know about it. You'll address this in a moment.

Meanwhile, give your hero some blood too. Drag the **MarineDeathParticles** prefab to be a child of the **BobbleMarine-Body**. Set **Position** to (**-1.68**, **5.29**, **0.63**) and **Rotation** to (**-90**, **0**, **0**).

In the Particle System, expand the **Collision** section and drag the **DeathFloor** to the **Planes** property. In the **Renderer** section, set the **Sorting Layer** to **Foreground**.

Click the **Simulate** button to see the marine gush like a fountain.

The particle systems are ready to go for the alien and space marine; you just need to activate them in code.

Activating particles in code

Unity allows for all sorts of adjustments to particle systems in code. In your case, you just need to know if the particle system is running or not.

In the Project view, select the **Scripts** folder and click the **Create** button. Choose **C# Script** and name it **DeathParticles**. Open the script in your editor.

Add the following instance variables:

```
private ParticleSystem deathParticles;
private bool didStart = false;
```

`deathParticles` refers to the current particle system and `didStart` lets you know the particle system has started to play.

You need to get a reference to the particle system, so add the following to `Start()`:

```
deathParticles = GetComponent<ParticleSystem>();
```

Add the following method:

```
public void Activate() {
  didStart = true;
  deathParticles.Play();
}
```

This starts in the particle system and informs the script that it started. In `Update()`, add the following:

```
if (didStart && deathParticles.isStopped) {
  Destroy(gameObject);
}
```

Once the particle system stops playing, the script deletes the death particles because they are meant to play only once.

Finally, to address the need for a collision plane that's set in code, add the following method:

```
public void SetDeathFloor(GameObject deathFloor) {
  if (deathParticles == null) {
    deathParticles = GetComponent<ParticleSystem>();
  }
  deathParticles.collision.SetPlane(0, deathFloor.transform);
}
```

The script first checks to see if a particle system has been loaded, which is necessary because the alien prefab is instanced and used immediately.

In case `Start()` hasn't been called and `deathParticles` isn't populated, this line populates it. Then it sets the collision plane.

Now to activate the particles — it's a similar process for the alien and the marine, but the alien requires a little set up work.

First, you need to get a reference to the particle system, and it's a child of the Alien GameObject, so you need code to get it.

Open **Alien** in your code editor, and add the following instance variable:

```
private DeathParticles deathParticles;
```

Then create the following method:

```
public DeathParticles GetDeathParticles() {
  if (deathParticles == null) {
    deathParticles = GetComponentInChildren<DeathParticles>();
  }
 return deathParticles;
}
```

The magic occurs in `GetComponentInChildren()`. It returns the first death particle script that it finds. There's only one script so there's no possibility of grabbing the wrong one.

In `Die()`, add the following before `Destroy(gameObject)`:

```
if (deathParticles) {
  deathParticles.transform.parent = null;
  deathParticles.Activate();
}
```

This makes the blood splatter when an alien dies. You have to remove the parent. Otherwise, the particles are destroyed along with the GameObject.

Open **GameManager** and add this instance variable:

```
public GameObject deathFloor;
```

This contains a reference to the death floor.

Add the following after `alienScript.OnDestroy.AddListener(AlienDestroyed);` in `Update()`:

```
alienScript.GetDeathParticles().SetDeathFloor(deathFloor);
```

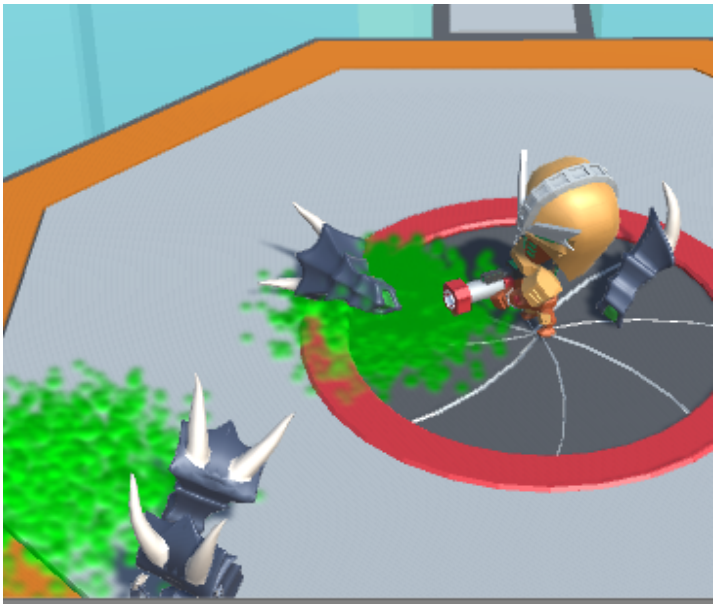
Finally **save** all your files and switch to Unity.

