Up to date for Unity 2018.1

BEAT 'EM UP GAME STARTER KIT

By Jonathan Simbahan

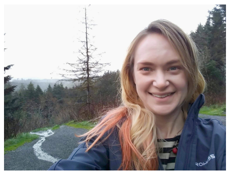
Beat ’Em Up Game Starter Kit

By Jonathan Simbahan

About the team

Jonathan Simbahan is the author of this book. Jonathan is a Philippines-based game programmer with a curious mind on a quest to make enjoyable games. Outside of game development, he finds simple joys in his numerous hobbies which are usually food-related.

Eric Van de Kerckhove is the technical editor of this book. Eric is a Belgian hobbyist game dev and has been so for more than 15 years. He has made a fair share of games over the years, mostly free ones for fun and as learning experiences. Eric leads the Unity team at raywenderlich.com.



Wendy Lincoln is the editor of this book. Wendy is a technical marketing project manager who likes words, games, and helping developers create really awesome tutorials. When she’s not working, you can find her on the beach or geeking out on technology.

Allen Tan is the final pass editor of this book. Allen is a game designer and developer who finds joy in the quirky and bizarre. He turned his passion for video games into a career by founding Monstronauts, an indie game studio from the Philippines. Together, he and his team set out to show the world their own brand of fun, one game at a time.

We’d also like to acknowledge the efforts of an additional individual who helped out with the book:

Maria Gelyn Lopez assisted with the writing portion of this book. Maria Gelyn works on an Enterprise Integrated System for a manufacturing company. She enjoys helping people choose and use IT solutions.

About the artist

Hunter Russell is a pixel artist and animator with experience on games like Duelyst and Ikenfell. Have a look at her portfolio at https://www.hrpixelart.com and her Twitter at https://twitter.com/_bigjerk, or contact her via email at hunter@hrpixelart.com..

Table of Contents: Overview

Table of Contents: Extended

Book license

Book source code and forums

This book comes with the source code for the starter and completed projects for each chapter. These resources are shipped with the digital edition you downloaded from store.raywenderlich.com.

We've also set up an official forum for the book at forums.raywenderlich.com. This is a great place to ask questions about the book or to submit any errors you may find.

Book updates

Since you’ve purchased the digital edition version of this book, you get free access to any updates we may make to the book!

The best way to get update notifications is to sign up for our monthly newsletter. This includes a list of the tutorials that came out on raywenderlich.com that month, any important news like book updates or new books, and a list of our favorite iOS development links for that month. You can sign up here:

- www.raywenderlich.com/newsletter

Introduction

What could possibly be a more amusing way to burn time than slaughtering a horde of bad guys with trusty right and left hooks? Creating your very own beat ‘em up game, of course!

Beat ‘em up games have been around since the inception of 8-bit games and experienced their glory days when early console gaming and arcade gaming were all the rage — long before the world turned to pixels in the early part of the 21st century. :]

Although they’re less popular today, you can’t deny that beat ‘em ups follow a winning format. The premise is straightforward: walk toward one side of the screen and punch the living daylights out of anything that moves. Games in this format are typically side-scrolling and two-dimensional.

A few classic beat ‘em ups have stood the test of time: *Double Dragon*, *Teenage Mutant Ninja Turtles*, *Golden Axe* and *Streets of Rage*.

Some games have managed to appeal to sense of nostalgia while employing modern game technologies like player progression or a touch-based interface. *Castle Crashers* and *Scott Pilgrim vs. the World* are good examples.

Seeing as how you have this starter kit in hand, I can deduce that you’re no stranger to the genre and have played most, if not all, of the aforementioned games. No wonder you picked up this book!

Let me take this opportunity to say *thank you* for keeping the beat ‘em up spirit alive and supporting raywenderlich.com. I sincerely hope that with the help of this starter kit, you’ll make a “knockout” game that *I* can play one day.

Want to see what it will look like? Come closer! Note the mobile controls, colored sprites, destructible objects, weapons, hit effects, HUD, etc.

What is a starter kit?

In the third edition of this popular book, we're walking you through building the game on Unity.

Note: Previous versions of this book were written for cocos2D and SpriteKit; at the time, these were popular for creating 2D games.

Once Unity released 2D tools, it quickly became *the* game creation engine because developers can create a game once and build it for a number of platforms. The dark days of building for one platform, then painstakingly building your ingenious game on another engine for a different platform are over!

More than a book, this full-flavored starter kit equips you with tools, assets, fresh starter projects for each chapter, and step-by-step instructions to create an addictive beat ‘em up game for Android and iOS. In addition to walking you through processes for building the game, you'll receive an intimate backstage tour of Unity!

Our objective is for you to walk away with enough knowledge of Unity so that you can start creating games on your own.

Each chapter builds on the last. You build out features one at a time, allowing you to learn at a steady, logical, and fun pace.

Components were designed with reusability in mind so you can easily customize the resulting game.

In the spirit of Unity’s motto “Build once, deploy anywhere”, you’ll be able to run the game on Android and iOS platforms.

We hope you enjoy this Unity edition as much as we had fun creating it, and thank you again for purchasing the Beat ‘Em Up Game Starter Kit Unity Edition!

Who should have this starter kit?

This starter kit is for beginner to intermediate developers who have at least poked around Unity, perhaps even built a game, but need guidance around how to create a beat ‘em up game.

You’ll get most from this book if you’ve fulfilled the prerequisites below, or have equivalent experience and pick up new IDEs and languages quickly.

However, if you’re a complete n00b with Unity or programming, it’s possible to follow along with this book, but you may experience a steep learning curve. There are many steps ahead, so don’t rush and do not expect perfection from yourself the first time through. Follow the steps carefully, copy and paste the provided code, and check your work against the screenshots for the smoothest learning experience.

Regardless of your skill level, remember that if you get stuck, you can revert to the starter kit for the current chapter, or pull up the final kit and run diff checks to identify where things went wrong.

Prerequisites

This book assumes you have a few programs installed and some skills before you dig into the first chapter:

- A computer — any operating system will do — with the latest version of Unity installed.
- Familiarity with Unity and the C# programming language

For the last chapter only:

- The Android SDK installed on your computer
- An Android device

- A computer running macOS with Xcode installed
- An iOS device
- Apple developer account

Note: If you are unfamiliar with Unity or C#, I recommend going through both parts of the “Introduction to Unity: Getting Started” tutorial:

Part 1: https://www.raywenderlich.com/147687/introduction-unity-getting-started-part-12

Part 2: https://www.raywenderlich.com/149036/introduction-unity-getting-started-part-22

There are a number of other Unity tutorials, including several on C#, available on our website if you want to dive deeper: https://www.raywenderlich.com/category/unity.

Lastly, if you don’t want to go through the hassle of getting the SDKs or devices, don’t worry about it! The book can be completed without testing on a mobile device — just skip the last chapter.

How to use this starter kit

There are several ways you can use the Beat ‘Em Up Game Starter Kit:

- First, you can look through the complete sample project and begin using it right away. You can modify it to make your own game or pull out snippets of code you find useful for your own project. As you look through the project, you can flip through the chapters and read up on any sections of code you’re not sure about. The beginning of this guide has a table of contents that can help, and the search tool is your friend!
- A second way to use the starter kit is to go through the chapters one by one and build the game from scratch. This is the best way to use this book because you’ll literally write each line of the main gameplay code.
- Third, you don’t necessarily have to do each chapter—each chapter includes a starting and final project, so you can skip around with confidence. You’ll find the final projects helpful for debugging and as points of reference for what it “should” look like.

Chapter overview

You start with nothing and build the game from the ground up! Here's what you can look forward to in each chapter:

1. Getting Started

In this first chapter, you'll take a crash course on Unity and get started with creating your Unity Project and main menu scene.

While creating the main menu for the game, you'll get your first taste of Unity and its interface. You'll also create a button and script that navigate to your second scene, the game scene.

2. Working with Tilemaps

This chapter is all about tile maps: smaller repeating sets of sprites to form up, a bigger, more elaborate picture. With this, you'll create the retro background sprites for your game. You'll also learn how sprites are drawn in the scene and you'll understand a key feature of Unity: prefabs! These allow you to save GameObjects, allowing you to create objects on demand.

3. Walk This Way

This chapter is all about setting up the game scene: implementing cool techniques that allows you to create a "2.5 dimensional" game with the engine. Being the cunning developer that you are, you'll accomplish this by rotating all assets in the game.

Don't forget about the hero! Introducing the Pompadoured Hero, the protagonist of the game! You'll learn how to process user input, animate a 2D character and create bounds to keep objects, especially the hero, within view! With the help of the keyboard inputs, you'll create a hero with stunning hair.

4. Running, Jumping and Punching

This chapter creates new actions for the hero, empowering him to be a genuine, bot-crushing warrior. You'll begin by simulating a directional double-tap gesture to make the hero run.

Building on your virtual control scheme, you'll create and add two custom buttons to make the hero perform other actions, such as jabbing. Then you'll enhance the 3D feel of the game with shadows and jumping.

Aside from new actions, you'll also learn an important process: refactoring! You'll re-organize the hero script, resulting in cleaner code that is easier to reuse and understand.

5. Bring on the Droids

In this chapter, you'll spawn an enemy robot for the hero to fight. You'll give each rust-bucket a special animation that will play when hit, and a life counter that will knock out the robot. You'll also work with the layering of physics objects and collision handling, which will let the hero beat up some helpless robots until they die. Poor, poor robots.

6. Brainy Bots

It's the revenge of the fallen! In this chapter, you'll develop artificial intelligence for the robots, implementing a system of weighted decisions that will determine each robot's action. This time, the robots won't stand idly by while the hero pummels them—they will aggressively fight back.

7. Pompadroid's Playbill

In this chapter, you'll enable your game to handle battle events: a situation requiring the hero to fight enemies until no enemy is left standing! You'll also control and define enemy spawns using ScriptableObjects, creating levels for the game.

Additionally, you'll enhance the game, and the hero's ego, by adding entrance and exit animations whenever a level starts or ends.

8. Power Attacks

Naturally, you'll want to trick out your hero with flashy moves. In this chapter, you'll dig deep and learn to extend the current actions system by giving the hero the ability to do a chained attack with a 1-2-3 punch. You'll also give him combination actions, such as a jump punch and a running kick.

At this point, the game will need a few swipes of polish, so you'll fix a few bugs and add a hit effect for everything that punches in the game. Additionally, you'll add logic so the hero gets knocked out when he takes too much damage!

9. Heads-up Display

You'll begin the chapter by creating a heads-up display (HUD) to keep the player informed of critical stats, such as hit points and when to move forward on the tile map. Then, you'll further polish the game with floating damage numbers and explosive hit animations for each attack.

Lastly, you'll add on-screen controls to enable the game for mobile devices: a directional pad (d-pad), as well as attack and jump buttons.

10. Big Bad Boss and Powerups

Boss characters are essential. Bigger, badder and more barbaric than the other antagonists, they are a hero's most formidable enemy. In this case, the boss will be an ape-like monster with a high-top fade. You do not want to get in its way!

To give the hero a fighting chance, you'll also add a powerup to the game: the Punch-a-tron Operational Work-gloves 300, or P0W 300. This glorious pair of mitts allow the hero to punch harder, giving him extra "oomph" when fighting the boss. You'll also create trash bin containers; when punched, they drop these mighty gloves!

11. Audio and Final Polishes

It's the homestretch! In this chapter, you'll add the final piece: audio. You'll add a catchy background theme song, along with hit effects and death sounds for all actors in the game.

To get the game ready for release, you'll fix bugs as well. By the end of this chapter, you'll have a complete and polished beat 'em up game!

12. Running on Mobile Devices

The game will be done at this point, but one final task remains! In this chapter, you'll run PompaDroid on a mobile device! You'll add platform-specific controls for Android, and create builds for Android and iOS to show to your friends!

Acknowledgements

I would like to thank several people for their assistance in making this starter kit possible:

- To Ray and the tutorial team: Thanks for the great tutorials that helped me in learning programming for iOS.
- To my family, for their patience and support in creating this book.
- To my colleagues at Monstronauts, who have motivated me in completing this book.
- To **Jeng**: for her patience and help editing this project. Without her, this book wouldn’t be as good as it is.
- And most importantly, the readers of raywenderlich.com and you! Thank you so much for reading our site and purchasing this starter kit. Your continued readership and support is what makes this all possible.

Chapter 1: Getting Started

Welcome to the first chapter, where you'll start your journey in the same way you start a game — through the title screen!

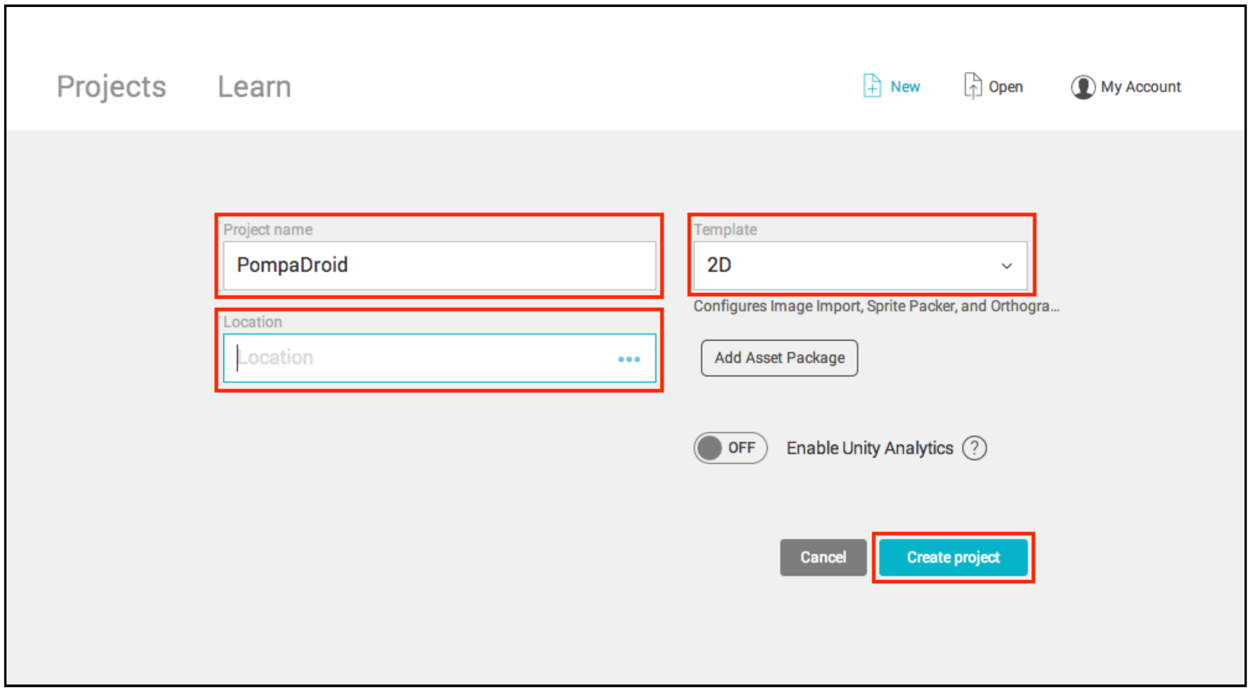
As you assemble the first bits of your game's UI, you'll go through the basics of Unity and familiarize yourself with the Unity editor. By the end of it, you'll be well on your way to making your very own **beat 'em up** game.

What are you waiting for? You've got games to build and things to beat up! Get to it!

Note: If you're already familiar with Unity, feel free to skip ahead a bit to the section *Creating the title Scene*. On the other hand, if you feel a little rusty or you're new to Unity, start from the top!

Creating a Unity project

Start up Unity and click **New**. You'll see the following window:



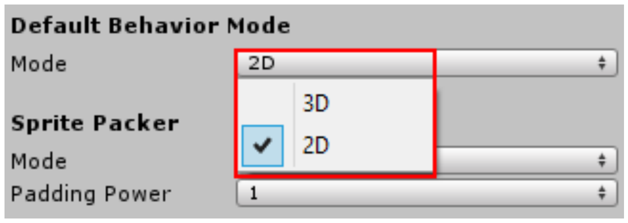
Project name: PompaDroid
Location: Location
Template: 2D
Configures Image Import, Sprite Packer, and Orthogra...

- Enter **PompaDroid** as the **Project Name**, and then select the **Location** to where you want to save the project.

Select **2D** as the **Template**. Finally, click the **Create Project** button.

Just like that, you created your very first Unity project. Since you went with 2D settings, your images will all import as 2D sprites instead of textures.

If you feel compelled to change this, you can do so under **Edit\Project Settings\Editor**. It is labeled as **Default Behaviour Mode**.



Note: You're probably wondering why you named the game PompaDroid. As you'll soon see, the main character has a funky pompadour hairdo, and he's about to beat up on a bunch of droids. My apologies in advance to the Android devs out there.

With the Unity editor now open, first things first: you need to set the target platform.

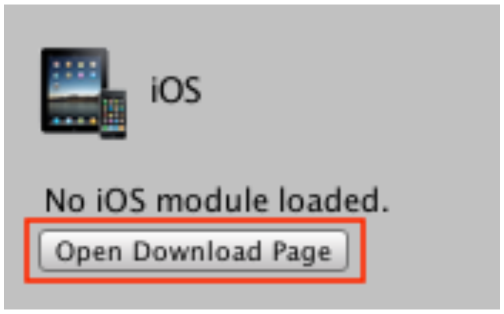
On the menu at the top, select **File\Build Settings**. Select **iOS** in the Platform section.

At the bottom of the window, click **Switch Platform**, and then check **Development Build**.

Close the window.

You've just set up your game to run on iOS devices. If you prefer to build for Android, just do the same thing but select Android instead of iOS. It's that simple to build the same game for a different platform!

Note: If you encounter a "no module loaded" error for the desired platform, just download the module by clicking **Open Download Page**. Go through the steps to get the install process going, then come back to set up your folders and go through the crash course. When installation finishes, you can just circle back to finish setting up the platform.



No iOS module loaded.

Clearing Starter Files

Depending on which Unity version you are using, starting a new project may or may not include starter files for you to use. These files are not necessary for PompaDroid. If your starter project is not empty, the following steps will help you delete all of the unused assets.

Create a new scene by choosing **File\New Scene** in the top menu. Once a new Scene has been loaded, select all files inside the Assets folder in the Project view, and right-click **Delete** to remove them from the project.

Setting up the Project Folders

Now is an excellent time to think ahead and set up a system for your project files — orderly assets are easier to find.

In the **Project window**, right-click the **Assets** folder and select **Create\Folder**. Name this folder **Scenes**.

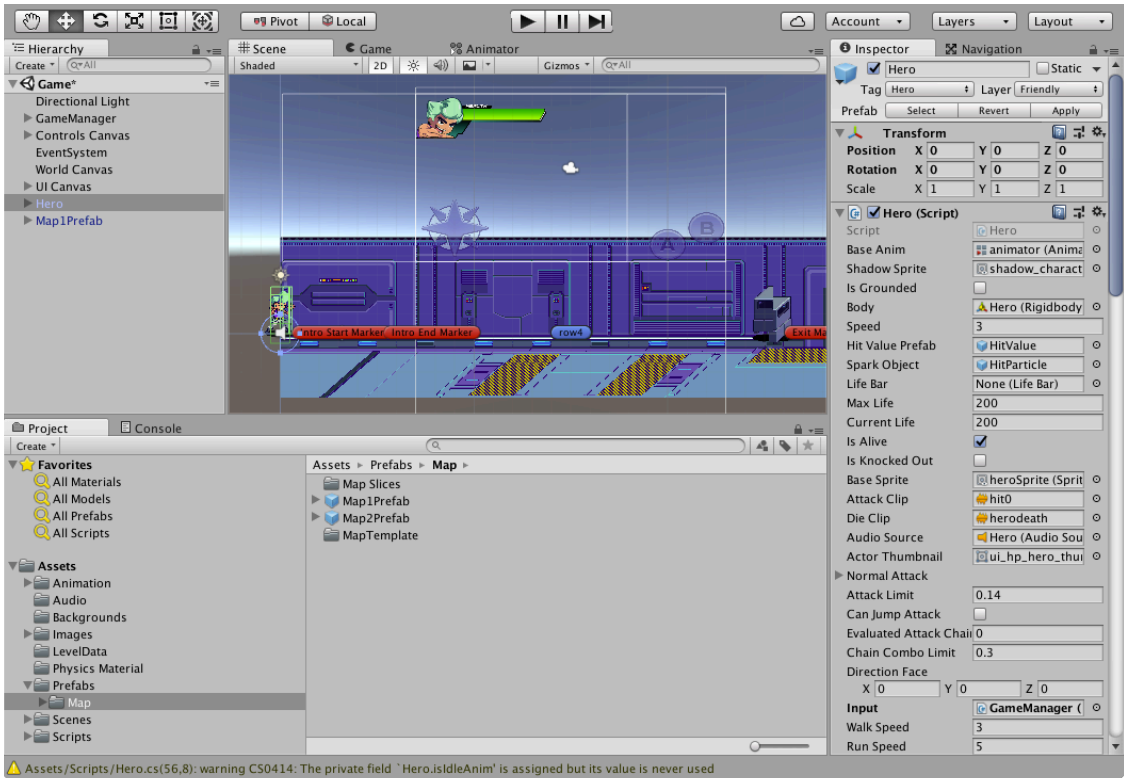
Repeat these steps and create **Images**, **Prefabs** and **Scripts** folders.

No surprises here — you'll save your various assets in these folders.

Unity crash course

You're at the point where it's time for a Unity editor crash course. Unity veterans may want to skip this part.

Note: If you want a not-so-quick explanation about using the Unity editor, you can read more about it at http://docs.unity3d.com/Manual/UnityOverview.html.



Hero   Static
Tag Hero   Layer Friendly
Prefab Select Revert Apply

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

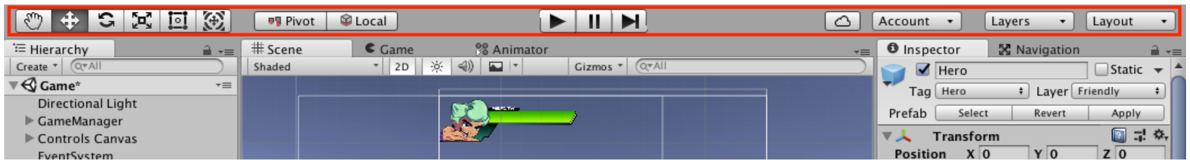
Hero (Script)
Script: Hero
Base Anim: animator (Anima
Shadow Sprite: shadow_charact
Is Grounded: ☐
Body: Hero (Rigidbody
Speed: 3
Hit Value Prefab: HitValue
Spark Object: HitParticle
Life Bar: None (Life Bar)
Max Life: 200
Current Life: 200
Is Alive: ✓
Is Knocked Out: ☐
Base Sprite: heroSprite (Sprit
Attack Clip: hit0
Die Clip: herodeath
Audio Source: Hero (Audio Sou
Actor Thumbnail: ui_hp_hero_thu
Normal Attack
Attack Limit: 0.14
Can Jump Attack: ☐
Evaluated Attack Chai: 0
Chain Combo Limit: 0.3
Direction Face: X 0 Y 0 Z 0
Input: GameManager (
Walk Speed: 3
Run Speed: 5

```
Assets/Scripts/Hero.cs(56,8): warning CS0414: The private field `Hero.isIdleAnim' is assigned but its value is never used
```

The Unity editor comprises several windows. Except for the toolbar, all windows can be detached, docked, grouped and resized. Go ahead, try dragging and dropping things to see what you're working with here.

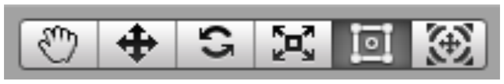
Its interface allows you to create a variety of layouts that suit personal and project needs, so your layout may look unique from other developers' layouts.

Toolbar



The toolbar contains essential tools you need for manipulating GameObjects.

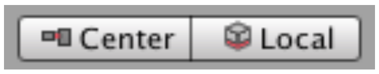
Transform tools



These tools let you manipulate the Scene view and its GameObjects. From left to right, these are:

- **Grab tool:** Allows you to pan around the Scene view.
- **Translate tool:** Used for moving GameObjects.
- **Rotate tool:** Allows you to rotate GameObjects.
- **Scale tool:** Used to scale GameObjects.
- **Rect tool:** Allows you to manipulate 2D elements in Unity (Sprites and UI).
- **Transform tool:** Allows you to move, rotate and scale using only one tool.

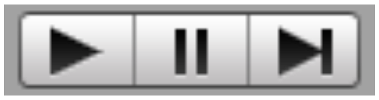
Transform gizmo toggles



Toggles are what you use to change how transform tools affect GameObjects. From left to right, these are:

- **Pivot Toggle:** Toggles whether transforms happen from the center of the GameObjects or around their pivot points. It’s useful when rotating GameObjects.
- **World Space & Local Space Toggle:** Toggles whether the transforms should work on world or local space.

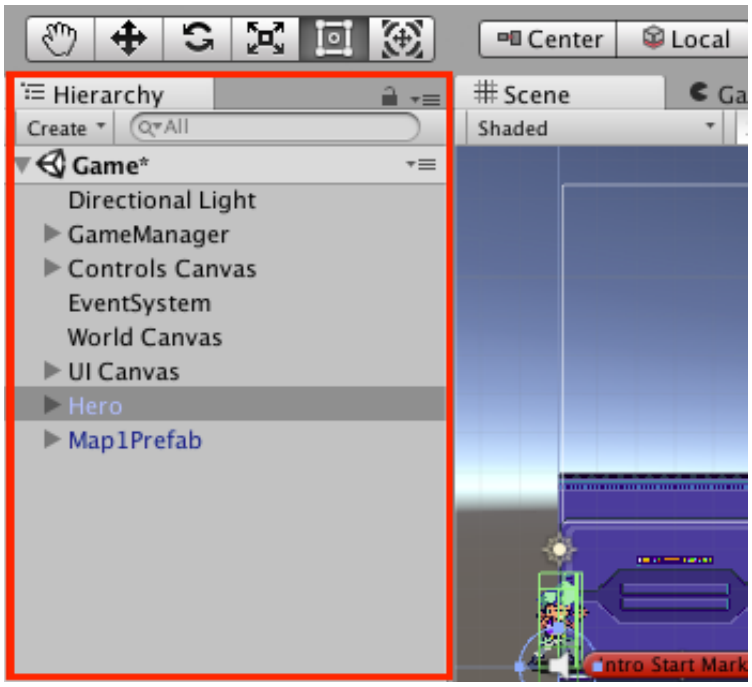
Playback buttons



These buttons allow you to run and test your game. From left to right, these are:

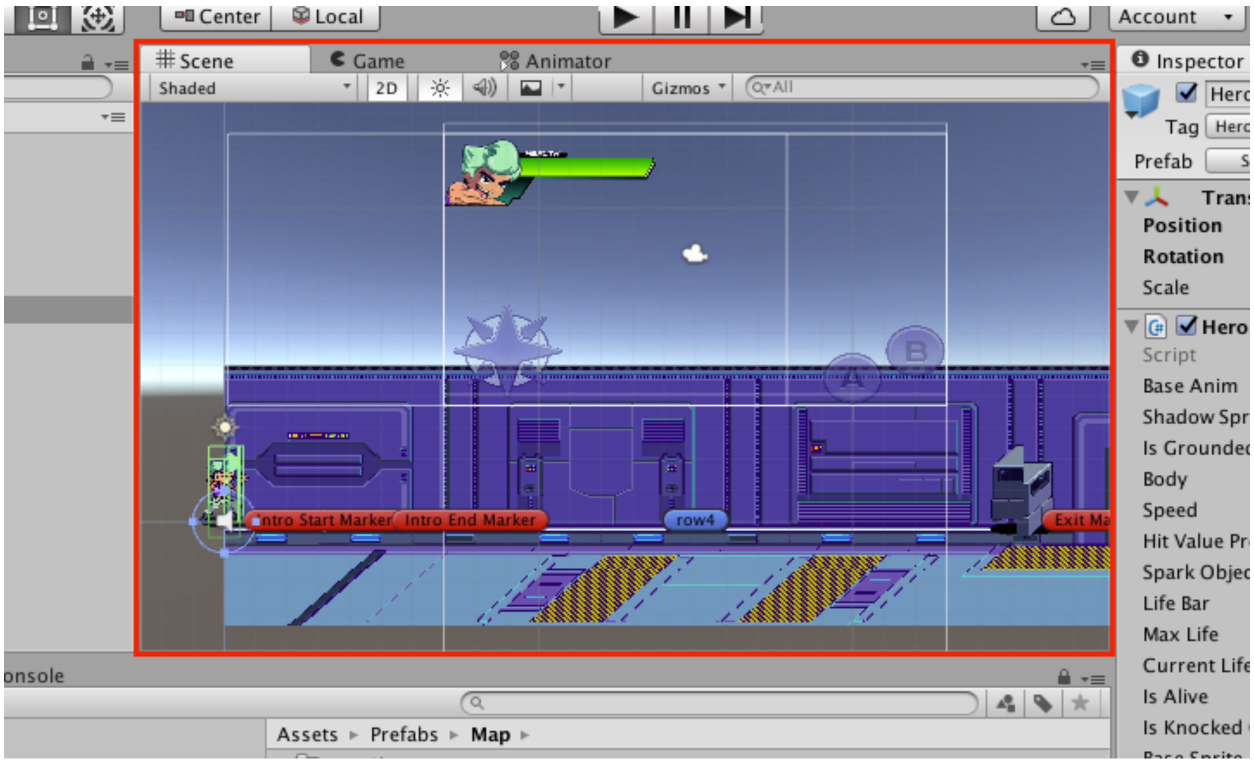
- **Play button:** Runs the current scene.
- **Pause button:** Pauses and resumes the game.
- **Step button:** Allows you to jump forward by one frame when the game is paused. It’s useful for hunting down pesky bugs.

Hierarchy



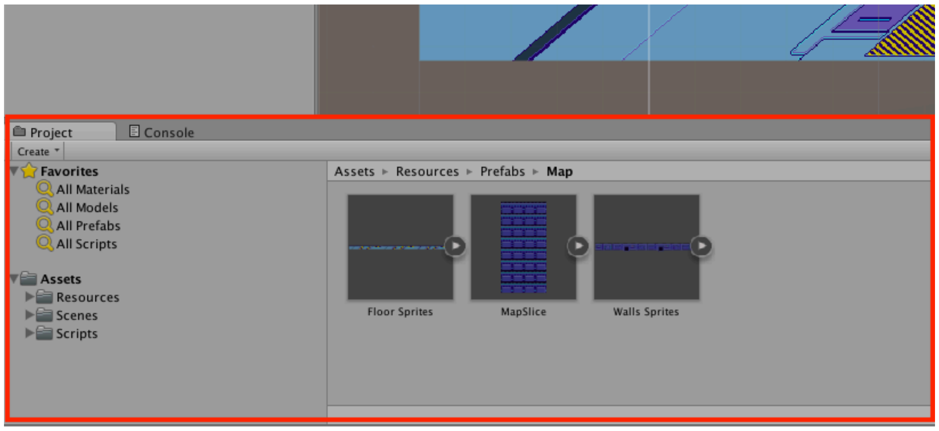
The Hierarchy is a complete list of all GameObjects in your current scene.

Scene view



The Scene view is a viewing window into the world you're creating. You'll be able to select, position and manipulate GameObjects here.

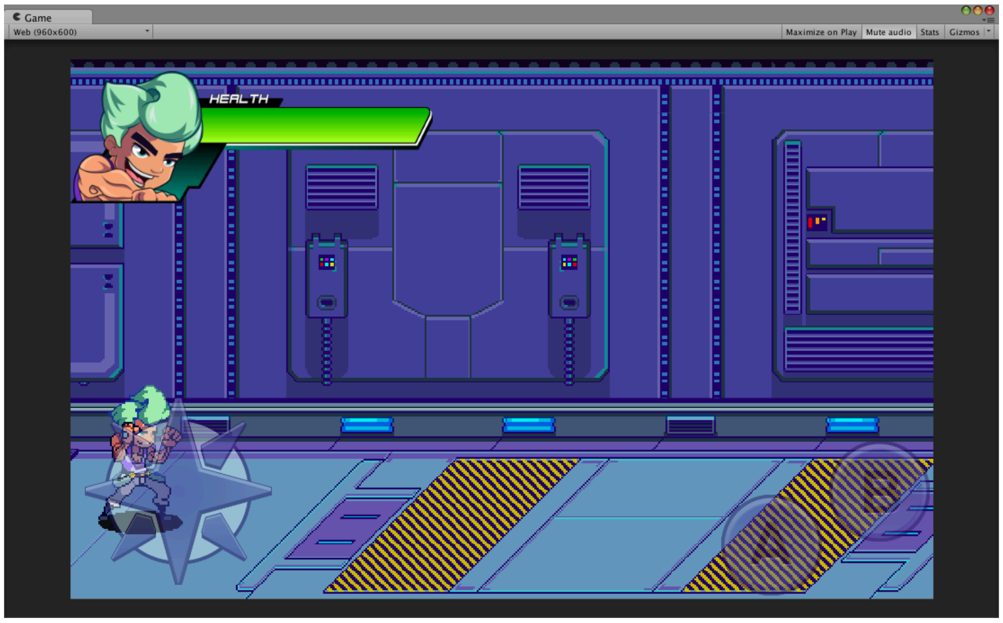
Project window



The Project window, sometimes referred to as the Project browser, contains all the assets that belong to your project. You can add, delete, search, rename and move assets here.

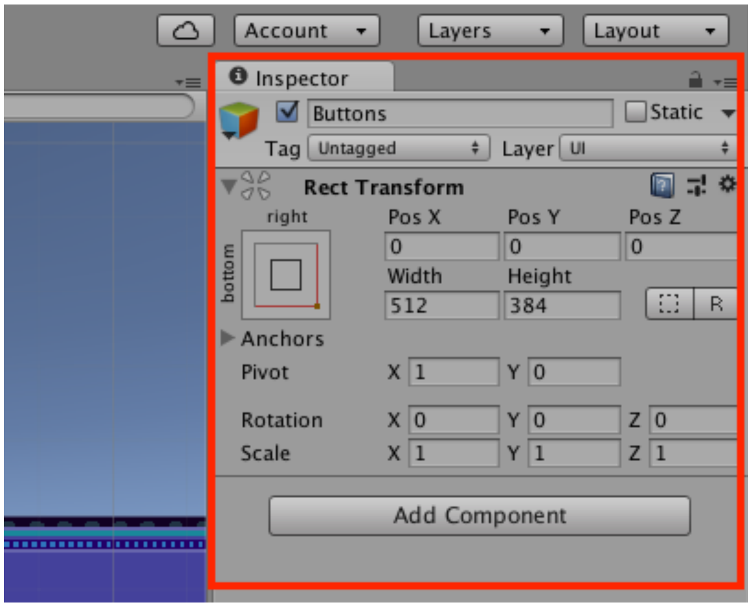
- The left-hand side shows your project's folder structure.
- The right-hand side shows all the assets contained in the folder you've selected on the left.

Game view



The Game view renders what the camera(s) sees. It's a decent representative of the final product.

Inspector window



Buttons
Static
Tag Untagged
Layer UI
right
bottom
Pos X 0
Pos Y 0
Pos Z 0
Width 512
Height 384
Pivot X 1 Y 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

The Inspector window is for viewing the GameObjects' properties, assets and other preferences and settings.

Basic Unity concepts

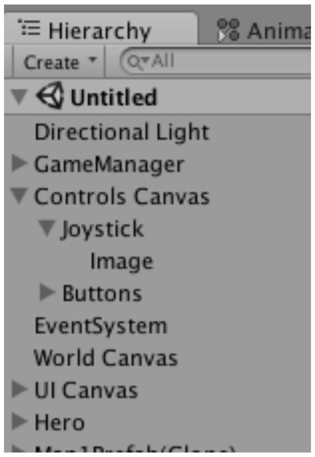
You have a basic understanding of how to get around in the Unity editor. Now get ready to absorb a few more core concepts that'll help you make the most of this book.

GameObjects

Meet *the* fundamental objects in Unity. Without them, you have…nothing.

They are the "things" that make up a game — literally what the name implies: objects in your game. They can be trees, lights, floors, cameras, a ball, a car, a slice of bacon, a hot buttered waffle, etc. (Hungry yet?)

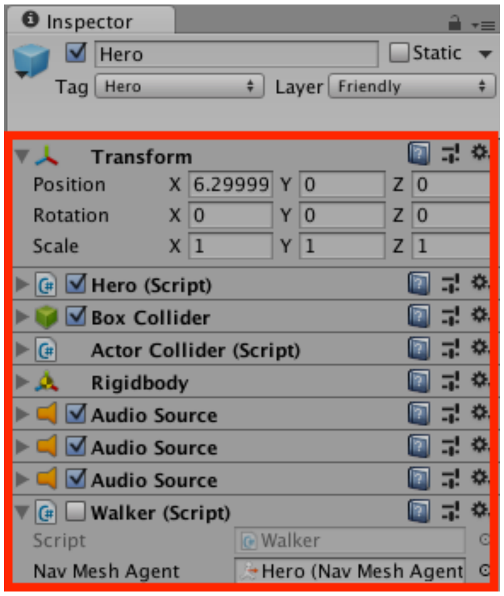
The Hierarchy window lists GameObjects used in the current scene.



GameObjects are basically containers that can contain components, which are building blocks that define a GameObject's capabilities. Components allow GameObjects to display images, play audio, think with an AI, handle physics, display 3D meshes and so much more!

Unity comes with many built-in components, but you can (and will) create some of your own by scripting — more on that later.

The following shows all the components that belong to the GameObject named Hero.


Inspector

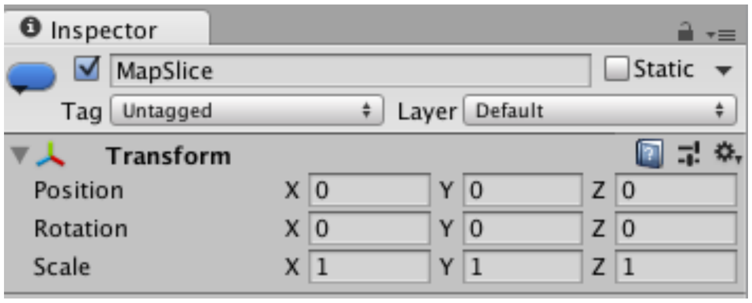
Hero ☐ Static

Tag Hero Layer Friendly

Transform
Position X 6.29999 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

☑ Hero (Script)
☑ Box Collider
Actor Collider (Script)
Rigidbody
☑ Audio Source
☑ Audio Source
☑ Audio Source
☐ Walker (Script)
Script Walker
Nav Mesh Agent Hero (Nav Mesh Agent

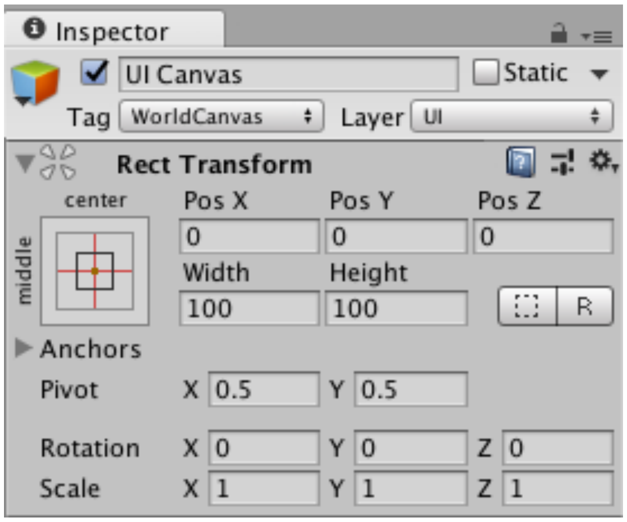
A GameObject doesn't fly solo; it always has a **transform** component that determines the GameObject's position, rotation and scale.



MapSlice

	X	Y	Z
Tag	Untagged	Layer	Default
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

UI elements (still GameObjects) have a much more complicated transform called **RectTransform**. You'll learn about them in a later chapter.



UI Canvas

Tag: WorldCanvas, Layer: UI

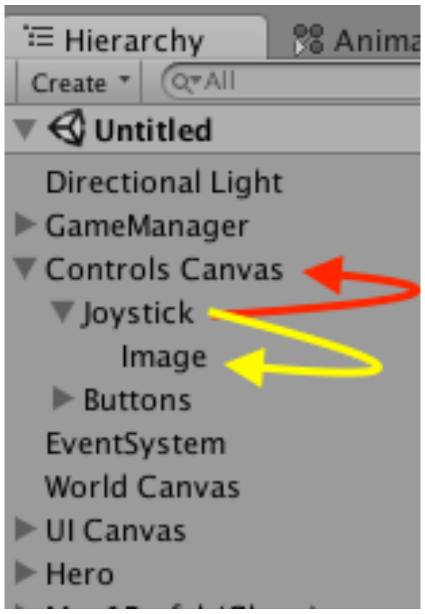
	Pos X	Pos Y	Pos Z
	0	0	0
	Width	Height	
	100	100	
Pivot	X 0.5	Y 0.5	
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Parenting

Parenting is simple. Well, at least it is in the Unity engine! Any GameObject can become the child of another GameObject. The Hierarchy view is where you see and manipulate children and parents. Child GameObjects are indented beneath their respective parents.

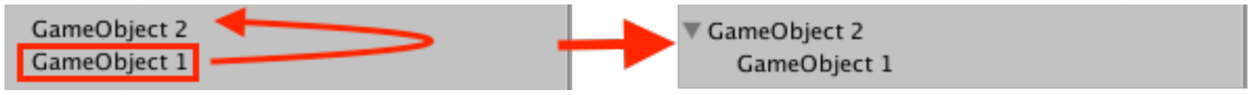
There are no limits on how many children a parent can have, but each child can only have one parent.

The next image shows you an example of such a relationship: Image is a child of Joystick, and Joystick is a child of Controls Canvas. It's just one big, happy family in there!



To make one GameObject a child of another, just drag one over another in the Hierarchy. To unparent, just do the opposite — it's as easy as pie!

In the example, GameObject 1 is now a child of GameObject 2:



You've finished the Unity crash course and have the basic understanding you need to start creating a game!

Don't worry if you're not totally clear on everything so far. You can always come back to this part to refresh your memory, and trust me when I say you'll become *very* familiar with the engine as you create PompaDroid.

Creating the title scene

You've created the project and are ready to build. When you first visit a new project, Unity greets you with a new, unsaved scene.

To save it, select **File\Save Scene** and name the scene **MainMenu**. Navigate to the Scenes folder and click **Save**.

Saving is simple enough. Look closer at the scene — contrary to what you might think it isn't empty. By default, Unity created a Main Camera GameObject.

You don't need it for the title scene, so select the **Main Camera** in the **Hierarchy** window, **right-click** and select **Delete**.

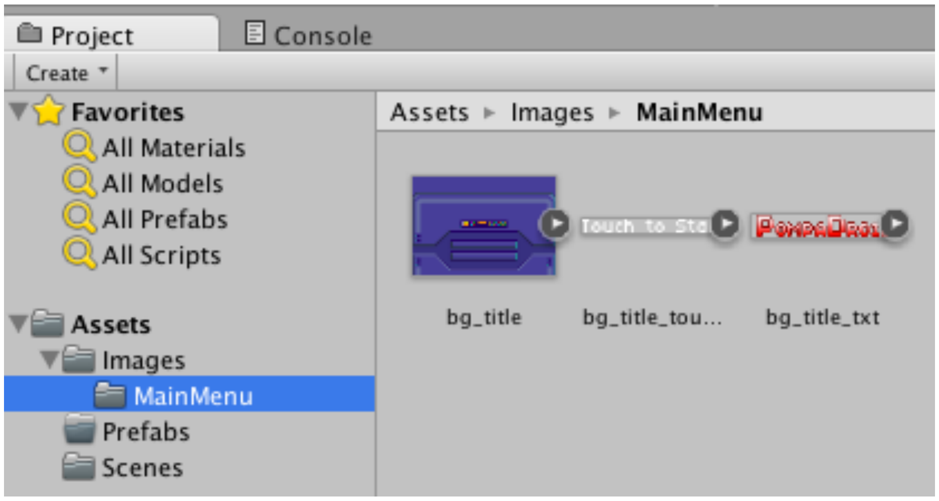
Your scene won't stay empty for long!

Find the **Images** folder under **Assets** in the **Project window**.

Right-click and select **Create\Folder**. Name it **MainMenu** — this is where you'll add all the assets you need to make the title screen.

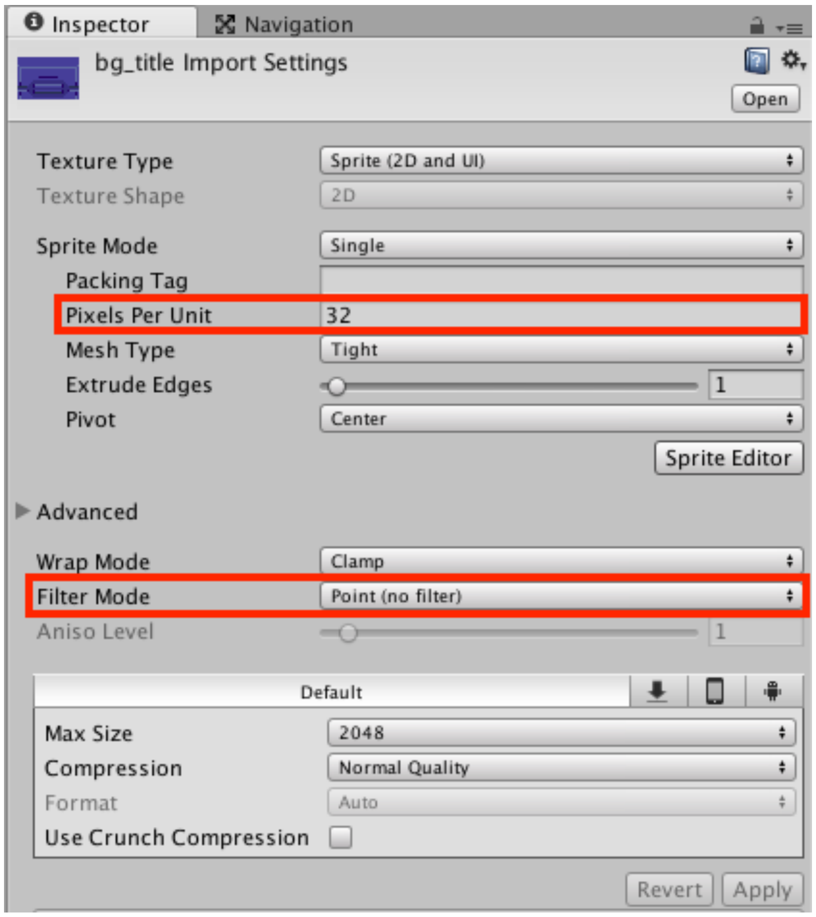
Open **MainMenu** then **right-click** it and select **Import New Asset**. Navigate to the **Chapter 1 Assets** folder that comes with this book.

Import **bg_title.png**, **bg_title_touchtostart.png** and **bg_title_txt.png** to **MainMenu**.



Note: An alternative way to import files into Unity is to drag the files directly into the Project window.

Now **select** the three assets you just imported in the Project window. Go to the **Inspector** and set **Pixels Per Unit** to 32 and set **FilterMode** to **Point**. Click on **Apply** to save your changes.



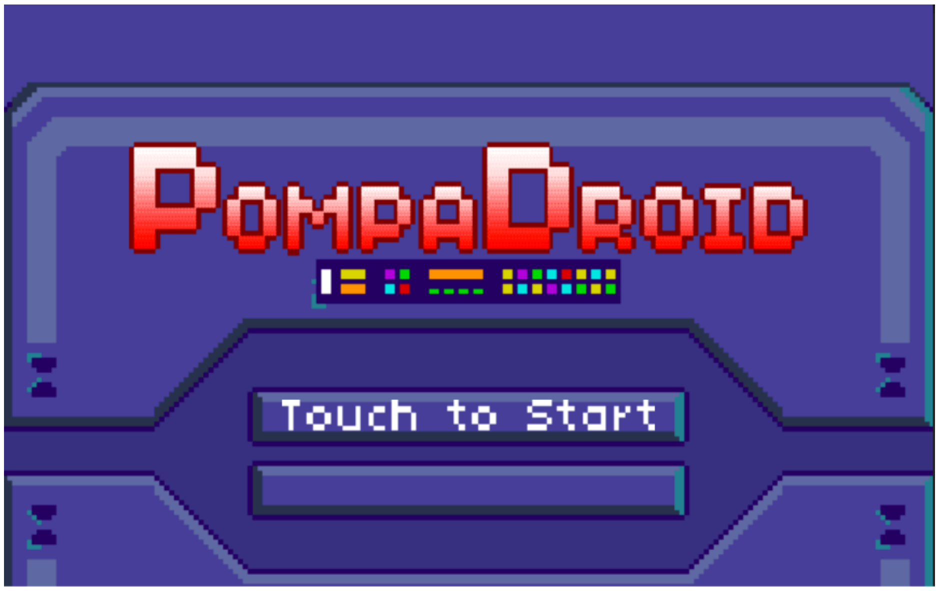
bg_title Import Settings

Texture Type	Sprite (2D and UI)
Texture Shape	2D
Sprite Mode	Single
Packing Tag	
Pixels Per Unit	32
Mesh Type	Tight
Extrude Edges	1
Pivot	Center
Wrap Mode	Clamp
Filter Mode	Point (no filter)
Aniso Level	1
Max Size	2048
Compression	Normal Quality
Format	Auto
Use Crunch Compression	☐

What did you just do?

- Setting the value of a **world space unit** in Pixels Per Unit means that 1 Unity world space unit equals 32 pixels.
- Setting Filter Mode to Point means textures will look blocky when magnified. Perfect for pixel art. This setting determines what happens when an image is stretched.