In the Project view, expand the Alien prefab and select the **AlienDeathParticles** GameObject. Click **Add Component** and under scripts, choose **DeathParticles**.

Next, select the **GameManager** in the Hierarchy and drag the **DeathFloor** to the **DeathFloor property** in the Inspector.

Delete the **Alien** from the hierarchy as that was only a temporary copy and you don't need it anymore.

Now play your game and shoot some aliens — you might need a mop.



Now for the marine. Select **MarineDeathParticles** in the Hierarchy and click **Add Component**. Under scripts, choose **DeathParticles**. Next, open the **PlayerController** in your editor.

Add the following instance variable:

```
private DeathParticles deathParticles;
```

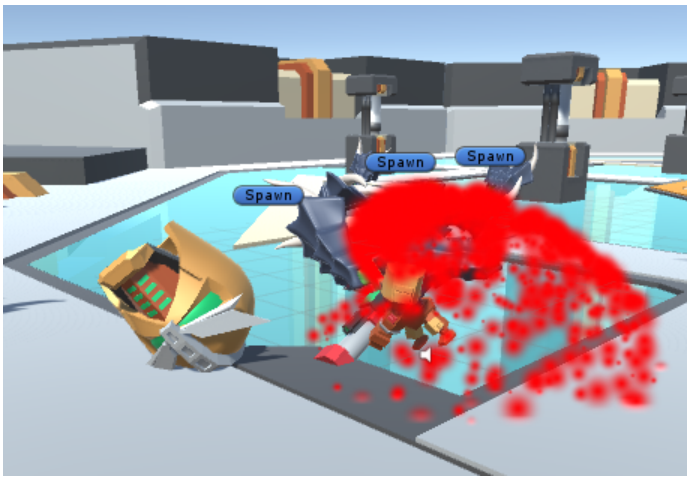
In `Start()`, add the following:

```
deathParticles =
  gameObject.GetComponentInChildren<DeathParticles>();
```

Finally, in `Die()`, add the following before `Destroy()`:

```
deathParticles.Activate();
```

Play your game and charge those aliens. You'll get a satisfying splatter when the marine dies.



Congratulations! You've lost the game!

Winning the game

Thankfully, you have an alternative to losing your head.

The winning sequence will run as follows:

- When the marine kills all the aliens, the hatch opens and the elevator rises.
- Once the space marine steps on the elevator, it ascends to the ceiling — perhaps to elevate the marine to an immortal or the next level. Who knows?

Take a deep breath, and grab your favorite caffeinated beverage — you're in the home stretch!

This section will be a great review for everything you've learned so far.

First, you need to create a collider for the elevator hatch so that it knows when the player steps on it.

In the Hierarchy, select the **BobbleArena** and click **Add Component**. From the Physics category, select **Sphere Collider**.

First, **check** the **Is Trigger** checkbox. Next set **Center** to (**0**, **0.22**, **0**) and **Radius** to **1.65**. Make sure to **uncheck the component** to make it inactive. It only needs to be active when the marine clears the arena.

Next, click **Add Component** and select **New Script**. Name it **Arena**, set the language to **C Sharp** and click **Create and Add**. Open the script in your code editor.

Add the following instance variables:

```
public GameObject player;
public Transform elevator;
private Animator arenaAnimator;
private SphereCollider sphereCollider;
```

You need each of these to win the game:

- You need to access both the `player` and the `elevator` to raise the marine to the top of the arena.
- The `arenaAnimator` will kick off the animation.
- The `sphereCollider` will initiate the entire sequence.

In `Start()`, add the following:

```
arenaAnimator = GetComponent<Animator>();
sphereCollider = GetComponent<SphereCollider>();
```

You should be used to this by now; in here, you’re getting access to the components.

There needs to be a chain of events to specify what happens after the marine steps onto the trigger.

You want the hero to ride the elevator towards the ceiling. Animating the player is one approach that might work, but it’s easier to make the marine a child of the elevator. As it goes up, the marine has no choice but to follow. Additionally, the camera will try to follow, so you need to disable it.

Add the following:

```
void OnTriggerEnter(Collider other) {
  Camera.main.transform.parent.gameObject.
```

```
      GetComponent<CameraMovement>().enabled = false;
    player.transform.parent = elevator.transform;
  }
```

The first line gets the camera then disables the movement. Then the player is made into a child of the platform.

You need to disable the player's ability to control the marine. Add the following to `OnTriggerEnter()`:

```
  player.GetComponent<PlayerController>().enabled = false;
```

This stops the player from turning, shooting or doing anything with the marine.

Now you need an audio cue to alert the player to the elevator's arrival:

```
  SoundManager.Instance.PlayOneShot(SoundManager.Instance.
    elevatorArrived);
```

You also need to kick off the animation, so add this line after the previous code you entered:

```
  arenaAnimator.SetBool("OnElevator", true);
```

This will start the animation state and work when the player enters the sphere collider, but remember that the collider is disabled. It should be active when the platform arrives but not before. **Save** and switch to Unity.

Select **BobbleArena** in the Hierarchy. In the Inspector, drag the **SpaceMarine** to the **player** property. Expand the **BobbleArena** and look for a child GameObject named **Elevator_Platform**. Drag it to the **Elevator** property in BobbbleArena.

You're down to the last step of calling your code. Thankfully, animation events can do this. How cool is that?

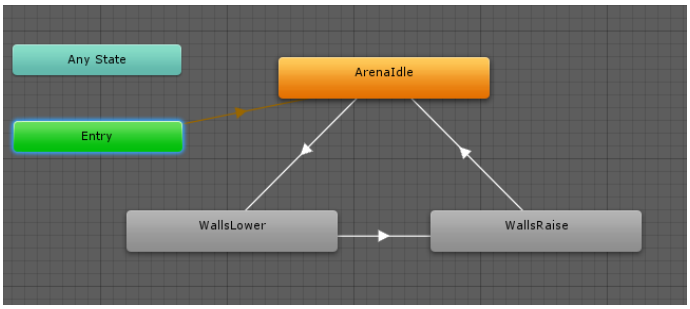
Animation events

An animation event is very much like a keyframe. You determine a point in the animation at which to run the code, and then you assign it a method. This approach allows precision control over when the code runs.

> **Note**: Remember, you can animate all properties, including the collider you've enabled.

You imported the animations for summoning and rising in Chapter 1 and just need to associate them with the Animator.

With **BobbleArena** selected in the Hierarchy, open the **Animator** window. You'll see the wall states you set up in Chapter 6, "Animation".

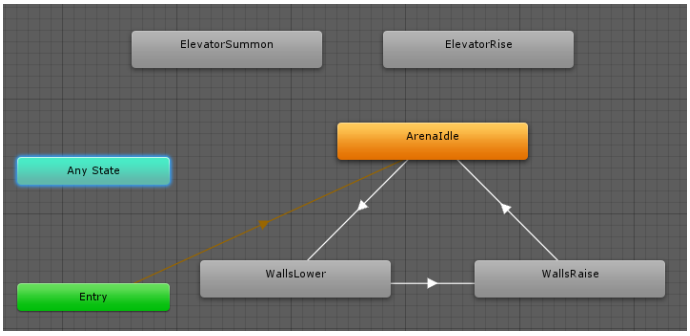


You need a new state to summon the elevator, and it should be accessible from all the states because a win may occur at any time. You could create a transition from each state, but Unity has you covered with the "Any State" state.