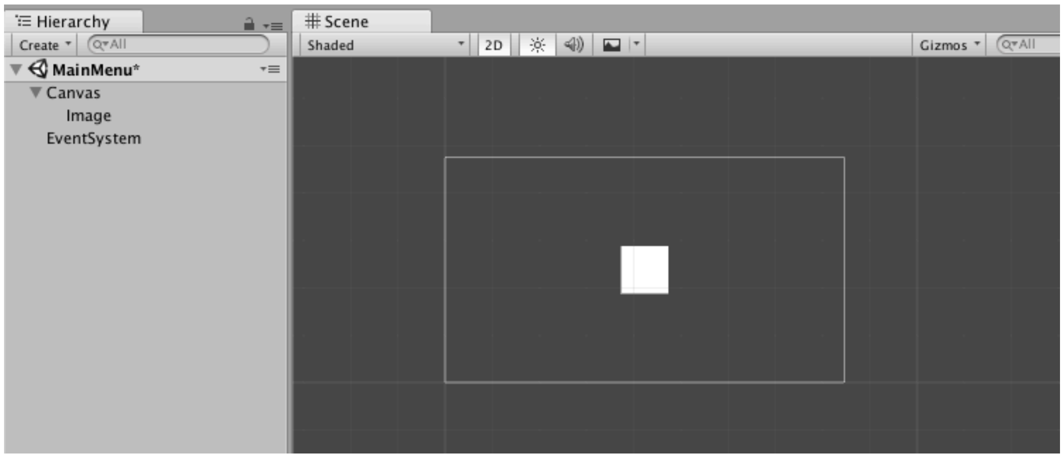
The assets are ready to use on the title screen — speaking of title screens, here's what yours will look like:



How are you going to make that? By using Unity's UI of course!

Set up the canvas

In the **Hierarchy** window, click **Create\UI\Image** to create three new GameObjects and a plain white box:



A **canvas** comprises three components: canvas, canvas scaler and graphics raycaster. You'll find them in the Inspector.

Remember that you'll need to create a canvas for all UI objects in Unity.

- The **Canvas** represents the space where the UI is drawn. You'll learn more about this later. For now, just keep in mind that all UI elements have an ancestor of a canvas component that renders on screen.

- The **Canvas Scaler** handles the overall size of the canvas. A common use case is when you want the game's UI to scale automatically to the current screen size.
- The **Graphics Raycaster** determines if a raycast can hit a canvas. It's important when setting up correct UI functionalities such as button clicks.

Secondly, Unity also created an **Image GameObject** as a child of the Canvas GameObject. It contains two components: a **Canvas Renderer** that's required to render a UI object on a canvas, and an **Image** component that draws the sprite set in the source image field.

Lastly, there's also a new **EventSystem GameObject** containing an **Event System** component that handles all the events the UI system uses. You won't be able to interact with UI elements without this component. It also contains a **Standalone Input Module** to handle game input.

Add the background image

Select the **Image** GameObject from the **Hierarchy**. Next, open **Images\MainMenu** from the **Assets** folder in the Project window, and drag **bg_title** to the **Source Image** field in the **Image** component.

Tadaaa! The white square on the screen now shows the bg_title image — you may need to zoom out to see it.

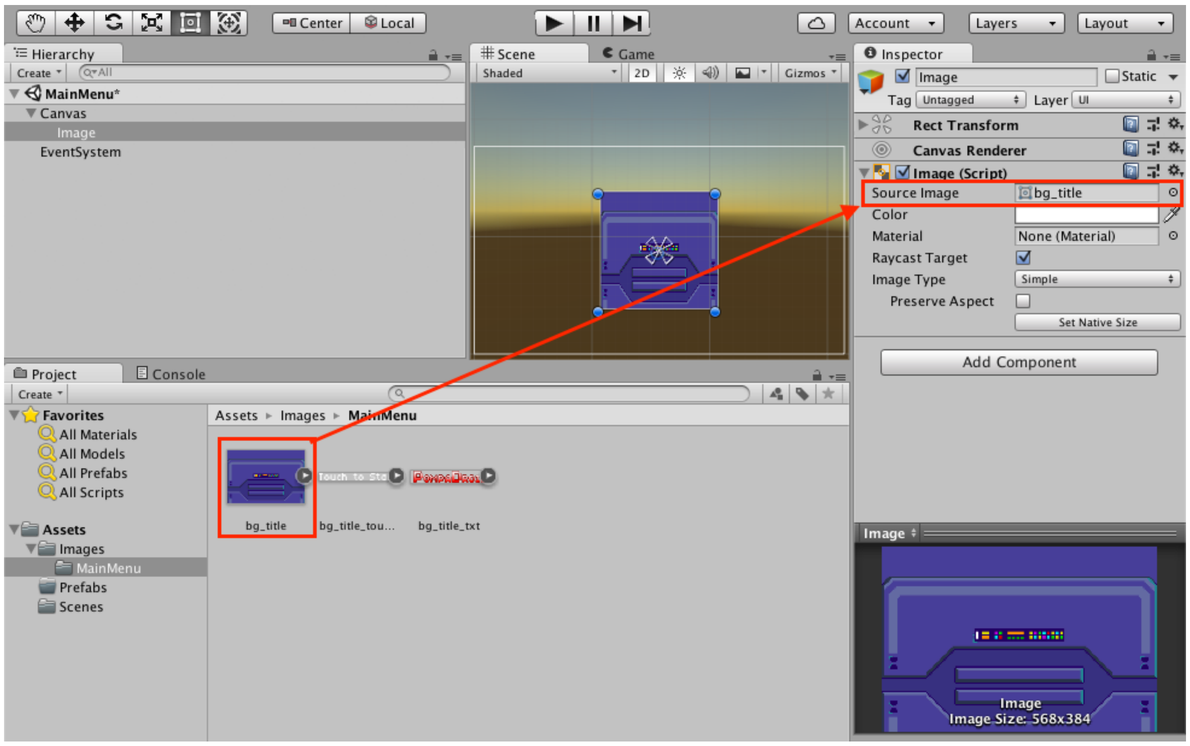


Image — Static
Tag Untagged — Layer UI
Rect Transform
Canvas Renderer
Image (Script)
Source Image: bg_title
Color
Material: None (Material)
Raycast Target: ✓
Image Type: Simple
Preserve Aspect: ☐
Image
Image Size: 568x384

Note: Another way to set the source image field is to click the knob on the right side to open a sprite selection window where you can select the sprite you want.

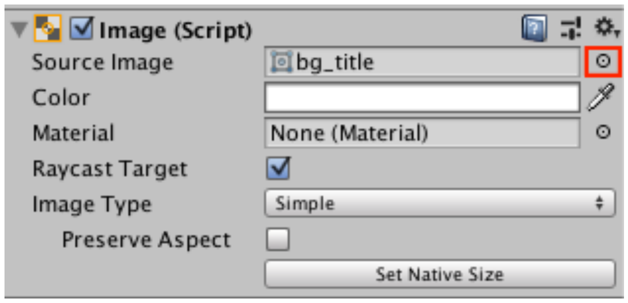


Image (Script)

Source Image: bg_title

Color

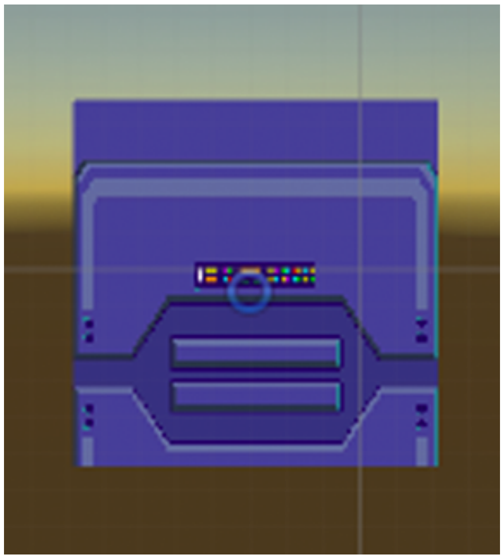
Material: None (Material)

Raycast Target: ✓

Image Type: Simple

Preserve Aspect: ☐

Take a closer look. It's there…but why is it shrunken, squished and squared off?

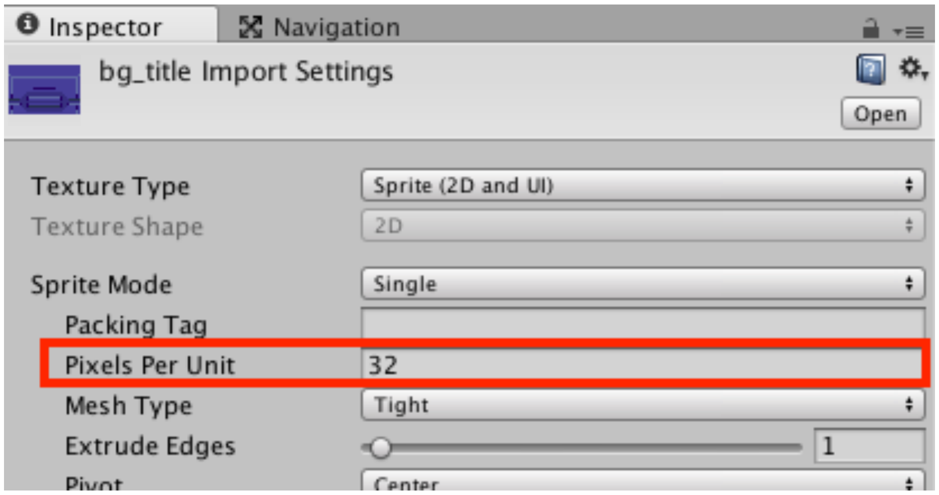


It definitely shouldn't look like this!

Make it pretty with the canvas scaler

First, you need to determine the resolution and pixel per unit import settings for bg_title.

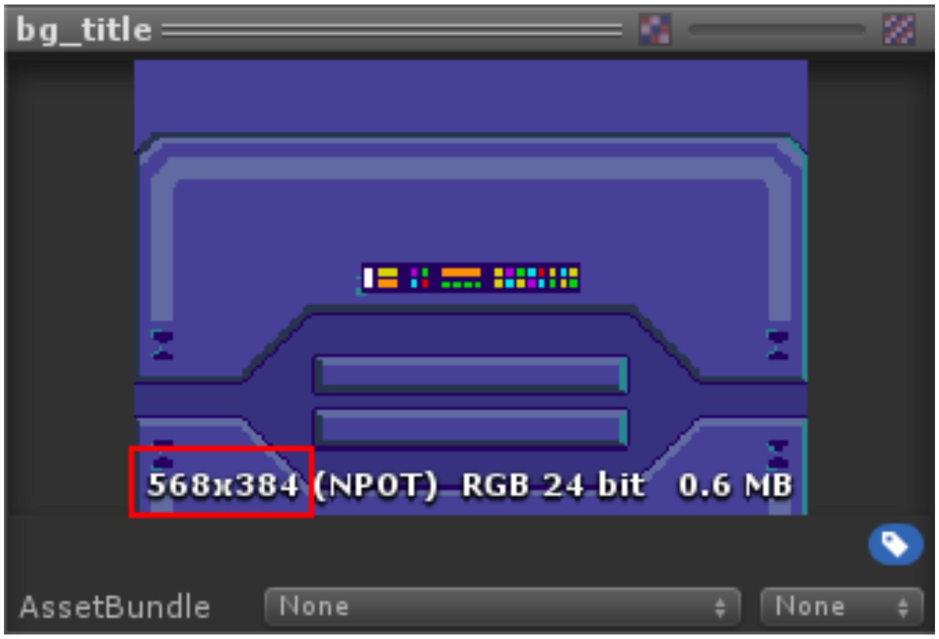
Select **bg_title** in the **Project window** and check the pixel per unit value.



bg_title Import Settings

Texture Type: Sprite (2D and UI)
Texture Shape: 2D
Sprite Mode: Single
Packing Tag:
Pixels Per Unit: 32
Mesh Type: Tight
Extrude Edges: 1
Pivot: Center

For the image resolution, select the **sprite** (bg_title) in the **Project window** and check its properties at the bottom of the **Inspector**.



bg_title

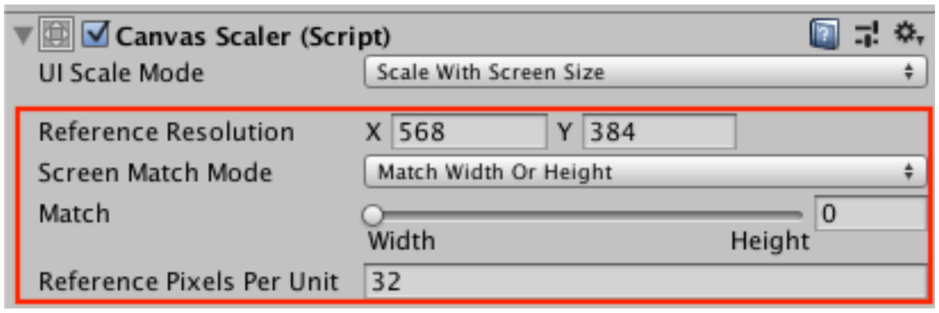
568x384 (NPOT) RGB 24 bit 0.6 MB

AssetBundle: None, None

You imported bg_title at 32 pixels per unit and its dimensions are 568 x 384 pixels, so you still need to configure the canvas scaler to match these settings. Remember, this component scale's the UI to fit the screen size.

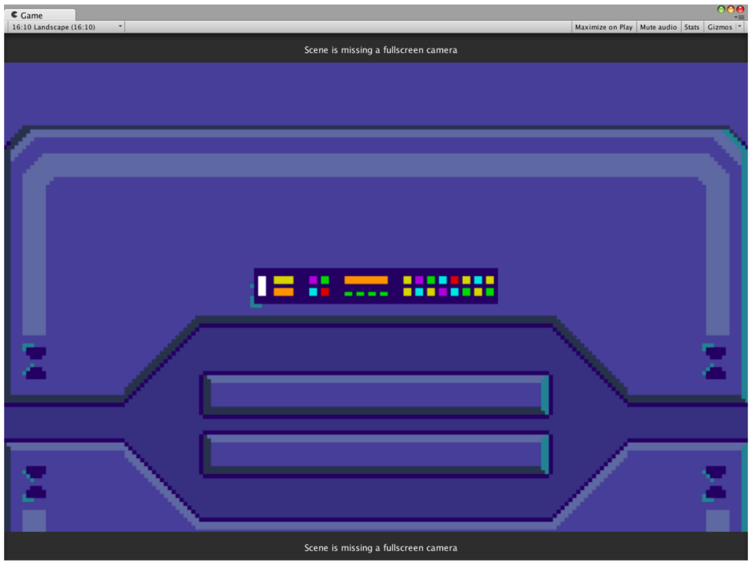
In the **Canvas Scaler** component of the **Canvas GameObject**, change **UI Scale Mode** to **Scale With Screen Size**. Set **Reference Resolution** to the sprite resolution of `568 x 384`.

Set **Screen Match Mode** to **Match Width or Height**, and **Match Value** to `0`. Finally, set **Reference Pixels Per Unit** to `32`.


Canvas Scaler (Script)
UI Scale Mode: Scale With Screen Size
Reference Resolution: X 568 Y 384
Screen Match Mode: Match Width Or Height
Match: 0 (Width – Height)
Reference Pixels Per Unit: 32

Now you can fill the screen with the background image.

Select the **Image** GameObject. Set both **PosX** and **PosY** to `0` to center the image, and in the **Image** component, click **Set Native Size**.


Scene is missing a fullscreen camera

Scene is missing a fullscreen camera

There you go! Now look at the Game view and change the resolution by selecting various devices from the top-left drop-down selector. You’ll notice the image stays fullscreen because the canvas scaler now knows to maintain its size at bg_title’s resolution!

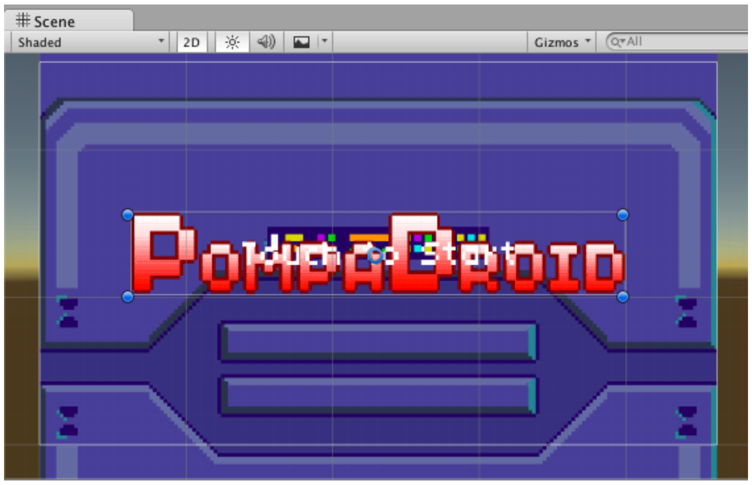
Add text sprites to the title screen

Now you have a lovely textured wall, but there are no prompts to tell players what to do next. This is a job for a text sprite!

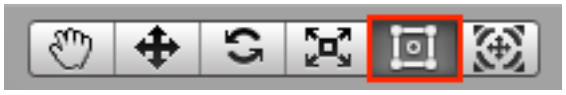
Create two more images by **right-clicking** in the **Hierarchy**, then selecting **UI\Image**. Rename the first Image GameObject to **TitleText** by **right-clicking** and selecting **Rename**.

Drag **bg_title_txt** to the Image component’s **Source Image**.

Rename the second Image GameObject to **TouchToStart** and then drag **bg_title_touchtostart** to the Image component's **Source Image**. Click the **Set Native Size** button on both Image components. Make sure that both TitleText and TouchToStart are children of Canvas.

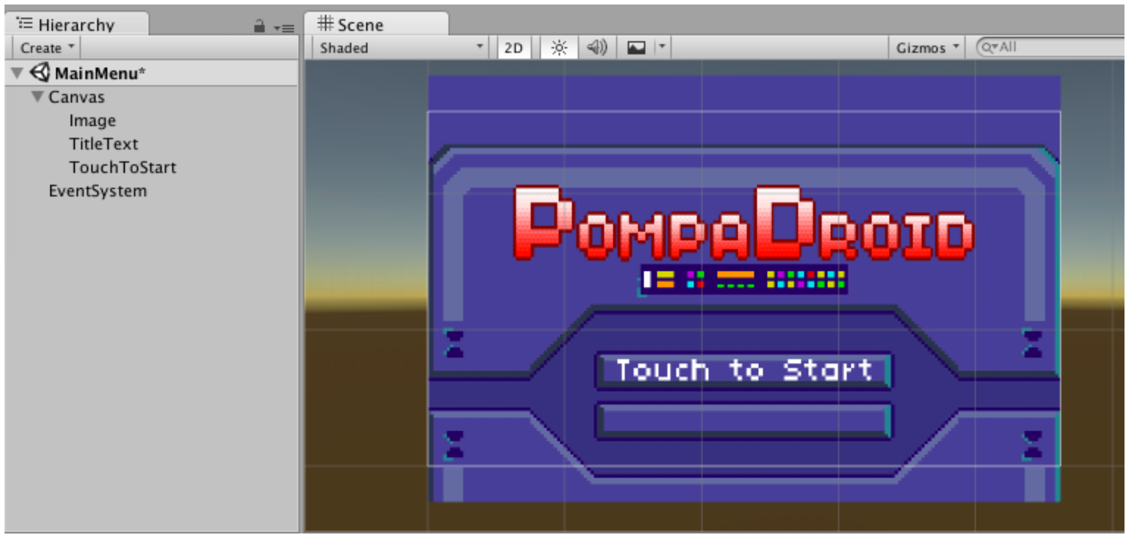


Hmmm, that's not quite right. Seems like a good time to go over repositioning things.



Select the **Rect Tool** on the toolbar — it allows you to change the position, rotation, scale and dimensions of 2D GameObjects.

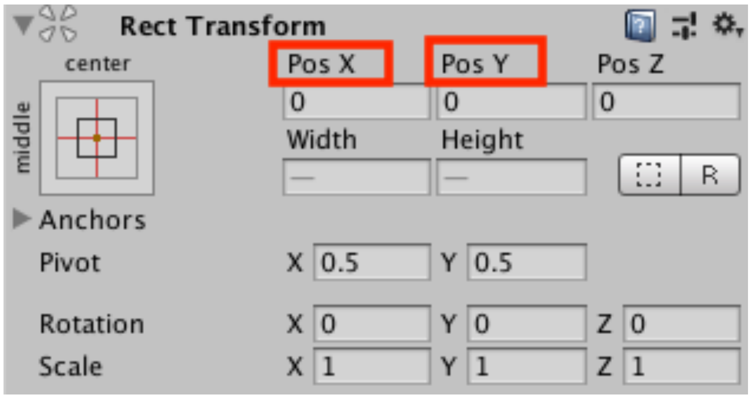
Select **TitleText** and drag it around the selection box in the Scene view, then do the same with **TouchToStart** until you have something similar to this:



If one of the sprites "disappears" after you drag it around, check the order of GameObjects in the Hierarchy and make sure your screen matches the above screenshot.

Now you've got a legitimate title screen!

Note: Alternatively, you can change positioning from each sprite's Rect transform component. Set a specific value by typing it in or dragging the text of the value you want to adjust.



	Pos X	Pos Y	Pos Z
	0	0	0
	Width	Height	
	—	—	
Pivot	X 0.5	Y 0.5	
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

This is the moment you've been waiting for — click the **Run** button on the **Toolbar**.

What happens when the game is running? Absolutely nothing! There is nothing to run...yet.

The next task is to allow the player to start the game from the title screen.

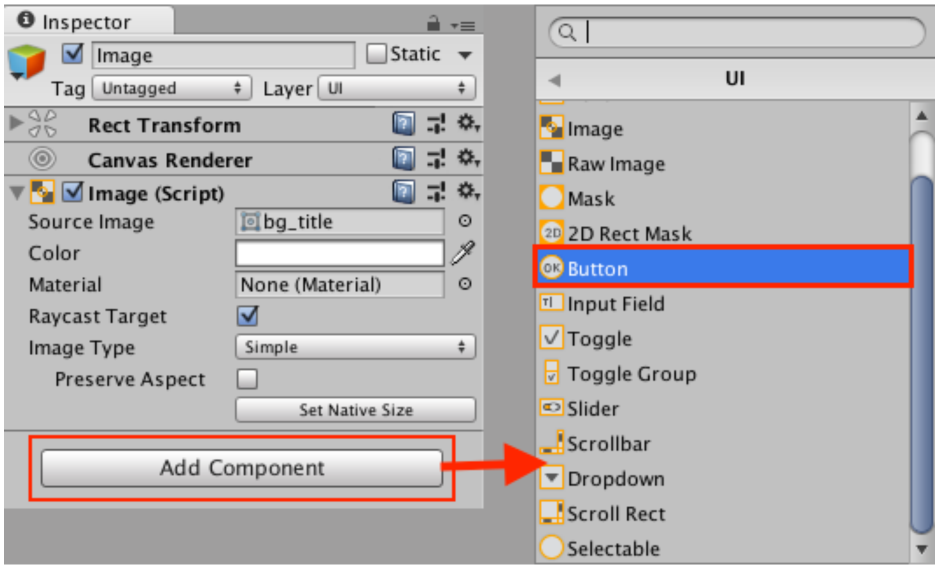
Buttons and scripts

In this section, you'll create controls and points of interaction on your scene. Here's where you'll work with scripting for the first time.

Create a button

Users naturally look for things to press, tap and swipe, even when you've made them non-essential. You should always add a button so the user has no doubts about how to start the game.

Select **Image** in the Hierarchy, and then click the **Add Component** button in the Inspector. Select **UI\Button**, and then on the newly created Button component, set **Transition** to **None**.



Note: Button transitions let you change the look of buttons when they're pressed, disabled, hovered over or selected. You don't *need* any transitions for this title screen, but don't let that stop you from experimenting with various button transition modes!

Currently, your button is merely an empty shell. You'll change that by adding a custom script as a component to your canvas GameObject and creating a corresponding script file in the root of the Assets folder.

Select the **Canvas** GameObject, click **Add Component** and select **New Script**. Name it **MainMenu**, set the language to **C Sharp** and then click **Create and Add**.

Find it under Assets in the Project window, **drag** it to the **Scripts** folder, and then **double-click** the script to open it in your code editor.

Change the contents of **MainMenu** to match the following:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class MainMenu : MonoBehaviour {
  void Start() {
  //1
    Application.targetFrameRate = 60;
  }

  //2
  public void GoToGame() {
  //3
    SceneManager.LoadScene("Game");
  }
}
```

Save the script and take a closer look at what you did:

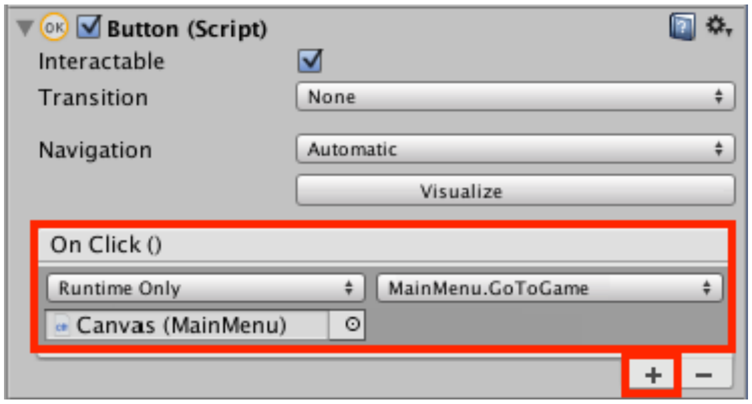
1. In the `Start` method, you changed `Application.targetFrameRate` to `60`, which limits gameplay to 60 FPS (frames per second). It's fast enough but helps your game avoid being a battery hog, which could easily happen if you left FPS uncapped.
2. You added a method named `GoToGame()` and made it `public` so you can call it from the button you created earlier.
3. `SceneManager.LoadScene()` loads another scene. In this case, a soon-to-be-created scene named "**Game**".

Save, close the editor and return to Unity.

Give the button something to call

The Button component needs to call the `GoToGame()` method.

Select the **Image** in the Heirarchy. In the **Button** component, click the **+** that's beneath the OnClick() field. **Drag** the Canvas GameObject to the field named **None (Object)**. Select **MainMenu\GoToGame()** from the drop-down to the right.



Interactable ✓
Transition: None
Navigation: Automatic
Visualize
On Click ()
Runtime Only
MainMenu.GoToGame
Canvas (MainMenu)

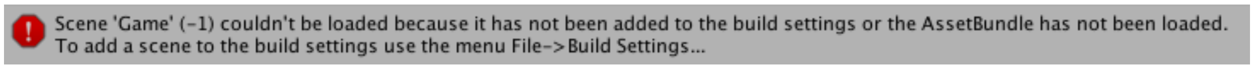
Well, that was pretty easy! Now your button calls `MainMenu.GoToGame()` whenever it's pressed!

Add a game scene

Start with saving your current scene by going to **File\Save Scene**. Afterwards, create a new, empty scene by clicking on **File\New Scene**. **Save** this new scene and put it inside the **Assets\Scenes** folder. Name it **Game**.

Open the MainMenu scene by **double-clicking** the **Assets\Scenes\MainMenu** scene file in the Project window.

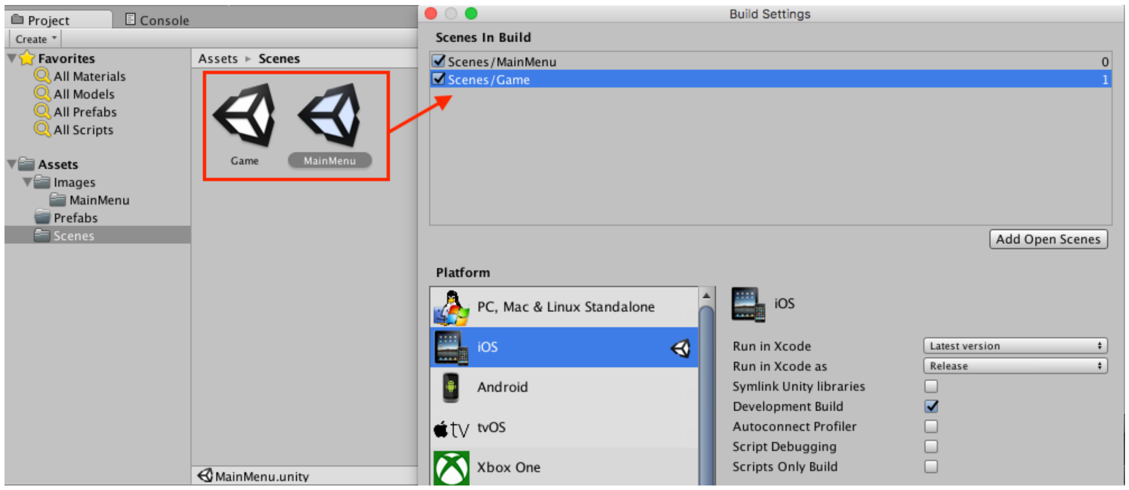
Run the game again and press anywhere on the screen.



```
Scene 'Game' (-1) couldn't be loaded because it has not been added to the build settings or the AssetBundle has not been loaded.
To add a scene to the build settings use the menu File->Build Settings...
```

Fail! Although you created a new scene, Unity doesn't know whether to include it in the game or not.

To add it, navigate to **File\Build Settings**. Drag the two scenes from the **Scenes folder** to the **Scenes in Build** field. If needed, removed other scenes that are added and drag the scenes to show **MainMenu** above the **Game** scene. Unity always starts from the top.



Scenes In Build

☑ Scenes/MainMenu	0
☑ Scenes/Game	1

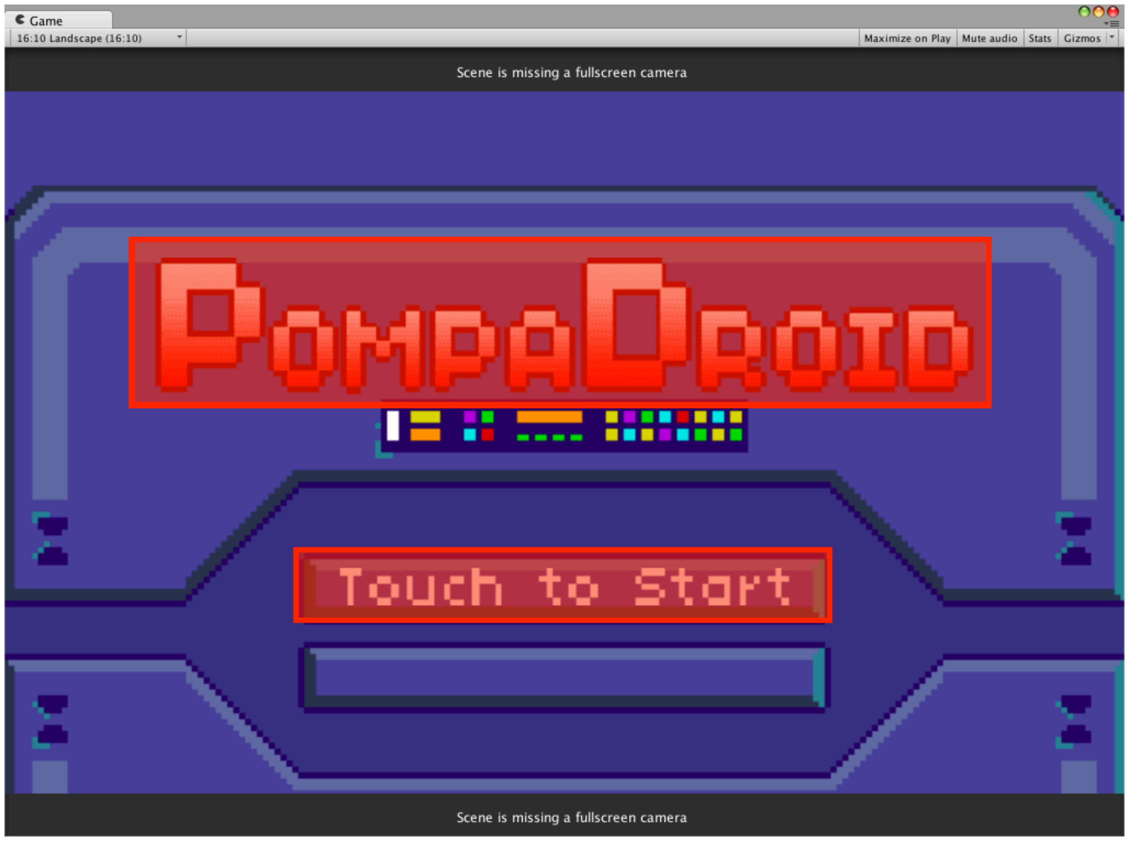
iOS

Run in Xcode	Latest version
Run in Xcode as	Release
Symlink Unity libraries	☐
Development Build	☑
Autoconnect Profiler	☐
Script Debugging	☐
Scripts Only Build	☐

Save the scene and run the game again. Click the background and there you go! You're looking at an awe-inspiring empty scene.

You might notice an issue while clicking things in the title scene. For instance, if you click the PompaDroid Logo or the Touch to Start text, nothing happens.

It's not really a bug, it's just that you haven't walked through making a transition to the game scene. Don't blame yourself for this one! :]



Just like bouncers keep the rif-raf out of a club, your sprites currently block click events from reaching the background. Your next task is to let the click events pass through.

Uncheck **Raycast Target** in the **Image** component of both **TitleText** and **TouchToStart** to fix this.

Run again and click the text logo. It now transitions to the game scene! Bug fixed!

Where to go from here?

Congratulations, you now have a functional title screen at your disposal! Not bad, especially since you're probably new to Unity.

At this point, you know how to:

- Set up a new Unity project
- Navigate the Unity editor
- Create a simple scene using Unity's UI
- And finally, load a new scene

You also dabbled with scripting and transforms.

Treat yourself to a well-deserved break! There's a lot to learn ahead. In Chapter 2, “Working with Tilemaps”, you’ll start work on the game scene's background.

Chapter 2: Working with Tiled Maps

In Chapter 1, “Getting Started”, you got the hang of using Unity’s UI system to display images, and you set up PompaDroid’s title scene.

Not quite the challenge you wanted? Strap yourself in because you’re about to feel the burn and learn some very cool stuff!

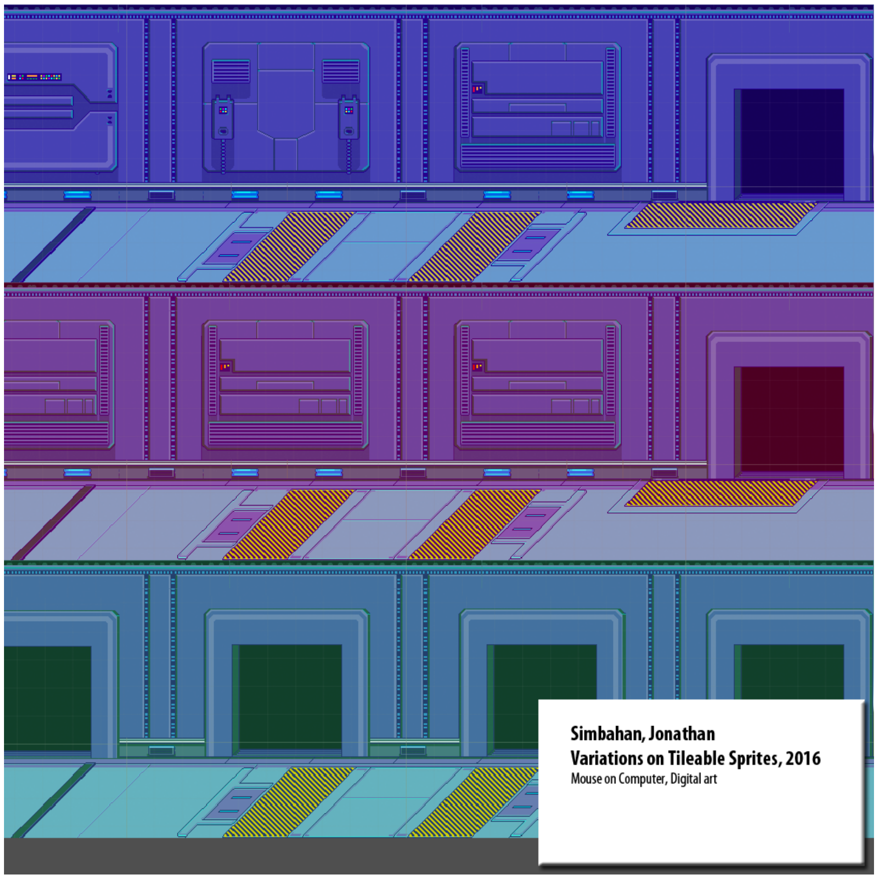
This chapter is all about Unity’s 2D sprite system and how to create and design levels for your beat ’em up game. You’ll also learn a cool map creation technique that's origins trace back to the earliest days of video game development — tiled maps!

Working with tiled backgrounds

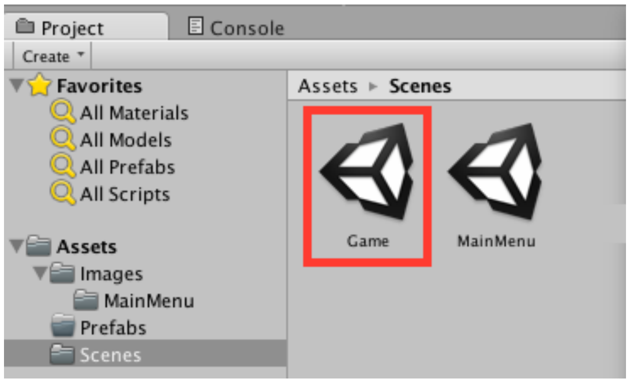
The background of PompaDroid will be composed of tiles, which are small, repeatable sprites positioned side-by-side to form a bigger picture.

Why go through this process? Tiling saves space and texture when comes to game backgrounds. Instead of creating one big sprite per level, you create a few images that you can shuffle around to create a variety of interesting backgrounds.

> **Note**: At the time of writing, Unity didn't have its own tilemap feature. As such, I created an advanced technique to generate tilemaps from sprites, which I'm about to share with you. Although you can make tilemaps in a similar way using the tilemap system found in Unity 2017 and onwards, the editorial team and I feel that this technique is worth learning.


Simbahan, Jonathan
Variations on Tileable Sprites, 2016
Mouse on Computer, Digital art

Switch to the game scene. Double-click the **Game** scene inside the **Scenes** folder in the Project window.



> **Note**: If you’re familiar with tilemaps in Unity, you’ll be pleased to learn that you can skip ahead. The files for this chapter have a package that contains a tiled wall and floor. Import **Complete Wall And Floor Sprites.unitypackage** from the **Unity Packages** folder and skip to the section named **Optimizing Tiled Assets**.

Importing tile assets

Import **GridSnapper.unitypackage** by **double-clicking** the package inside the "Unity Packages" folder included with the book, or by selecting **Assets\Import Package\Custom Package** on the menu bar.

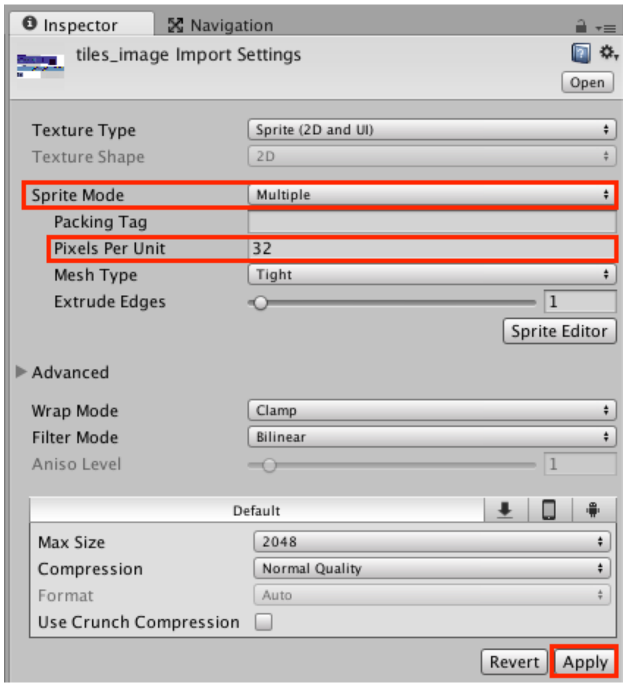
Make sure you’ve imported **GridSnapper.cs** *and* **GridSnapperInspector.cs**.

These scripts make tiling much easier. Although we could explain how they work right here, right now, it would be a little distracting. Visit the appendix section later to learn more.

Import and configure the spritesheet

Create a new folder under **Assets\Images**, name it **Background** and import **tiles_image.png** included with the chapter. Make sure the **Texture Type** is **Sprite (2D and UI)** and set the **Sprite Mode** to **Multiple**.

Set **Pixels Per Unit** to 32 and click **Apply**.

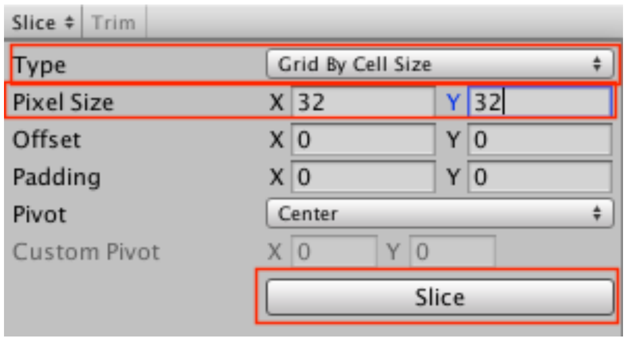


tiles_image Import Settings

Texture Type	Sprite (2D and UI)
Texture Shape	2D
Sprite Mode	Multiple
Packing Tag	
Pixels Per Unit	32
Mesh Type	Tight
Extrude Edges	1
Advanced	
Wrap Mode	Clamp
Filter Mode	Bilinear
Aniso Level	1
Default	
Max Size	2048
Compression	Normal Quality
Format	Auto
Use Crunch Compression	☐

Locate and click the **Sprite Editor** button in the Inspector — this is where you’ll slice the sprite into many small, "tile-able" sprites.

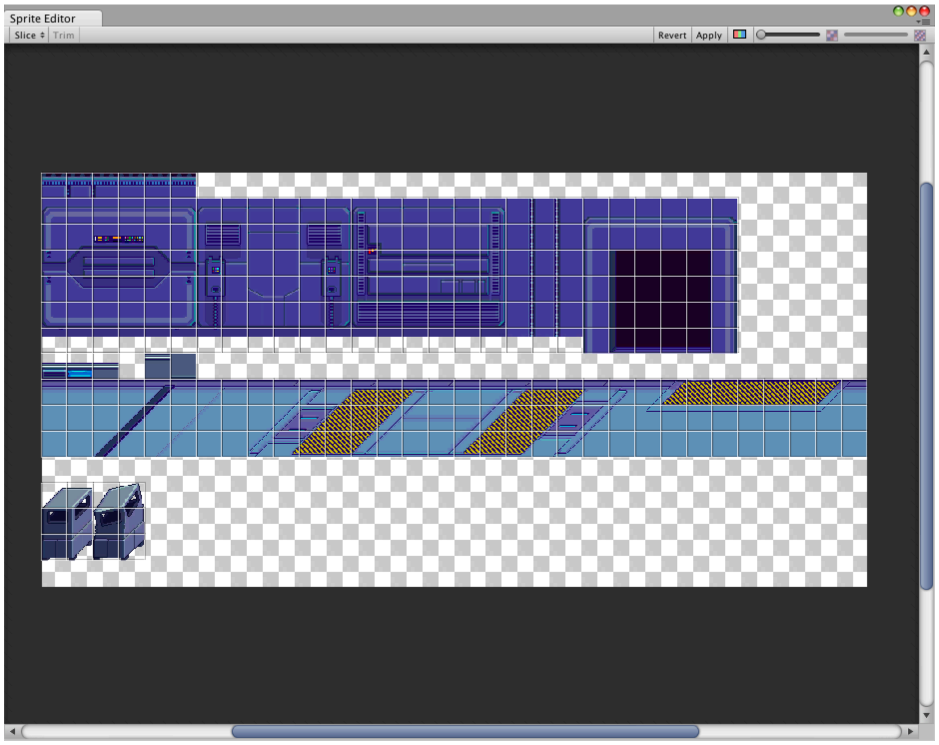
Select **Slice** in the top-left of the window. Set **Type** to **Grid By Cell Size** and **Pixel Size** to `(X:32, Y:32)`.


Slice | Trim

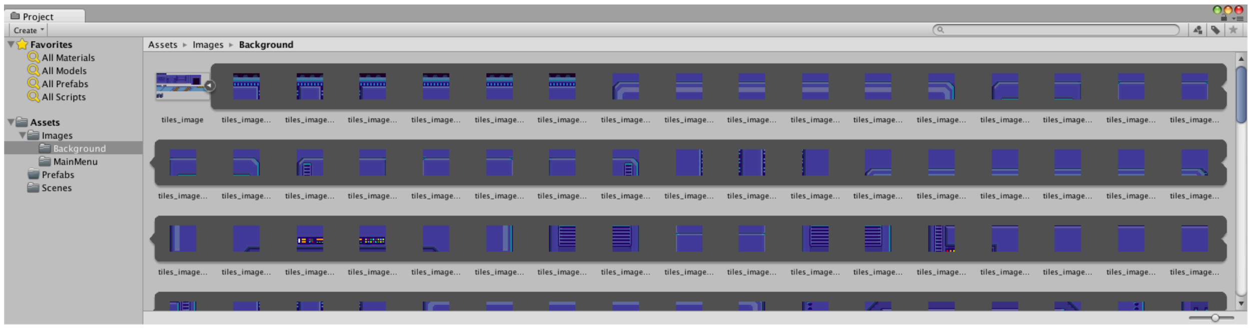
Type: Grid By Cell Size
Pixel Size: X 32 Y 32
Offset: X 0 Y 0
Padding: X 0 Y 0
Pivot: Center
Custom Pivot: X 0 Y 0

Click the **Slice** button then click **Apply** in the top-right to save the sprites. **Close** the window.

Congratulations! You've sliced and diced tiles_image and have sprites to fashion into a background.



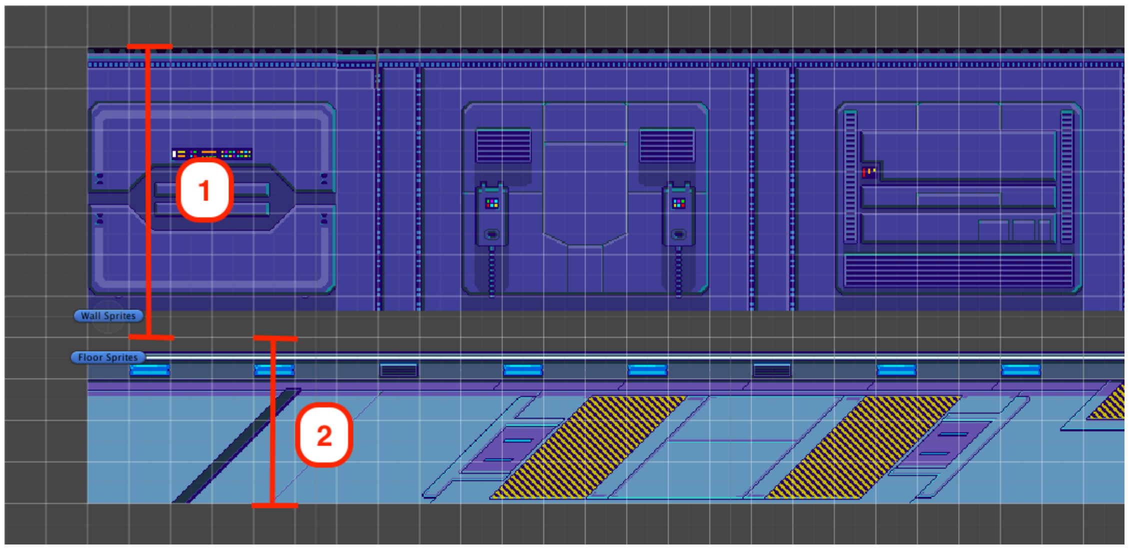
Take a look at the Project window. Now tiles_image has a triangle next to its name. Click it to expand and reveal the tiles.



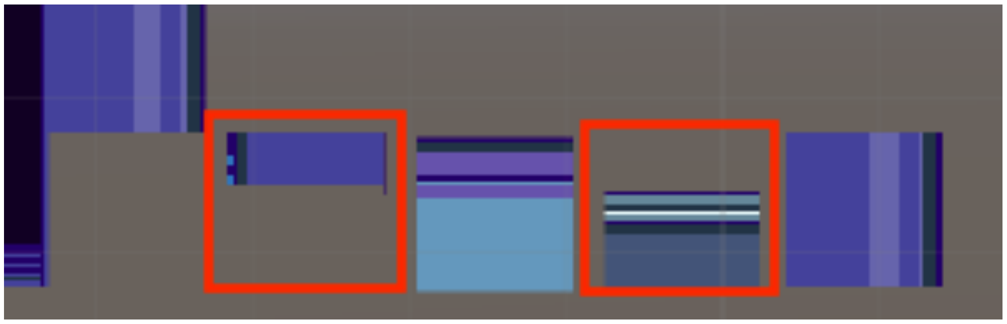
Assembling tiles

Now comes the interesting part of assembling tiles to make a background. You’ll need two tilemaps: one for the wall sprites and one for the floor sprites.

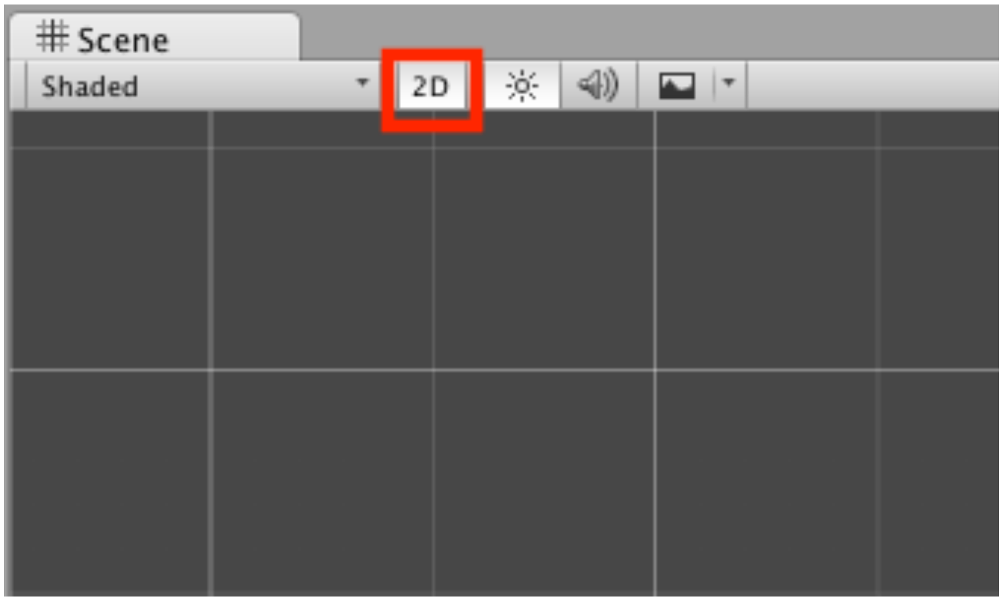
- The wall tilemap will be seven tiles high and start at the bottom-left.
- The floor tilemap will be four tiles high and start at the top-left.



Note that you have half-tiles for both tilemaps. If you pair them up, they should fit perfectly together. Look at how the wall half-tile matches up with the floor half-tile.



Make sure to enable **2D mode toggle** in the **Scene view** to render the scene in Orthographic projection so that all assets are now 2D.



Meet the GridSnapper

If you’ve ever seen a professional lay “real” tiles, you probably saw them use tools to create perfect lines and corners.

While you don’t need a bunch of tools to lay sprites neatly, you'll do yourself a solid by creating a **GridSnapper**, which is an empty GameObject you'll make for each surface and a reference point for each type of sprite you’ll lay down.

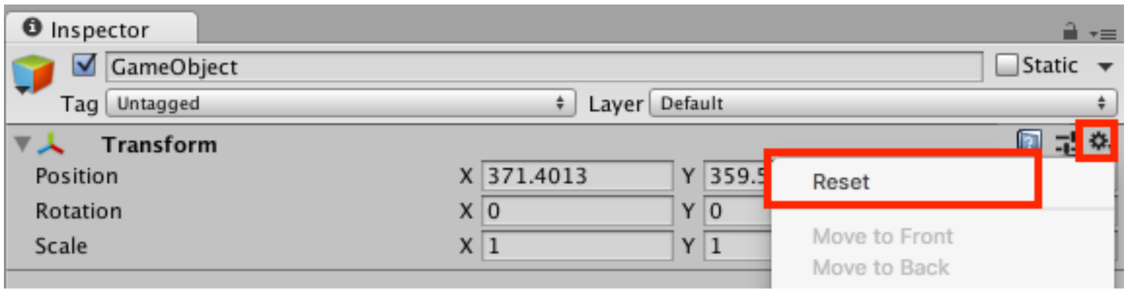
It'll save you from the pains of manually setting each tile by snapping them into place. In the next steps, you’ll make this tool.

Open the **game scene** from the Project window.

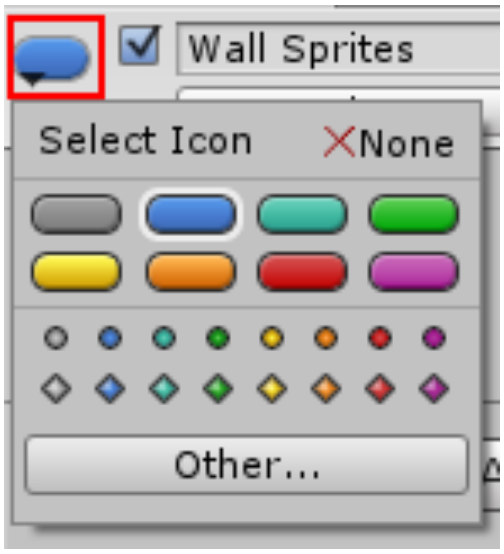
In the Hierarchy, right-click and select **Create Empty** to make a new GameObject and name it **Wall Sprites**. This GameObject will be the parent to all wall tile sprites.

In the **Inspector**, reset Transform values by setting position, rotation and scale to `(X:0, Y:0, Z:0)`, `(X:0, Y:0, Z:0)`, and `(X:1, Y:1, Z:1)`.

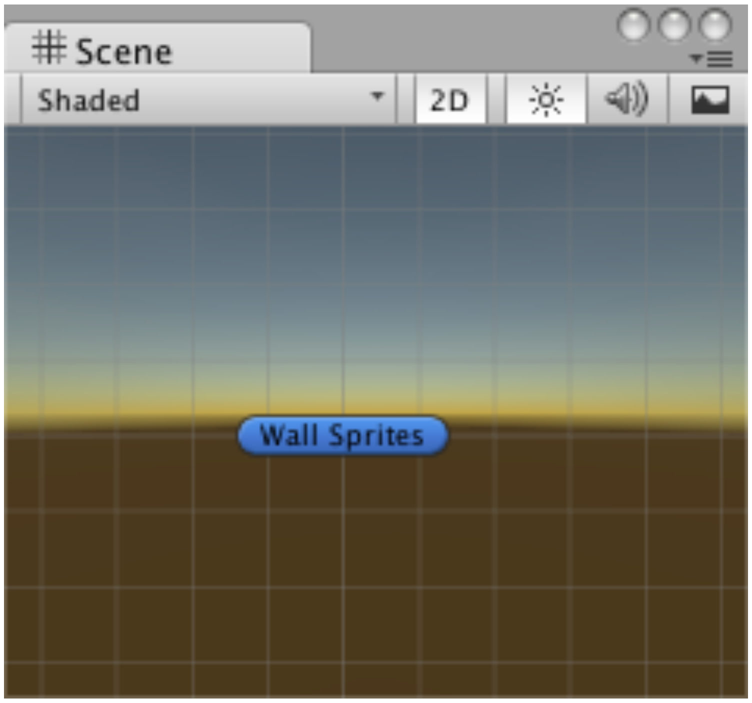
Note: Another way to reset is by **clicking** the **gear** on the right side of the **Transform** component and selecting **Reset**.



Click on the **multi-colored cube** at the top-left of the **Inspector** to open a small window. In here, you can pick the icon that represents the GameObject in the Scene view — **select** one of the **colored capsules**.



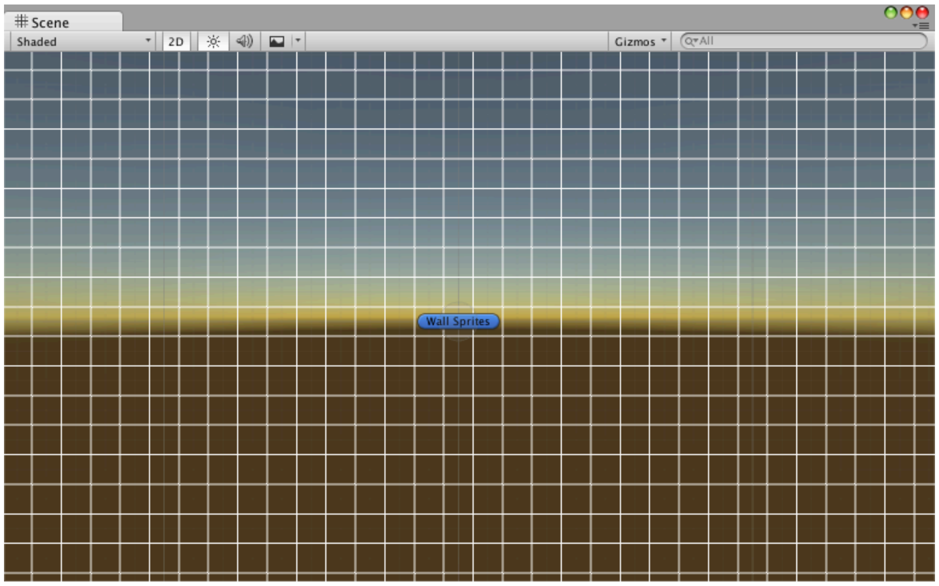
Now your GameObject shows as a colored capsule label in the Scene view.



Neat, isn't it? It's a reference point that's for your eyes only that you'll use when adding tile sprites. It will not show in the actual game.

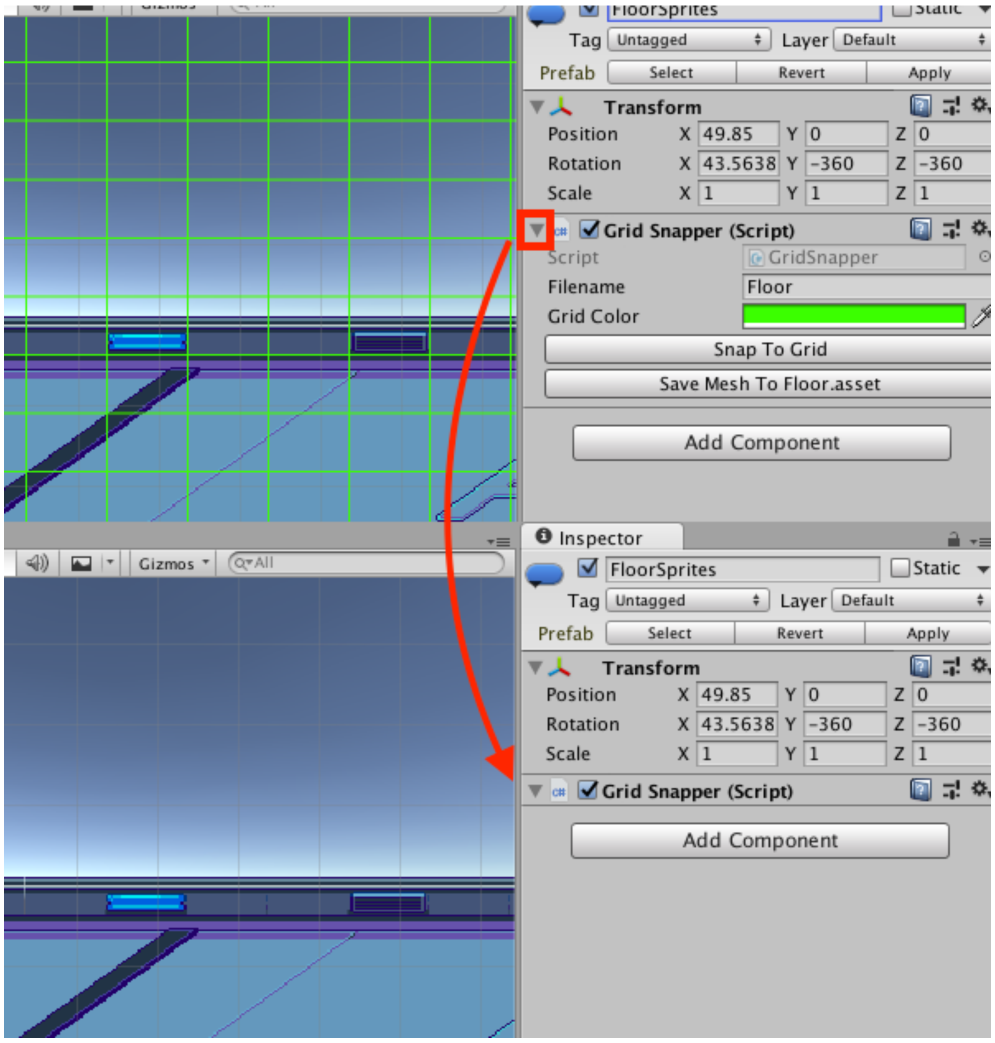
Click **Add Component\Script** and choose the **GridSnapper** to add it as a component and observe what happens next.

Almost like magic, grid lines appear!



The GridSnapper makes life much easier because it marks where tiles should go. Try using different colors when you're working with multiple GridSnappers; you can change the field grid color as needed to stay organized.

Note: Are you tired of grid lines interrupting your workflow? Click the GridSnapper component's **name** in the **Inspector** to collapse the component and disable the grid.



FloorSprites Static
Tag Untagged Layer Default
Prefab
Transform
Position X 49.85 Y 0 Z 0
Rotation X 43.5638 Y -360 Z -360
Scale X 1 Y 1 Z 1
Grid Snapper (Script)
Script GridSnapper
Filename Floor
Grid Color 🟩

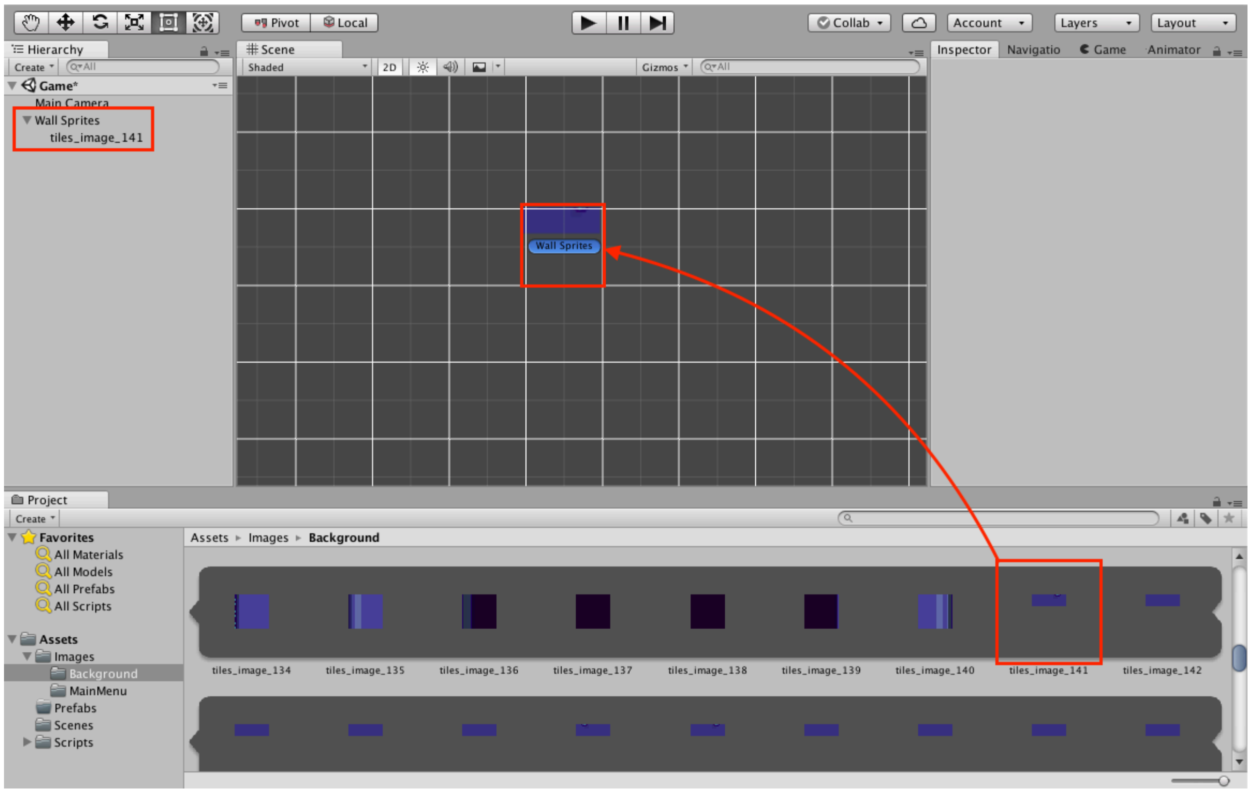
FloorSprites Static
Tag Untagged Layer Default
Prefab
Transform
Position X 49.85 Y 0 Z 0
Rotation X 43.5638 Y -360 Z -360
Scale X 1 Y 1 Z 1
Grid Snapper (Script)

Tile it up

Select **tiles_image_141** under **tiles_image** in the **Project window** and **drag** it onto the **Scene view**. Doing this creates a new GameObject with a SpriteRenderer component. The sprite **tiles_image_141** is also set as the SpriteRenderer's sprite field — make it a **child** of **Wall Sprites**.

Finally, **reset** the Transform component to center the sprite.

Note: Selecting and dragging multiple sprites on the Scene view won't add multiple SpriteRenderers. By default, Unity assumes you're making an animation with those sprites. Press **Alt** on PC or **Option** on Mac while dropping to create separate tiles without accidentally animating things.

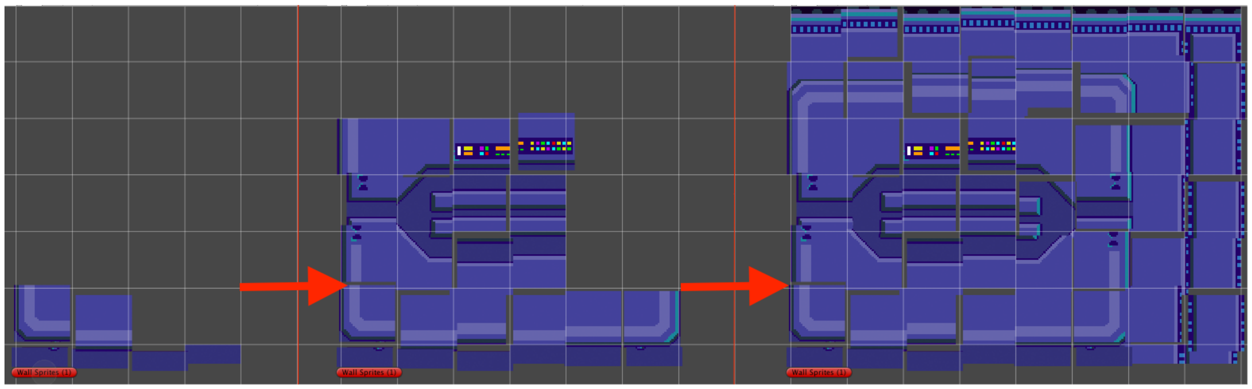


You've just laid your first tile — the bottom-left wall half-tile to be precise. Just continue to add the adjacent tiles (141-167 make up the bottom row), and position them next to the previous tiles.

Placement doesn't have to be precise, but you *do* need to remember to add tiles as children of the Wall Sprites GameObject.

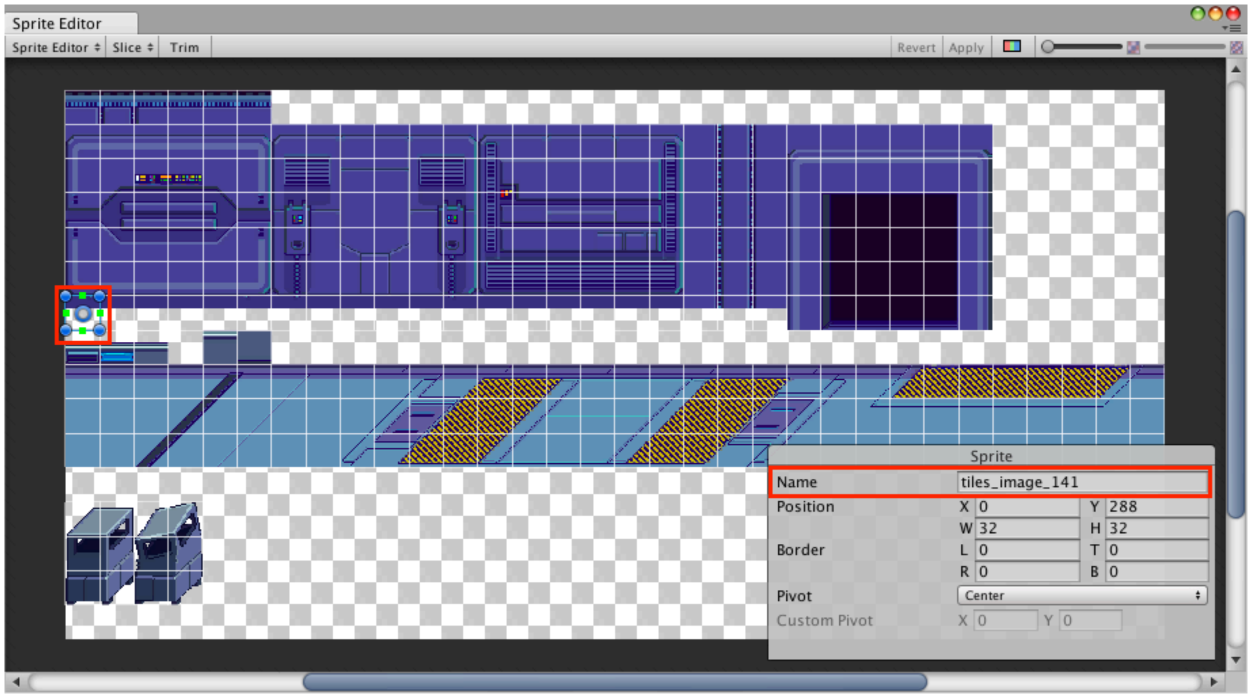
Add the tiles that go above your starting tile. Start the next row by placing titles_image_114 just above 141, then use the screenshot below to help yourself find and place the right tiles to build out the next five rows.

In this case, you should build up from the bottom. It's easier and more efficient.



Remember that wall sprites are seven tiles tall, including the first tile you placed.

Note: Depending on your version of Unity, the tile number might differ from the one listed above. No worries! You can check the tile names and if they fit together by referring to the original *tiles_image* in the Sprite Editor, which you'll find by selecting the **sprite** in the Project window and clicking the **Sprite Editor** button in the Inspector. Simply select any tiles in question to identify them by name.



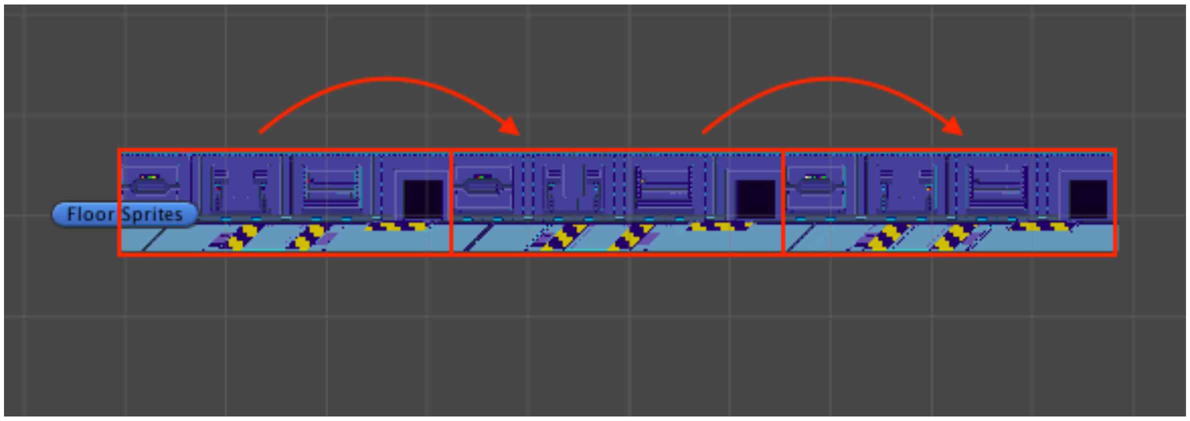
Sprite
Name tiles_image_141
Position X 0 Y 288
W 32 H 32
Border L 0 T 0
R 0 B 0
Pivot Center
Custom Pivot X 0 Y 0

Make copies of tiled sprites

Keep adding tiles until you have a tileable chunk, where the left-most tiles can join with the right-most tiles. These chunks will repeat to complete the map. In the example below, the chunk is made from the with the three square designs, and then the door design. **Select** all children of the **Wall Sprites** GameObject. **Right-click** and select **Duplicate**.

In the **toolbar**, select the **Transform** tool. With the duplicate tiles still highlighted in the Hierarchy, **drag** the newly copied sprites to the right in the Scene view.

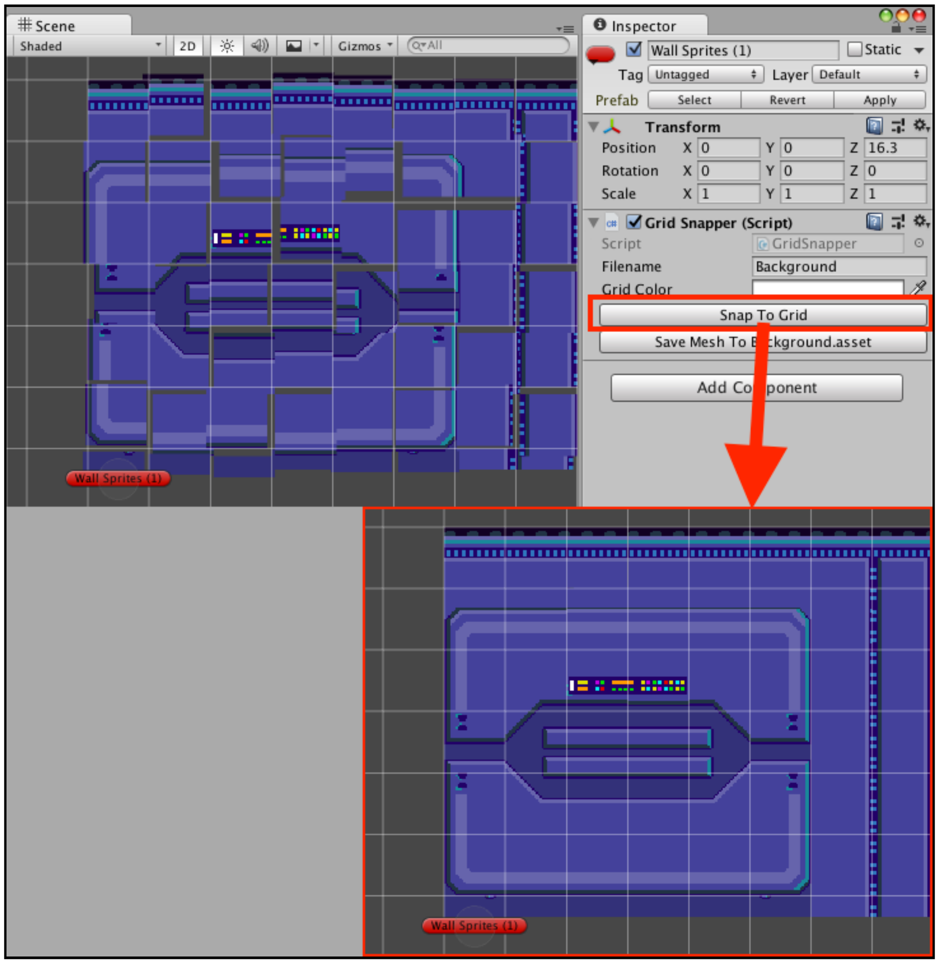
Keep making copies until the level is as long as you want it to be.



Snap it all together

It’s tricky to position tiles exactly as when they're all misaligned like that. GridSnapper to the rescue!

Select the **Wall Sprites** GameObject and go to the Grid Snapper component in the Inspector — click **Snap To Grid**. GridSnapper will snap the tiles to the nearest possible grid.



Lay down a floor

Now that you've made the wall, you need to make a floor.

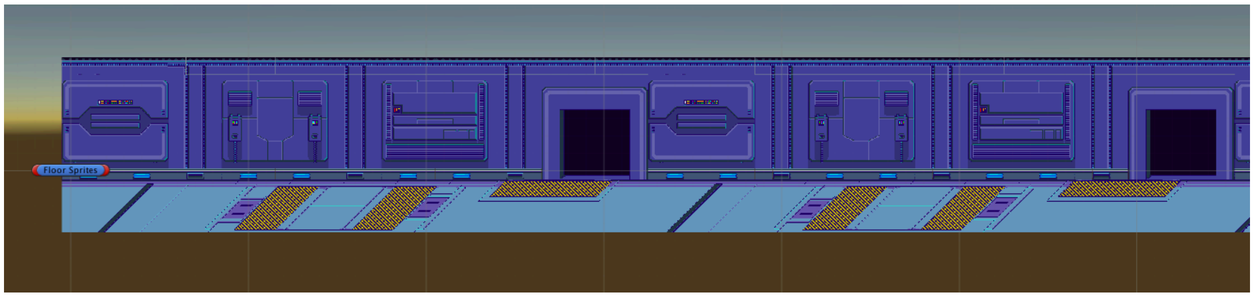
Create an empty GameObject and **name** it **Floor Sprites**, and then **reset** its Transform. **Change** the icon to a different colored capsule.

Add the GridSnapper component. Select **tiles_image_168** from **tiles_image** in the Project window and **drag** it onto the Scene view. Make this GameObject a **child** of **Floor Sprites** GameObject.

Now do the same as you did with the wall sprites, but this time, make it four tiles tall.

In this case, work downward from the initial tile.

Keep at it until you have a sizable level. Remember to **Snap To Grid**!



Congratulations, you finished the hardest part! Take a moment to save your scene and maybe even leave your chair for a few minutes.

There's a little more to set up before you're ready for the next chapter.

Prefab creation

If you've ever wondered how to create expansive games without painstakingly building each and every element, you're in luck. This section is all about using prefabs to save time, energy and a good bit of frustration.

In fact, I'm pretty sure the wall and floor sprites are begging to be your first prefabs.

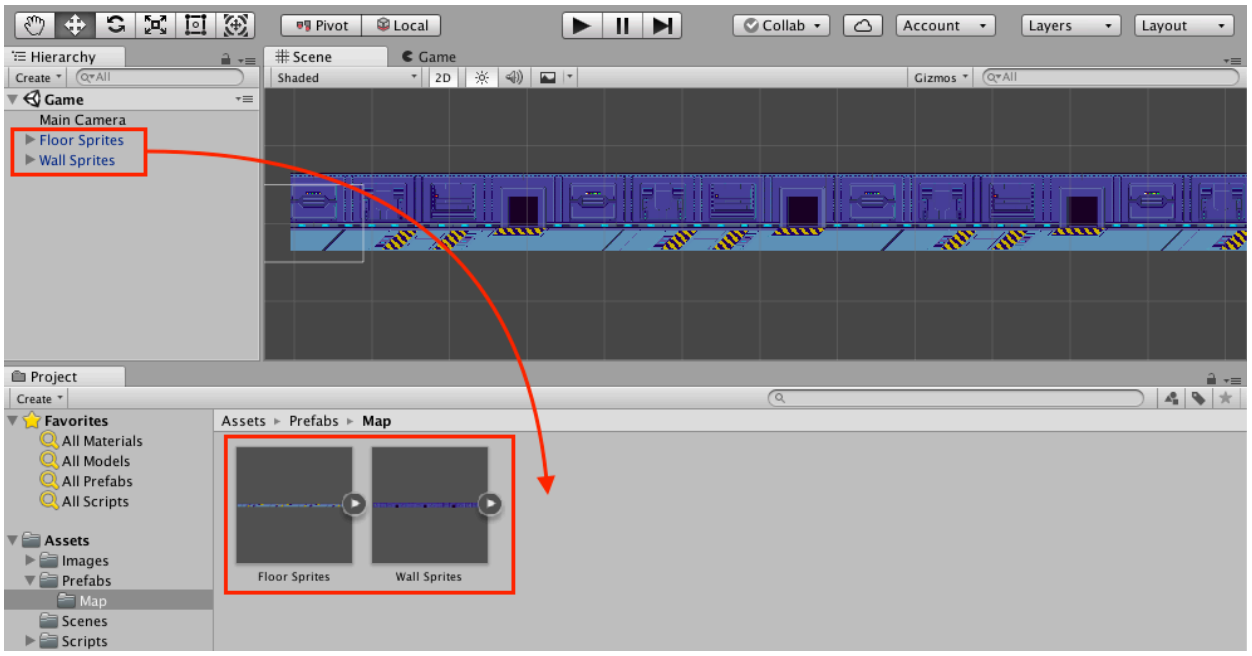
The first step is saving the wall and floor sprite GameObjects so that you can reference them in the future. Wait, what? *Save* GameObjects? How is that even possible?

A prefab is a template of a GameObject that stores the object and its children. It's like the master copy. If you make a change to the original prefab, its instances also change.

Save the sprites!

Go to the Project view, right-click the **Prefabs** folder, and create a child **Map** folder inside of that.

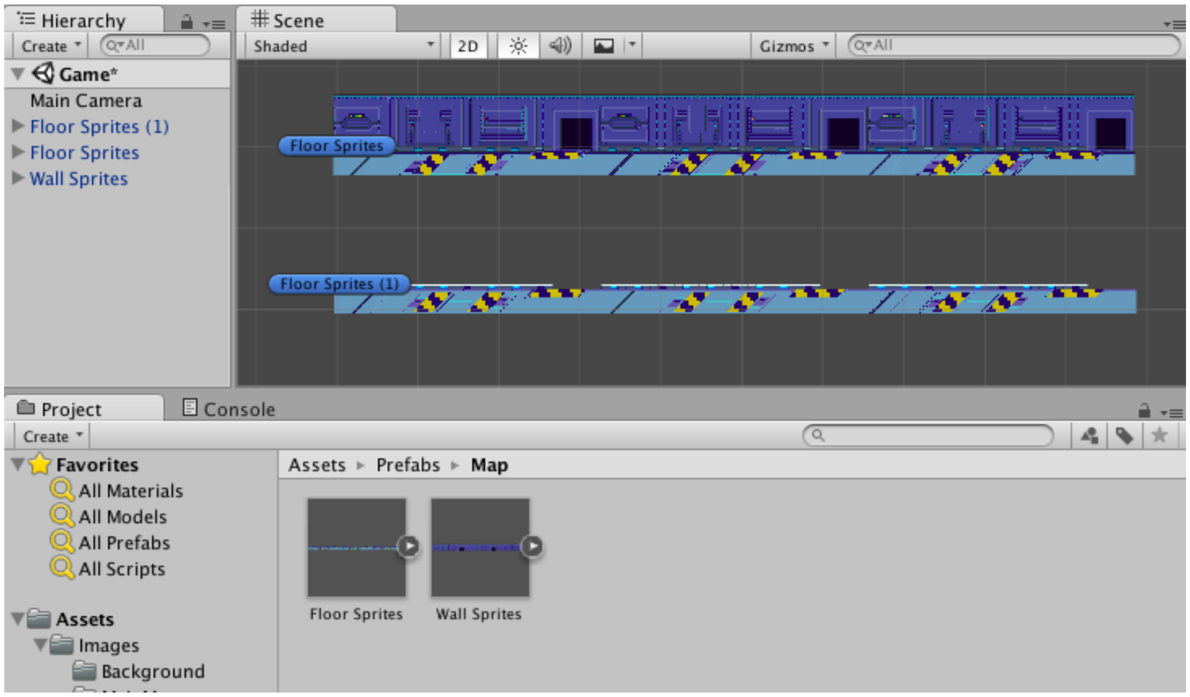
Drag the **Floor Sprites** GameObject from the **Hierarchy** to the **Project window** and repeat with the **Wall Sprites**.



You've just created two prefabs in Assets\Prefabs\Map. Both the floor and wall sprites turned blue in the Hierarchy to let you know that these GameObjects are linked to a prefab.

Note: You can unlink a GameObject from a prefab by going to **GameObject\Break Prefab Instance** in the menu at the top.

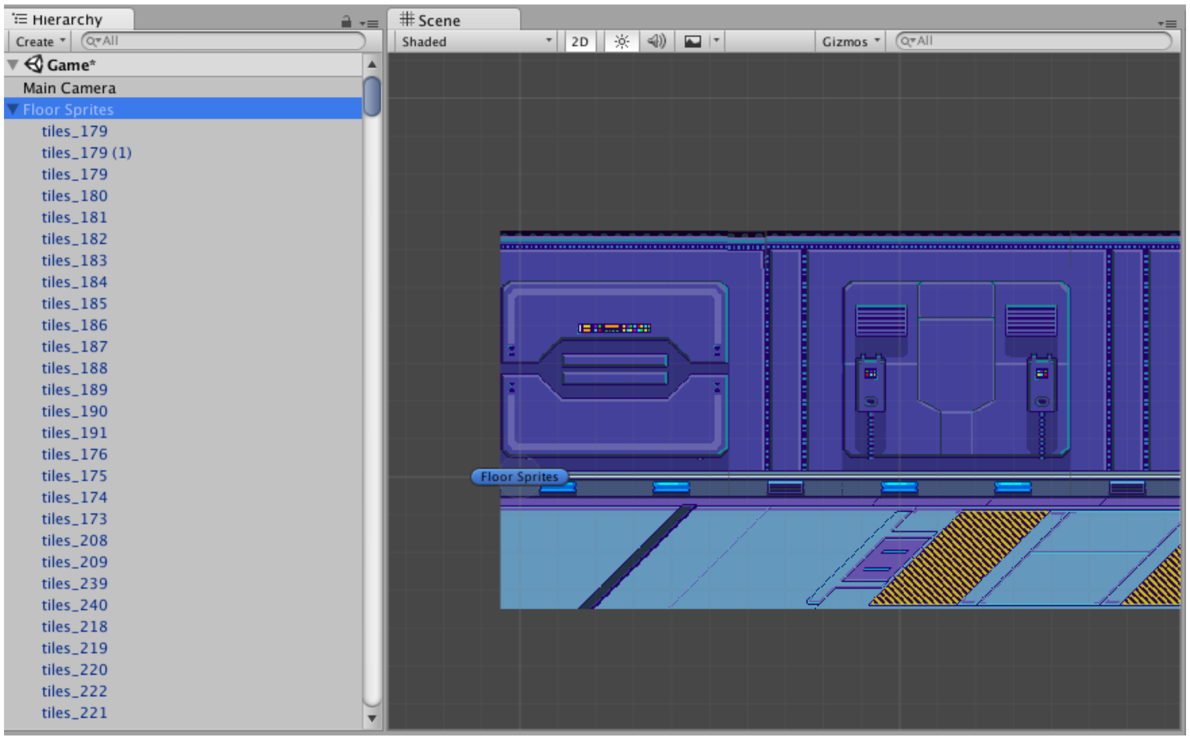
Just for fun: Try **dragging** the **Floor Sprites** prefab to the **Scene view**. Now you have two Floor sprites. Amazing! But you only need one, so delete the copy of Floor Sprites you just made.



You won't use the prefabs for the tilemaps you're making, but they are handy backups and make it easy to create variations in the future.

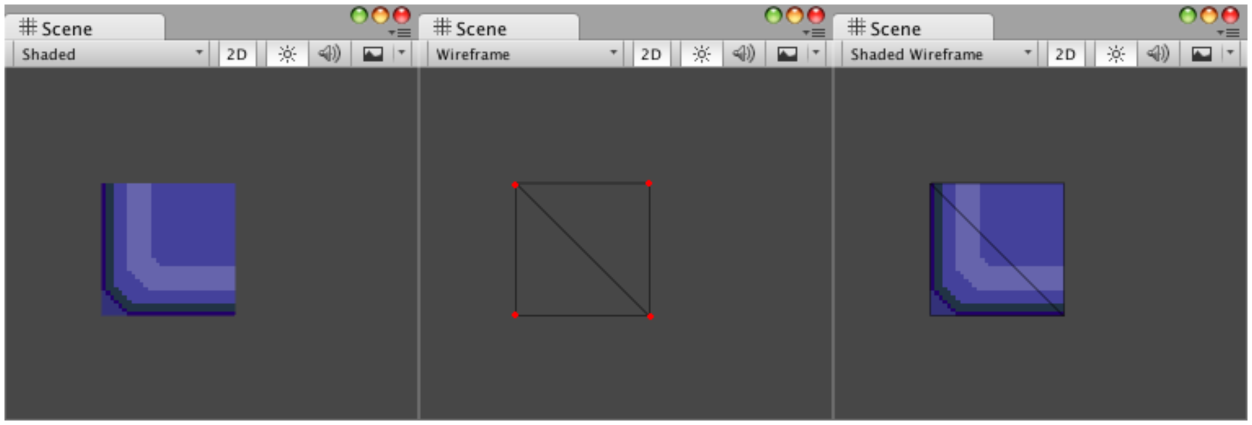
Optimizing tiled assets

Have a look at the Hierarchy. Wow, so many sprites!

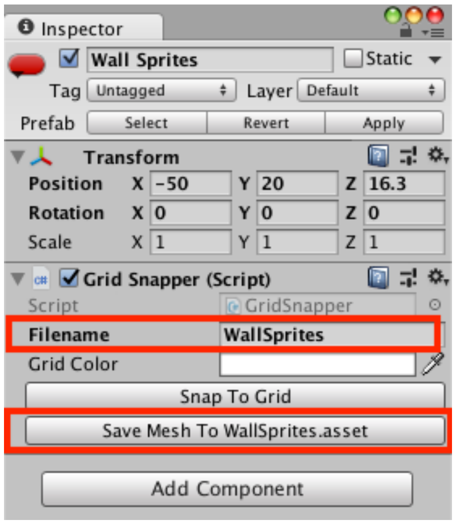


A crowded Hierarchy is a bad idea. You'll tidy it up by merging all the sprites into one object. By doing so, you'll turn the wall and floor sprites into meshes.

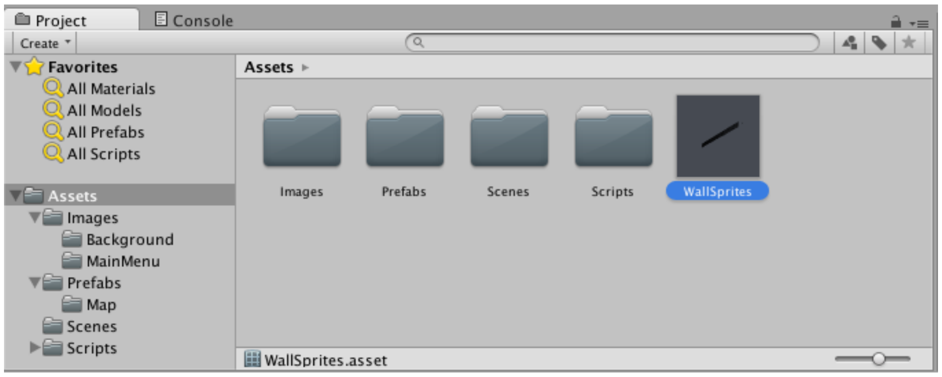
A **mesh** is simply a collection of points, named **vertices**, that are connected by lines, called **edges**. 2D sprites are essentially four connected points in a 3D space. Add something to draw on top of that, such as an image, and you have a 2D sprite.



Locate the **GridSnapper** component of the **Wall Sprites** GameObject, change the **filename** to **WallSprites** and click **Save Mesh to WallSprites.asset**.

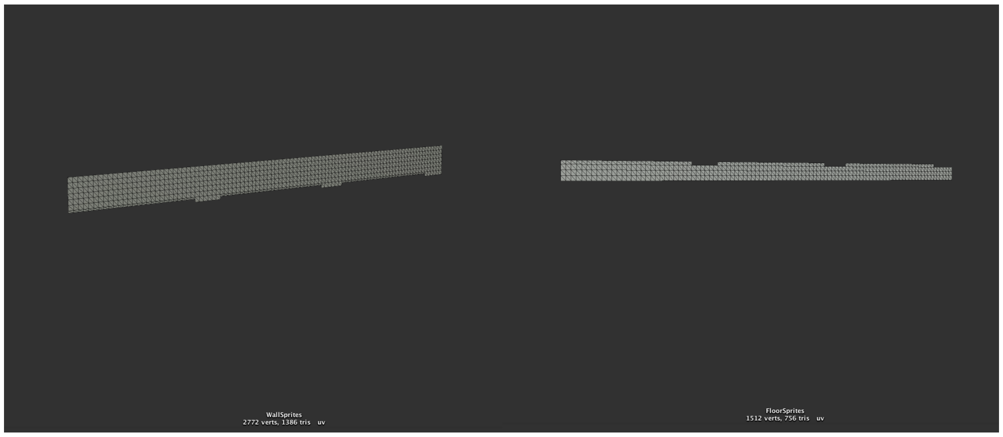

Inspector
Wall Sprites | Static
Tag Untagged | Layer Default
Prefab Select Revert Apply
Transform
Position X -50 Y 20 Z 16.3
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Grid Snapper (Script)
Script GridSnapper
Filename WallSprites
Grid Color
Snap To Grid
Save Mesh To WallSprites.asset
Add Component

Now you have a new WallSprites asset in the root of your Assets folder. It's a mesh, but you can't use it just yet.



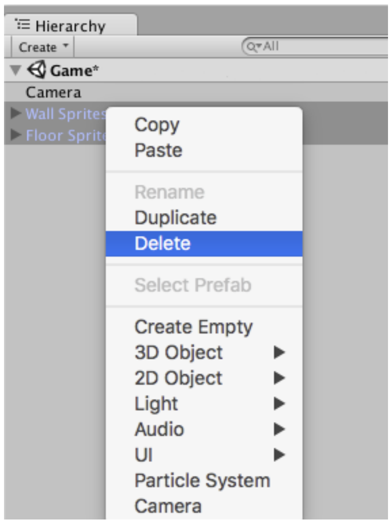
Do the same for the **Floor Sprites** GameObject, but this time, change the **filename** to **FloorSprites**.

If you check the mesh assets you made, they should look similar to this:



In the **Hierarchy**, remove **Wall Sprites** and **Floor Sprites** then select **both GameObjects**. **Right-click** and select **Delete**.

Don't worry; you saved the maps in the Prefabs\Map folder!

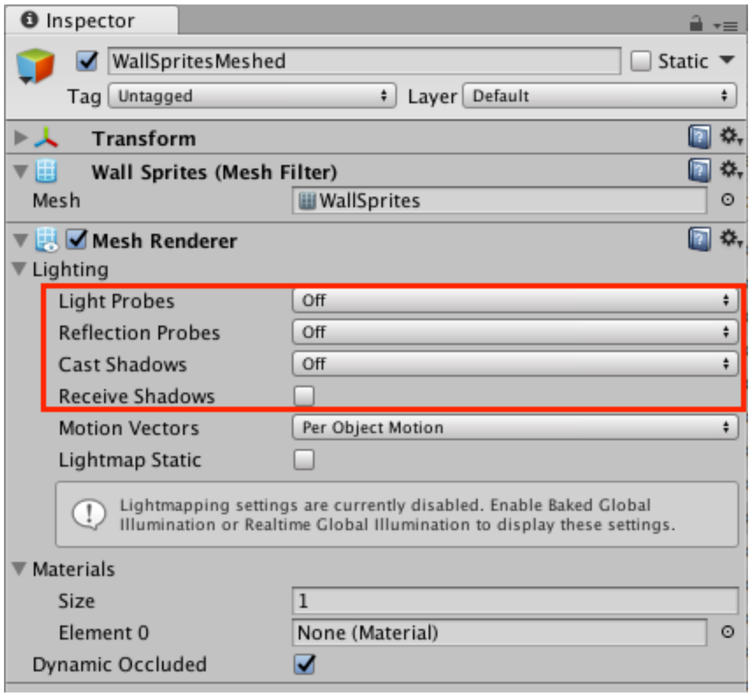


Add the meshes to the Scene

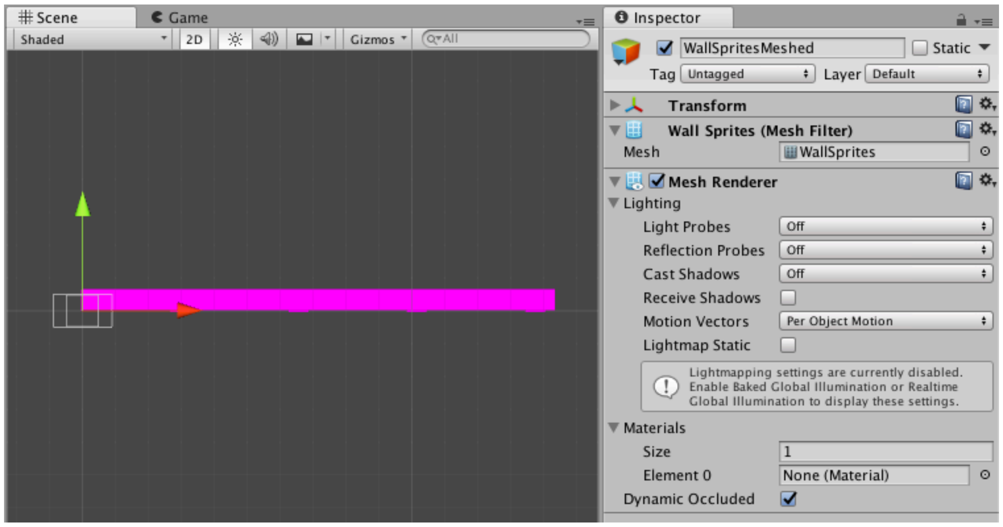
Create a new GameObject, name it **WallSpritesMeshed** and **reset** its transform. **Add** a **Mesh\Mesh Filter** component and a **Mesh\Mesh Renderer** component to it.

In the **Mesh Renderer** component, set **Cast Shadows** to **Off** and **uncheck Receive Shadows**. Set **Light Probes** and **Reflection Probes** to **Off**.

No need for reflections or shadows in a 2D game like PompaDroid. :]


Inspector
WallSpritesMeshed — Static
Tag: Untagged — Layer: Default
Transform
Wall Sprites (Mesh Filter)
Mesh: WallSprites
Mesh Renderer
Lighting
Light Probes: Off
Reflection Probes: Off
Cast Shadows: Off
Receive Shadows: ☐
Motion Vectors: Per Object Motion
Lightmap Static: ☐
Lightmapping settings are currently disabled. Enable Baked Global Illumination or Realtime Global Illumination to display these settings.
Materials
Size: 1
Element 0: None (Material)
Dynamic Occluded: ✓

Now drag the **WallSprites mesh** from the **Assets** folder to the **Mesh** field of the **Mesh Filter** you just created. This is the result:



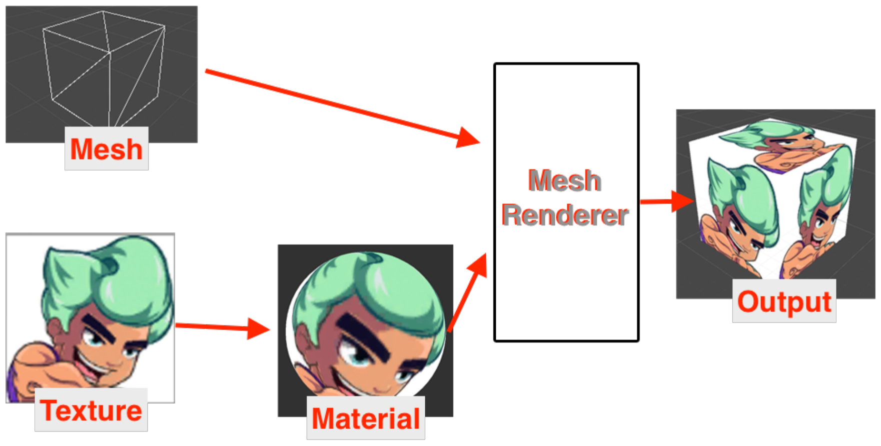
Inspector
WallSpritesMeshed — Static
Tag: Untagged — Layer: Default
Transform
Wall Sprites (Mesh Filter)
Mesh: WallSprites
Mesh Renderer
Lighting
Light Probes: Off
Reflection Probes: Off
Cast Shadows: Off
Receive Shadows: ☐
Motion Vectors: Per Object Motion
Lightmap Static: ☐
Lightmapping settings are currently disabled. Enable Baked Global Illumination or Realtime Global Illumination to display these settings.
Materials
Size: 1
Element 0: None (Material)
Dynamic Occluded: ✓

A brief explanation of mesh renderers

Did you build a fancy fuchsia mesh? Um, no, you didn't. At least not if you've been following these instructions!

To understand why you got that, you need to understand how a mesh renderer works:

- The Mesh Filter component is required so the mesh renderer can access the mesh to be rendered.
- Once it has access to the mesh field of the MeshFilter, the Mesh Renderer component draws the mesh on screen with the materials specified in the materials field.



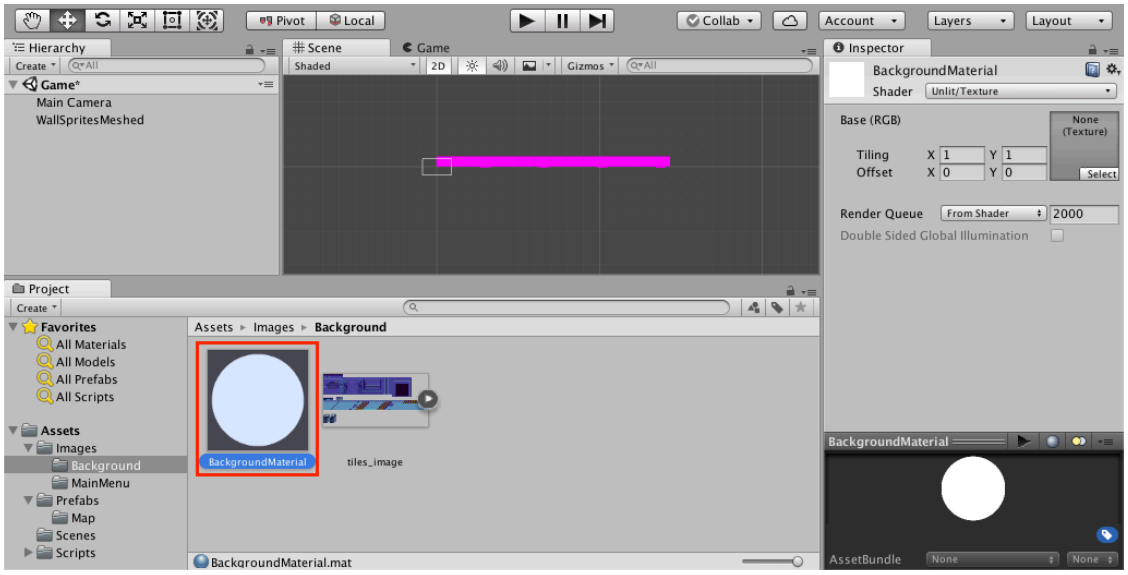
Materials and shaders

Materials are assets that define how a surface should render, and they determine what to do through shaders.

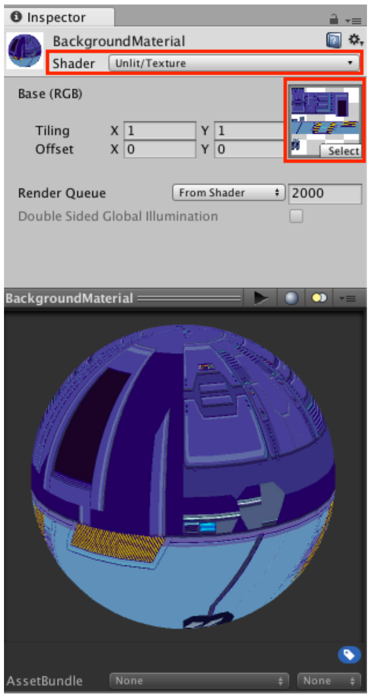
Shaders are special code files used to calculate the color of each pixel on the screen. They can also compute how light behaves on a given material.

You currently have a mesh but *no material*. It's solid magenta because that's what the Mesh Renderer draws when there's no defined material.

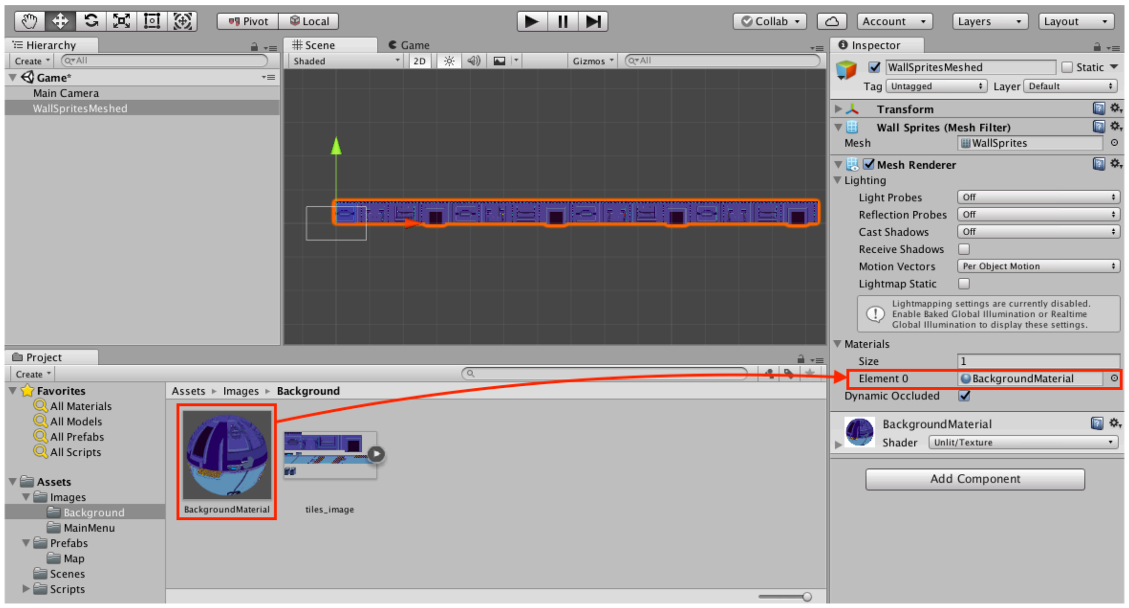
Navigate to the **Assets\Images\Background** folder and click **Create** at the top of the Project window and select **Material**. Name it **BackgroundMaterial**.



Select the **material** and set the **shader** to **Unlit/Texture**. Now drag **tiles_image** to the **Base(RGB) texture** field.



Select the **WallSpritesMeshed** GameObject, click the triangle next to **Materials** and **drag** the **Background Material** to **Element 0**.



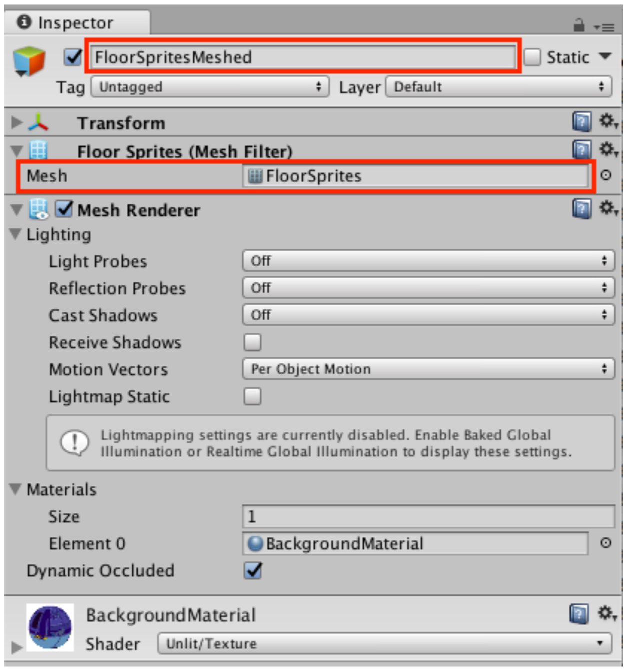
WallSpritesMeshed
Tag Untagged Layer Default
Transform
Wall Sprites (Mesh Filter)
Mesh WallSprites
Mesh Renderer
Lighting
Light Probes Off
Reflection Probes Off
Cast Shadows Off
Receive Shadows
Motion Vectors Per Object Motion
Lightmap Static
Lightmapping settings are currently disabled. Enable Baked Global Illumination or Realtime Global Illumination to display these settings.
Materials
Size 1
Element 0 BackgroundMaterial
Dynamic Occluded ✓
BackgroundMaterial
Shader Unlit/Texture

There you go! You've transformed a bunch of tiles into a nice mesh.

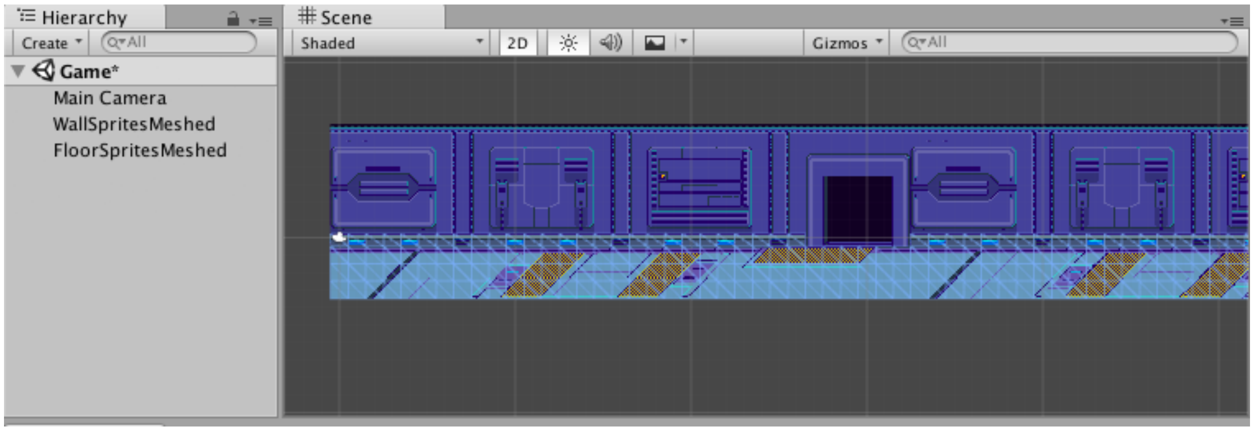
Lay down the floor

Now you just need to add the floor and you're all set!

In the Hierarchy, **duplicate** the **WallSpritesMeshed** GameObject and rename it to **FloorSpritesMeshed**. Replace WallSprites in the **Mesh** field with the **Floor Sprites** mesh.


Inspector
FloorSpritesMeshed Static
Tag Untagged Layer Default
Transform
Floor Sprites (Mesh Filter)
Mesh FloorSprites
Mesh Renderer
Lighting
Light Probes Off
Reflection Probes Off
Cast Shadows Off
Receive Shadows
Motion Vectors Per Object Motion
Lightmap Static
Lightmapping settings are currently disabled. Enable Baked Global Illumination or Realtime Global Illumination to display these settings.
Materials
Size 1
Element 0 BackgroundMaterial
Dynamic Occluded ✓
BackgroundMaterial
Shader Unlit/Texture

Whew!! All that work! Those sprites are now two simple, clean, textured meshes! Don’t you just love how clean and organized your workspace looks? :]



Where to go from here?

Nice work. Your tiled map is a thing of beauty! You have a sweet background and prefabs to help you create more levels.

In this chapter, you:

- Set up a tiled map for your level
- Optimized the tiled map by combining multiple sprites into meshes

You've got a running start on your beat ’em up game! Next, you’ll meet and work with the hero of the game, funky hairdo and all.

Chapter 3: Walk This Way

You've set the scene with a sleek mesh for the floor and background. Now you'll add some punch to the game by introducing the hero!

You've already added sprites to your game and the process for the hero is similar, so that's going to be easy enough.

However, a hero doesn't just stand around like some stiff, stale statue. Heroes must move, punch and maneuver to inspire the masses and destroy any droids that dare step in their way.

In this chapter, you'll add the hero and give him signature moves. Along the way, you'll learn about:

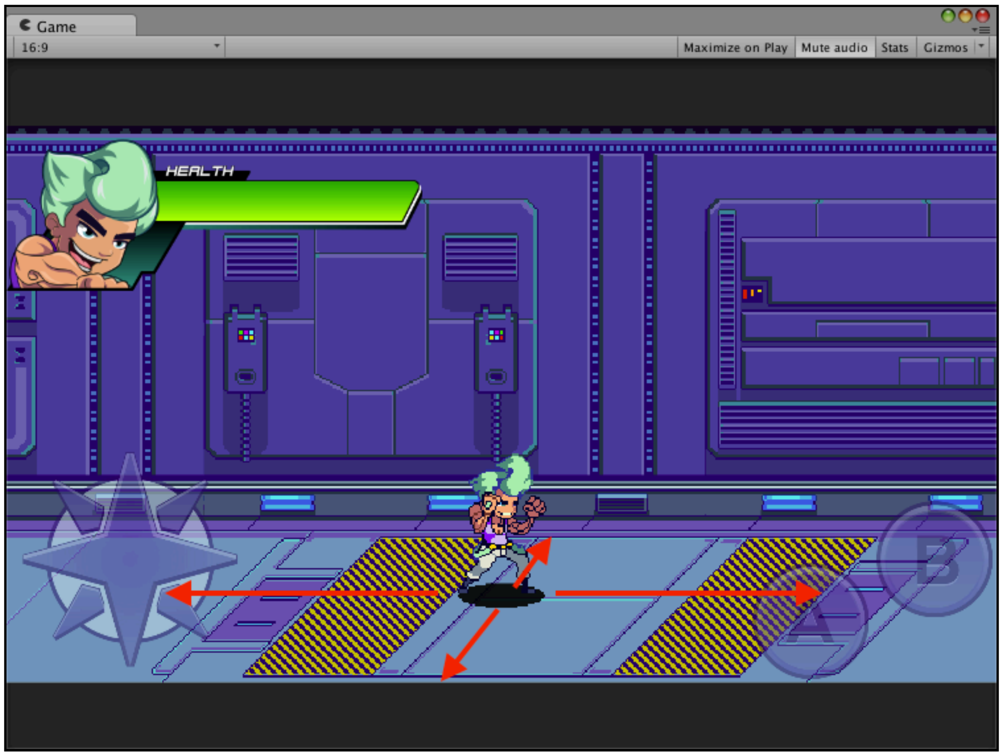
- Unity's animation system, **Mecanim**, and physics engines in general
- Moving the hero around the screen and responding to **user input**
- Animating a 2D character
- Following a GameObject with a camera