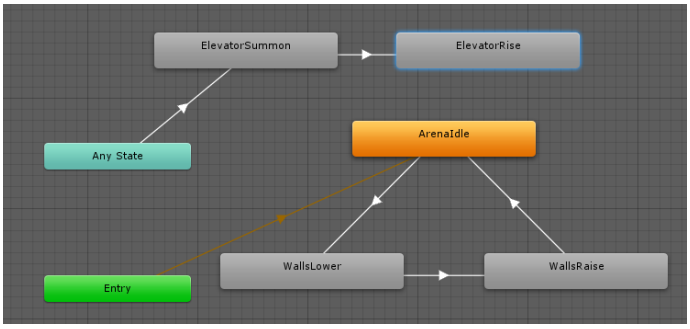
In the Animator, **right-click** anywhere and select **Create State\Empty**. Select **New State**, and name it **ElevatorSummon** in the Inspector. Set **Motion** to be the **ElevatorSummon** animation clip and **Speed** to `0.5` to slow the elevator to half its normal speed. As-is, it runs a little fast and stops before its audio clip ends. Slowing it down keeps the elevator in sync with its audio.

Create another empty state, call it **ElevatorRise** and assign the **ElevatorRise** animation clip to it.

The animator should look as follows:



Create a transition from **AnyState** to **ElevatorSummon** and another from **ElevatorSummon** to **ElevatorRise**. It should look like the following:



Now you need a condition. Click the **parameters** tab then **+**, choose **Trigger** and name it **PlayerWon**.

Select the transition from **AnyState to ElevatorSummon**. In the Inspector, add a condition and set it to **PlayerWon**. You don't have to set a value for the condition because it's a trigger.

Why use a trigger? By creating a transition from Any State, then any state can reach the condition, including the destination of that transition.

If you were to use a boolean condition like you did before, once you transition from Any State to ElevatorSummon, the animator would repeat the transition until the bool value switches to false. It would creates an infinite loop, but using a trigger means the transition only occurs once.

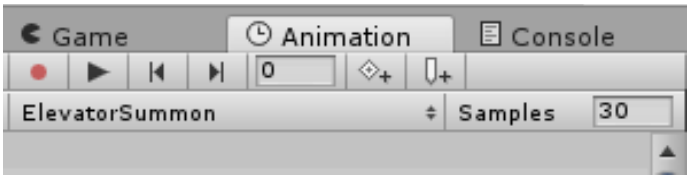
Next, create a new **Bool** parameter and name it **OnElevator**.

Select the transition between **ElevatorSummon** and **ElevatorRise**. In the Inspector, **uncheck Has Exit Time** and give it a new condition: **OnElevator** is equal to **true**.

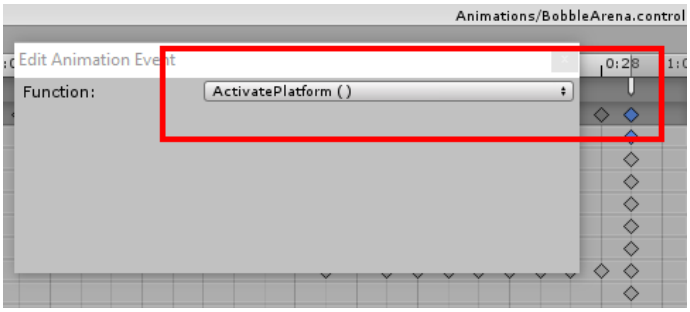
Open **Arena** in your code editor and add the following method:

```
public void ActivatePlatform() {
  sphereCollider.enabled = true;
}
```

Save and switch back to Unity. Select **BobbleArena** in the Hierarchy and open the **Animation** window. From the animation drop-down, select **Elevator Summon**.



Over the last keyframe (which is at 0:28), **right-click** the gray bar above the frame counters and select **Add Animation Event**. You'll see a white pip appear over that frame, and this is where you call your code. You'll also see a dialog. From the drop-down, select **ActivatePlatform()**.



All the public methods attached to the current GameObject show in this dialog. Remember, you just wrote `ActivatePlatform` to enable the sphere collider on top of the elevator once the player has won the game, so you can detect when the player enters that area.

You're down to writing your last bit of code for Bobblehead Wars! Whoa!