- Keeping GameObjects in the scene

By the end of the chapter, you'll have the hero confidently strutting his stuff!

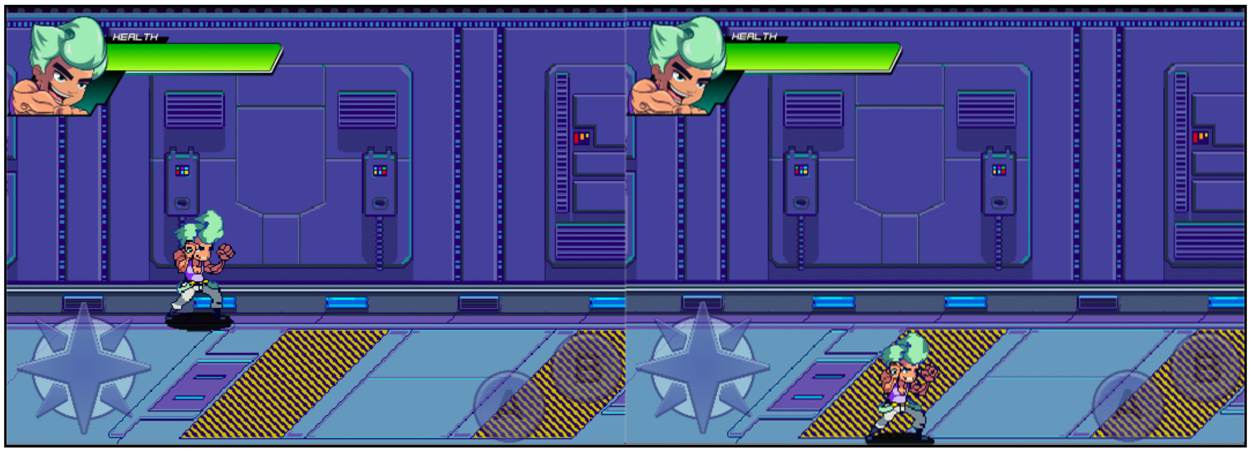
Walking in style

Before creating your beat 'em-up masterpiece, think through how the hero will move and some of the other gameplay basics.

In most beat-em-up games, the hero moves horizontally and vertically.



In this style of game, the hero also maintains its size, meaning that the character looks the same regardless of its distance from you.



Size and projection: A matter of perspective

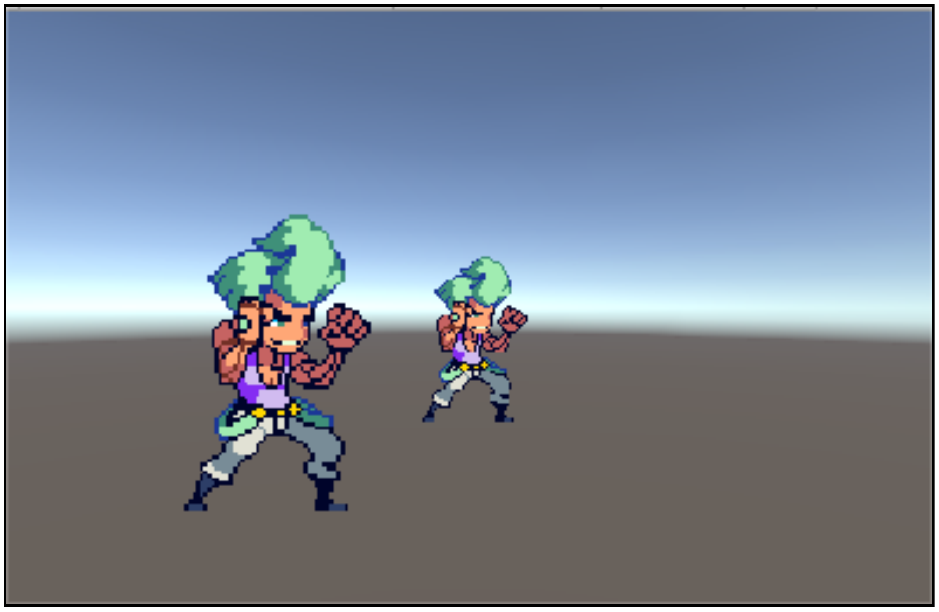
Unity includes a variety of projection modes that let you control perspective in your game.

In this section, you'll go over the basics you need to know to set up the game scene. You don't need to do anything other than read and absorb.

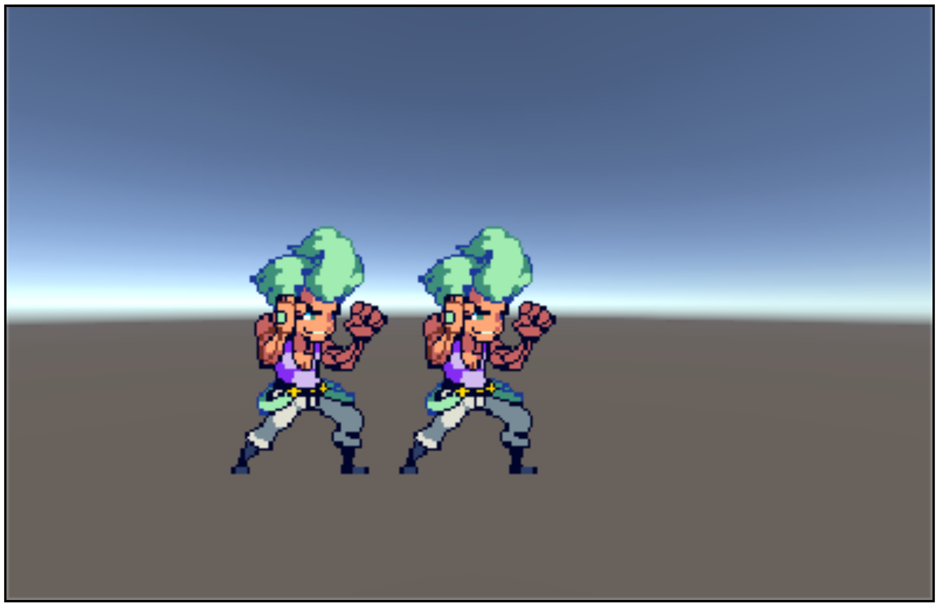
Cameras are objects that capture and convert the game world you design to an image on screen for the player through a process known as **projection**.

Graphics projection is the process by which three-dimensional data renders to a two-dimensional screen. The two most common projection modes are **orthographic** and **perspective**.

Perspective shows objects the same way you see them in the real world where distant objects appear smaller than those that are close. In the below image, you see how the center-most sprite, which is further away, looks smaller than the left-most sprite.



Orthographic, on the other hand, ignores the distance of objects from the camera. Objects near and far are the same size at all points along the camera's view. The image below shows the same scene rendered with an orthographic perspective.



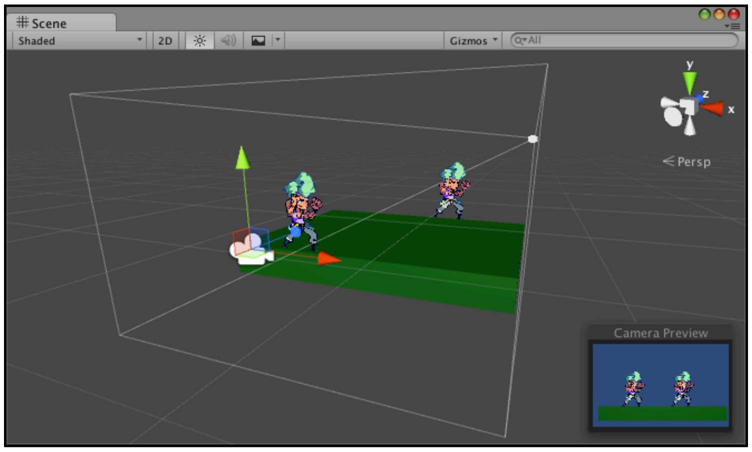
Which should you choose for PompaDroid? Orthographic projection, of course! He exists in a flat, narrow world, so keep it simple with othrographic.

On to character movement. If everything looks the same size, how does the player know whether the hero needs to come closer or move away? How do you show movement across the scene?

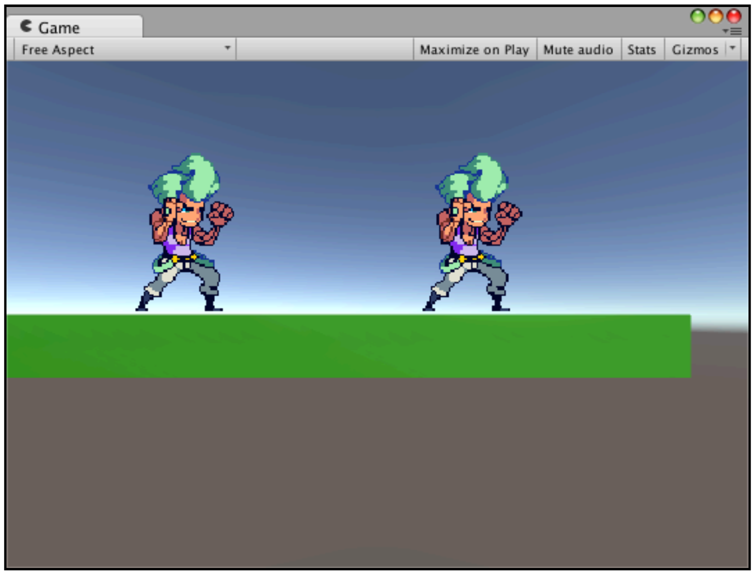
Left-to-right movement is easy to solve. You just move the hero and camera left to right to create movement.

How about top-to-bottom movement? Moving the camera in and out of the scene might do it, but you could really mess up gameplay with all that movement. This sounds like a job for rotation!

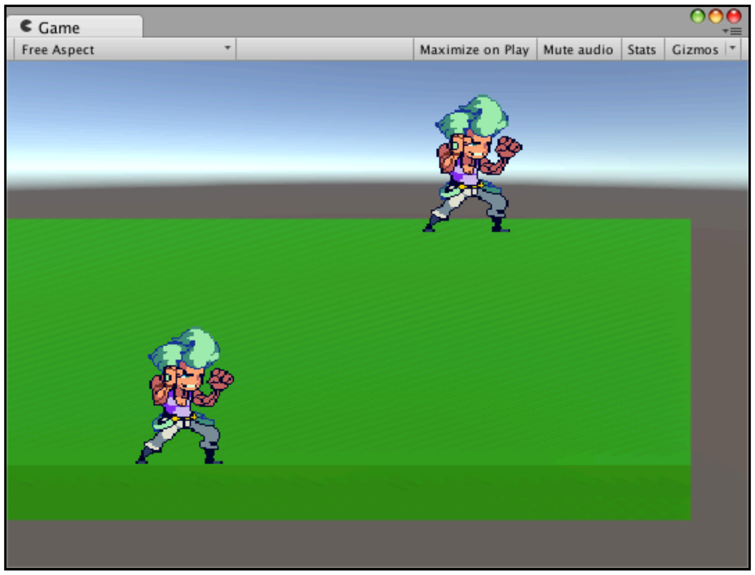
Imagine the scene only contains a green box and two sprites at opposite ends, like this:



This is how a camera with no rotation would show the scene:

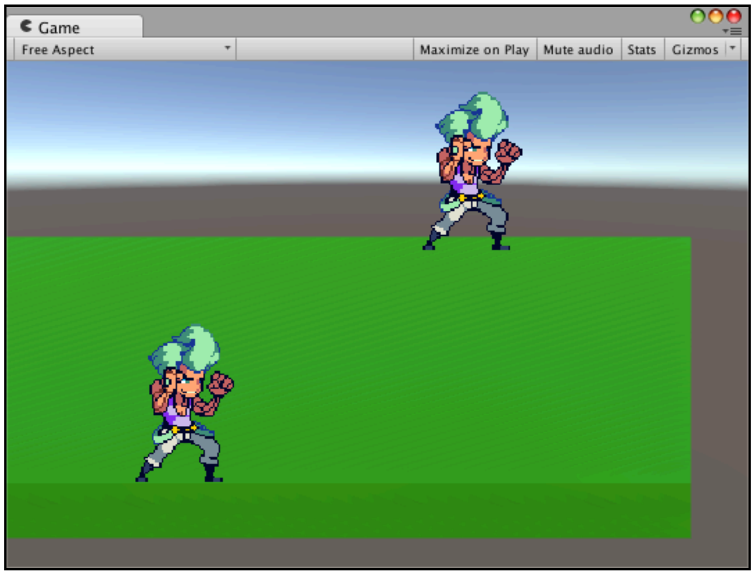


If you tilted the camera downward by 30 degrees, you'd show the sprites against the floor, like this:



Does the hero look like he shrank to you? Maybe a monster came along and stepped on him — or, maybe it's the camera angle messing with things.

The issue is that the camera is looking at the sprite at an angle, just as you told it to do. Rotating the sprite 30 degrees on the x-axis corrects the problem:



Now they look tough — and correctly proportioned.

Note: A 30-degree rotation provides enough tilt to give the 2D world some depth. You can adjust this to your liking, but this book will stick with 30 degrees.

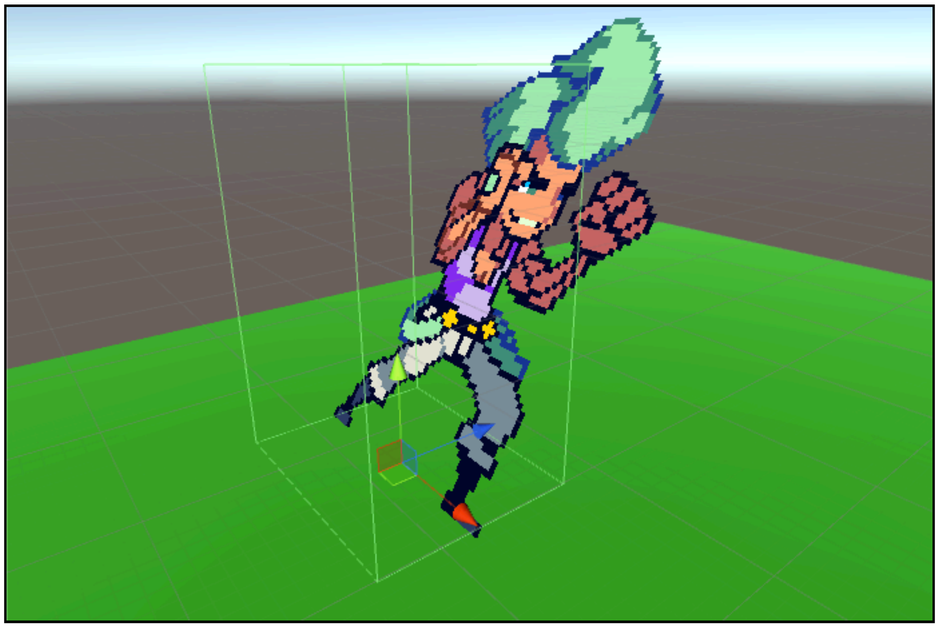
Turn your attention to **Unity's Physics Engine**!

Unity physics comes in two dimensions: **2D** and **3D**.

Unity has an integrated 3D physics engine name **PhysX** that you'll use for this game. The reason why you'll use 3D physics is that it allows depth calculation. In contrast, 2D physics doesn't calculate depth unless you add additional code.

In short, using 3D physics makes creating a beat-em-up game much easier than working with 2D physics.

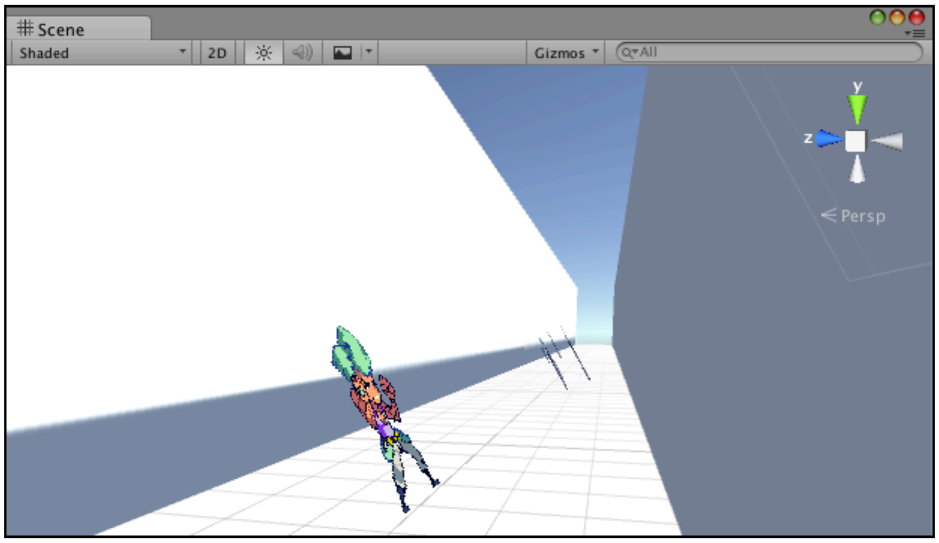
PompaDroid will use simple box colliders to approximate characters in 3D space. Unlike the sprites used in the game, this box is not rotated.



Everything the hero encounters — including enemy droids — will utilize box colliders. Boxes require relatively few physics calculations, so you can have an army of enemies on screen, and the game will still run buttery-smooth on mobile devices.

One more thing to go over before you dig in: unfortunately, the hero must never escape the screen. Did you see his hair and menacing grimace? It looks like he tangled with a uranium core or something! Who knows what would happen if he escaped.

You'll basically imprison him and all his enemies inside a physics tunnel that'll look like this:



Don't sweat the white walls. They'll be invisible to the camera but not to the physics engine. The characters will not be allowed to pass through these walls!

The life of a hero isn’t an easy one. :]

With all that said, here’s the recipe you’ll follow to add movement to PompaDroid:

- A camera with orthographic projection, rotated by 30 degrees along the x-axis
- Sprites tilted backward 30 degrees to “cancel” the camera’s rotation
- Configuring your game to use Unity’s 3D physics engine to handle collisions

Set Up Your Game Environment

Now you’re to the point where you’ll start doing things, so grab a beverage, get into your zone and prepare to build!

In this section, you’ll set up the camera and sprites, then implement a state machine to help with animations. Setting these pieces up next will make your life a little easier.

Camera

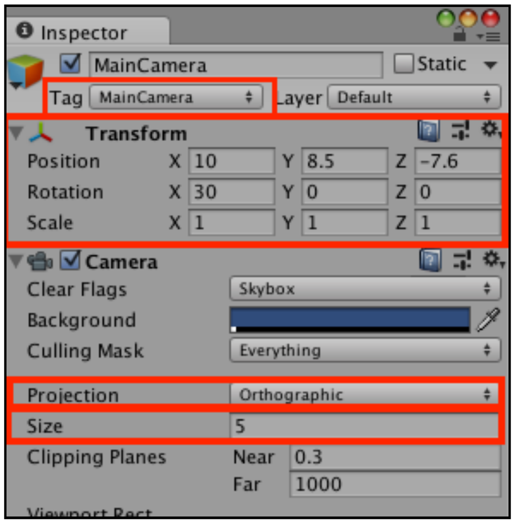
First, open the **Game** scene. Look in the Hierarchy for a camera. If you don’t have one, right-click and select **Camera** to add one.

Rename **Camera** to **MainCamera**, and then go to its transform and set the **Position** to `(X:10, Y:8.5, Z:-7.6)`, **Rotation** to `(X:30, Y:0, Z:0)` and **Scale** to `(X:1, Y:1, Z:1)`. Change its **Tag** to `MainCamera` if it is shows up as `Untagged`.

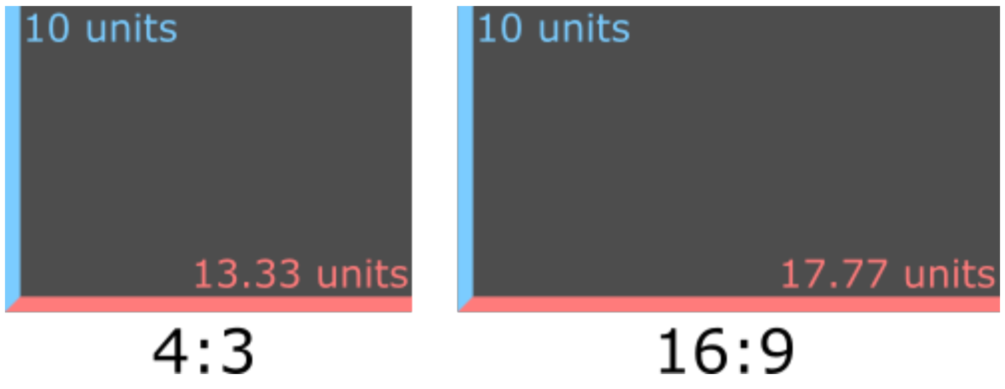
The coordinates look like magic arbitrary numbers, but they serve a specific purpose. The x-coordinate affects which part of the map can be seen horizontally. With a rotated camera, both the y and z-coordinates affect how much of the map can be seen vertically. As you will see later, the camera position you set represents the ideal view of the map's starting point.

Note that you rotated the camera 30 degrees downward along the x-axis, as discussed in the previous section.

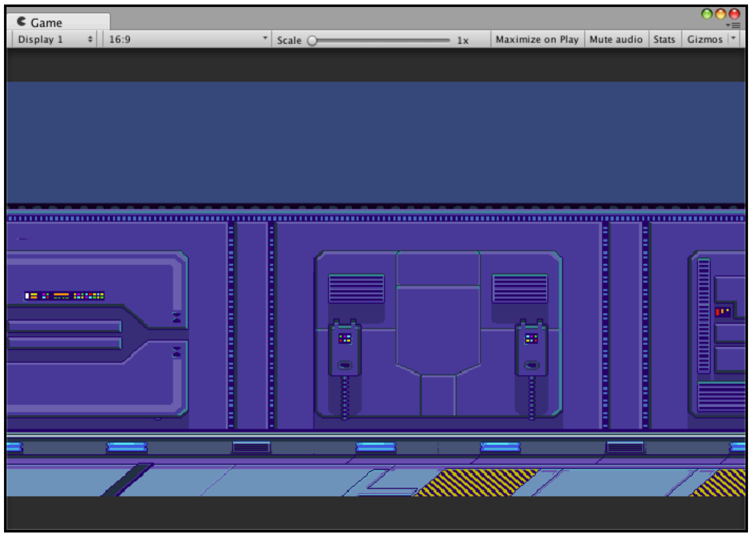
Set the **Projection** to **Orthographic** and **Size** to `5`.


Inspector
MainCamera Static
Tag MainCamera Layer Default
Transform
Position X 10 Y 8.5 Z -7.6
Rotation X 30 Y 0 Z 0
Scale X 1 Y 1 Z 1
Camera
Clear Flags Skybox
Background
Culling Mask Everything
Projection Orthographic
Size 5
Clipping Planes Near 0.3
Far 1000
Viewport Rect

The camera’s size is half the height of an orthographic camera in world units and independent of the camera’s aspect ratio. For example, a camera with a size of 5 and aspect ratio of 4:3 would be 13.33 units wide and 10 units tall, while a 16:9 aspect ratio would be 17.78 units wide and 10 units tall.



With that done, you need to work on the sprites’ rotation. Suddenly the hero’s playground looks a bit wonky.

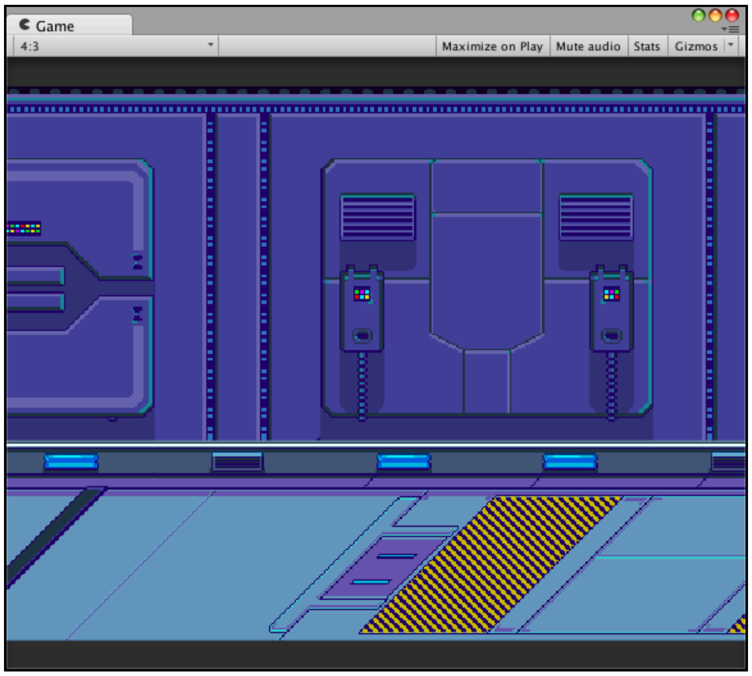


Create a new empty GameObject and name it **TileMaps**. This will act as the parent of the background meshes you made in the previous chapter.

Open up the Inspector, and in the transform, set **Position** to `(X:0.5, Y:-0.28, Z: 4.67)`, **Rotation** to `(X:30,Y:0, Z:0)` and **Scale** to `(X:1, Y:1, Z:1)`.

In the Hierarchy, drag **WallSpritesMeshed** and **FloorSpritesMeshed** over TileMaps to make them its children. **Reset** the transform for both children.

Now the background meshes should look correct because they’re facing the camera at the same angle the camera is rotated. Good work!



What do you think is next? The hero, of course! But before you add controls or the hero, you'll create an animation system — it's a little extra work now that'll pay off later.

Implementing the state machine

In most games, characters "move" because of various animations, each of which portrays an action.

Knowing how to run an animation isn't enough — you also need to know when to play it. For example, when the hero is going slowly, the game should use a walking animation. When the player makes him jump, the hero should look like he's sprung into action.

There are a few ways to transition between a sprite's animations, and for this book, you'll work with a state machine.

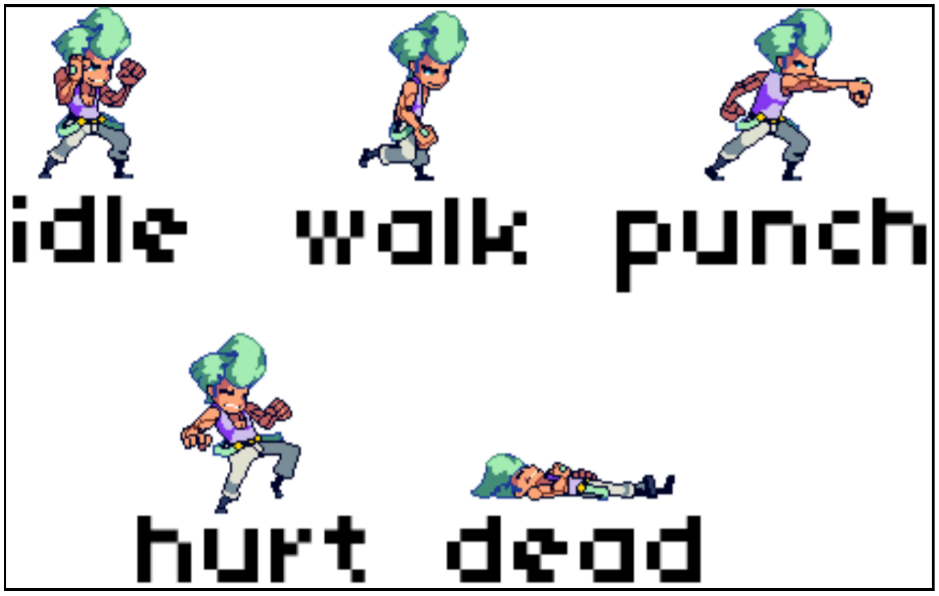
A **state machine** is an algorithm that tracks the state of a given object and executes various actions based on its current state. State machines can have only one active state at any given time but can transition between states based on triggers you define.

To understand this better, imagine the main character and list the things it can do:

- Idle
- Walk
- Punch

Now think about the requirements for each with the assumption — for the sake of simplicity — that only one activity can happen at once:

- If the state is idle then it can't walk or punch.
- If the state is walk then it can't idle or punch.
- If the state is punch then it can't idle or walk.



Your hero will have three states that are under the user's control. There will also be a hurt and dead state, but these will be controlled by code.

Import **Hero Walk and Idle Sprites.unitypackage** from the **Unity Packages** folder. It contains the sprites you'll use to animate walk and idle states. You'll also see a shadow sprite for the hero.

You might have noticed the excessive padding on the hero sprite assets — the sprite is tiny compared to the image size! This is to accommodate all of the hero's actions.

All images in an animation need to be the same size. When you play an animation, the game replaces the current sprite with the next, and when they're all sized the same, you get a clean effect that works similarly to a flipbook.

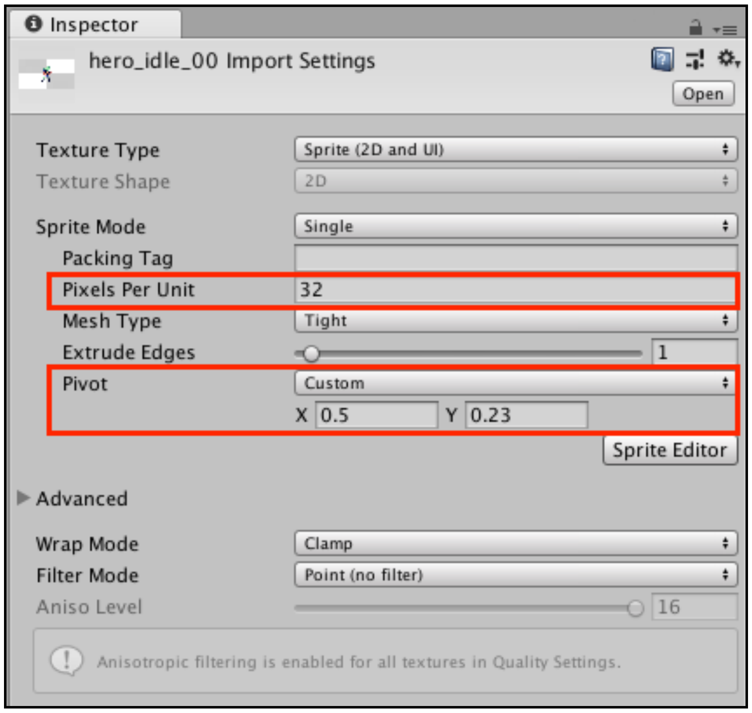
In some animations, the object needs to move away from the center of the canvas. For instance, when the hero jumps to make a kill or falls dead after a run-in with a trap.

There are two things you need to do to work with these kinds of animations:

- Make the canvas bigger, so there's room to place the sprite at the right spot to accommodate relative animations.
- Set it so that when the character moves around the screen, such as when the user presses or taps the movement button, the entire character canvas moves within the layer.

The imported sprites are set to `32` pixels per unit, and their pivot is at `(0.5, 0.23)` of the total image size. You can see it in the **Sprite Import Settings**.

Pivots are points around which a sprite will rotate and scale and uses normalized values. The values of 0.5 and 0.23 represent percentages, specifically - 50 percent of the width from the left, and then 23 percent of the height from the bottom.



Texture Type: Sprite (2D and UI)
Texture Shape: 2D
Sprite Mode: Single
Packing Tag:
Pixels Per Unit: 32
Mesh Type: Tight
Extrude Edges: 1
Pivot: Custom
X 0.5 Y 0.23
Wrap Mode: Clamp
Filter Mode: Point (no filter)
Aniso Level: 16
Anisotropic filtering is enabled for all textures in Quality Settings.

Adding the hero

This game doesn't deserve its name until it has procured a properly pompadoured protagonist!

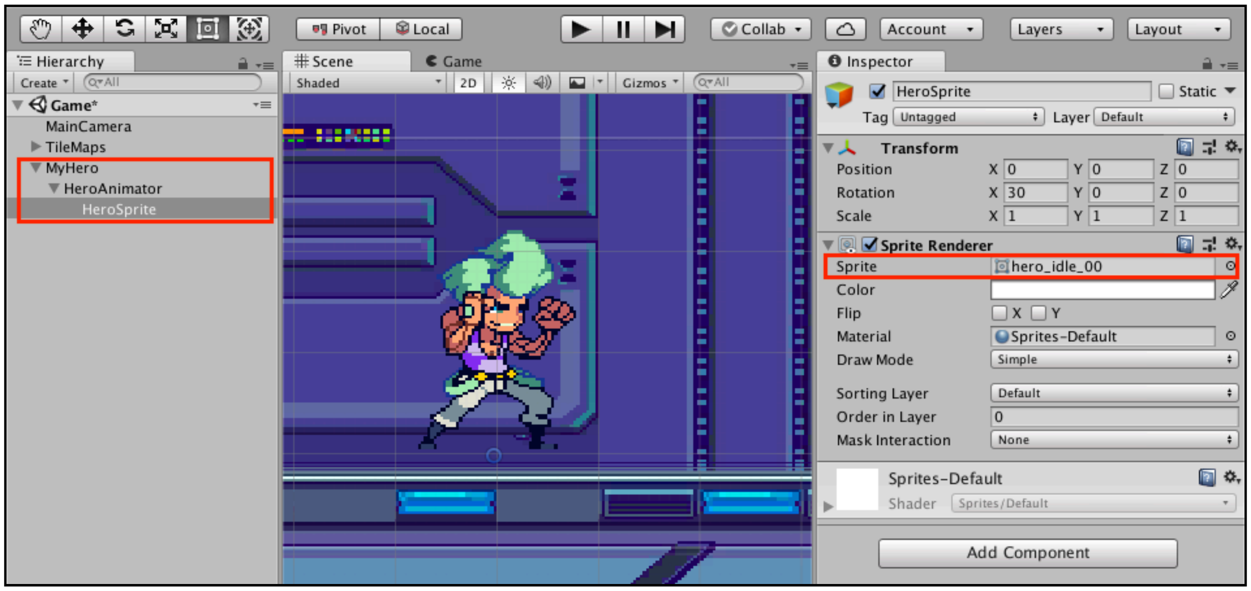
In the Hierarchy, create a new empty GameObject, **reset** its Transform, set its **Position** to `(X:5, Y:0, Z:0)` and change its **Name** to **MyHero**.

Create another empty GameObject and make it MyHero's child, **reset** its Transform and **name** it **HeroAnimator**.

In the Hierarchy, create a **2D Object\Sprite**. With this, you're making a new GameObject that has a SpriteRenderer component. **Name** this new GameObject **HeroSprite**.

Make **HeroSprite** a child of **HeroAnimator** and **reset** its Transform and set its **Rotation** to `(X:30, Y:0, Z:0)`.

Drag the **hero_idle_00** sprite from the **Assets\Images\Sprites\Hero** folder in the Project view to **HeroSprite's** SpriteRenderer **Sprite** property.



Game*
MainCamera
TileMaps
MyHero
HeroAnimator
HeroSprite

HeroSprite Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 30 Y 0 Z 0
Scale X 1 Y 1 Z 1

Sprite Renderer
Sprite hero_idle_00
Color
Flip X Y
Material Sprites-Default
Draw Mode Simple
Sorting Layer Default
Order in Layer 0
Mask Interaction None

Sprites-Default
Shader Sprites/Default

Hero created — well, at least in the sprite is set up. There's still a ton of (fun) work ahead!

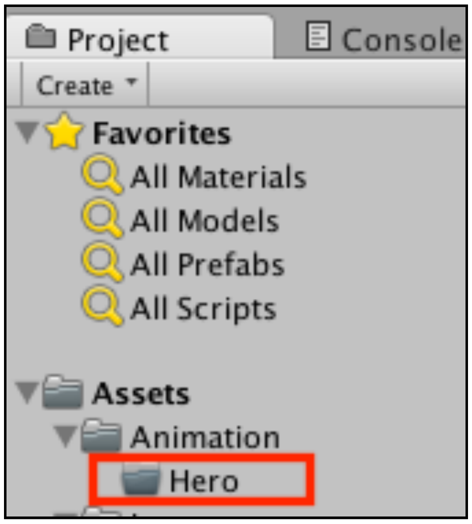
Press the **Play** button. Notice that the pompadoured guy is as about as lifelike as a painted wall. He's an extreme wallflower because you haven't given him the gift of motion, aka animation.

Click **Play** again to stop playback.

Animating the hero

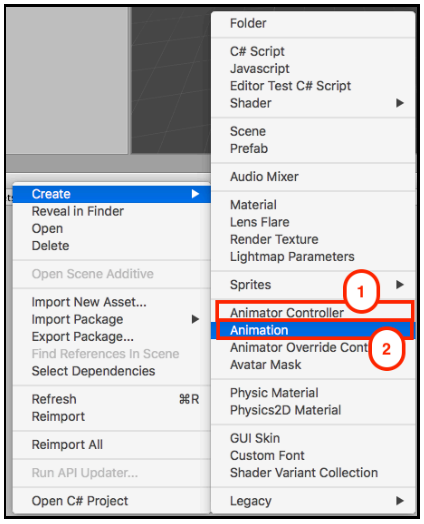
In the Project view, open the **Assets** folder, create a new folder inside of it and name it **Animation**. You'll save all animation for the game in here.

Double-click the **Animation** folder, make a new folder then name it **Hero**. This is for the hero's animations.

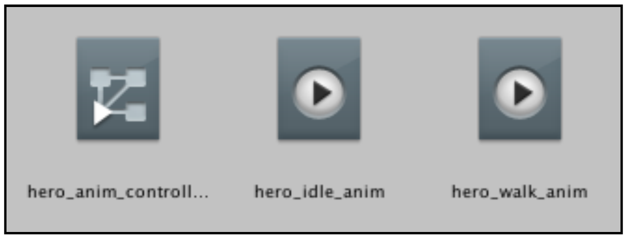


Now you're in the part where you add animation clip assets. Open the **Animation\Hero** folder and **right-click** to toggle a menu.

1. Create an **Animator Controller** and name it **hero_anim_controller**.
2. Create two **Animation** assets and name them **hero_idle_anim** and **hero_walk_anim**.



You made three new assets: an animator controller asset and two animation clips. All three are integral parts of Unity's animation system — Mecanim.



Mecanim in a nutshell

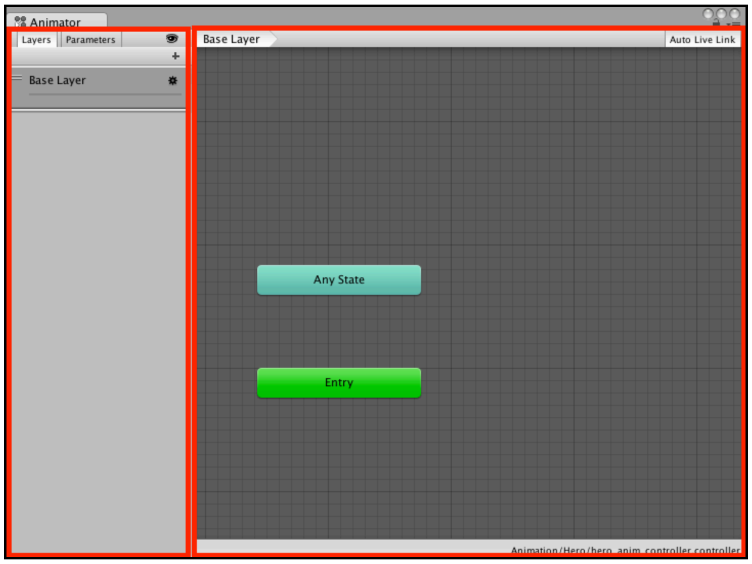
Mecanim is the system that handles animations in Unity. It uses animator controllers that contain state machines to handle various animation states.

Each state contains an animation clip that defines which animation will play once a certain state is reached. When animations finish or other conditions are met, you can implement transitions to move to other states.

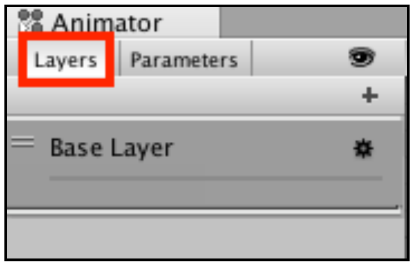
Animator controllers handle the state, mix and sequence of animations. They can be used by GameObjects through the **Animator** component.

Animator controllers can be edited in the **Animator view**, which is accessible from the top menu bar via **Window\Animator**. Open it up and select **hero_anim_controller** in the Project view so that there’s something in the Animator view to look at while you read on.

The Animator view has two main sections: **Layers and Parameters** and **Layout**.

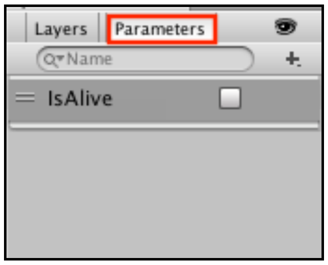


The layers and parameters pane contains two tabs: the animation **Layers** tab and the events **Parameters** tab.



In **Layers** you employ various state machines to handle multiple layers of animation. One way you'd use them is for making multiple body parts move at the same time. It allows a multi-layered approach that's essential when making complex 3D animations.

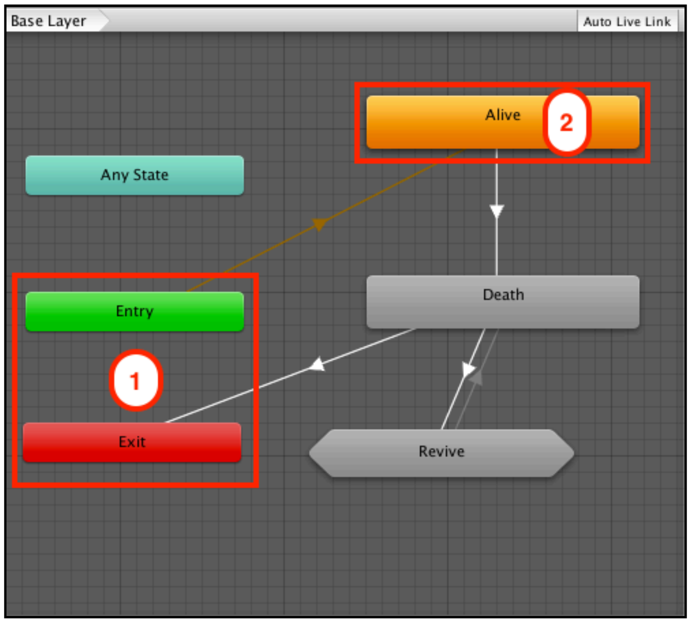
All that said, PompaDroid is less complex; you'll stick to the default layer, aptly named **Base Layer**.



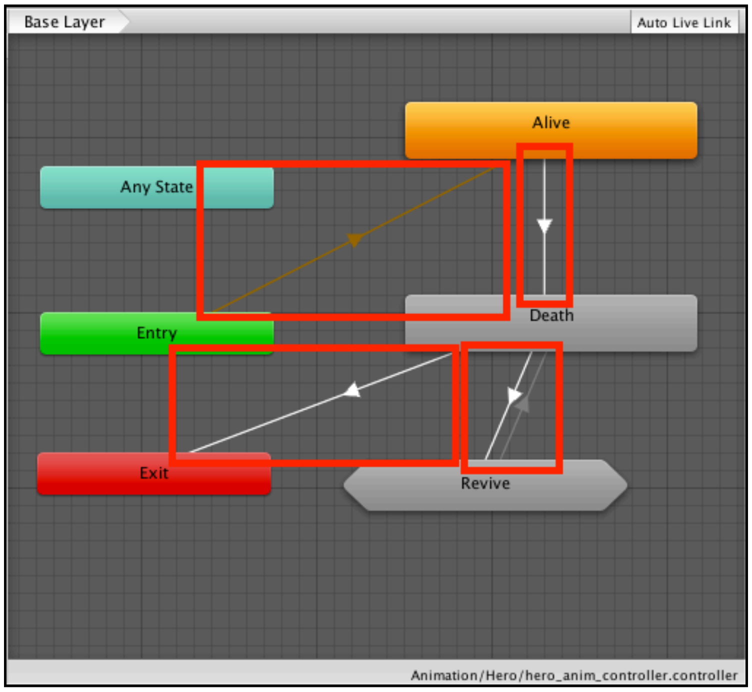
In **Parameters**, you find variables — aka parameters — that affect the animation. Later you will create a Boolean named `IsAlive` that enables the Animator controller to transition to a death animation when it's set to false. These parameters are also accessible outside the Animator controller via other components and scripts.

Turn your attention to the layout area on the right side of the Animator view. Although you won't see them at this point, just know that each Animator contains an **Entry** and an **Exit** node.

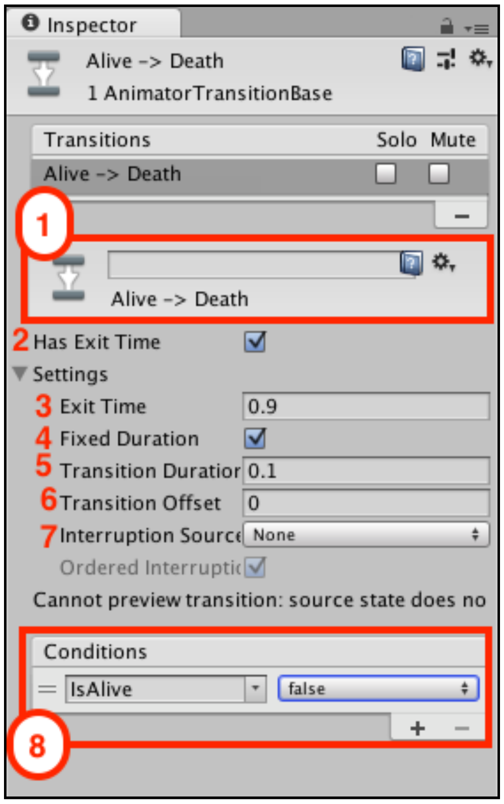
1. The **Entry** node defines which node is called first. The **Exit** node is used when the animation finishes.
2. In this case, Entry calls the default state which is represented as the **Alive** node.



Nodes are connected by **State Transitions** that are depicted as arrows in this image:



Each transition has many parameters that can modify what happens between states. To see these parameters, you'd select a transition and check the Inspector:



Alive -> Death
1 AnimatorTransitionBase

Transitions	Solo	Mute
Alive -> Death	☐	☐

Alive -> Death

Has Exit Time ✓
Settings
Exit Time 0.9
Fixed Duration ✓
Transition Duratior 0.1
Transition Offset 0
Interruption Source None
Ordered Interruptic ✓
Cannot preview transition: source state does no

Conditions
IsAlive false

1. **Transition name:** The name of this transition. You use this when checking animation states.

2. **Has Exit Time:** Determines whether this transition's condition can take effect any time or only during the state's exit time. Check the box to limit the animation to an exit time of your choosing.

3. **Exit Time:** Sets the time, in a normalized value, that must pass before the transition takes effect — `0.25` would be 25 percent. Check **Has Exit Time** to enable this parameter.

4. **Fixed Duration:** Toggles between using seconds or normalized time to set a transition's duration.

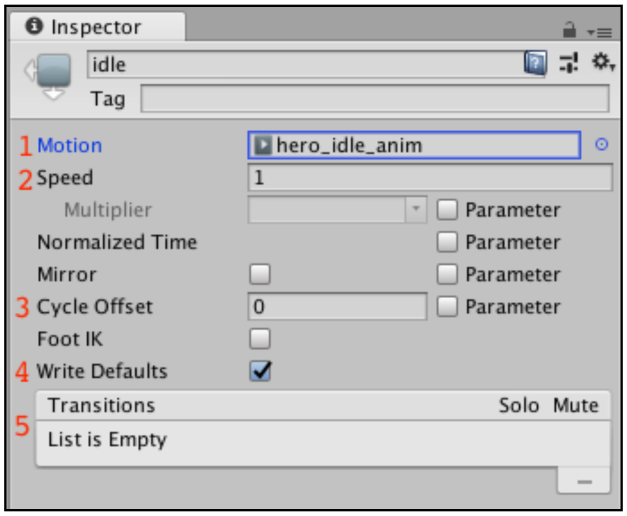
5. **Transition Duration:** Defines how long a transition should last. You also see this visualized in the transition graph.

6. **Transition Offset:** Sets how long the next state's animation should wait before playing. With normalized values, `0.1` means the next animation would start playing at 10 percent of its timeline. In this book, the offset value is always zero.

7. **Interruption Source:** Sets if and how the transition may be interrupted by other transitions.

8. **Conditions:** Here you set the deciding factors that determine when to transition to another node. In the image above, when **IsAlive** is set to false, the Animator transitions from the Alive state to the Death state.

Animation states also have properties that define how they handle animation:



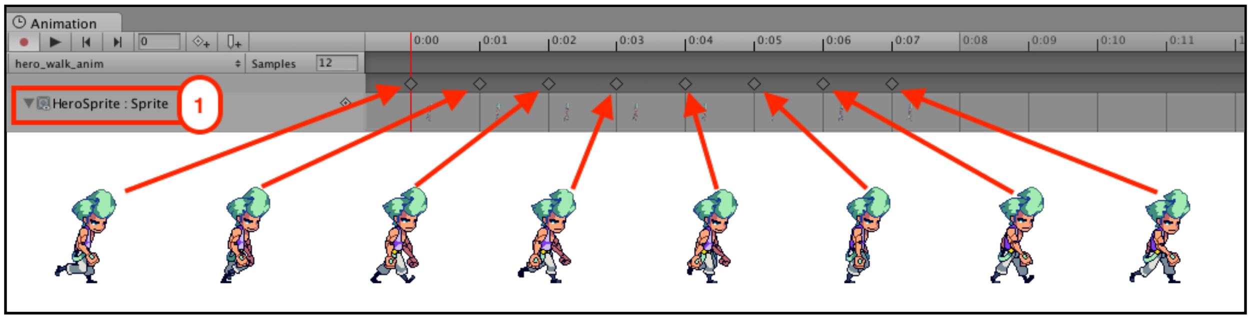
Motion: hero_idle_anim
Speed: 1
Multiplier [ ] Parameter
Normalized Time [ ] Parameter
Mirror [ ] [ ] Parameter
Cycle Offset: 0 [ ] Parameter
Foot IK [ ]
Write Defaults [✓]
List is Empty

1. **Motion:** The animation clip used by this state.

2. **Speed:** The speed multiplier for the animation; it's `1` by default.
3. **Cycle Offset:** The number of frames by which the loop animation should offset.
4. **Write Defaults:** All unanimated properties stay in their default states when this is enabled. For example, if you only animate the X position, the Y and Z position will stay `0`.
5. **Transitions:** A list of all transitions connected to this animation state.

Each animation state has an animation clip in its **Motion** parameter. Clips are the heart and soul of an animation: They contain data necessary to carry out an animation.

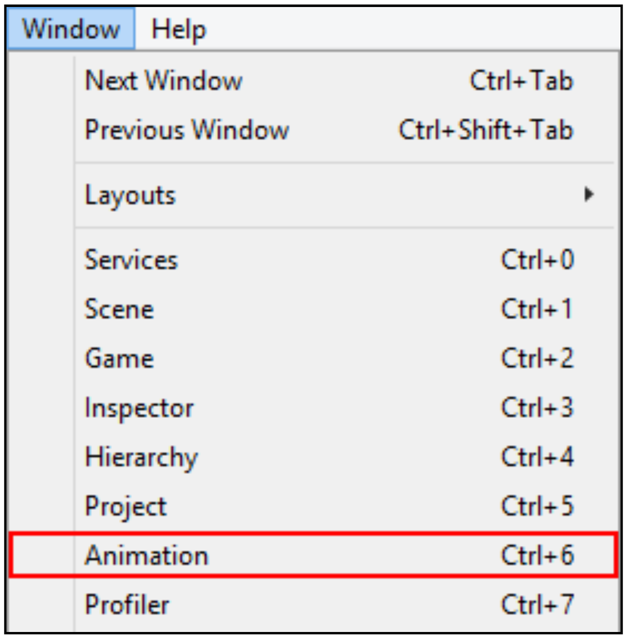
Animation clips are values or properties of components that change over a certain period of time. To visualize the concept, consider the image below, which breaks down **hero_walk_anim**:



You see the hero's individual "walk" sprites. Each arrow points to the corresponding frame on the timeline — these are **keyframes**, which are snapshots of the component's values. The **Sprite** property, noted as **1**, changes with each frame.

Playing the clip simply means it cycles through the keyframes and changes the HeroSprite's sprite value, creating an illusion of movement.

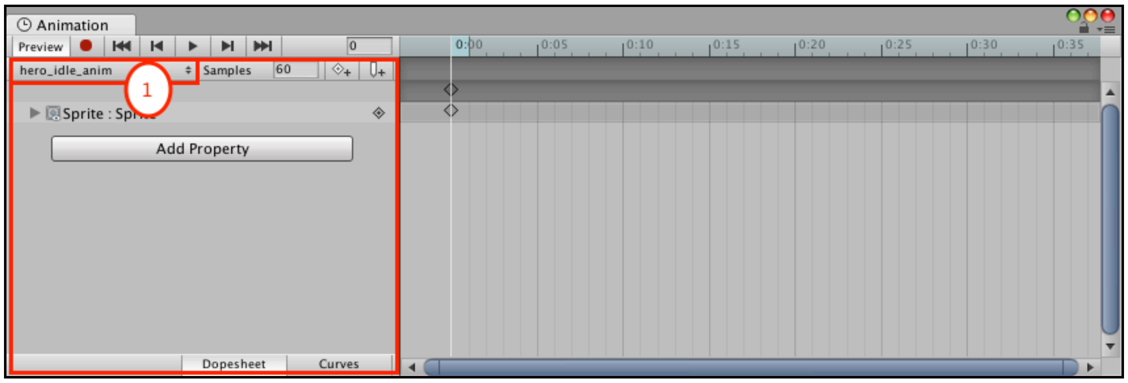
Animation clips can be viewed and edited in the Animation window, located under **Window\Animation** in the file menu. Drag the **Animation** window to Unity now, but remember, you won't see anything in there just yet — you're still just reading.



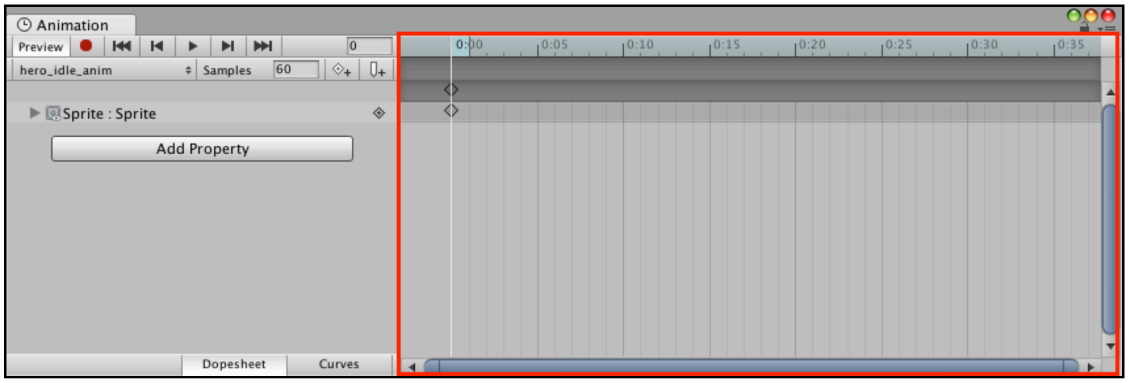
The Animation view has two parts: the **Animated Properties List** on the left and the **Animation timeline** on the right.

The **Animated Properties List** contains the properties and public variables that your animation modifies. If the property or member variable shows in the Inspector, then you can animate it!

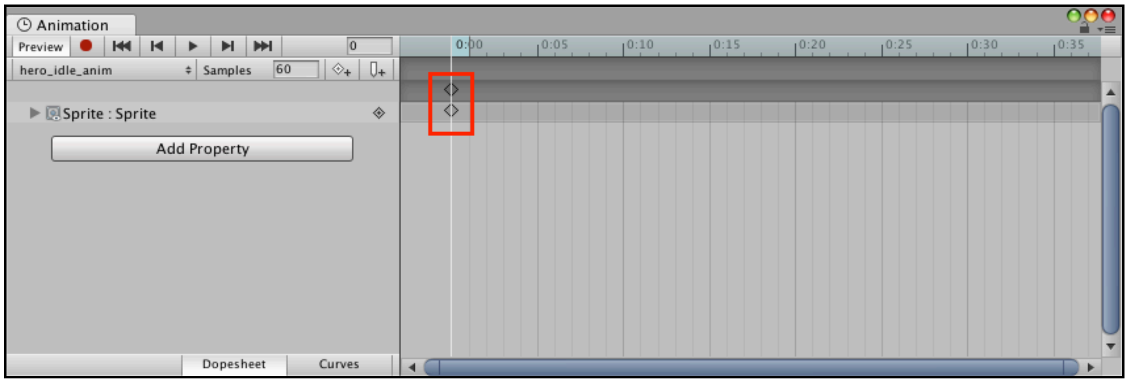
Use the top-left drop-down, highlighted as 1, to switch between animation clips attached to the same Animator.



The right is the **Animation timeline**. It comprises frames represented by vertical bars.

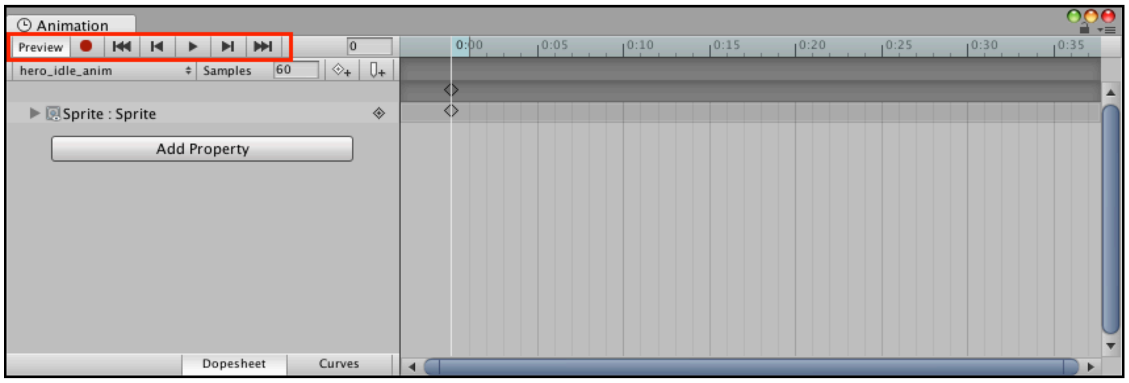


The **diamond icon** denotes keyframes in the timeline.



If Mecanim just considered the keyframes, then animations would look cheap and choppy. Hence, the system interpolates between keyframes to calculate values for each animation frame.

However, PompaDroid uses frame-by-frame animation for the characters, so interpolation is largely unnecessary.



The Animation view lets you record and playback an animation — controls are in the upper-left corner.

Recording is very simple: click the **record** button and change the properties that you want to animate. Changing any property of the selected Animator or its children will create a keyframe at the current value on the timeline. You'll cover this a bit later.

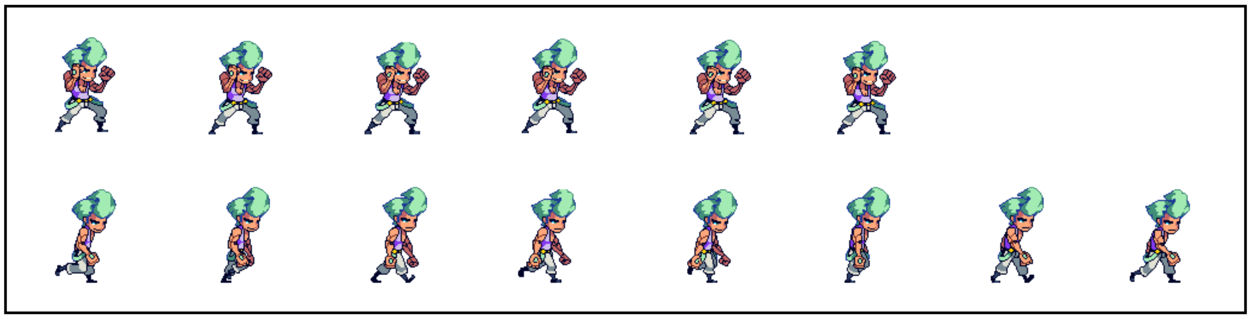
The Animation view has two view modes that show in the bottom-right of the properties pane: **Dope Sheet** and **Curves**.

- **Dope Sheet**: In this mode, you can view multiple properties for a specific animation and adjust keyframe values very easily.
- **Curves**: This provides a resizable graph where you can inspect values for animated properties. It lets you exert total control over animations. There is a cost though: too much control can make it hard to show all the data on screen smoothly, i.e., gameplay can suffer.

You'll use Dope Sheet mode for PompaDroid

Congratulations! You've just finished reading Mecanim in a nutshell, and now you're equipped with the knowledge to make the hero walk and idle in style! Time to get to work.

Creating pompadoured animations

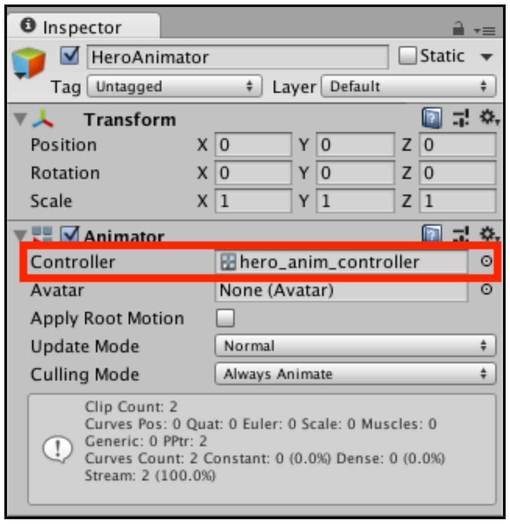


The hero comes with six idle sprites and eight walking sprites. Notice that perfect pompadour is never out of place. These sprites will play sequentially and loop to create the effect of movement.

I bet you're already thinking about how to set them up, so let's get to it!

Select **HeroAnimator**, which is a child of **MyHero**. Click **Add Component** then **Miscellaneous/Animator**.

Drag **hero_anim_controller** from the **Assets/Animation/Hero** folder to the **Controller** property of the **Animator** component.



HeroAnimator ☐ Static
Tag: Untagged Layer: Default

	X	Y	Z
Position	0	0	0
Rotation	0	0	0
Scale	1	1	1

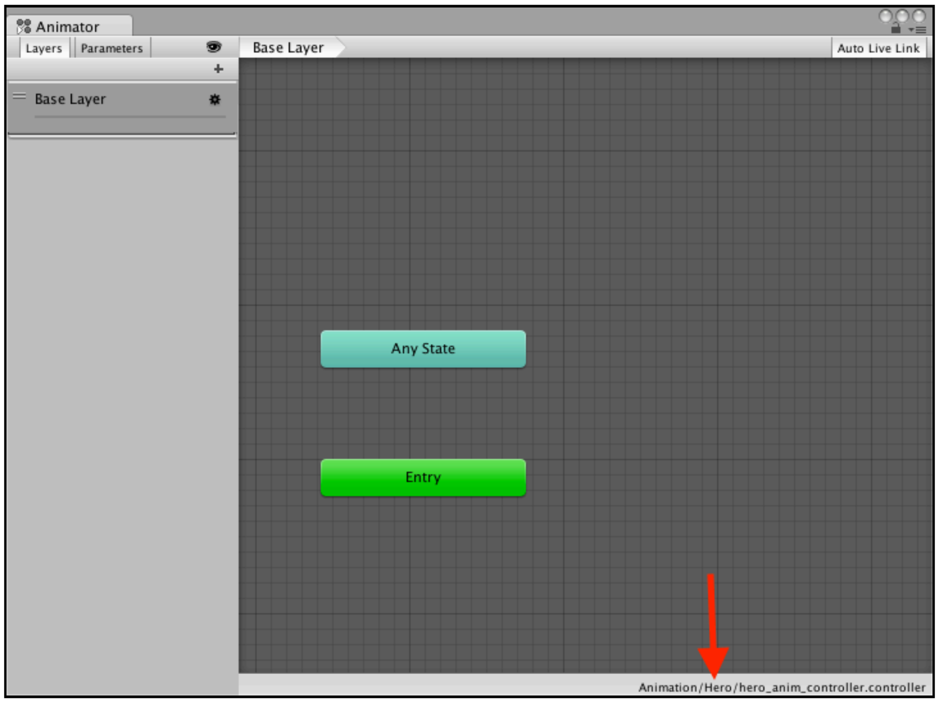
Controller: hero_anim_controller
Avatar: None (Avatar)
Apply Root Motion: ☐
Update Mode: Normal
Culling Mode: Always Animate

```
Clip Count: 2
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0
Generic: 0 PPtr: 2
Curves Count: 2 Constant: 0 (0.0%) Dense: 0 (0.0%)
Stream: 2 (100.0%)
```

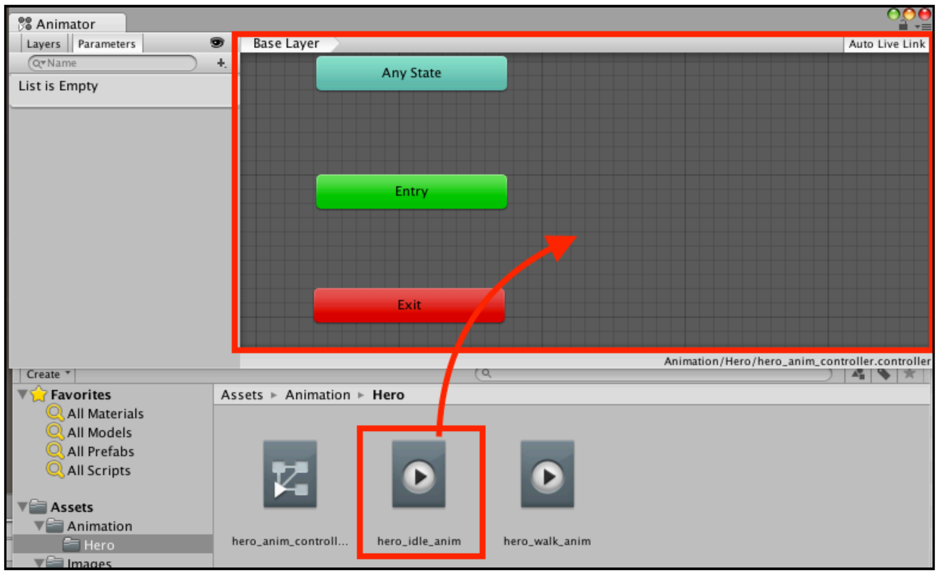
With that, you've set up the Animator. Now you'll add the animation itself.

Open the **Animator** window, navigate to **Assets/Animation/Hero** and select **hero_anim_controller**.

The Animator window should reflect the animation state machine for **hero_anim_controller**. Verify this by checking the name on the bottom-right of the panel.

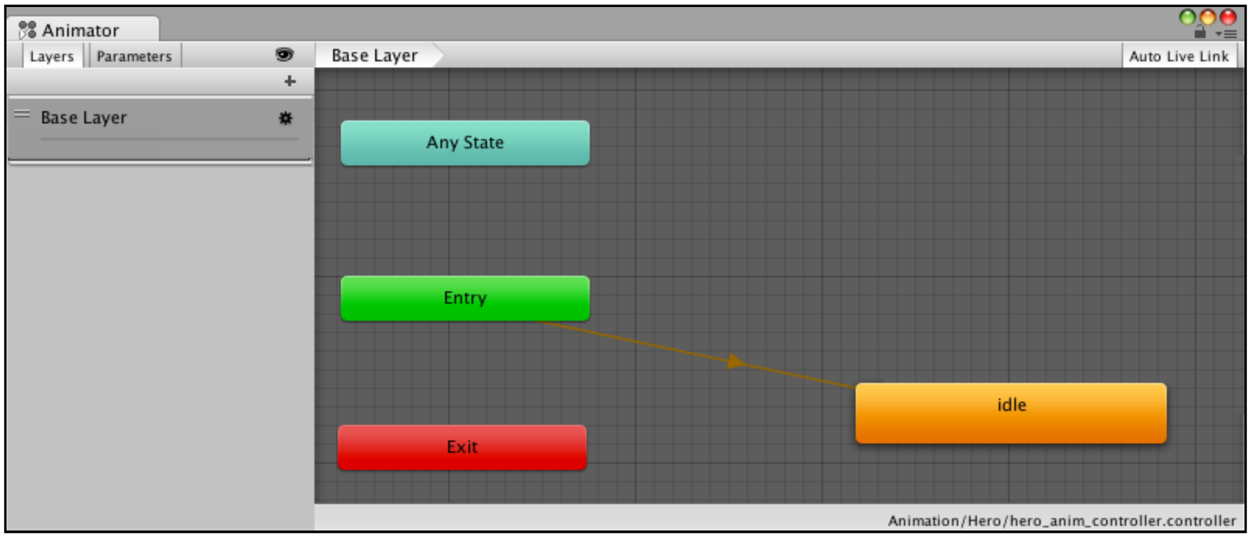


The state machine is empty. To add an Animation state, drag **hero_idle_anim** to the Animator window, like this:



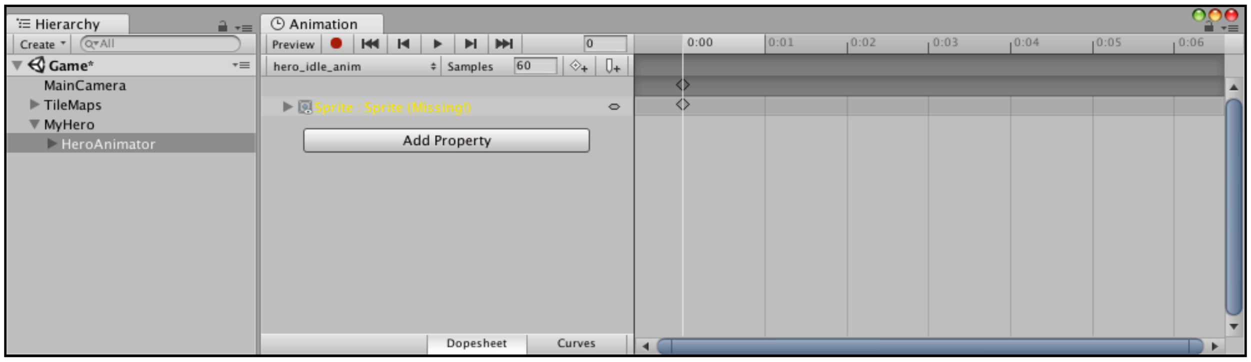
This creates a new Animation State named **hero_idle_anim** with a motion parameter that uses the **hero_idle_anim** clip. As the first Animation state for this controller, it's also the default, as indicated by the yellow color.

Select the **hero_idle_anim** state in the Animator view and rename it **idle** in the Inspector.



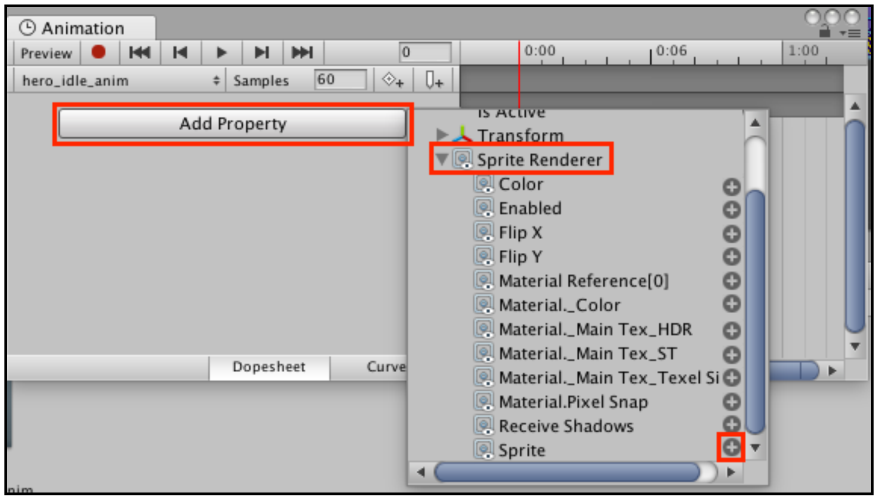
The **idle** state is now part of the hero animation state machine. Now you're ready to add an animation to **hero_idle_anim** — when idle, the pompadoured protagonist will bounce slightly to show that he's ready for action.

Select **HeroAnimator** in the Hierarchy. Open the Animation view by going to **Window\Animation** in the menu. Because HeroAnimator is an Animator component, you can edit the clips to be used by the Animator state machine in the Animation view.

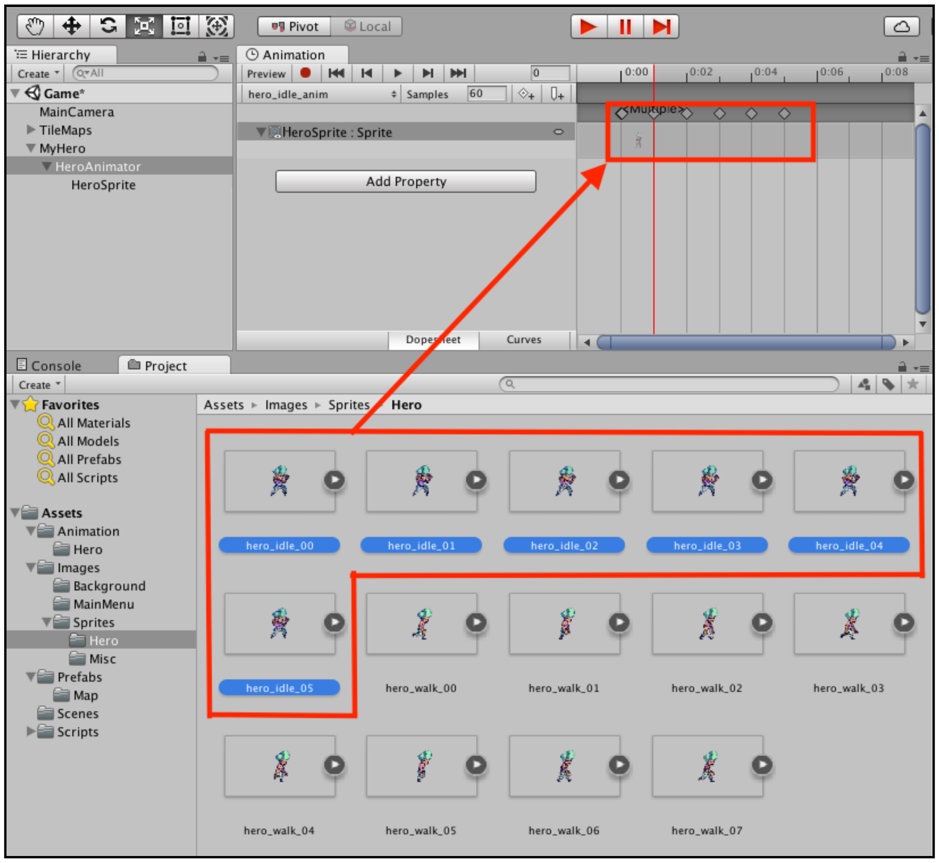


Select **hero_idle_anim** in the top-left drop-down of the Animation view.

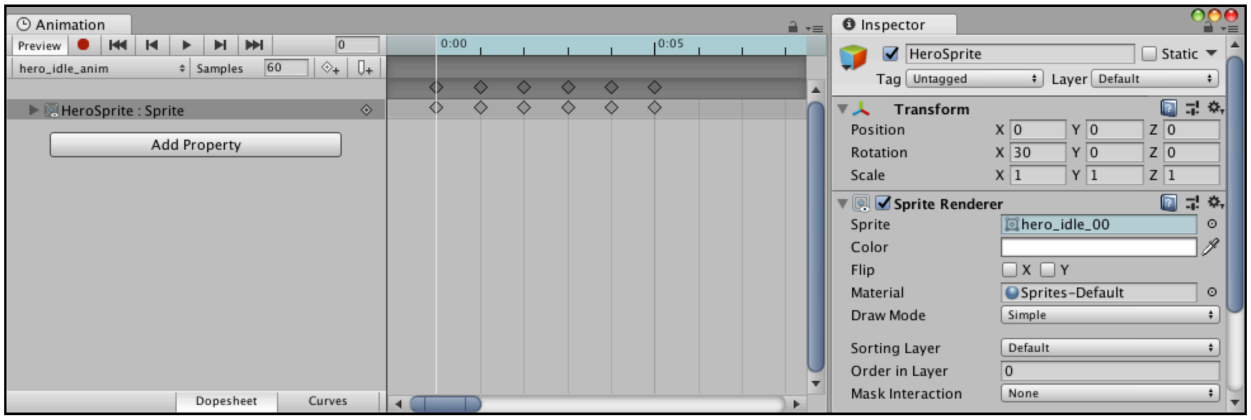
Click **Add Property** on the Animated Properties List. Select **HeroSprite\SpriteRenderer** and click **+** to the right of **Sprite**. You've just added the **HeroSprite: Sprite** property to the Animation timeline.



In the Project window, go into **Sprites\Hero** and select **hero_idle_00–hero_idle_05**. **Drag** them all to the first frame of the **HeroSprite: Sprite** parameter in the Animation timeline.

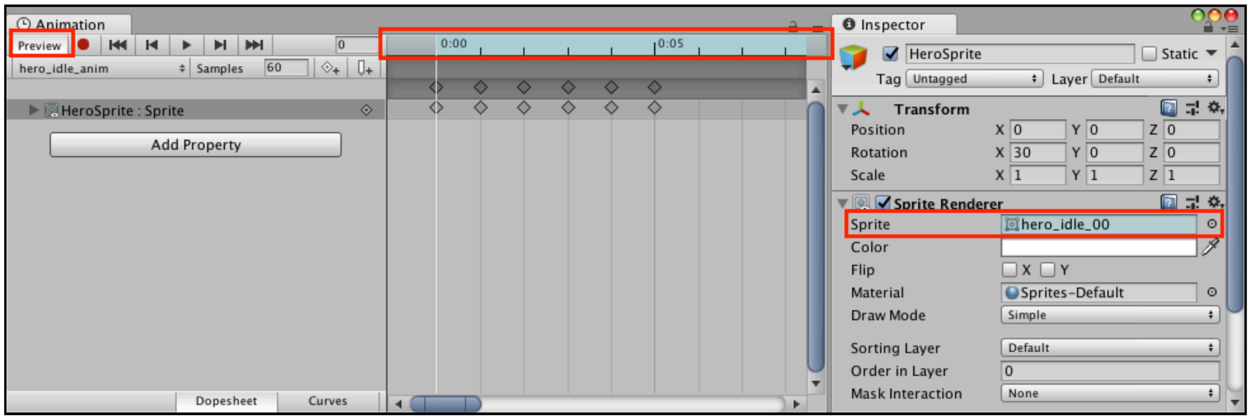


The timeline should show one keyframe per sprite, similarly to the image below:



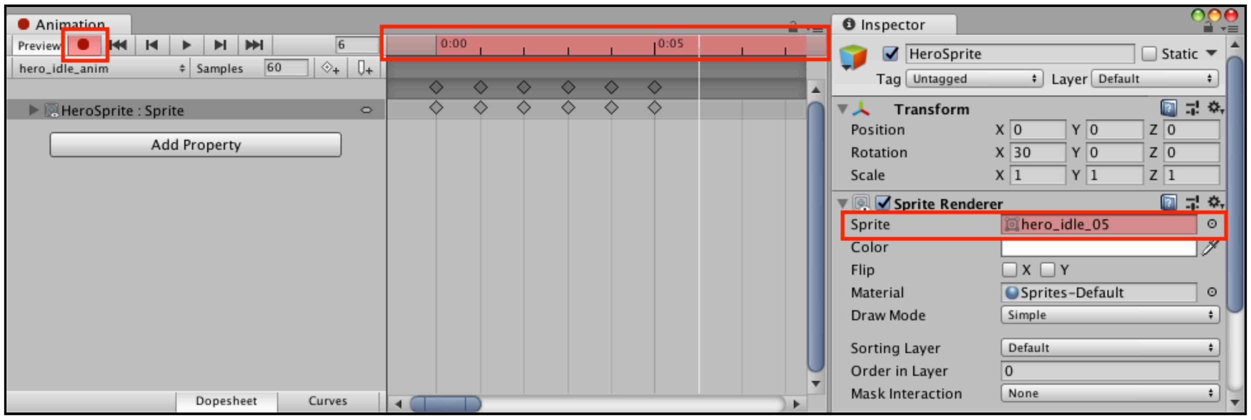
Scrub through the timeline and check if the sprite animation is correct. You should have six frames, with the first being **hero_idle_00** and the last being **hero_idle_05**. If there's an extra keyframe at the end of the timeline, right-click it and select **Delete Key**.

When you select any frame in the timeline, Unity will enter preview mode. This mode is easy to distinguish. Note how the Preview button in the Animation playback is selected and the timeline is blue.



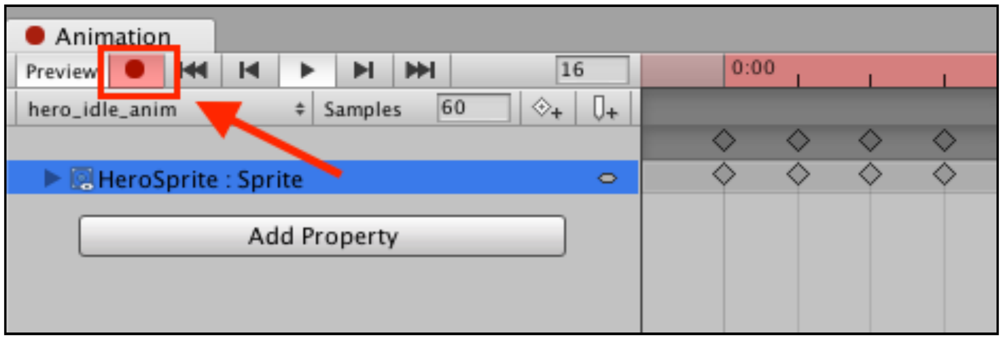
Changes to values that are being animated are **not** saved. Unity highlights these properties in blue. Of course, that means any values that are not highlighted **will** be saved — be careful!

If you want to modify the animation you are creating, click the Record button. This will change the Animation window so that it has a red timeline, and the record button shows as selected.

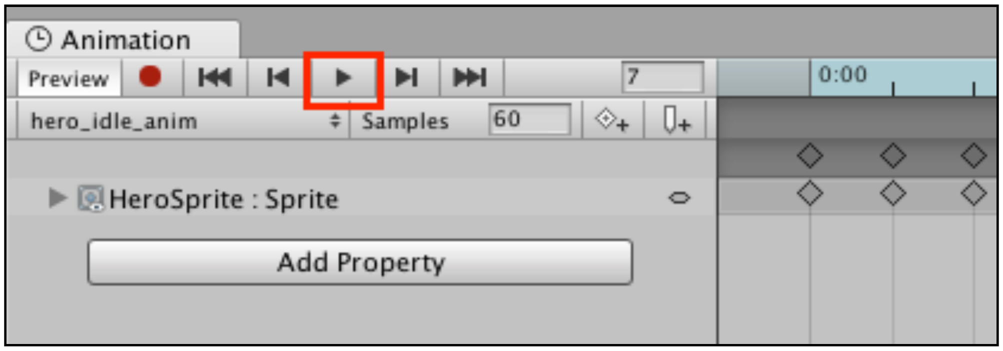


Unity records and keyframes any values that change.

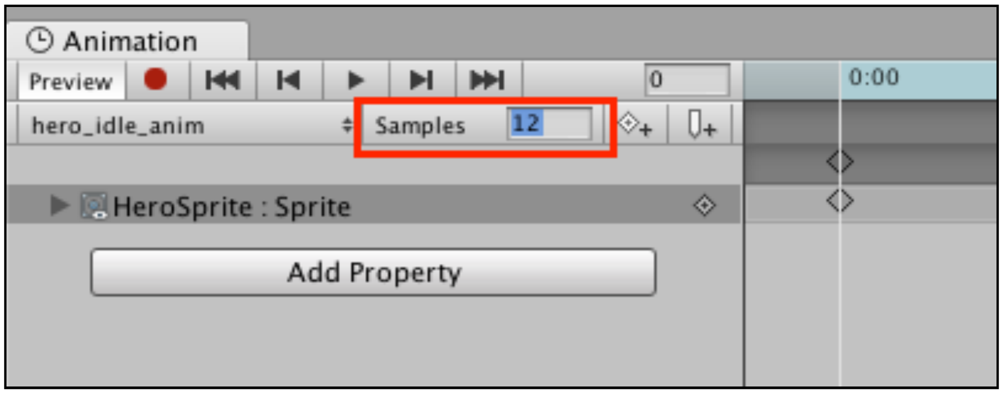
To disable record mode, just click the Record button in the playback tools.



By the way, you just finished your first animation! Press **Play** in the Animation view to see it. It'll play the entire sprite animation in the Scene view. Watch the red scrubber move across the frames to see which one is currently showing.



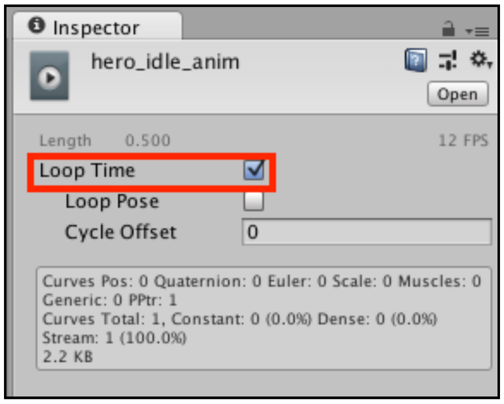
The hero is probably moving a bit fast for the effect to look right. Adjust this by setting **Samples** value to 12, which will adjust the selected animation's frames per second (FPS). The hero's idle sprites were originally designed for 12 FPS, but you can play with this value if you want.



Make sure you **stop record mode** before proceeding.

Press the **Play button** in the toolbar. The hero bobs left and right but stops abruptly. That's not going to cut it during gameplay.

Select **hero_idle_anim** animation in the Project window and check **Loop Time** in the Inspector.



Length 0.500 12 FPS
Loop Time ✓
Loop Pose
Cycle Offset 0

Curves Pos: 0 Quaternion: 0 Euler: 0 Scale: 0 Muscles: 0
Generic: 0 PPtr: 1
Curves Total: 1, Constant: 0 (0.0%) Dense: 0 (0.0%)
Stream: 1 (100.0%)
2.2 KB

Play the game again. Congratulations! He's no longer some dull, static image.

Adding input

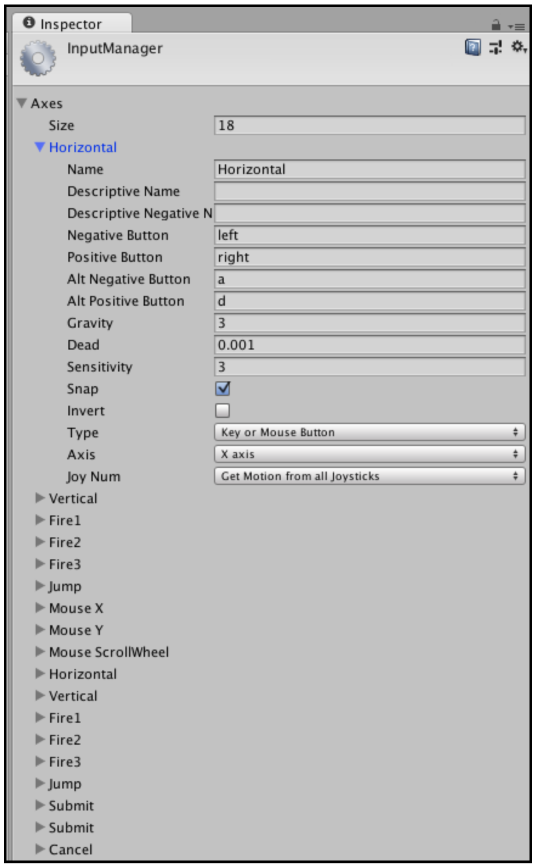
There are more animations to add, but you'll get to those later. Right now, it's time to give the player control of the character.

PompaDroid requires the following inputs:

- A button to **punch**
- A button to **jump**
- A joystick for **movement**

Unity has an `Input` class that interprets input from various devices, including keyboard and touch. Its values are configured in the **Input Manager**.

Go to **Edit\Project Settings\Input** to open the Input Manager in the Inspector.



Inspector

InputManager

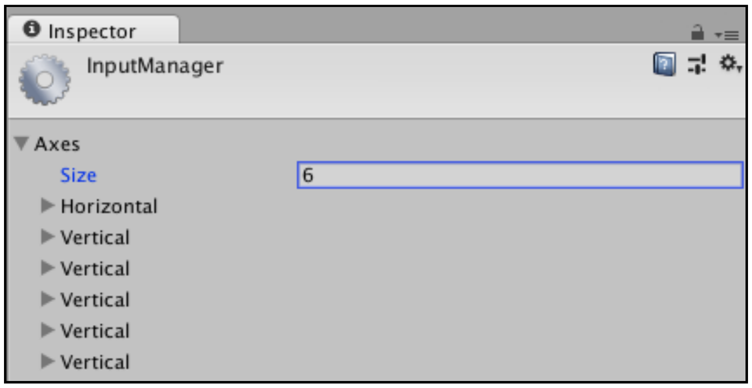
Axes

Size	18
Horizontal	
Name	Horizontal
Descriptive Name	
Descriptive Negative N	
Negative Button	left
Positive Button	right
Alt Negative Button	a
Alt Positive Button	d
Gravity	3
Dead	0.001
Sensitivity	3
Snap	✓
Invert	
Type	Key or Mouse Button
Axis	X axis
Joy Num	Get Motion from all Joysticks

Vertical
Fire1
Fire2
Fire3
Jump
Mouse X
Mouse Y
Mouse ScrollWheel
Horizontal
Vertical
Fire1
Fire2
Fire3
Jump
Submit
Submit
Cancel

Expand **Axes** to see all the inputs currently handled by the engine. Set **Size** to 2 to delete all the other unneeded axes — you'll only need **Horizontal** and **Vertical**. Expand **Horizontal** to show all its values, then clear the values for **Alt Negative Button** and **Alt Positive Button**. Do the same thing for the **Vertical** axis.

Now, add four new elements by setting the **Size** to 6 and pressing **Enter**. This will duplicate the vertical input four times:



Inspector

InputManager

Axes

Size 6

Horizontal
Vertical
Vertical
Vertical
Vertical
Vertical

- Name the first duplicate **Attack**. Clear all values with **Button** in their name. Set **Positive Button** to **a**.

- Name the second duplicate **Jump**. Clear all values with **Button** in their name. Set **Positive Button** to **s**.
- Name the third duplicate **Submit**. Clear all values with **Button** in their name. Set **Positive Button** to **a**.
- Name the fourth duplicate **Cancel**. Clear all values with **Button** in their name. Set **Positive Button** to **s**.

Submit and **Cancel** are necessary for the user interface (**UI**). If you don't declare these, then Unity will throw a bunch of warnings.

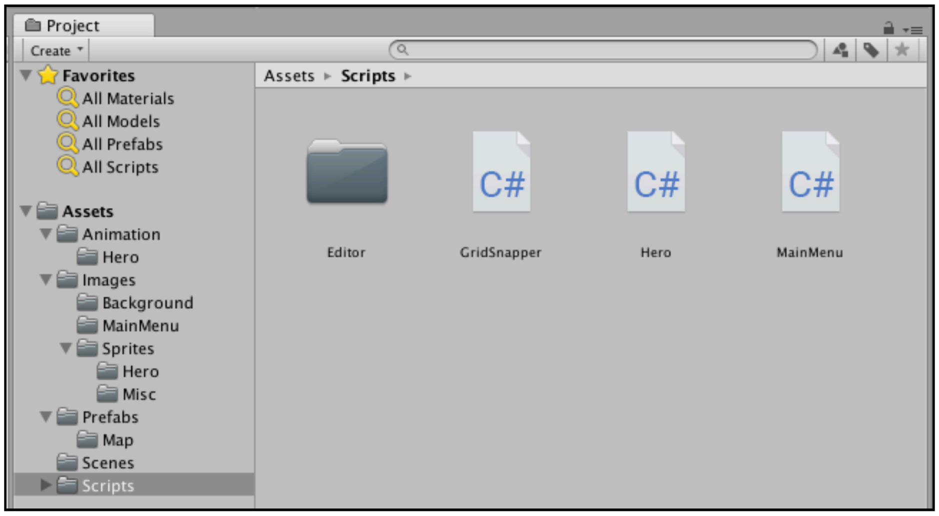
And with that, you've built out the input controls necessary to make PompaDroid work. On to getting the guy moving.

Moving the hero around, part 1

To implement movement, the hero needs three basic components:

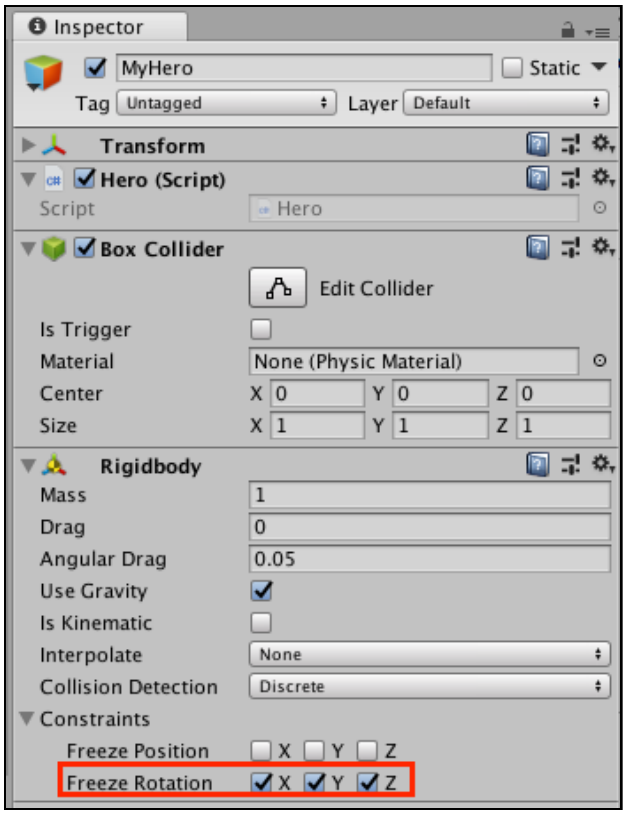
1. **Hero** script: A custom script to handle the hero, including movement.
2. **Box Collider**: A secondary GameObject that approximates the hero's body in the physics world. It detects collisions with other entities such as walls, floors, enemies and punches.
3. A **Rigidbody**: Allows the associated GameObject to be controlled by the physics engine. Working in conjunction with the attached box collider, they let the physics engine determine the result when forces are applied to it.

Navigate to the **Assets\Scripts** folder in the Project view and **right-click** in the empty space then select **Create\C# Script** and name it **Hero**.



Select **MyHero** in the Hierarchy and drag the **Hero** script to it. Click **Add Component**, select **Physics** then add a **Box Collider**. Repeat to add a **Rigidbody** and under its **Constraints**, find **Freeze Rotation** and check **X, Y, and Z** to disable rotation on the hero's Transform.

MyHero should look like this now:



Inspector

✓ MyHero ☐ Static

Tag Untagged Layer Default

Transform

✓ Hero (Script)

Script Hero

✓ Box Collider

Edit Collider

Is Trigger ☐

Material None (Physic Material)

Center X 0 Y 0 Z 0

Size X 1 Y 1 Z 1

Rigidbody

Mass 1

Drag 0

Angular Drag 0.05

Use Gravity ✓

Is Kinematic ☐

Interpolate None

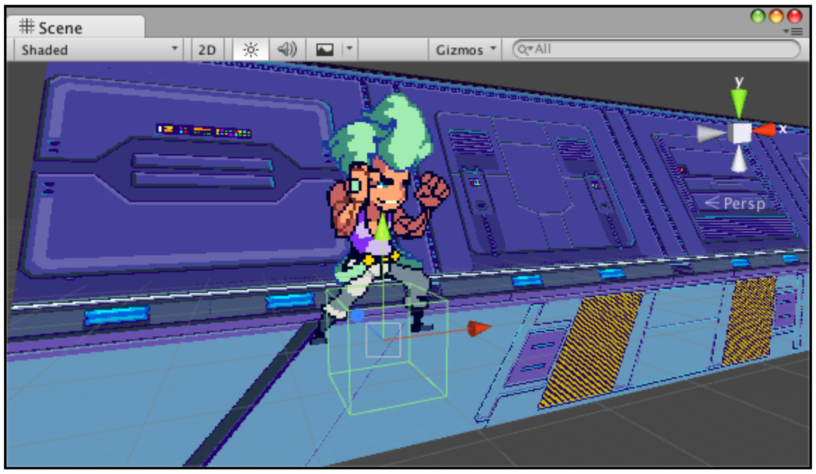
Collision Detection Discrete

Constraints

Freeze Position ☐ X ☐ Y ☐ Z

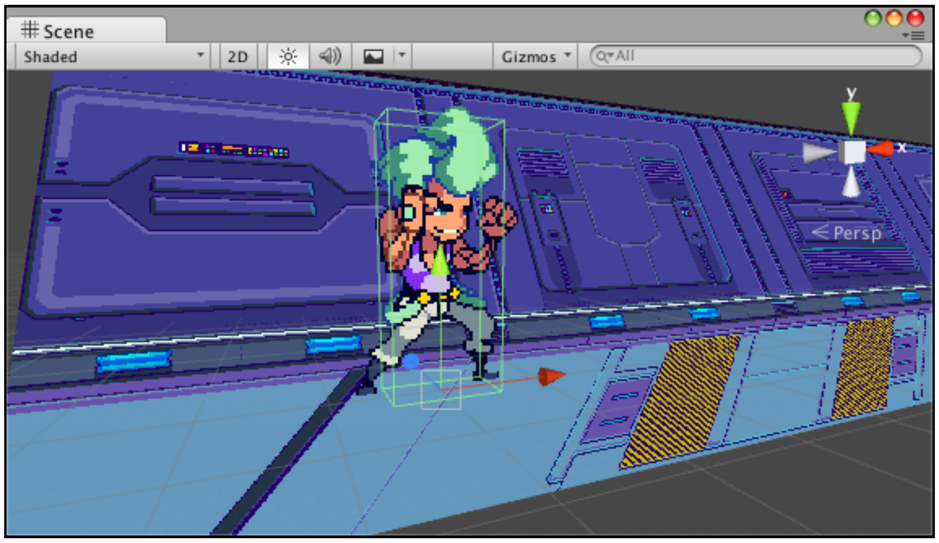
Freeze Rotation ✓ X ✓ Y ✓ Z

Look at the Scene view. You'll see the green box collider, but it's at the feet of the hero and does a poor job of approximating his body:



Note: If you're seeing a 2D square instead of a 3D cube, turn on 3D mode by clicking on 2D in the upper left corner of the Scene view.

Correct this by setting the **Center** value of MyHero’s **Box Collider** to `(X:0, Y:1.2, Z:0)` and set the **Size** to `(X:1, Y:2.4, Z:0.5)`. Now he should be mostly inside the green cube.



Press the **Play button** and test the game. Whoops? He’s dropping down through the floor. What a horrible fate!

It’s happening because there is no “solid” floor beneath his feet, so he’ll be pulled downwards by gravity for eternity — or at least until you press play again.

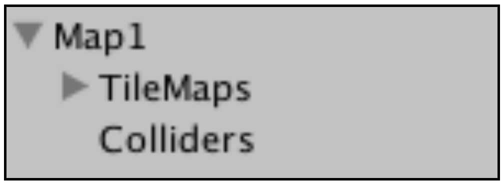
Tunnel vision

Although the power to walk through walls and floors can be a useful trait for a superhero, it does a pompadoured protagonist no good. He needs a floor to walk on and walls to keep him in play. You’ll implement more **Box Colliders** to ultimately solve the problems, but first, a little background reconstruction is in order.

In the Hierarchy, **right-click** and **Create Empty**. **Reset** its Transform and name it **Map1**. It’ll serve as the root of all the level objects.

Make **TileMaps** a child of **Map1** and set its **Position** to `(X:0.5, Y:-0.28, Z:4.67)`, **Rotation** to `(X:30, Y:0, Z:0)` and **Scale** to `(X:1, Y:1, Z:1)`. If, for some reason, the hero isn’t standing on the floor, play around with the **Z position** of **TileMaps** until you find the right value.

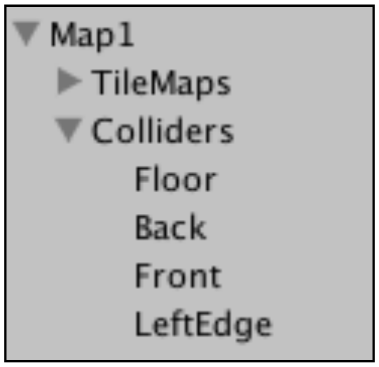
Create a new empty GameObject and name it **Colliders**. Make it a child of **Map1** and **reset** its Transform. This will serve as the parent of all the colliders in this level. You should have the same hierarchy as this:


Map1
TileMaps
Colliders

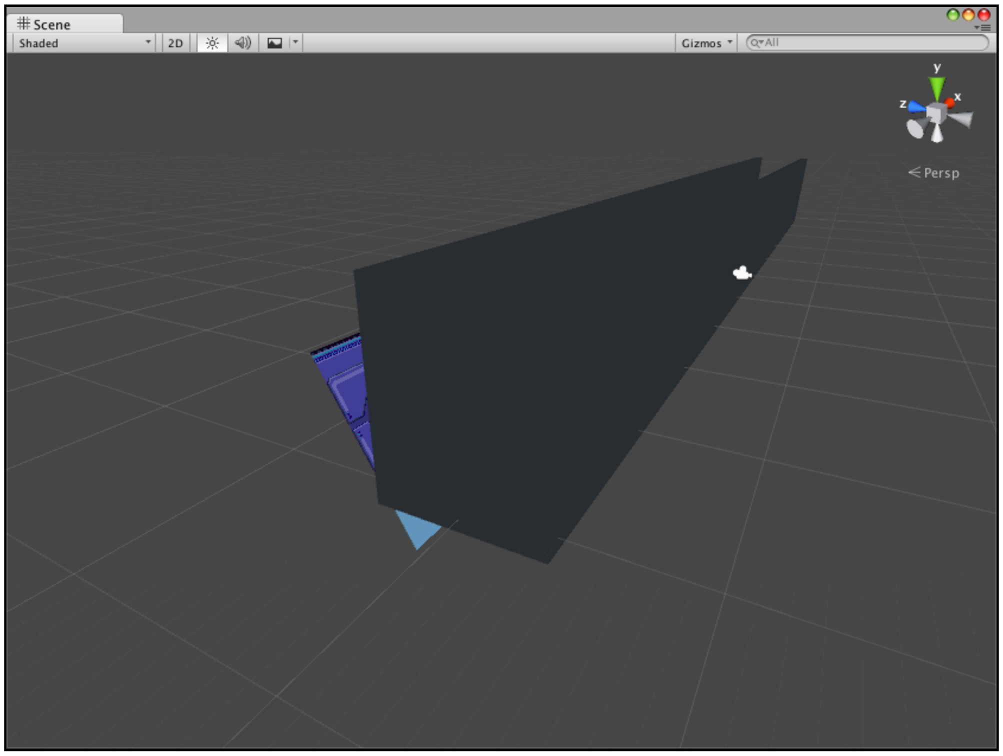
Now create the box colliders for the floor and walls:

- **Right-click** and select **3D Object\Cube** to create a cube and name it **Floor**. Make it a child of **Colliders**. Set its **Position** to `(X:50, Y:-0.5, Z:0)`, **Rotation** to `(X:0, Y:0, Z:0)` and **Scale** to `(X:100, Y:1.0, Z:10)`.
- Create another cube and name it **Back**. Make it a child of **Colliders**. Set its **Position** to `(X:50, Y:4.5, Z:3.5)`, **Rotation** to `(X:0, Y:0, Z:0)` and **Scale** to `(X:100, Y:10, Z:1)`. This cube will be the back wall of the map — the side furthest from the camera.
- Create a cube and name it **Front**. Make it a child of **Colliders**. Set its **Position** to `(X: 50, Y:4.5, Z:-3.5)`, **Rotation** to `(X:0, Y:0, Z:0)`and **Scale** to `(X:100, Y:10, Z:1)`. This defines the front wall closer to the camera.
- Create the last cube and name it **LeftEdge**. Make it a child of **Colliders**. Set its **Position** to `(X:-0.5, Y:4.5, Z:0)`, **Rotation** to `(X:0, Y:90, Z:0)` and **Scale** to `(X: 8, Y:10, Z:1)`.

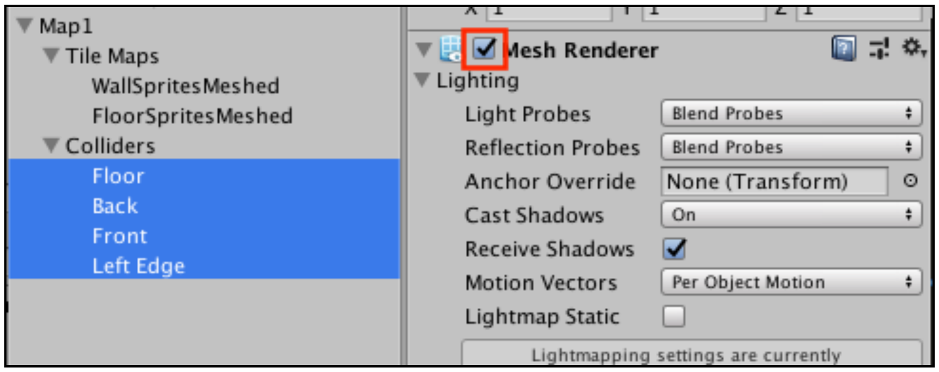
Your **Map1** hierarchy should now look this:


Map1
TileMaps
Colliders
Floor
Back
Front
LeftEdge

Four boxes now surround your hero, hiding him from the camera.



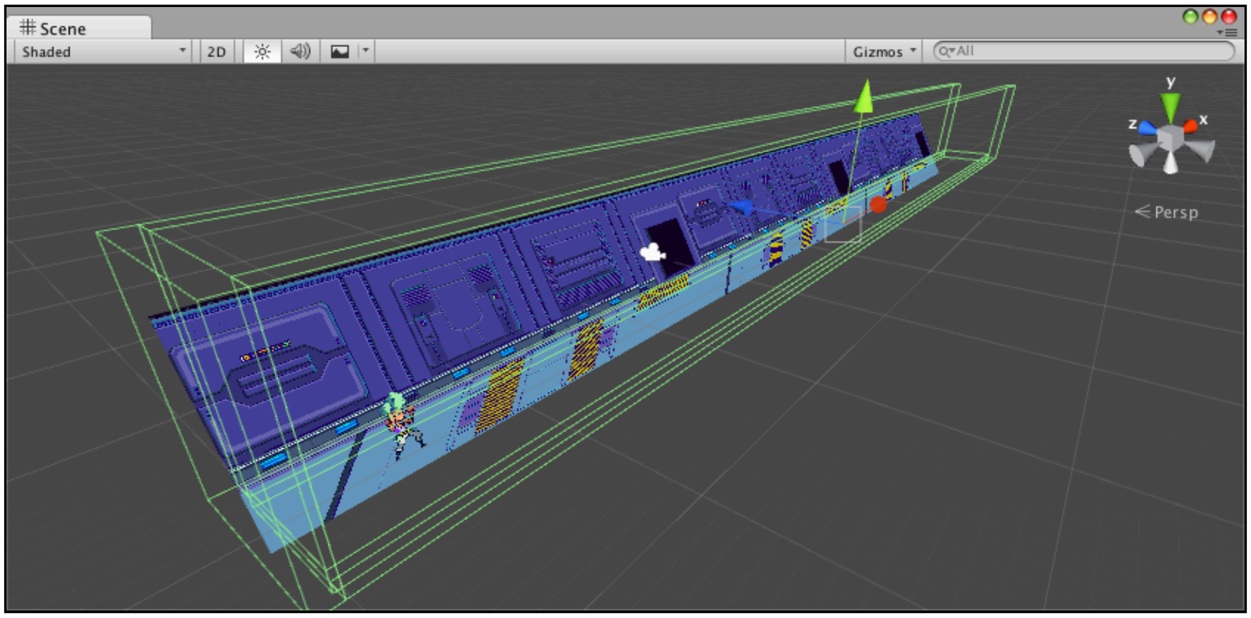
Disable all the **Mesh Renderers** of the newly created cubes to be able to see him again. Select the **Floor, Back, Front** and **LeftEdge** and **uncheck** the checkbox on their **Mesh Renderer** components.



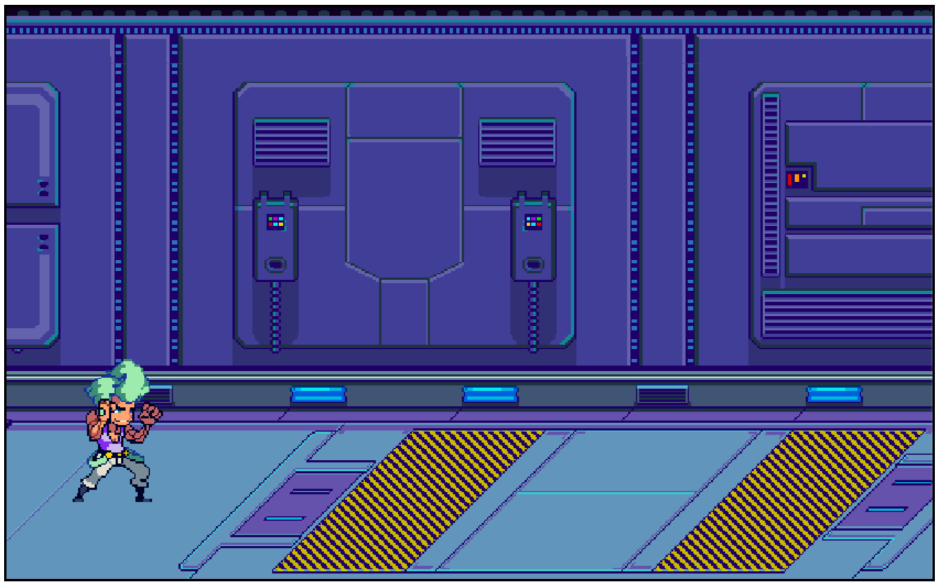
Map1
Tile Maps
WallSpritesMeshed
FloorSpritesMeshed
Colliders
Floor
Back
Front
Left Edge

Mesh Renderer
Lighting
Light Probes: Blend Probes
Reflection Probes: Blend Probes
Anchor Override: None (Transform)
Cast Shadows: On
Receive Shadows: ✓
Motion Vectors: Per Object Motion
Lightmap Static: ☐
Lightmapping settings are currently

Select **Colliders** and look at the Scene view. The result should be something like this:



The **Box Colliders** show as green outlines. Now that you've set the floor, press **Play** and see what happens. Now he's on — not in — the ground because there's a collider beneath him:



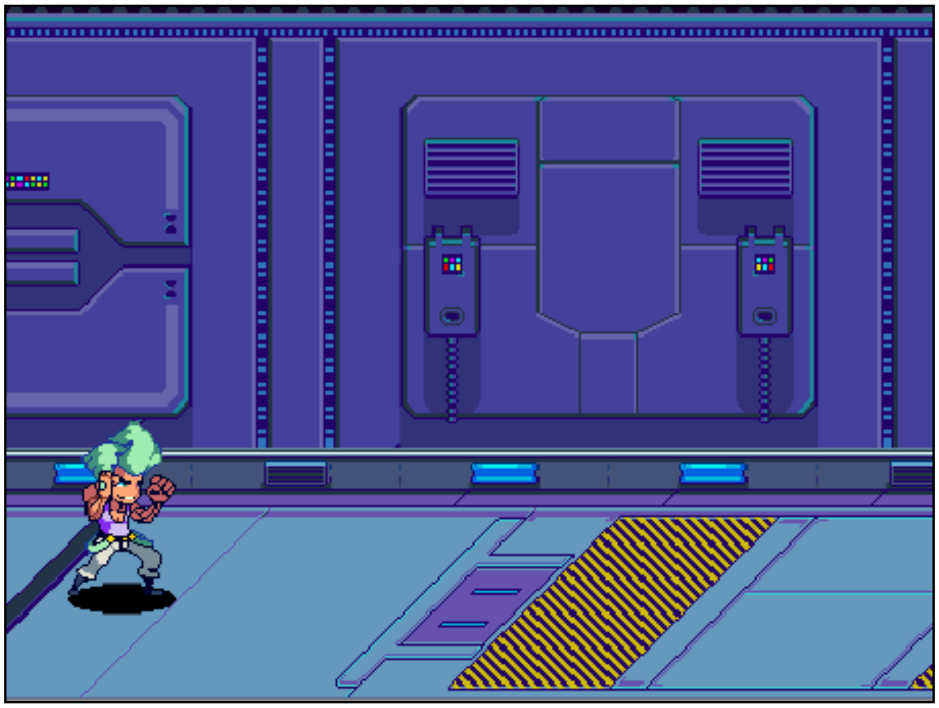
Yet something still seems a little off! The character appears to be floating ever so slightly over the floor instead of standing on it. A shadow beneath his feet will make him look like he's part of the world.

The shadow can be a black circle sprite that needs to be drawn below the hero:

No fancy effects needed; a plain and simple shadow is sufficient.

Go to the Hierarchy and **right-click** in empty space then select **Create\2D Object\Sprite** to make a new sprite game object. Make it a child of **MyHero** and rename it **ShadowCharacter**.

Set its **Sprite** to the **shadow_character** sprite located in **Assets\Images\Sprites\Misc** in the Project view. Set its Transform **Position** to `(X:0, Y:0, Z:0.1)`, **Rotation** to `(X: 30, Y:0, Z:0)` and **Scale** to `(X:1, Y:1, Z:1)`.



Much better! The hero is now firmly planted on the ground — amazing what a little shadow will do for the scene.

You're making great progress. You just finished up adding physics, which is a prerequisite for making the player move.

Moving the hero around, part 2

In this section, you'll edit the **Hero** script to make him walk like he's never walked before!

Open the **Hero** script and remove the `Start()` and `Update()` methods that are added by default. Add the following variables to the `Hero` class:

```
//1
public Animator baseAnim;
public Rigidbody body;
public SpriteRenderer shadowSprite;

//2
public float speed = 2;
public float walkSpeed = 2;

//3
Vector3 currentDir;
bool isFacingLeft;
protected Vector3 frontVector;
```

1. The `baseAnim` variable references the Animator you created for the hero's animation. The `body` refers to the Rigidbody that will handle the physics of the **Hero**, and `shadowSprite` references the shadow beneath the hero's feet.

2. `walkSpeed` defines how fast the hero moves around the map and `speed` is the current speed; `walkSpeed` is assigned to the `speed` value when the hero is walking.

3. `currentDir` contains the actual direction the hero will be moving. `isFacingLeft` is true whenever the hero faces left and false when he faces right. Lastly, the `frontVector` contains the direction the hero is facing. Its value should be `(X:-1, Y: 0, Z:0)` when the hero faces left and `(X:1, Y:0, Z:0)` when he's facing right.

Now for the methods of the `Hero` script. These are the blocks of code that make the hero actually do stuff. First, add the following method below the variable declarations.

```
void Update() {
  //1
  float h = Input.GetAxisRaw ("Horizontal");
  float v = Input.GetAxisRaw ("Vertical");


  //2
  currentDir = new Vector3(h, 0, v);
  currentDir.Normalize();

  //3
  if ((v == 0 && h == 0)) {
    Stop();
  } else if ((v != 0 || h != 0)) {
    Walk();
  }
}
```

The `Update()` method is native to Unity's MonoBehaviour lifecycle. It is called once every frame. Here, you do the following:

1. You check for user input by storing the value of the `horizontal` input in the `h` variable, and the `vertical` input in the `v` variable.
2. You store the `h` and `v` values to the `currentDir` Vector3. Horizontal input is translated to the hero's x-axis movement while vertical input is translated to movement along the z-axis.
3. Finally, you check for horizontal or vertical input. If no input is found, then you call `Stop()` — a method you will write later to make the hero stop. When either of the two inputs is found, then you call `Walk()` — a method you will also write later to get the hero moving.

You're not done yet. At this point, your code won't compile because you haven't written the `Stop()` and `Walk()` methods yet. Still in the `Hero` script, add the following methods right below `Update()`.

```
//1
public void Stop() {
  speed = 0;
  baseAnim.SetFloat("Speed", speed);
}

//2
public void Walk() {
  speed = walkSpeed;
  baseAnim.SetFloat("Speed", speed);
}
```

These two methods allow you to directly control the speed of the hero. Remember that both these methods are called in `Update()` and are mutually exclusive depending on the presence of any user input related to movement.

1. In `Stop()`, you set the hero's `speed` to `0` to stop him from moving. You also set his Animator's `Speed` parameter to `0` to ensure that the hero returns to the idle animation.
2. You do the opposite in `Walk()`. First, you copy the value of `walkSpeed` to `speed` to make the hero move. Likewise, you also set its Animator's `Speed` parameter to the same value as `speed` to make his walk animation play later.

The hero won't be able to move just yet. All you've really done at this point is to change the hero's speed. To move the hero, you need to get his **Rigidbody** to move. Still in the `Hero` script, add the following methods.

```
//1
void FixedUpdate() {
  Vector3 moveVector = currentDir * speed;
  body.MovePosition (transform.position + moveVector *
Time.fixedDeltaTime);
  baseAnim.SetFloat ("Speed", moveVector.magnitude);

  //2
  if (moveVector != Vector3.zero) {
    if (moveVector.x != 0) {
      isFacingLeft = moveVector.x < 0;
    }
    FlipSprite (isFacingLeft);
  }
}

//3
public void FlipSprite(bool isFacingLeft) {
  if (isFacingLeft) {
    frontVector = new Vector3(-1, 0, 0);
    transform.localScale = new Vector3(-1, 1, 1);
  } else {
    frontVector = new Vector3(1, 0, 0);
    transform.localScale = new Vector3(1, 1, 1);
  }
}
```

`FixedUpdate()` is another native Unity method similar to `Update()`. It is called once every fixed framerate frame at precise time intervals, while `Update()` is called once every actual framefrate frame at varying time intervals. With that said, let's step through this last code block one section at a time:

1. In `FixedUpdate()`, you calculate how much the hero's **Rigidbody** should move by multiplying his `currentDir` with his `speed`. You then call `body.MovePosition()` to position the hero's **Rigidbody** according to the `moveFactor`. The hero's Animator's `Speed` parameter is also updated.

2. Still in `FixedUpdate()`, you check whether the hero is facing right or left when he is moving — or in code lingo, when `moveVector` is not zero. You flip the hero's GameObject accordingly using the `FlipSprite()` method.

3. This method simply flips **MyHero**. If the hero is moving to the left, `FlipSprite` flips the object horizontally by setting its Transform **Scale** to `(X:-1, Y:1, Z:1)`, otherwise it makes the hero face right by scaling it to `(X:1, Y:1, Z:1)`. The `frontVector` variable is also updated in this method, which will come in handy in the future.

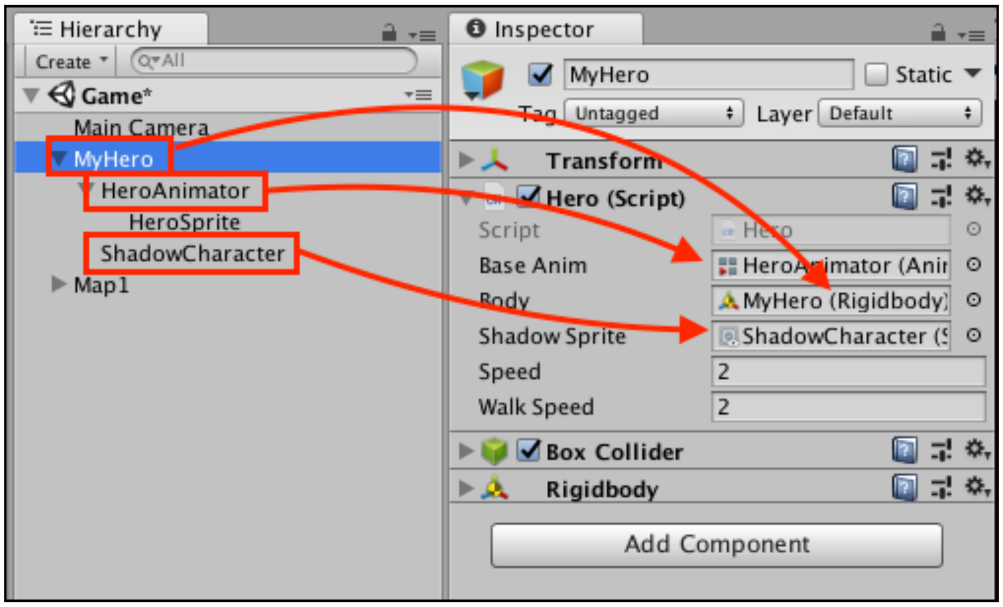
Note: To do physics calculations as accurately as possible, Unity needs a fixed time interval between frames, so any physics-related code — especially those related to **Rigidbody** — should be in `FixedUpdate()` instead of `Update()`. A higher or lower FPS would result to shorter or longer time intervals between calls to `Update()` and this can cause unstable physics calculations.

Whew, that's a lot of code! You still need to link the variables in the Inspector for it to work though.

Save the **Hero** script and return to Unity. In the Hierarchy, select **MyHero**. Drag **HeroAnimator** to the **Base Anim** parameter of the **Hero** component to assign the animator.

Drag **MyHero** to the **Body** parameter of the **Hero** component to make sure the script has access to the hero's **Rigidbody** for movement.

Finally, drag **ShadowCharacter** to the **Shadow Sprite** parameter of the **Hero** component to link the shadow.

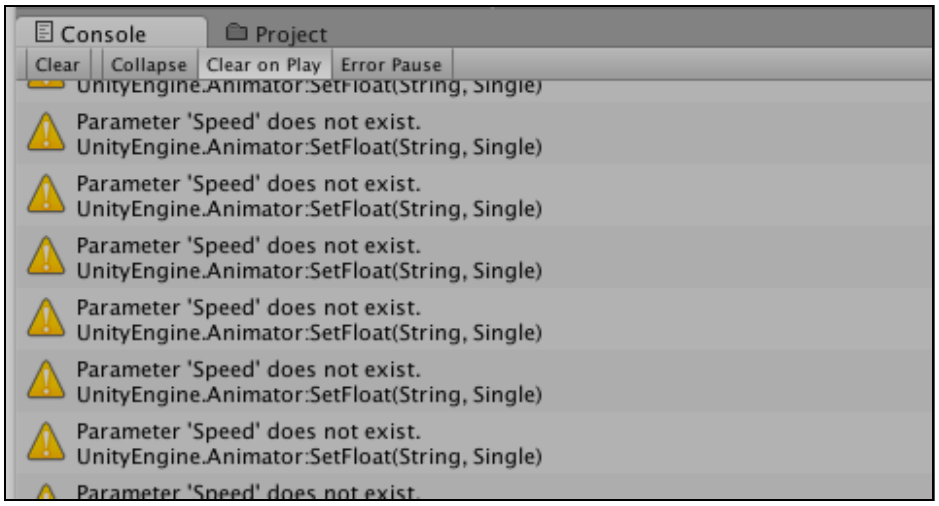


MyHero
Tag Untagged
Layer Default
Transform
Hero (Script)
Script: Hero
Base Anim: HeroAnimator (Anir
Body: MyHero (Rigidbody)
Shadow Sprite: ShadowCharacter (S
Speed: 2
Walk Speed: 2
Box Collider
Rigidbody

Press the **Play** button and watch what happens.

Pressing the **Arrow keys** on your keyboard moves the **Hero** around! But what in the heck is he doing? Is that some kind of new dance move — the pompadour pop, perhaps? Whatever it is, it sure isn't walking.

He's sliding around like that because HeroAnimator only plays **hero_idle_anim**. Take note of this message in the console:



```
UnityEngine.Animator:SetFloat(String, Single)
Parameter 'Speed' does not exist.
UnityEngine.Animator:SetFloat(String, Single)
Parameter 'Speed' does not exist.
UnityEngine.Animator:SetFloat(String, Single)
Parameter 'Speed' does not exist.
UnityEngine.Animator:SetFloat(String, Single)
Parameter 'Speed' does not exist.
UnityEngine.Animator:SetFloat(String, Single)
Parameter 'Speed' does not exist.
UnityEngine.Animator:SetFloat(String, Single)
Parameter 'Speed' does not exist.
UnityEngine.Animator:SetFloat(String, Single)
Parameter 'Speed' does not exist.
```

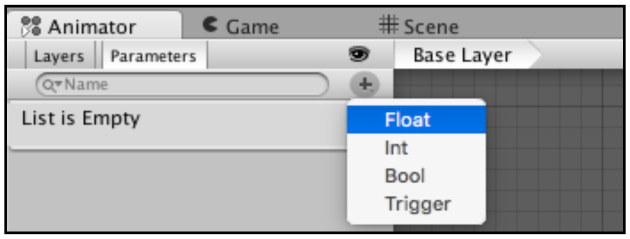
When the script detects input, the physics pushes the hero object around the map, exactly as you programmed it to do. It also notifies the **Animator** about the **Speed** of the movement but the **Animator** clearly doesn't know that the hero has other moves.

Note: You could have added the other animations earlier, along with the idle animation, but then you'd have missed out on this unforgettable learning experience!

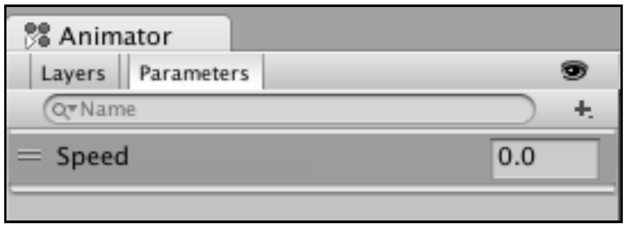
Creating a state machine

Now you can see where the walking animation comes into play. You'll spend the rest of this section doing prework for implementing the walk animation: creating a state machine.

Select **HeroAnimator** and open the **Animator** view. Now select the **Parameters** tab in the left pane of the **Animator** window. Click the **+** button at the top-right, select **Float** and name the parameter **Speed**.

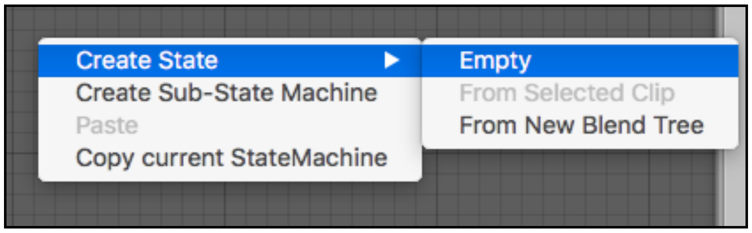


You just created a parameter that is accessible by script and has the power to control which animation plays.

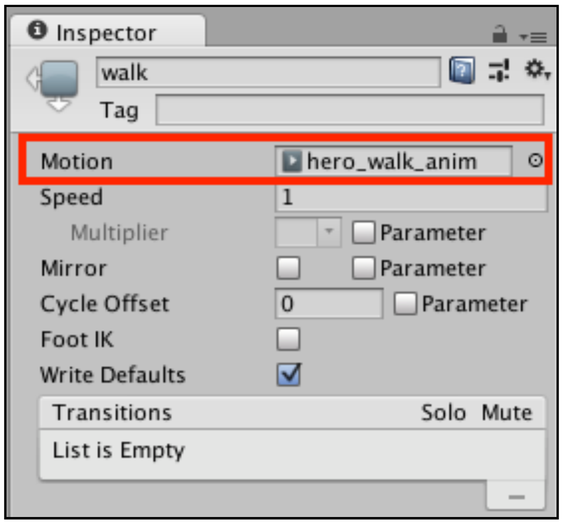


Speed 0.0

Create a new empty **Animation State** by **right-clicking** somewhere in the Layout area of the **Animator** window and selecting **Create State\Empty**.

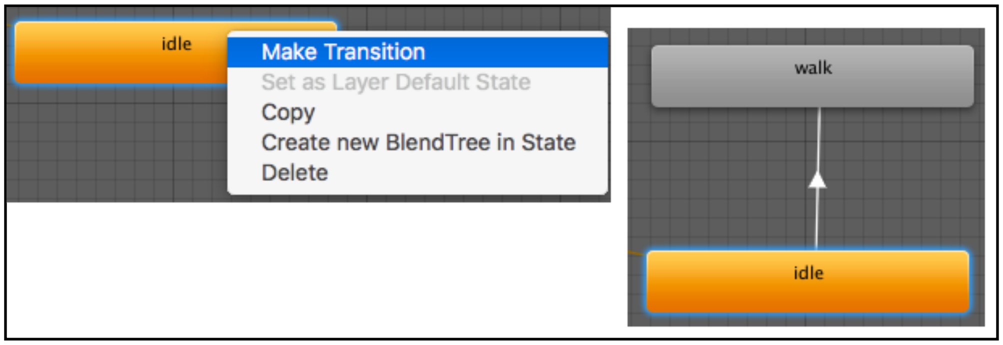


Select the new animation state and name it **walk**. Drag **hero_walk_anim**, which is located in the **Assets/Animation/Hero** folder, to the **Motion** field.



walk
Tag
Motion hero_walk_anim
Speed 1
Multiplier ☐ Parameter
Mirror ☐ ☐ Parameter
Cycle Offset 0 ☐ Parameter
Foot IK ☐
Write Defaults ✓
Transitions Solo Mute
List is Empty

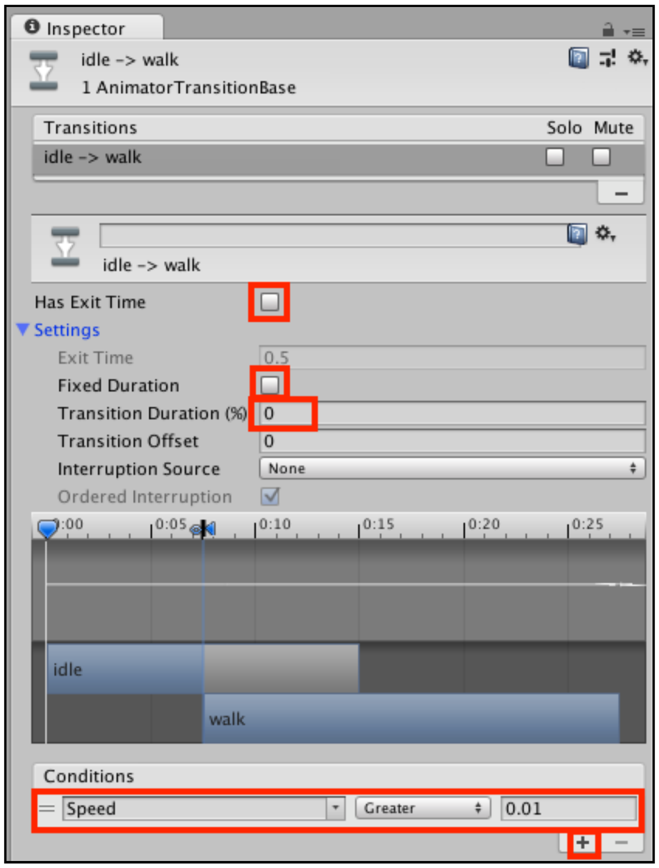
In the Animator Layout, **right-click** the idle animation, click **Make Transition** and select the **walk** state. This will create a transition from the idle to the walk state.



Select the transition arrow from idle to walk state and configure the transition in the Inspector:

Uncheck **Has Exit Time** then expand **Settings** and uncheck **Fixed Duration**. Set **Transition Duration** to `0`.

Click the **+** button at the bottom of the **Conditions** list. Click the **Parameter** drop-down and set **Speed** as the parameter to be evaluated. Set the condition to **Greater** with a value of `0.01`.


Inspector
idle -> walk
1 AnimatorTransitionBase

Transitions	Solo	Mute
idle -> walk	☐	☐

idle -> walk

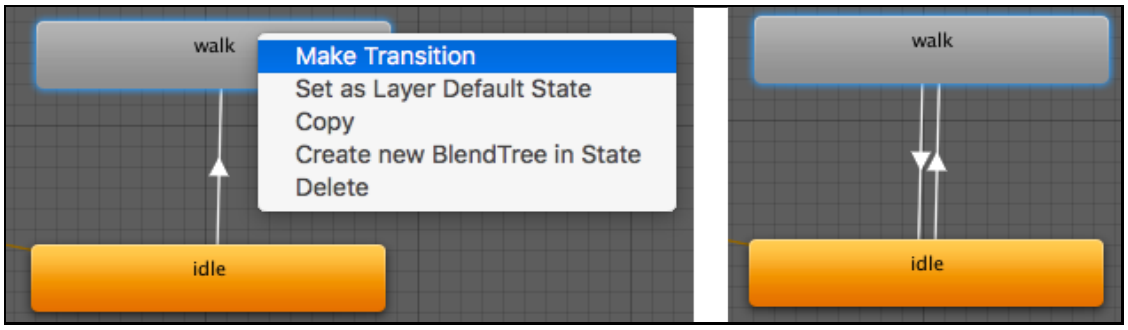
Has Exit Time ☐
Settings
Exit Time 0.5
Fixed Duration ☐
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Conditions
Speed Greater 0.01

You've created a new condition: when the animator is in idle state and Speed is greater than `0.01`, it will transition to the walk state.

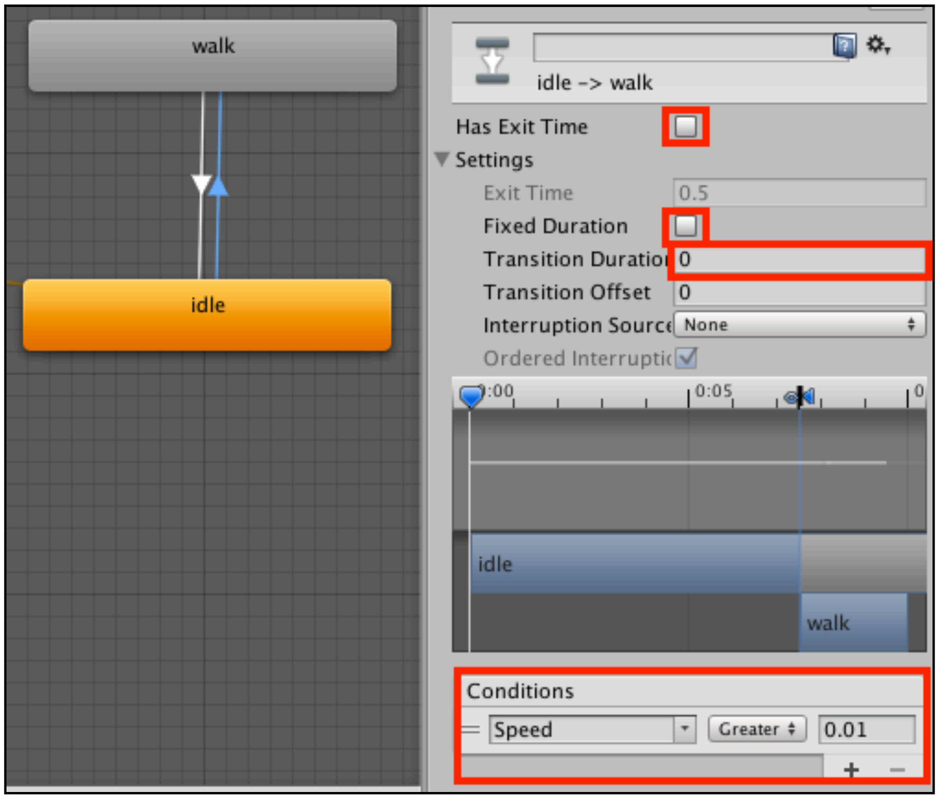
Adding the walk to idle transition is the last thing to do in the Animator view, at least for this section:

Right-click the **walk** animation, click **Make Transition** and select the **idle** state. This will create a transition from the walk to the idle state.



Next, select the transition arrow from walk to idle, and in the Inspector:

Uncheck **Has Exit Time** and **Fixed Duration**. Set **Transition Duration** to `0`. Add a new **Condition** by clicking the + button. Set the parameter to **Speed** and set the **condition** to **Less** with a value of `0.01`.



idle -> walk

Has Exit Time ☐

Settings

Exit Time 0.5

Fixed Duration ☐

Transition Duratio 0

Transition Offset 0

Interruption Source None

Ordered Interruptic ✓

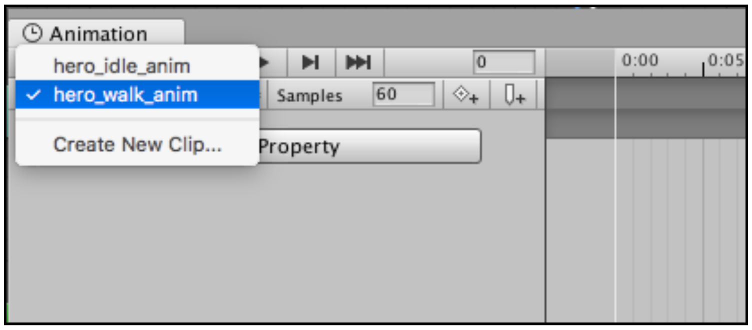
Conditions

Speed Greater 0.01

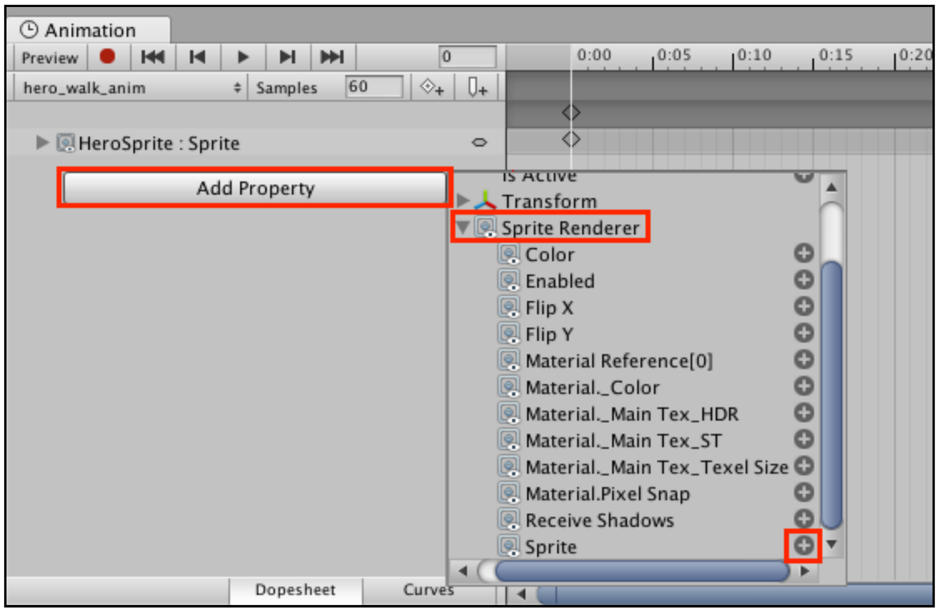
The state machine is done! In the next part, you're going to make the **walk** animation.

The walk animation

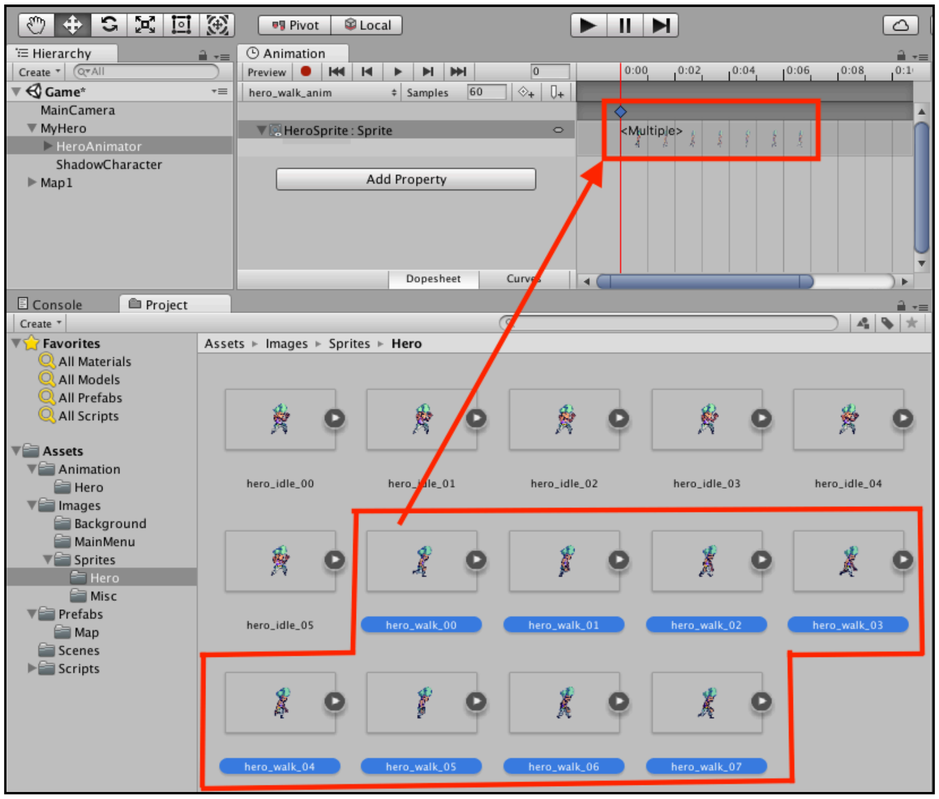
Select **HeroSprite** in the Hierarchy. Open the **Animation** view. Change the selected **animation** in the top-left drop-down to **hero_walk_anim**.



Click the **Add Property** button on the Animation Property view. Select **HeroSprite > SpriteRenderer** and click the **+** icon on the right side of the **Sprite** field. This will add the **HeroSprite : Sprite** property to the Animation timeline.

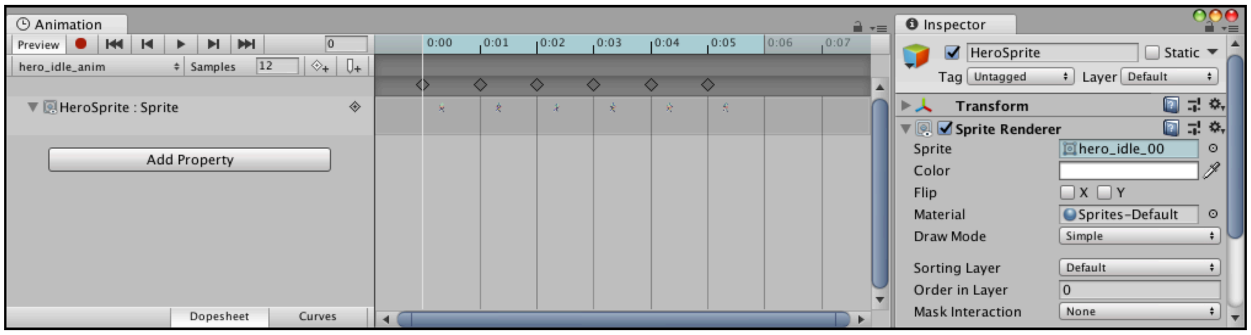


Select all the walk sprites — from **hero_walk_00–hero_walk_07** in the **Images\Sprites\Hero** and drag them to the newly created **HeroSprite: Sprite** parameter in the timeline. As it did when you made the idle state, Unity should add a keyframe per sprite in the timeline.



If there's a keyframe at **1:00** then delete it by **right-clicking** and selecting **Delete Key**. Also, set the **Samples** value to **12** like you did with the idle animation.

Scrub through the timeline to make sure the sprite animation is correct. You should have eight frames, with the first frame being **hero_walk_00** as the **HeroSprite: Sprite** value and the last being **hero_walk_07**.

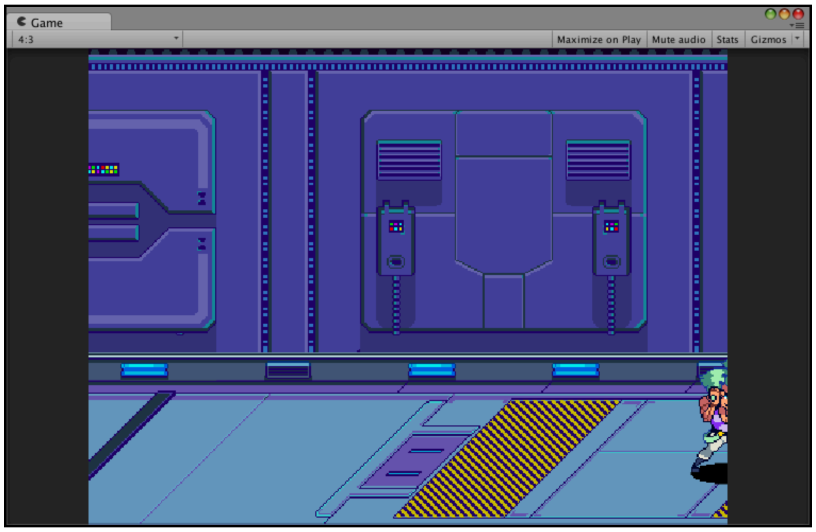


Remember to enable **Loop Time** in the **hero_walk_anim** asset so the animation plays continuously!

Press the **Play** button again and move the hero around using the **arrow keys**. Look at the confident strut! When you stop giving input, the game transitions him back to the idle state.

Congratulations, You now have a fully functional character walking around! Now try this: walk your green pompadoured self to the edge of the visible screen and keep going.

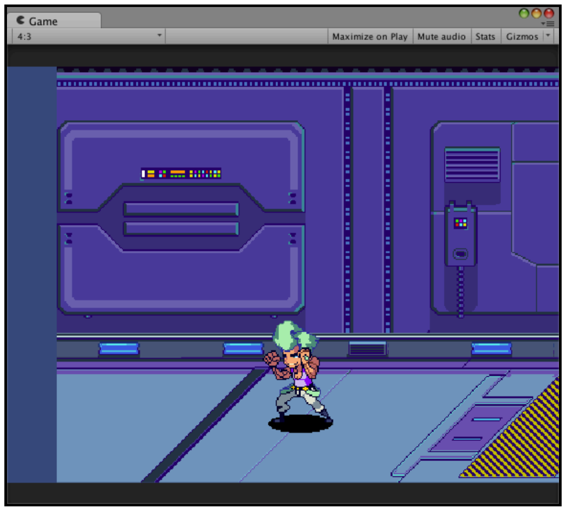
He struts right off the screen! The issue is not with the hero — it's that the camera doesn't follow him. You'll write a script to fix the issue that gives him a virtual personal camera crew.



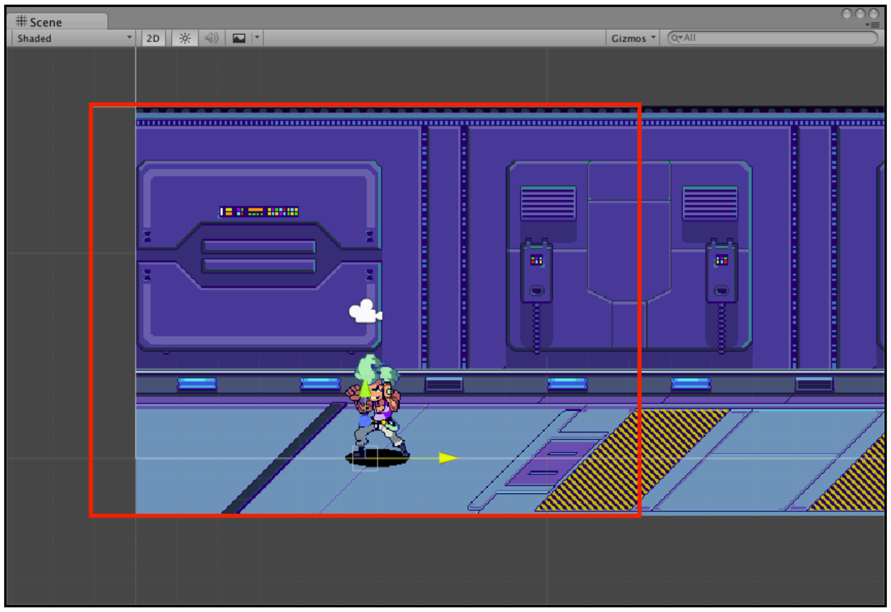
Camera controls

In this section, you'll create a `CameraBounds` class to keep the hero in view.

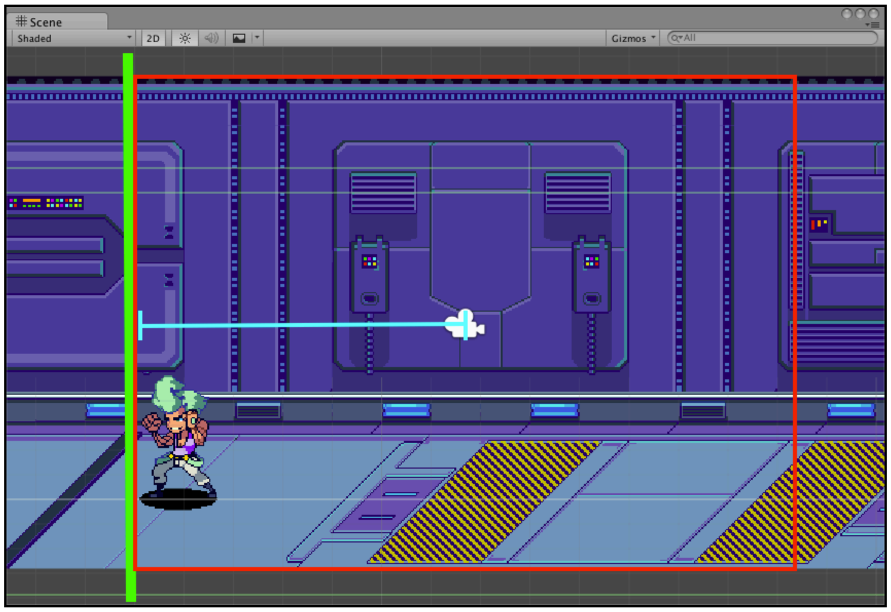
Although the camera needs to follow and keep him at the center of the screen, there are times when you need it to be static. For example, when he reaches an edge of the map. If the camera doesn't take these exceptions into account, this could happen:



The camera shows the edge of the map because its view, which is depicted by the red rectangle in the next screenshot below, extends past the edge of the background and reveals empty space.



The camera needs to stop moving towards an edge when it's near one. Here's a diagram to illustrate further:

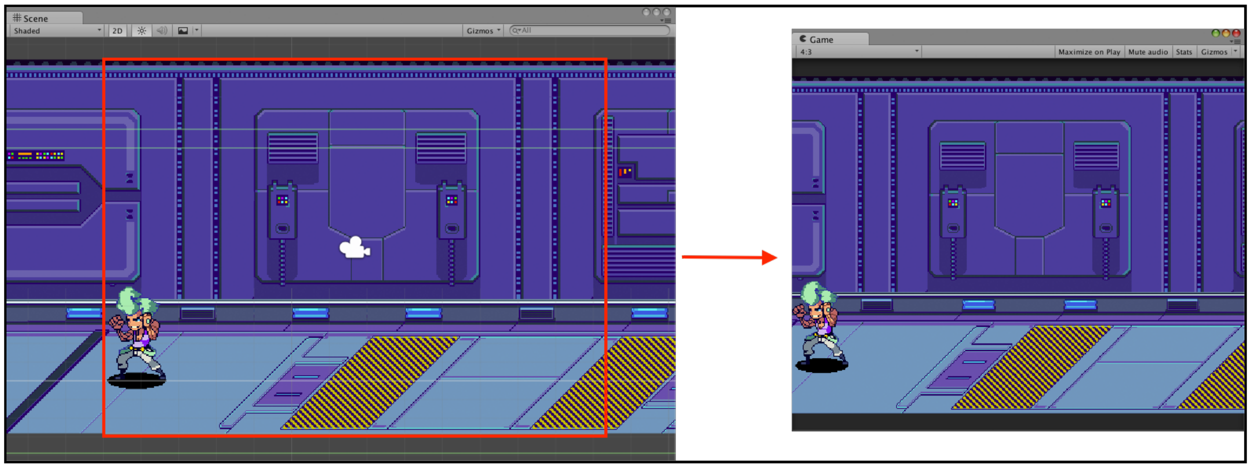


The green line represents the edge of the map as far as the camera is concerned. Similar to a collider around a sprite, the line isn't the actual edge of the map . It's offset a bit because the game needs space to animate the hero walking into the part of the map that's out of frame.

The camera shouldn't continue when its view box or **frustum** (depicted in red) hits this virtual edge.

To set this up, you need the value that equals 50 percent of the width of the camera's view, which is shown above in **light blue**. You'll use that value to calculate the point where the camera should stop moving.

The image below is what you're aiming for in this section.



Sounds like it's time to script! In the **Assets/Scripts** folder in the Project view, create a new C# script and name it **CameraBounds**.

Drag **CameraBounds** to the **MainCamera**.

Open the script and replace its contents with the following code:

```
using System.Collections;
using UnityEngine;

//1
[RequireComponent(typeof(Camera))]
public class CameraBounds : MonoBehaviour {
  //2
  public float minVisibleX;
  public float maxVisibleX;
  private float minValue;
  private float maxValue;
  public float cameraHalfWidth;

  //3
  private Camera activeCamera;

  //4
  public Transform cameraRoot;

  //5
  void Start() {
    //6
    activeCamera = Camera.main;

    //7
    cameraHalfWidth =
      Mathf.Abs(activeCamera.ScreenToWorldPoint(
        new Vector3(0, 0, 0)).x -
      activeCamera.ScreenToWorldPoint(
        new Vector3(Screen.width, 0, 0)).x) * 0.5f;
    minValue = minVisibleX + cameraHalfWidth;
    maxValue = maxVisibleX - cameraHalfWidth;
  }

  //8
  public void SetXPosition(float x) {
    Vector3 trans = cameraRoot.position;
    trans.x = Mathf.Clamp(x, minValue, maxValue);
    cameraRoot.position = trans;
  }
}
```

1. `[RequireComponent(typeof(Camera))]` defines an attribute that adds additional behavior to your methods, class and variables. The `RequireComponent` attribute requires any GameObject attached to this script to have a Camera component attached to it. It ensures that certain components appear together on the same GameObject.

2. These variables are necessary for `CameraBounds` to work. `minVisibleX` and `maxVisibleX` define the virtual edges of the map. `minValue` and `maxValue` stores the actual calculated limits of the camera's x-position. `cameraHalfWidth` stores the calculated half-width of the camera's view frustum.

3. `activeCamera` stores a reference to the scene's `MainCamera`.

4. `cameraRoot` is the target transform that will move left and right to follow the player.

5. `Start()` is a method that's native to Unity's GameObject lifecycle. It's called once at the start of the lifecycle.

6. This gets the MainCamera and stores a reference to it in the `activeCamera` variable.

7. The first line calculates half the width of the camera's view by transforming the screen's left-most and right-most points from screen space to world space equivalents using the camera's `ScreenToWorldPoint` method. Then it takes the absolute distance between these points as the camera's half-view width. The `minValue` and `maxValue` variables are assigned via simple addition and subtraction with the `minVisibleX` and `maxVisibleX` along with `cameraHalfWidth`.

8. `SetXPosition()` is the method that moves the `cameraRoot` GameObject. It'll try to move the object's x-position to the x-parameter of this method. It also limits the value of the x parameter between the allowed `minValue` and `maxValue` variables using the `Mathf.Clamp()` method.

Save the script and return to the editor.

The **CameraBounds** script is ready to go, but that `SetPostion(float x)` method is sitting idle — for the moment. You'll build a **GameManager** script to handle that task!

One manager to rule them all

A **GameManager** is a script that handles the general state of the whole game. It's a common feature in Unity games that has many purposes. In the case of PompaDroid, it'll:

- Check if the hero is **alive or dead**
- Determine when and where enemy droids should **spawn**
- Manage **input** controls
- Check player **victory conditions**
- Handle **camera events**, such as following the player and stopping the camera when a battle ensues.

Note: The **GameManager** script will be something you revisit and append as you progress through this book.

Create a new **C# script** in the **Assets/Scripts** folder and name it **GameManager**. Open it up, delete everything and add the following:

```
using UnityEngine;

public class GameManager : MonoBehaviour {
  //1
  public Hero actor;
  public bool cameraFollows = true;
  public CameraBounds cameraBounds;

  //2
  void Start() {
    cameraBounds.SetXPosition(cameraBounds.minVisibleX);
  }

  //3
```

```
    void Update() {
      if (cameraFollows) {
        cameraBounds.SetXPosition(actor.transform.position.x);
      }
    }
  }
```

Here’s what you just did in there:

1. You declared the required variables: `actor` will hold a reference to the `Hero` script, `cameraFollows` indicates whether the camera should follow the hero or not, and `cameraBounds` holds a reference to the `CameraBounds` script.

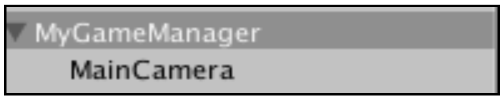
2. You set the initial position of the camera to the `minVisibleX` value with the `Start()` method.

3. In the `Update()` method, you move the camera’s x-position to the player’s x-position if the `cameraFollows` bool is set to `true`.

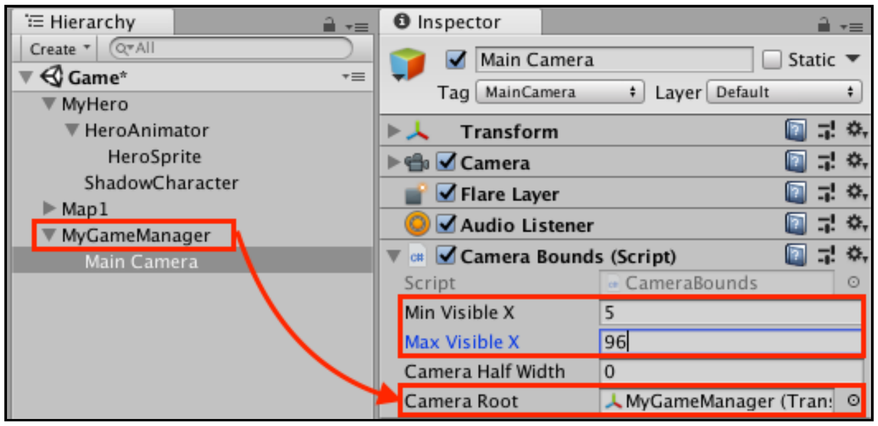
Save the script and go back to Unity. Now that you’ve configured the script, it’s time to assemble the Game Manager.

In the Hierarchy, **Create Empty** and **name** it **MyGameManager**. **Reset** its Transform. Drag the **GameManager** script from the Project view to the Inspector.

Make **MyGameManager** the parent of **MainCamera** then go to the Transform for **MainCamera**. Set the **Position** to `(0, 8.53, -7.61)`, **Rotation** to `(30, 0, 0)` and **Scale** to `(1, 1, 1)`.

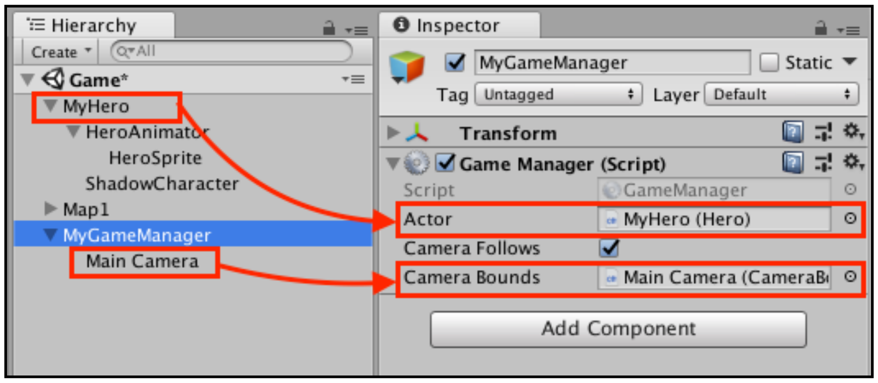


Drag **GameManager** to the **Camera Root** field of **CameraBounds** if it’s not there already. Set its **Min Visible X** to `5` and **Max Visible X** to `96`.



Main Camera
Tag MainCamera
Layer Default
Script CameraBounds
Min Visible X 5
Max Visible X 96
Camera Half Width 0
Camera Root MyGameManager (Tran:

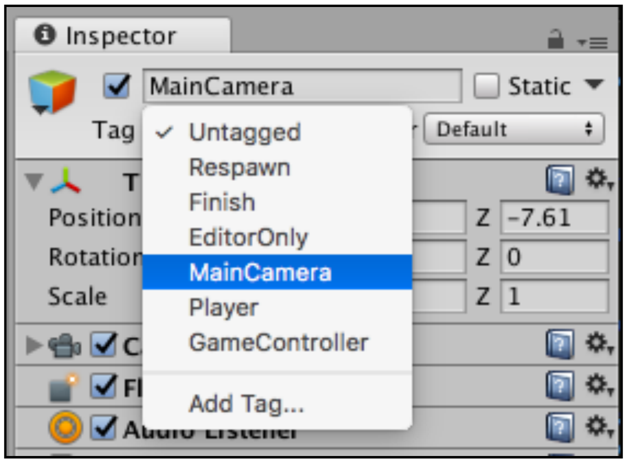
Drag **MyHero** to the **Actor** field of **MyGameManager**, and then drag the **MainCamera** to the **Camera Bounds** field of the **GameManager**.



Change the MyHero **Transform** to `(X:7, Y:0 Z:0)` and click the **Play**!

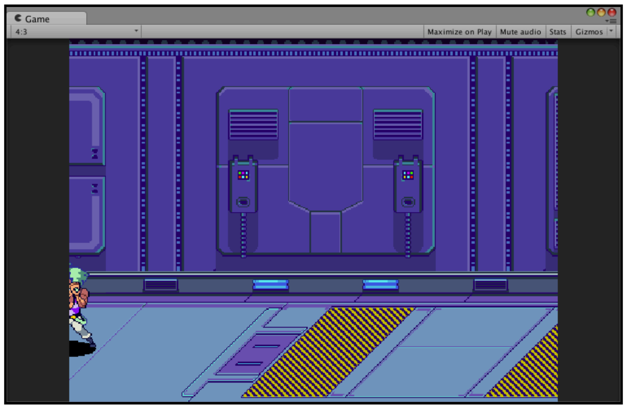
The camera should follow the hero as you move him left and right. When the hero moves far to the left, the camera stops following because there’s nothing there!

Note: If you get an error regarding a NullReferenceException in the CameraBounds script, it is probably caused by not specifying a MainCamera for the scene. The fix is simple: select **MainCamera** and set its **Tag** to **MainCamera**.



You can also stop the camera following the hero by unchecking the **Camera Follows** checkbox on the **CameraBounds** component.

You solved one problem and revealed another: the camera stops when you move too far to the left or right. Our hero literally has one foot out the door.

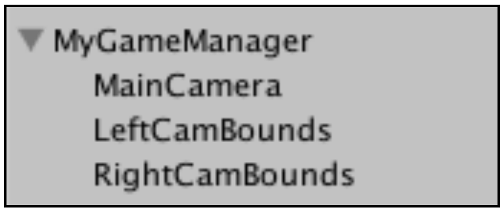


Not to worry — the fix is pretty simple. You're missing walls to prevent the hero from walking outside the camera's view.

In the Hierarchy, create a new **3D GameObject\Cube** and make it a child of **MyGameManager**. Name this GameObject **LeftCamBounds**. Set its **Position** to `(-8, 4.5, 0)`, **Rotation** to `(0, 90, 0)` and **Scale** to `(8, 10, 1)`.

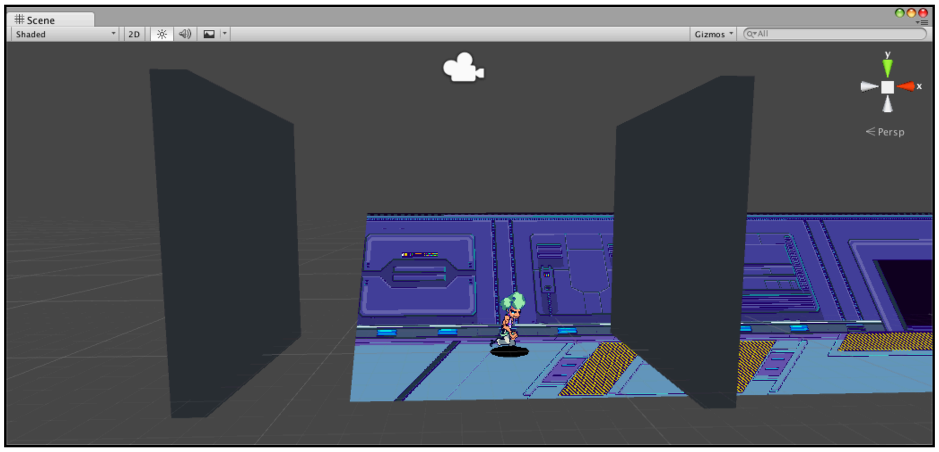
Duplicate the LeftCamBounds by selecting it, **right-clicking** and choosing **Duplicate**. Rename the new GameObject to **RightCamBounds**. Set its Transform's position, rotation and scale to **(8, 4.5, 0)**, **(0, 90, 0)** and **(8, 10, 1)** respectively.

The **MyGameManager** hierarchy should now look like this:

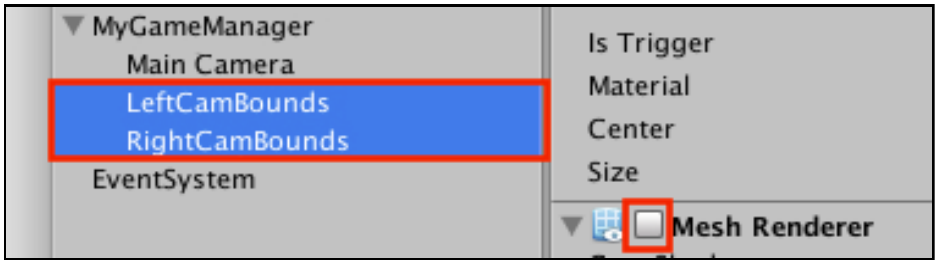


MyGameManager
MainCamera
LeftCamBounds
RightCamBounds

Check the scene. Two boxes should show up on either side of the hero:



Select **RightCamBounds** and **LeftCamBounds** in the Inspector and disable their **Mesh Renderer** components.



Well, you have walls, which is great. However, they won't position themselves correctly when the **aspect ratio** changes. Think for just a moment about all the screen sizes and aspect ratios out there...

The point is that you need to factor that in and support numerous aspect ratios when you make any game.

The **CameraBounds** script can easily reposition walls since it already knows the camera's half-view width value.

Open the **CameraBounds** script and add the following just below `public Transform cameraRoot;` — the last variable declaration:

```
//1
public Transform leftBounds;
public Transform rightBounds;
```

Next, add the following lines inside the `Start()` method, after the `maxValue = maxVisibleX - cameraHalfWidth;` line.

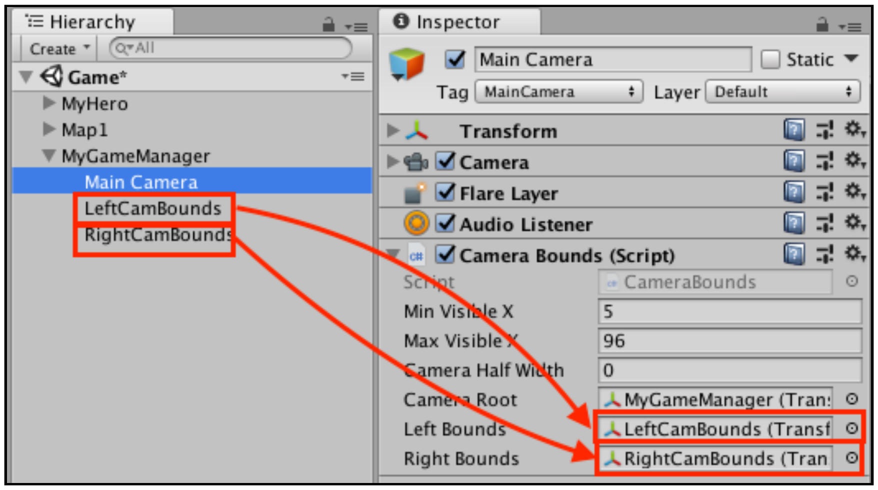
```
//2
Vector3 position;
position = leftBounds.transform.localPosition;
position.x = transform.localPosition.x - cameraHalfWidth;
leftBounds.transform.localPosition = position;

position = rightBounds.transform.localPosition;
position.x = transform.localPosition.x + cameraHalfWidth;
rightBounds.transform.localPosition = position;
```

Here's what these code snippets do:

1. Adds references to the `wall` GameObjects that you'll use in the `Start()` method.
2. Makes calculations to move the wall to the edge of the camera's view. Subtracts or adds the `cameraHalfWidth` value to the center of the camera to determine the edge. Moves the `leftBounds` and `rightBounds` objects to these edges.

Save and return to the editor. All that's left is to set the references in the scene:

Drag **LeftCamBounds** to the **Left Bounds** field on the **MainCamera**. Then drag **RightCamBounds** to the **Right Bounds** field on the **MainCamera**.



Main Camera
Tag: MainCamera
Layer: Default
Script: CameraBounds
Min Visible X: 5
Max Visible X: 96
Camera Half Width: 0
Camera Root: MyGameManager (Tran:
Left Bounds: LeftCamBounds (Transf
Right Bounds: RightCamBounds (Tran

Hit the **Play** button and play the game. The hero won't be able to escape the camera view, nor will he be able to live on the edge.

Where to go from here?

Great job! You accomplished a lot in this chapter. Some of the highlights include:

- Learned about Unity's **Mecanim** and physics engines
- Moved the character around the game
- Animated and transitioned between the hero's idle and walk states
- Implemented input for character movement
- Followed the hero around the map with a camera

Feel free to take a well-deserved break! In the next chapter, you'll add the more complex animation states to make the hero punch, jump and run across the map.

Chapter 4: Running, Jumping, and Punching

As you ease into this chapter, think about how far you've come. Level? Check. Hero? Check. He can even walk and idle.

Although it would be a little premature to throw him into a pit of droids — he has no defense and moves slower than your average snail — it's safe to say you've made a lot of progress!

It's time to add some pep to his step by giving him the ability to run. You'll also give him jumping and punching skills, which are standard movements for a beat 'em up game.

In this chapter, you'll:

- Move the hero around the map faster with **running**
- Give the hero the ability to reach greater heights by **jumping**
- Add an **Actor** superclass to the **Hero** script to enhance script reusability
- Teach the hero about basic **punching**

By the end of this chapter, he'll be ready to turn bots into scrap metal!

Run, sprite, run!

In the last chapter, you left the hero alone in a long, empty corridor with nothing to do but a walkabout. He’s already bored. You could increase the hero’s `walkSpeed` attribute so he moves faster, but here’s a better option: implement that running animation.

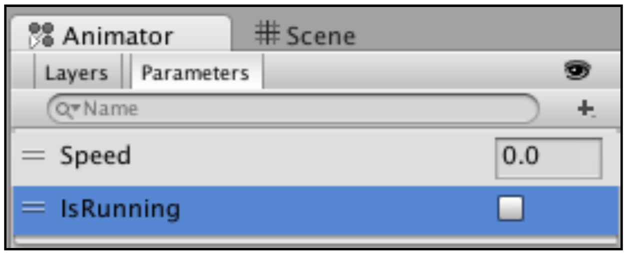
To add running, you’ll:

- Increase the movement speed
- Play the run animation
- Set up inputs to trigger running

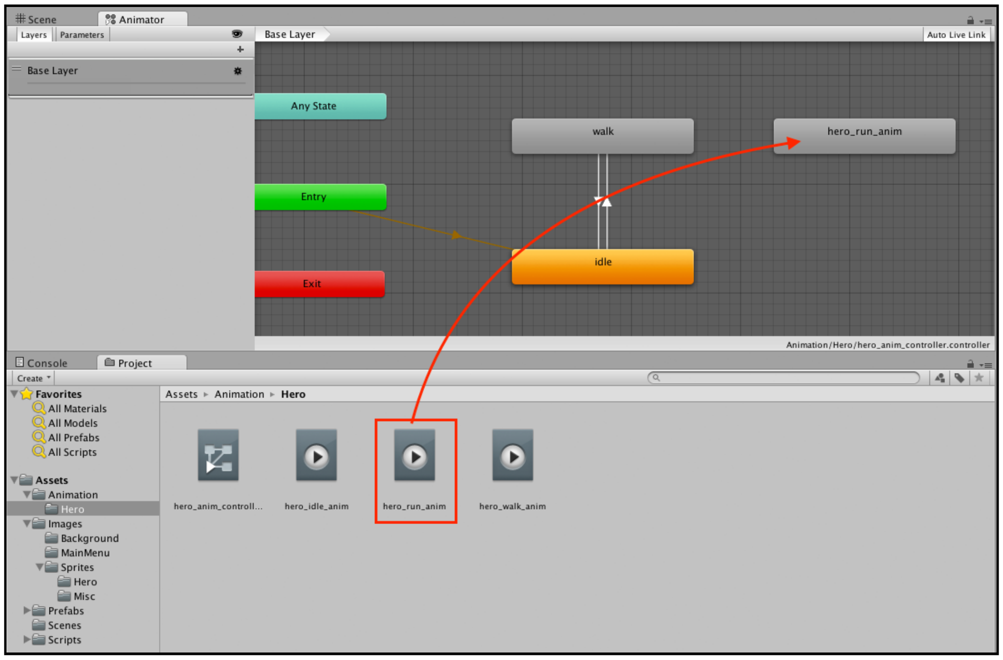
Import **Hero Run Animation.unitypackage** from the **Unity Packages** folder. It contains the **hero_run_anim animation** clip and all the sprites for the running animation.

In the **Animation\Hero** folder in the Project view, double-click **hero_anim_controller.controller** to open the Animator view. **Drag** the **Animator** view into Unity and dock it next to the Editor so that it’s handy (if it’s not there already).

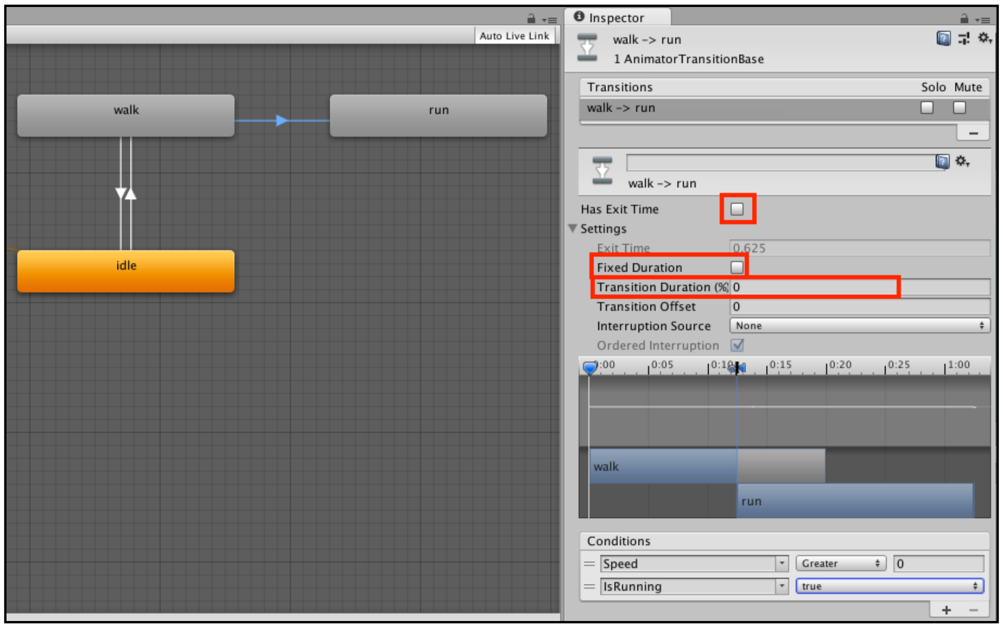
In the **Animator**, click the **Parameters** tab then the little + button. Select **Bool** and name it **IsRunning**. When set to true, this new parameter will trigger the running animation.



Drag **hero_run_anim** to the Animator view to create a new **Animation State** for it. Name it **run**.



One simply does not go from idle to run without some kind of transition, so you'll implement transitions between the other states and run. Right-click **walk**, pick **Make Transition**, and then click on **run**. Select the transition and go to the Inspector. Uncheck **Has Exit Time** and **Fixed Duration** in the transition settings and set **Transition Duration** to `0`.



walk -> run
1 AnimatorTransitionBase

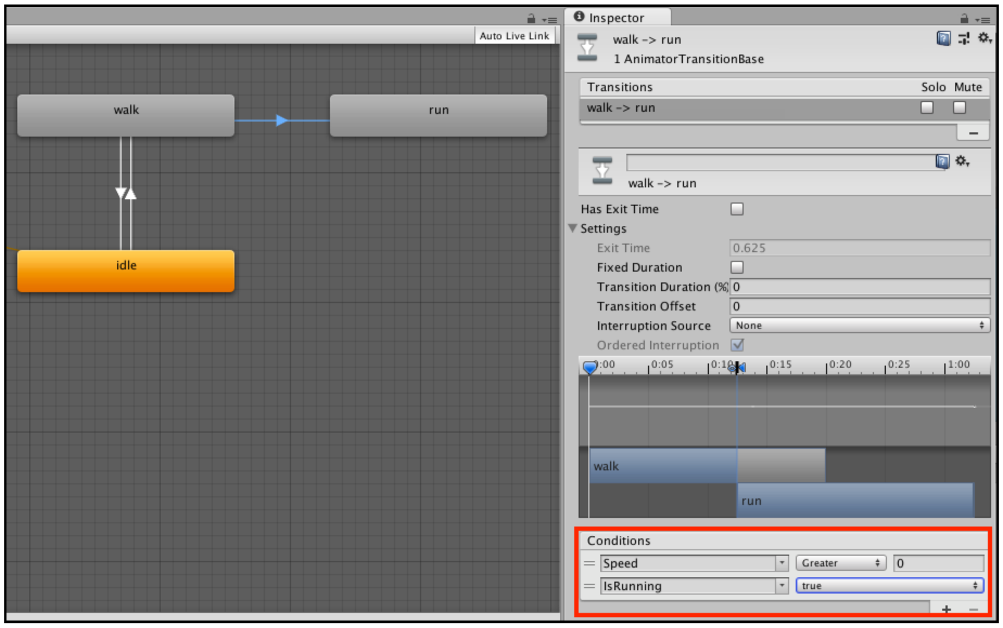
Transitions Solo Mute
walk -> run

walk -> run

Has Exit Time ☐
Settings
Exit Time 0.625
Fixed Duration ☐
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Conditions
Speed Greater 0
IsRunning true

Add two conditions: **Speed** is greater than `0.01` and **IsRunning** is **true**.



Inspector
walk -> run
1 AnimatorTransitionBase

Transitions	Solo	Mute
walk -> run	☐	☐

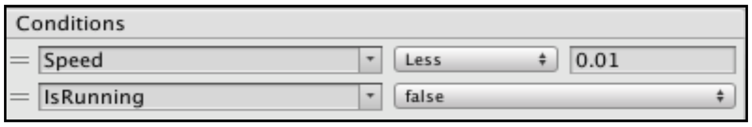
walk -> run

Has Exit Time ☐
Settings
Exit Time 0.625
Fixed Duration ☐
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Conditions

Speed	Greater	0
IsRunning	true	

Next, add a **transition** from run to walk, uncheck **Has Exit Time** and **Fixed Duration** and set **Transition Duration** to `0`. Add these conditions: **Speed** is less than `0.01` and **IsRunning** is **false**.

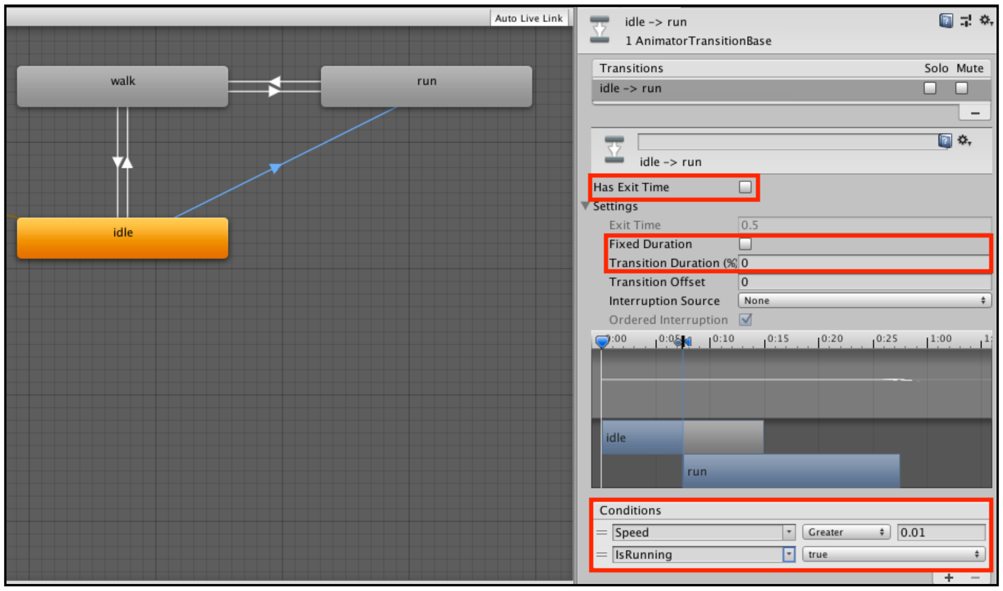

Conditions

Speed	Less	0.01
IsRunning	false	

These parameters make it so the hero must be moving faster than 0.01 and the **IsRunning** boolean must be true to trigger the running animation; otherwise, he'll walk if moving and idle if not.

You're making decisive progress towards giving your hero some serious moves. Next, you'll add transitions between idle and run as well as run to idle.

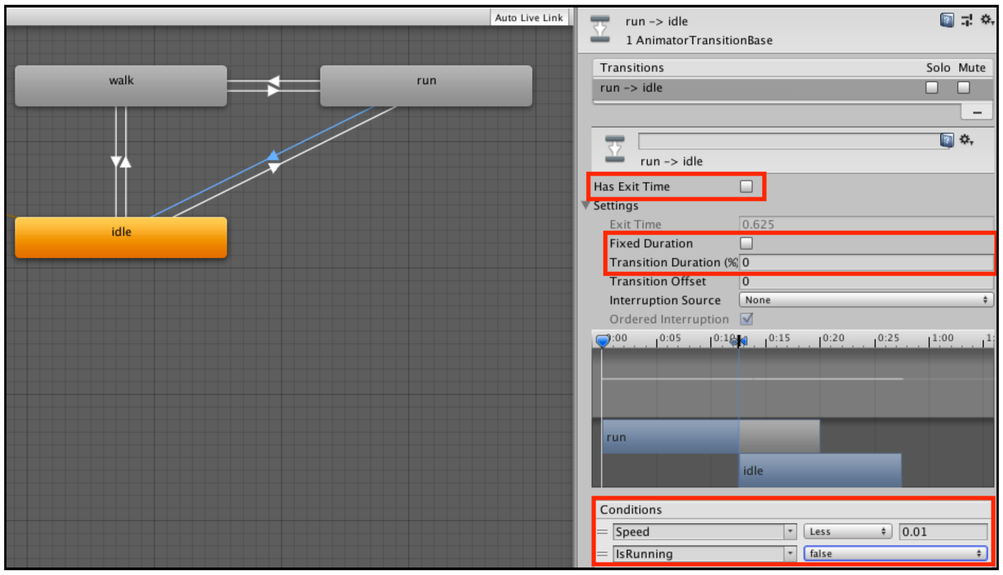
Create a new **transition** from the idle to the run state. Uncheck **Has Exit Time** and **Fixed Duration** and set **Transition Duration** to `0`. It takes the same conditions as walk to run, so add the condition **Speed** is greater than `0.01` and **IsRunning** is **true**.



idle -> run
1 AnimatorTransitionBase
Transitions Solo Mute
idle -> run
Has Exit Time ☐
Settings
Exit Time 0.5
Fixed Duration ☐
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓
Conditions
Speed Greater 0.01
IsRunning true

The last transition is from running to idle — you know, for when he needs to catch his breath after shredding some droids.

Create a new **transition** from run to idle, uncheck **Has Exit Time** and **Fixed Duration** then set **Transition Duration** to `0`. Add these conditions: **Speed** is less than `0.01` and **IsRunning** is **false**.



run -> idle
1 AnimatorTransitionBase
Transitions Solo Mute
run -> idle
Has Exit Time ☐
Settings
Exit Time 0.625
Fixed Duration ☐
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓
Conditions
Speed Less 0.01
IsRunning false

You've wired up the transitions like a pro, but you've got some scripting to do before he actually runs. Open the **Hero** script and add the following just under the `walkSpeed` member declaration:

```
public float runSpeed = 5;
```

This declares the runSpeed variable and sets it at five.

Add these right below:

```
bool isRunning;
bool isMoving;
float lastWalk;
public bool canRun = true;
float tapAgainToRunTime = 0.2f;
Vector3 lastWalkVector;
```

These are variable declarations needed to make the Hero run.

Finally, replace the `Update()` method with the new version below:

```
void Update() {
  float h = Input.GetAxisRaw ("Horizontal");
  float v = Input.GetAxisRaw ("Vertical");

  currentDir = new Vector3(h, 0, v);
  currentDir.Normalize();

  //1
  if ((v == 0 && h == 0)) {
    Stop ();
    isMoving = false;
  } else if (!isMoving && (v != 0 || h != 0)) {
    //2
    isMoving = true;
    float dotProduct = Vector3.Dot (currentDir,
lastWalkVector);

    //3
    if (canRun && Time.time < lastWalk + tapAgainToRunTime &&
dotProduct > 0) {
      Run ();
    } else {
      Walk ();

      //4
      if (h != 0) {
        lastWalkVector = currentDir;
        lastWalk = Time.time;
      }
    }
  }
}
```

You’ll get an error about the `Run` method not existing. Ignore it. You’ll resolve it shortly.

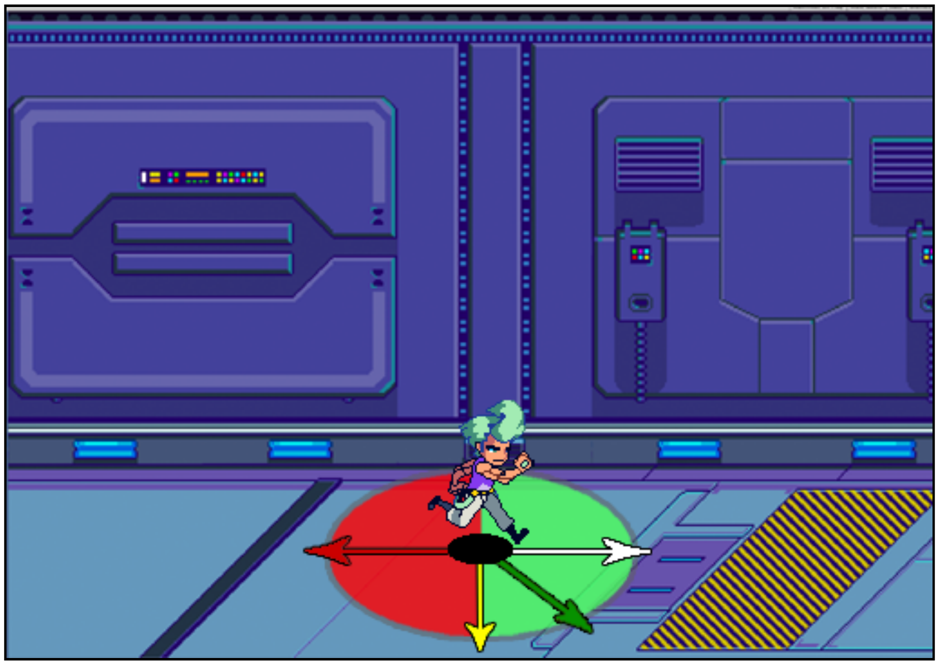
The first four lines of code should be familiar, since these were retained from the old version of `Update()`. To refresh your memory, the existing code simply gets directional user input and stores them in the appropriate variables. Let's go over the new lines of code one by one:

1. When there is no user input, you call `Stop()` and set the newly added `isMoving` boolean to false to indicate that the hero is not moving.
2. When there is user input in any direction, you first set `isMoving` to true to indicate that there is movement. Next, you get the dot product between the last movement direction and the current movement direction to determine if the user pressed the same direction twice. More on this later.
3. A positive dotProduct means the same direction was pressed twice. If both inputs occur within the time interval you set in `tapAgainToRunTime`, you call `Run()`, otherwise, you call `Walk()`. In short, this code makes the hero run whenever the same direction is rapidly pressed twice — also known as a double-tap.
4. Lastly, you store the current movement direction, and the current time — for use in the next call to `Update()` — as `lastWalkVector` and `lastWalk` respectively. You only do this for horizontal movement.

Note: `dotProduct` is a powerful operation you can use with vectors. With it, you can check if two input vectors face in the same direction; a value of 1 means they face the same way and -1 means they face in opposite directions. If the directions are perpendicular, the value is 0. To check if two button presses are pointing in the same direction, you just check if the dot product is greater than 0.

In the image below, the hero is moving in the direction of the white arrow. The green zone represents the possible paths that would trigger `run`.

In the green zone, the white and green arrows represent a `dotProduct` result that's greater than 0. Red represents where the `dotProduct` is less than zero — aka any negative value. The yellow arrow is perpendicular to the direction of the hero. Thus, it represents a `dotProduct` of 0.



Next, add the `Run()` method to the **Hero** script, ideally just below the closing bracket of the `public void Walk()` method.