Open the **GameManager** in your code editor and add a public reference for the arena animator:

```
public Animator arenaAnimator;
```

Add the following to initiate the game's end:

```
private void endGame() {
  SoundManager.Instance.PlayOneShot(SoundManager.Instance.
    elevatorArrived);
  arenaAnimator.SetTrigger("PlayerWon");
}
```

This kicks off the entire process and you just need to call it. In `AlienDestroyed()`, add the following before the closing brace:

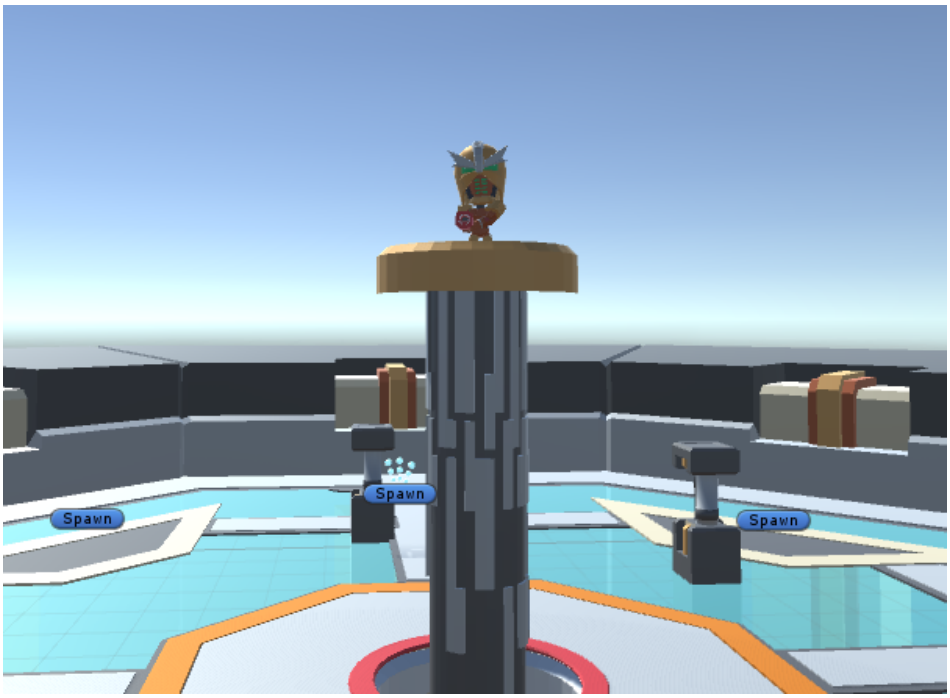
```
if (totalAliens == 0) {
  Invoke("endGame", 2.0f);
}
```

You've done a lot here, so let's review how the entire end-game process works:

1. Once the count of aliens hits 0, you schedule the `endGame()` method to be called after 2 seconds.
2. This fires the "PlayerWon" trigger on the arena animator.
3. Once the arena animator detects this trigger, it plays the elevator summon animation.
4. You configured the elevator summon animation to fire an animation event when the animation completes, to call the `ActivatePlatform()` method.
5. This enables a collider on the elevator.
6. When the player hits the collider, `OnTriggerEnter()` is called on the arena. This attaches the player to the elevator and sets the "OnElevator" boolean on the arena animator.
7. This causes the arena animator to play the elevator rise animation.
8. Phew! If you understand this sequence, pat yourself on the back - you've learned a lot about Unity!

Save the code and head back to Unity. Select **GameManager** in the Hierarchy and drag the **BobbleArena** to the **arenaAnimator** property.

Now for the final play test. Kill all the aliens then step on the elevator. Victory is yours!



Where to go from here?

Congratulations on completing your first Unity game! It's no small feat, but now you can see that building games is fun and probably easier than you thought when you decided to pick up this book.

In the process of finishing Bobblehead Wars, you learned some new concepts, including:

- **Unity Events** and how to use them inside your code.
- **Particle System** and how it can make some cool effects.
- **Animation Events** and how they can be used to call code from your animation clips.

Believe it or not, you've barely explored Unity's capabilities and have a lot more to look forward to in this book.

In the next section, you'll put your skills to use to create a first-person shooter that features some rather angry robots. Pick up your gun and get blasting. You're in for a fun ride!

Conclusion

We hope you've enjoyed this book as much as we enjoyed making it! Unity is an amazing platform for building games that look incredible and are a joy to play. We can't wait to see what you'll produce on your own, now that you're armed with all the information you covered in this book.

If you have any questions or comments as you continue to develop with Unity, please stop by our forums at http://www.raywenderlich.com/forums and share your thoughts.

This is only the first part of a book that contains several other sections. If you are interested in learning more, head over to:

http://www.raywenderlich.com/store/unity-games-by-tutorials

Brian