```
public void Run() {
  speed = runSpeed;
  isRunning = true;
  baseAnim.SetBool("IsRunning", isRunning);
  baseAnim.SetFloat("Speed", speed);
}
```

This method sets the current speed to the hero's run speed and tells the Animator to change the animation accordingly.

Next, add the following inside the `Stop()` and `Walk()` methods, just before `baseAnim.SetFloat("Speed", speed);`

```
isRunning = false;
baseAnim.SetBool("IsRunning", isRunning);
```

This line updates the IsRunning parameter in the Animator, which in turn disables the run animation.

Save the Hero script, return to the editor and play the game! **Press** any **direction key** twice and **hold** the second press. Running mode engaged!

You can play around with his speed by changing the `runSpeed` property of the Hero in the Inspector. It's set to a value of `5`. Adjust and tweak it to your liking.

Your hero is now up and running! Cool, right?

He's going to get a little bored unless you give him the ability to escape the ground.

Jumping up and falling down

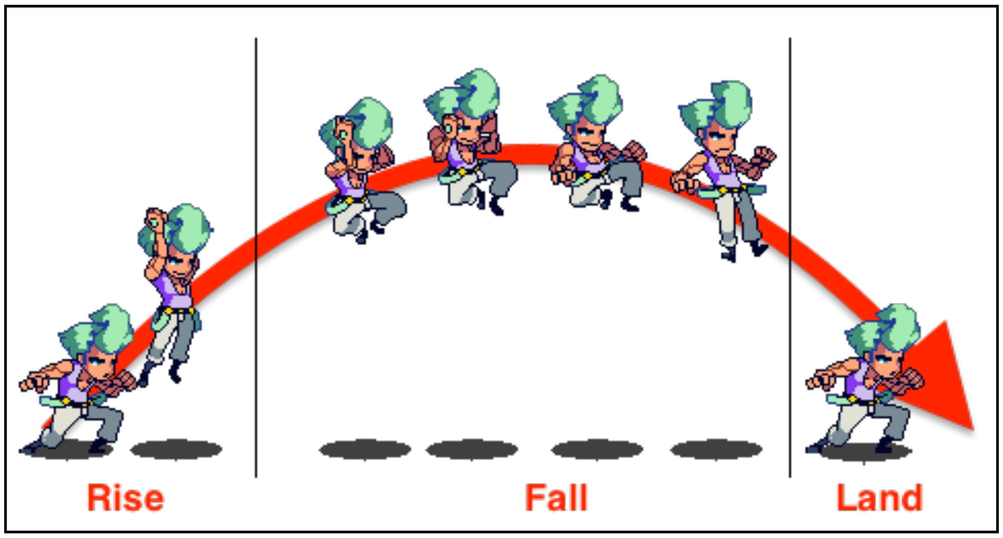
Making the hero jump is easier in practice than it seems when you're reading pages and pages of written instructions. TL/DR: It's easier than it seems at first!

Here's how it will work: when the player presses the jump button, the hero will play his jump animation, flying into the air as gracefully as a ballerina.

You'll introduce physics controls that'll "nudge" the hero with a bit of force at the start of the jump and let the physics engine handle how he should fall to create a natural-looking effect. He'll power upwards and fall back to the ground on a bit of an arc.

The jump animation comprises three phases: **Rise**, **Fall** and **Land**:

- **Rise** is when he's about to jump. He'll lean down slightly to prepare the big leap! This part of the animation will have the same duration regardless of jump height.
- **Fall** is when the hero is airborne. His feet will be in the air, and he'll fall towards the ground. This animation has a variable duration. When the hero does a little hop it'll be shorter but will last longer when he makes a big jump.
- **Land** is when the hero reconnects with the floor. He bends his knees to absorb the force. This animation, like the rise phase, is a fixed duration.

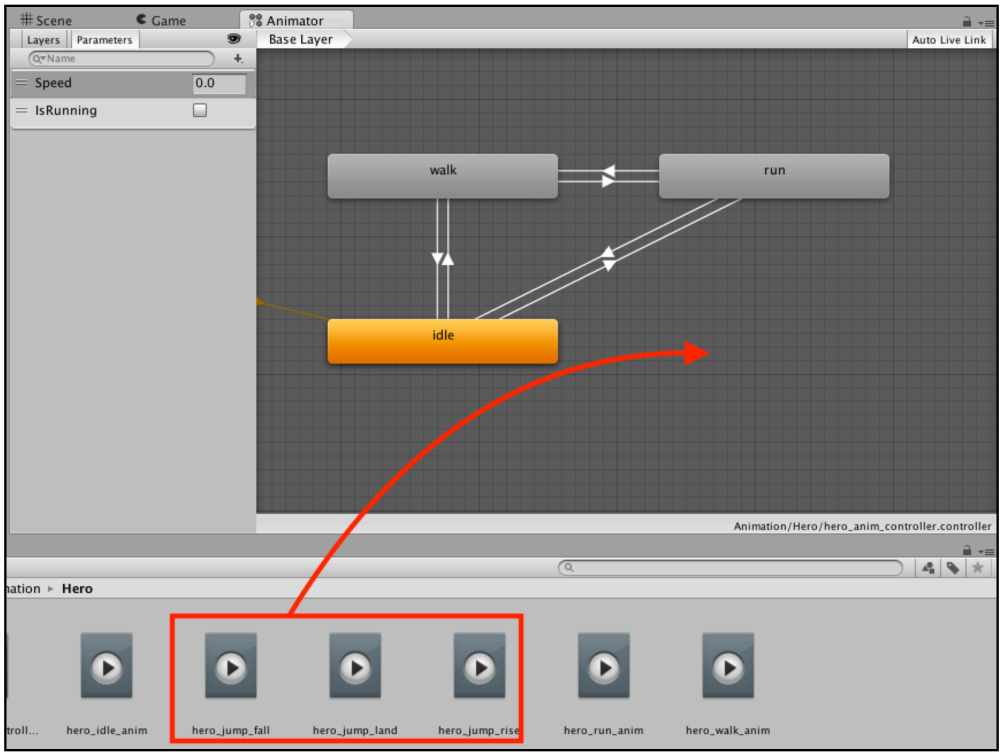


Setting up jump animation states

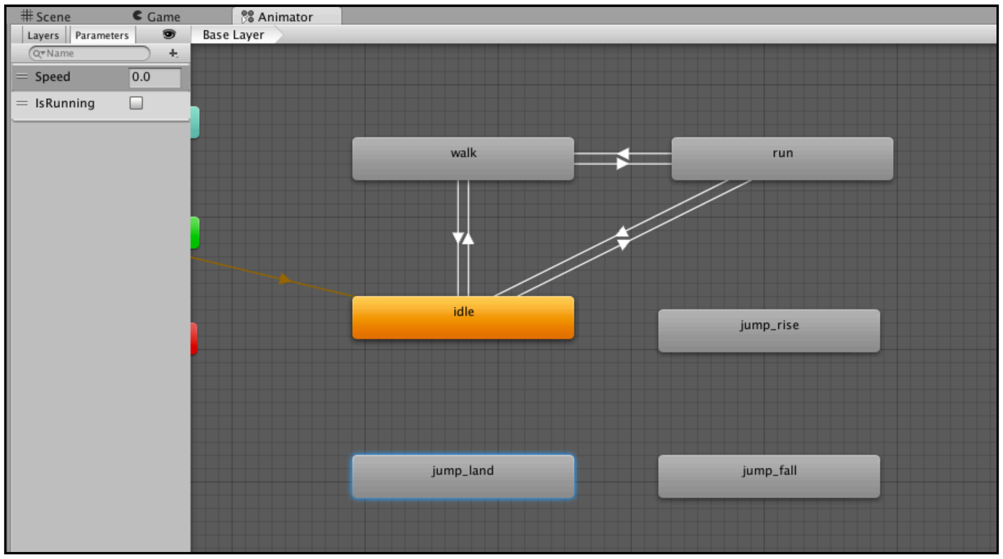
You'll need animation states to represent the various phases.

Import **Hero Jump Animation.unitypackage** from the **Unity Packages** folder. It has three animation clips and all the needed sprites.

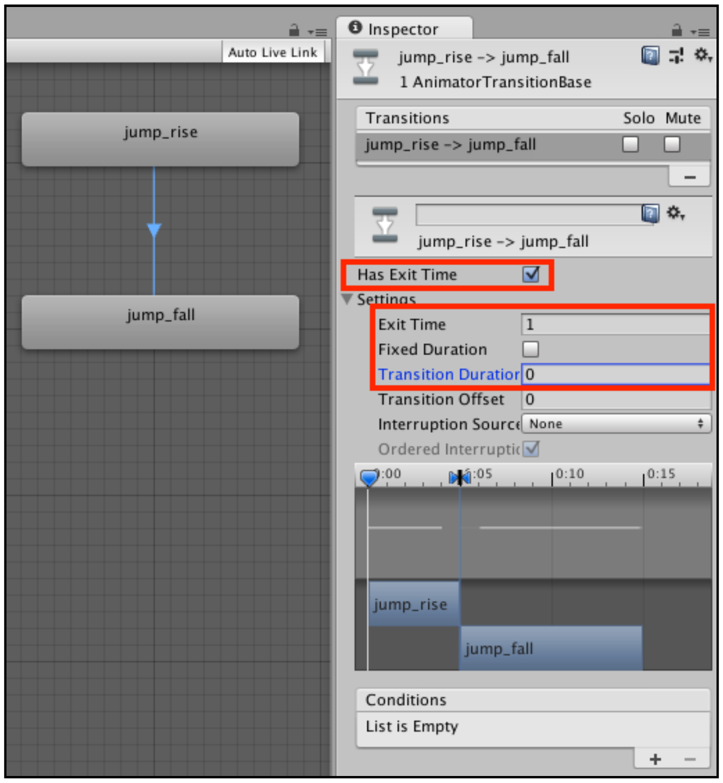
Double-click **hero_anim_controller.controller** in the **Animation\Hero** folder in the Project view to open it in the **Animator**. Add the **hero_jump_rise**, **hero_jump_fall** and **hero_jump_land** animation clips to the **hero_anim_controller** animator controller.



Remove the **hero_** prefix from the states so that you have three new states: **jump_rise**, **jump_land** and **jump_fall**. With that, you've added the animation states to the controller. Now you need transitions between phases.



Add a **transition** from the **jump_rise** to the **jump_fall** state then select it so you can edit its parameters. Keep **Has Exit Time** checked, set **Exit Time** to 1. Also, uncheck **Fixed Duration** and set **Transition Duration** to `0`.



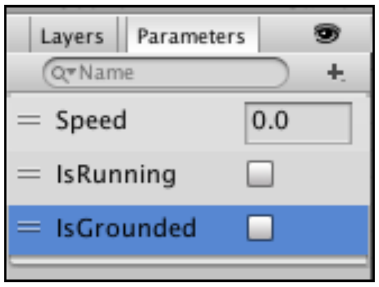
Has Exit Time ✓
Settings
Exit Time 1
Fixed Duration ☐
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

The jump_rise phase needs to complete its animation before transitioning to jump_fall. Turning on exit time and giving it a value tells Unity to transition to jump_fall *only* when jump_rise finishes.

Nicely done! The animator now knows that when the hero jumps, it must play the rise clip and transition to the fall clip after rise finishes. You've made it easy to detect because rise uses a fixed animation duration.

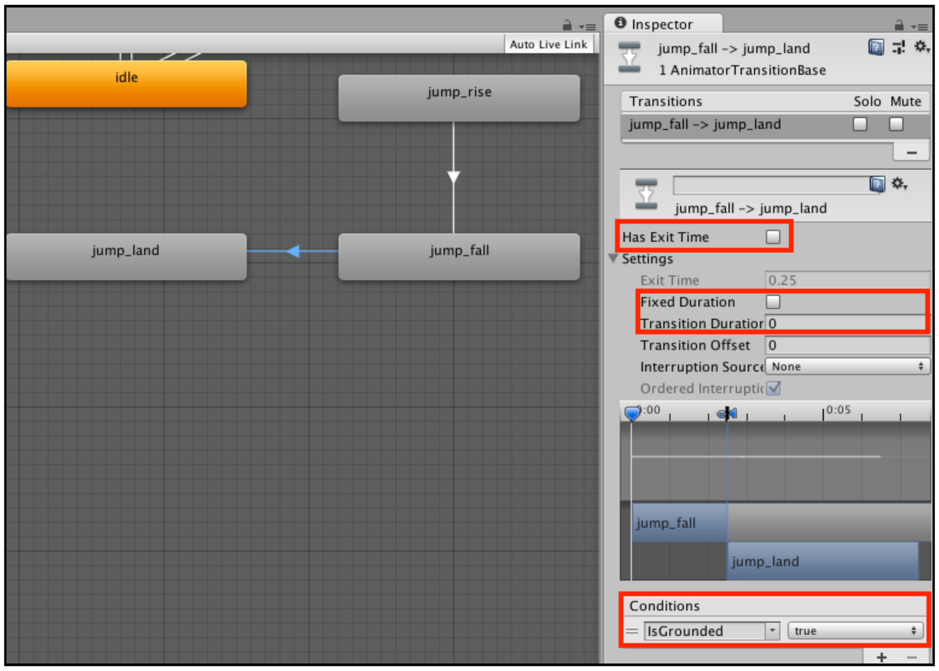
However, the fall clip has a variable duration, so a higher jump means it'll play longer than it would for a shorter jump. The game should transition to land *only* when the hero touches the floor.

In the **Events Parameter** of the **Animator** window, add a new **IsGrounded** boolean. It'll be your go-between from **jump_fall** to **jump_land**. This parameter will be set to **true** when the hero is in a collision with the floor.



Speed	0.0
IsRunning	☐
IsGrounded	☐

Create a new **transition** from **jump_fall** to **jump_land**. Select it, uncheck **Has Exit Time** and **Fixed Duration** then set **Transition Duration** to `0`. Add a new condition where **IsGrounded** is **true**.



jump_fall -> jump_land
1 AnimatorTransitionBase

Transitions	Solo	Mute
jump_fall -> jump_land	☐	☐

jump_fall -> jump_land

Has Exit Time ☐

Settings

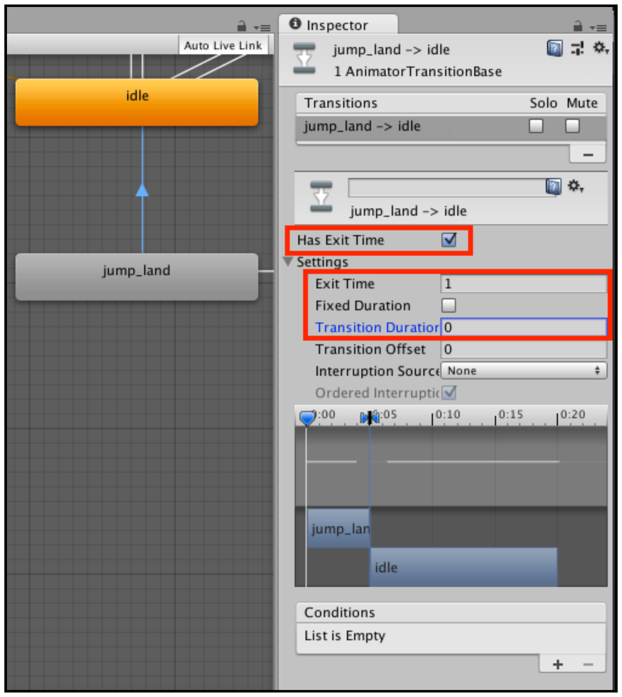
Exit Time	0.25
Fixed Duration	☐
Transition Duration	0
Transition Offset	0
Interruption Source	None
Ordered Interruption	✓

Conditions

IsGrounded	true

Add the final transition from **jump_land** to **idle**. Keep **Has Exit Time** checked, set **Exit Time** to `1`, uncheck **Fixed Duration** and set **Transition Duration** to `0`. This will make the Animator wait for **jump_land** to finish before it transitions to **idle**.

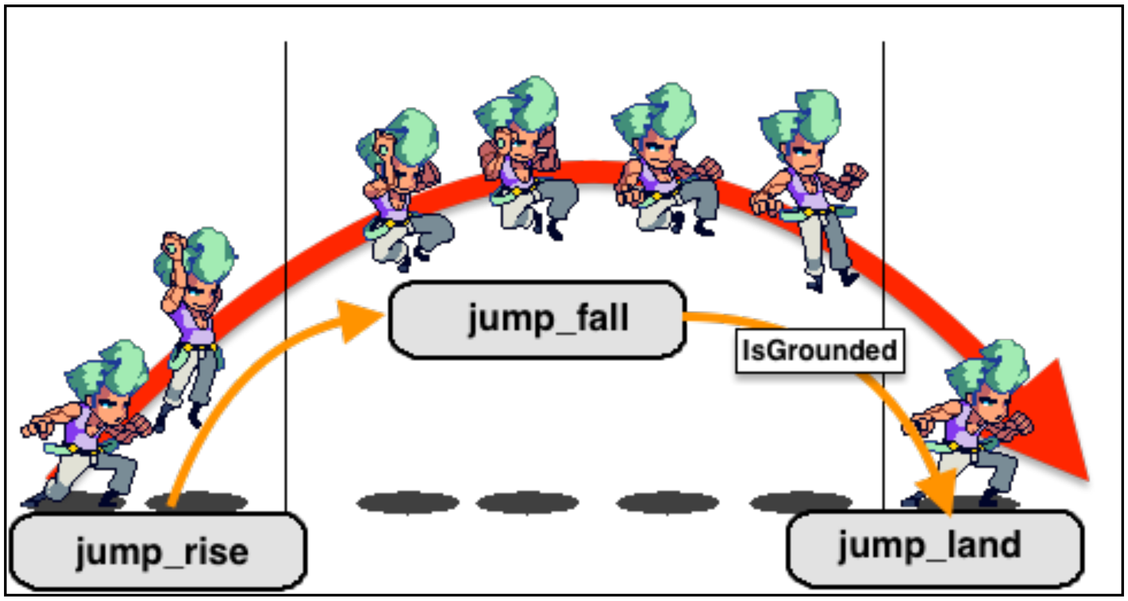
In effect, the hero will land and go back to bobbing, just waiting for the next enemy to show its face.



The jump animation flow is now complete!

To sum it up:

1. The jump starts with **jump_rise**.
2. It transitions to **jump_fall** when **jump_rise** is complete.
3. It plays **jump_land** when the parameter **IsGrounded** is **true**.
4. Finally, it transitions back to **idle** when **jump_land** completes.



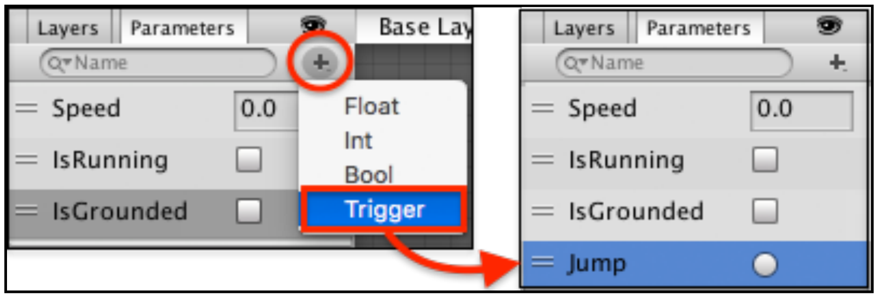
Adding conditions for jumping

Our pompadoured protagonist should only be able to jump when he's idling, walking or running. There will be other states, but trust me, you don't want him jumping about after he's died — sorry, no zombies allowed.

You need a transition from idle, walk and run to the **jump_rise** state. As you know, the player will press the jump button, which will trigger the jump animation. Specifically, it'll trip a parameter that ultimately tells the state machine to transition to the jump_rise state.

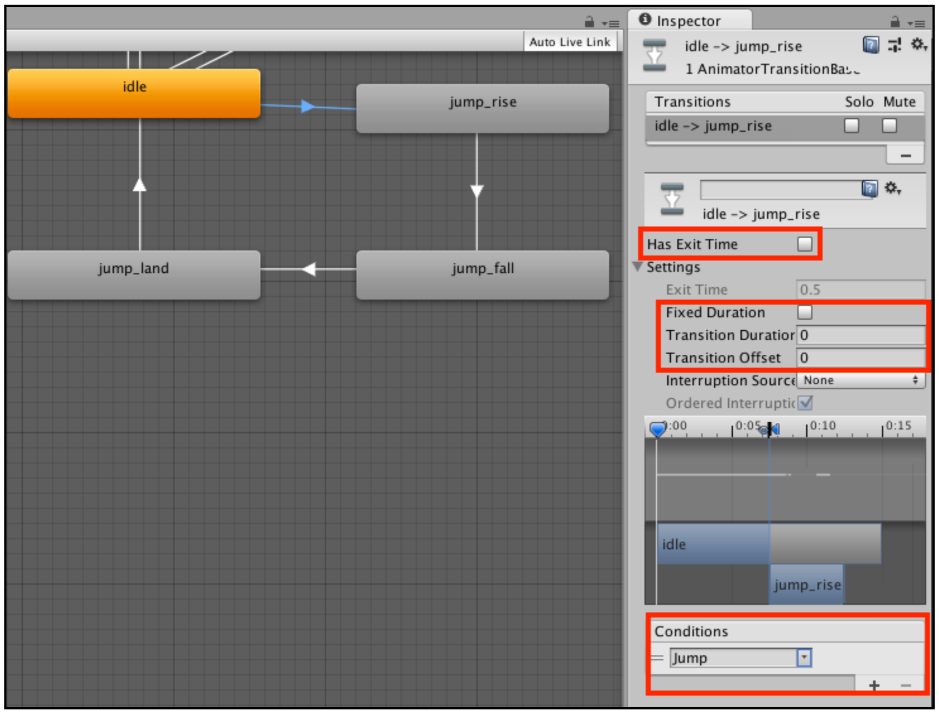
This isn't just any old parameter. It's a **Trigger** type, which acts like a boolean parameter but it automatically sets itself to false once any transition has used it.

Add a new **Trigger** parameter to **hero_anim_controller** and name it **Jump**.



Add a **transition** from **idle** to **jump_rise**. Uncheck **Has Exit Time** and **Fixed Duration** the set **Transition Duration** to `0`.

Add a **condition** and choose the **Jump** parameter. This time you won’t include any condition values because triggers don’t need to be set.



idle -> jump_rise
1 AnimatorTransitionBase

Transitions	Solo	Mute
idle -> jump_rise	☐	☐

idle -> jump_rise

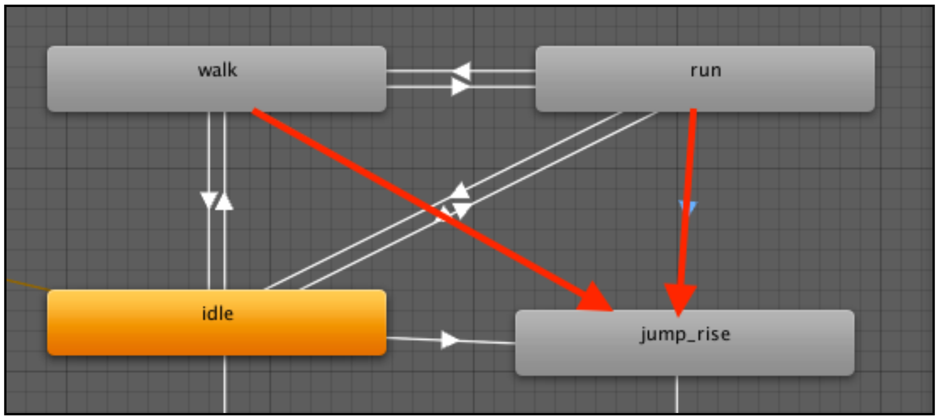
Has Exit Time ☐

Settings

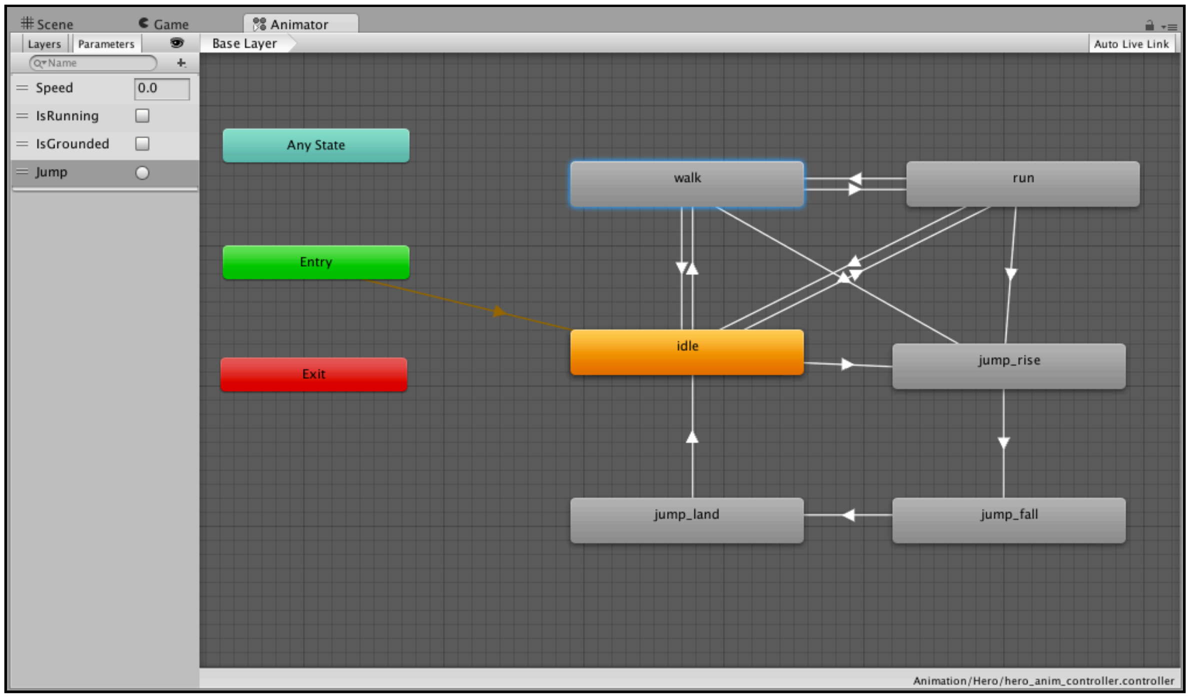
Exit Time	0.5
Fixed Duration	☐
Transition Duratior	0
Transition Offset	0
Interruption Source	None
Ordered Interruptic	✓

Conditions
Jump

Now add more transitions from **walk** to **jump_rise** and **run** to **jump_rise**. Make the settings and conditions the same as the **idle** to **jump_rise** transitions.



Your animator should look like similar to this:

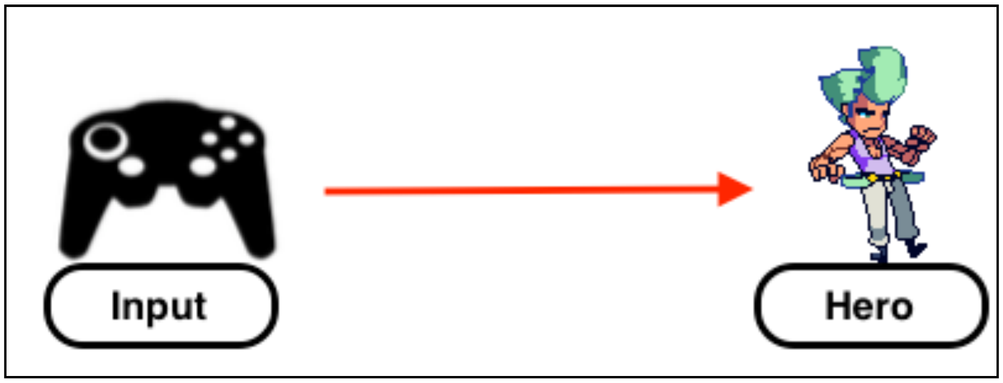


Okay, the animator is all set up. Off to scripting with you!

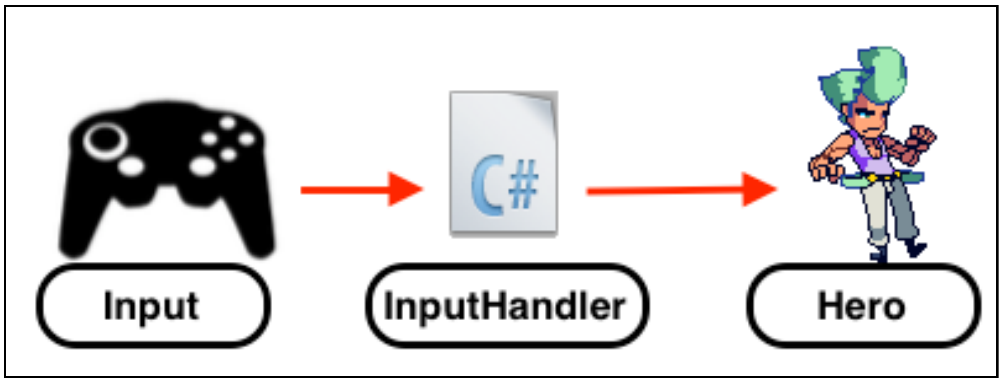
Jumping in code

Obviously, the pompadoured hero should jump when the player presses the jump button. But there’s a little more to it: Remember how I mentioned that the duration of jump_fall is variable? You need to make the UI allow for short and high jumps.

Until this point, the hero has received game input directly from the `Input` class.



You’ll create a mediator script which will allow you to process input and detect long button presses before passing the message to the Hero. You’ll name this mediator **InputHandler**.



Introducing InputHandler

Create a new **C# script** named **InputHandler** in the **Scripts** folder. Replace its contents with the following:

```
using UnityEngine;

public class InputHandler : MonoBehaviour {

  float horizontal;
  float vertical;
  bool jump;

  float lastJumpTime;
  bool isJumping;
  public float maxJumpDuration = 0.2f;

  //1
  public float GetVerticalAxis() {
    return vertical;
  }

  public float GetHorizontalAxis() {
    return horizontal;
  }

  public bool GetJumpButtonDown() {
    return jump;
  }

  void Update() {
    //2
    horizontal = Input.GetAxisRaw("Horizontal");
    vertical = Input.GetAxisRaw("Vertical");

    //3
    if(!jump && !isJumping && Input.GetButton("Jump")) {
      jump = true;
      lastJumpTime = Time.time;
      isJumping = true;
    } else if(!Input.GetButton("Jump")) {
      //4
      jump = false;
```

```
        isJumping = false;
      }

      //5
      if(jump && Time.time > lastJumpTime + maxJumpDuration) {
        jump = false;
      }
    }
  }
```

After declaring member variables at the top, this script:

1. Declares the methods `GetVerticalAxis()`, `GetHorizontalAxis()`, and `GetJumpButtonDown()`. The Hero script will call these methods to determine if there is movement or jump input.
2. Stores the `horizontal` and `vertical` input axes in the `Update` method but doesn't change them.
3. Stores the `jump` variable. When the player presses the jump button, `jump` is set to `true` until one of the following conditions are met:
4. Either the player releases the jump button…
5. Or, the jump button press is too long — the maximum button duration is stored in the `maxJumpDuration` variable.

Save the **InputHandler** script.

Add the following declarations inside of **Hero.cs** below the other variable declarations:

```
//1
bool isJumpLandAnim;
bool isJumpingAnim;

//2
public InputHandler input;

//3
public float jumpForce = 1750;
private float jumpDuration = 0.2f;
private float lastJumpTime;

//4
public bool isGrounded;
```

1. The `isJumpLandAnim` and `isJumpingAnim` variables are where you store whether the jump is currently playing or not, and they constantly update in the `Update()` method.
2. This reference to the **InputHandler** script powers the input for the **Hero** script.

3. These are your jump variables — `jumpForce` controls how much force to add when the hero jumps, `jumpDuration` detects higher jumps, and `lastJumpTime` is the last time the hero jumped.

4. The `isGrounded` variable detects if the **Hero** is in a collision with the floor.

Still in **Hero.cs**, find the `Update()` method and add these to the beginning of the method:

```
isJumpLandAnim =
baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_land");
isJumpingAnim =
baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_rise") ||
baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_fall");
```

These lines update the variables that store whether the hero is jumping or not.

While you're in `Update()`, replace the declaration of `float h` and `float v` with the following lines:

```
float h = input.GetHorizontalAxis ();
float v = input.GetVerticalAxis ();

bool jump = input.GetJumpButtonDown();
```

Still in the `Update()` method, add these lines at the end, right before the last closing curly brace:

```
if (jump &&
  !isJumpLandAnim &&
  (isGrounded || (isJumpingAnim && Time.time < lastJumpTime +
jumpDuration))) {
  Jump(currentDir);
}
```

This block closes up some of the conditions. It allows the hero to jump when the player presses the appropriate input *and* the hero isn't in the midst of landing *and* he `isGrounded` on the floor, or the `jumpDuration` hasn't expired.

For reference, your new `Update()` should now look like this:

```
void Update() {
  isJumpLandAnim =
baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_land");
  isJumpingAnim =
baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_rise") ||
baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_fall");

  float h = input.GetHorizontalAxis ();
  float v = input.GetVerticalAxis ();
```

```
    bool jump = input.GetJumpButtonDown();

    currentDir = new Vector3(h, 0, v);
    currentDir.Normalize();

    if ((v == 0 && h == 0)) {
      Stop ();
      isMoving = false;
    } else if (!isMoving && (v != 0 || h != 0)) {
      isMoving = true;

      float dotProduct = Vector3.Dot (currentDir,
  lastWalkVector);

      if (canRun && Time.time < lastWalk + tapAgainToRunTime &&
  dotProduct > 0) {
        Run ();
      } else {
        Walk ();

        //3
        if (h != 0) {
          lastWalkVector = currentDir;
          lastWalk = Time.time;
        }
      }
    }

    if (jump && !isJumpLandAnim &&
        (isGrounded || (isJumpingAnim && Time.time < lastJumpTime +
  jumpDuration))) {
          Jump(currentDir);
    }
  }
```

You're going to get an error saying the `Jump` method doesn't exist yet. Ignore it because you'll add it shortly.

Add the following methods inside **Hero.cs**:

```
  void Jump(Vector3 direction) {
    //1
    if (!isJumpingAnim) {
      baseAnim.SetTrigger ("Jump");
      lastJumpTime = Time.time;
      //2
      Vector3 horizontalVector = new Vector3(direction.x, 0, direction.z) *
  speed * 40;
      body.AddForce(horizontalVector,ForceMode.Force);
    }
    //3
    Vector3 verticalVector = Vector3.up * jumpForce* Time.deltaTime;
    body.AddForce(verticalVector,ForceMode.Force);
  }

  //4
```

```
void OnCollisionEnter(Collision collision) {
  if (collision.collider.name == "Floor") {
    isGrounded = true;
    baseAnim.SetBool("IsGrounded", isGrounded);
    DidLand();
  }
}

//5
void OnCollisionExit(Collision collision) {
  if (collision.collider.name == "Floor") {
    isGrounded = false;
    baseAnim.SetBool("IsGrounded", isGrounded);
  }
}

//6
void DidLand()
{
  Walk();
}
```

That's a big chunk. Let's step through what you're doing in there:

1. Here, the `Jump` method becomes the mechanism that applies the jump force of the hero.
2. `Jump` is called until the player stops pressing the jump button or the `jumpDuration` expires. At the beginning of every jump, it applies a jump force in the direction the hero character is facing.
3. When the `Jump` method is called, this applies a vertical force to move the player upwards.
4. The `OnCollisionEnter(Collision collision)` is a built-in method for MonoBehaviours. It is automatically called whenever another collider hits the attached collider. Its role is to detect when the hero collides with a GameObject named "Floor" and to update the **IsGrounded** parameter of `baseAnim`.
5. `OnCollisionEnter(Collision collision)` is another built-in method that is called whenever an attached collider detects that another collider has stopped colliding with it. Here, you use it to detect when the hero is in the air. It also updates the `IsGrounded`" parameter of `baseAnim`.
6. Finally, `DidLand` is called when the hero collides with the floor. When that happens, you simply call `Walk()` to get the hero walking again.

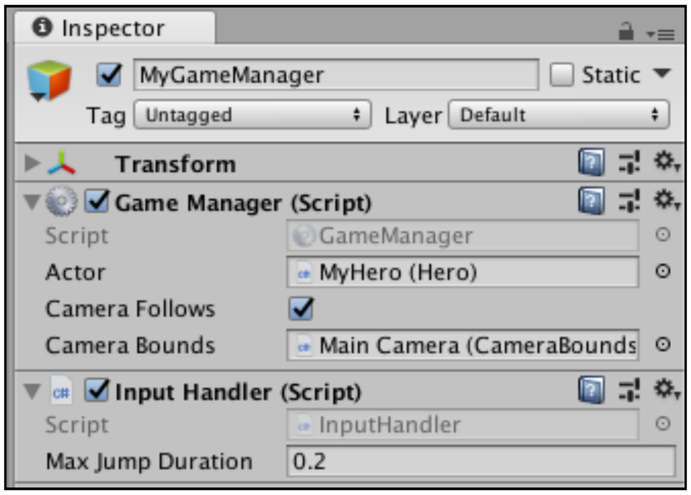
Next, still in **Hero.cs**, replace the `FixedUpdate()` method with the following:

```
void FixedUpdate() {
    Vector3 moveVector = currentDir * speed;

    if(isGrounded){
      body.MovePosition (transform.position + moveVector *
  Time.fixedDeltaTime);
      baseAnim.SetFloat ("Speed", moveVector.magnitude);
  }

    if (moveVector != Vector3.zero) {
      if (moveVector.x != 0) {
        isFacingLeft = moveVector.x < 0;
      }
      FlipSprite (isFacingLeft);
    }
  }
```

You added a new condition before letting the hero move. When `isGrounded` is false, the player won't be able to move the hero because the pompadoured protagonist is airborne. Remember that in `OnCollisionEnter`, you set `isGrounded` to true when the hero collides with the floor, and to false when the hero jumps in `OnCollisionExit`.

Save the script and return to the editor. For now, you're finished with the jumping code and are down to adding a few references to wire it all together!

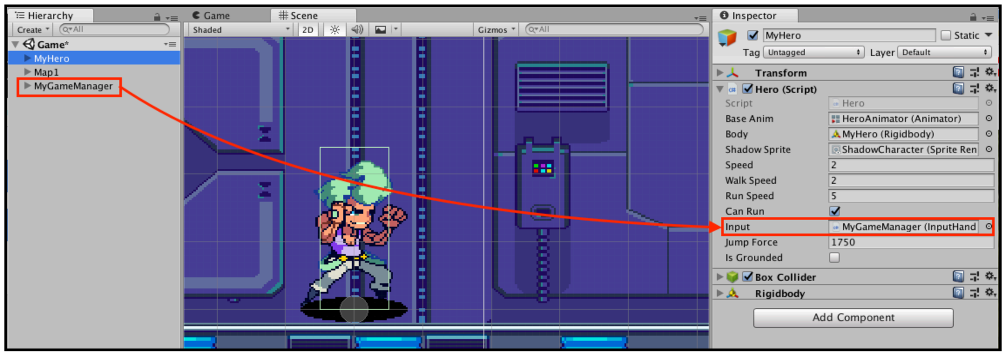
In the Hierarchy, select **MyGameManager** and add a new **InputHandler** component.



Script: GameManager
Actor: MyHero (Hero)
Camera Follows: ✓
Camera Bounds: Main Camera (CameraBounds

Script: InputHandler
Max Jump Duration: 0.2

Set the **Input** property of the **Hero** to that **InputHandler** by clicking the **selection** button and selecting **MyGameManager** from the list.



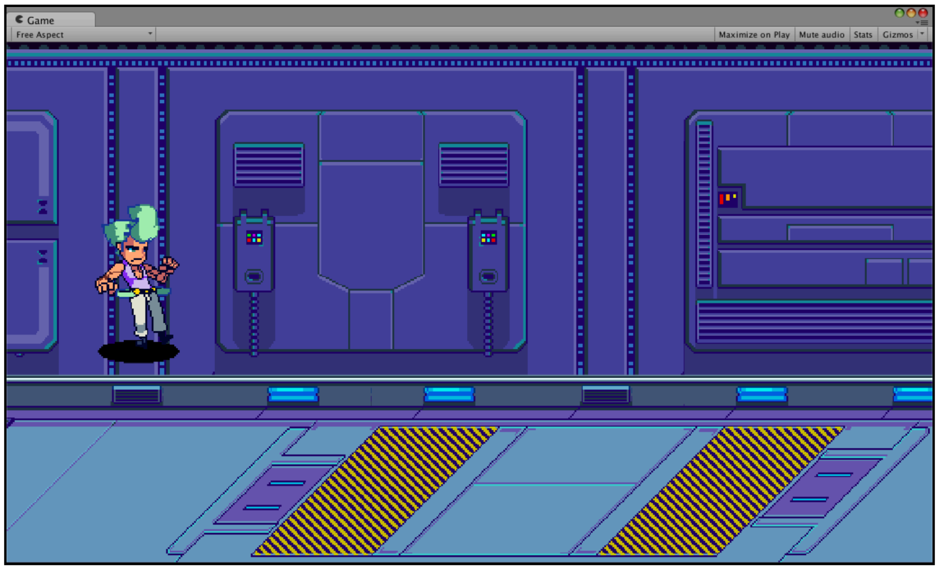
MyHero ☐ Static
Tag Untagged Layer Default

Hero (Script)

Script	Hero
Base Anim	HeroAnimator (Animator)
Body	MyHero (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Ren
Speed	2
Walk Speed	2
Run Speed	5
Can Run	✓
Input	MyGameManager (InputHand
Jump Force	1750
Is Grounded	☐

Run the game and press the **S** key to jump. Hold it down longer to jump higher and tap it really fast to make him hop. :]

What is up with that shadow? It’s a bit clingy, don’t you think?

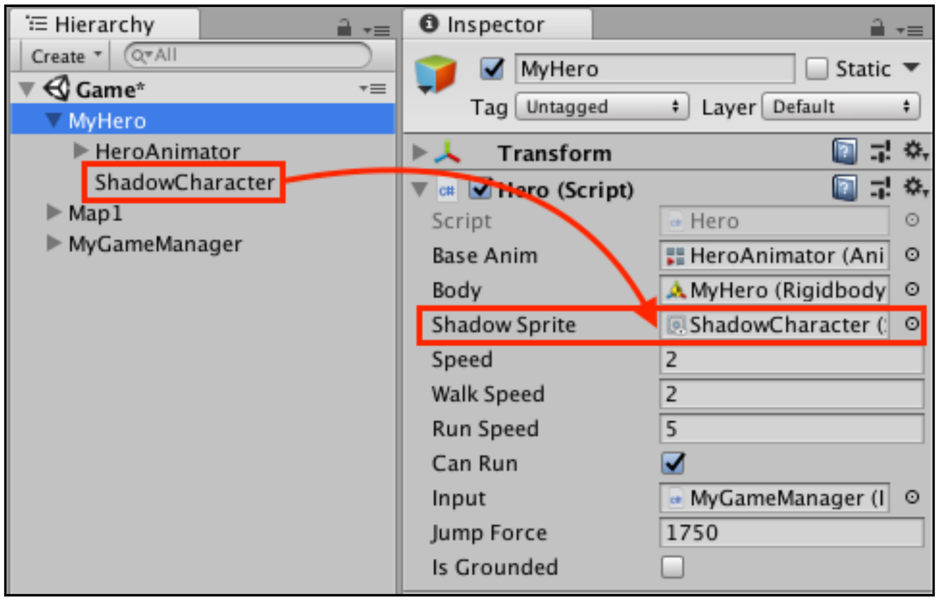


Open the **Hero** script and add this block before the closing brace `}` in the `Update()` method:

```
Vector3 shadowSpritePosition = shadowSprite.transform.position;
shadowSpritePosition.y = 0;
shadowSprite.transform.position = shadowSpritePosition;
```

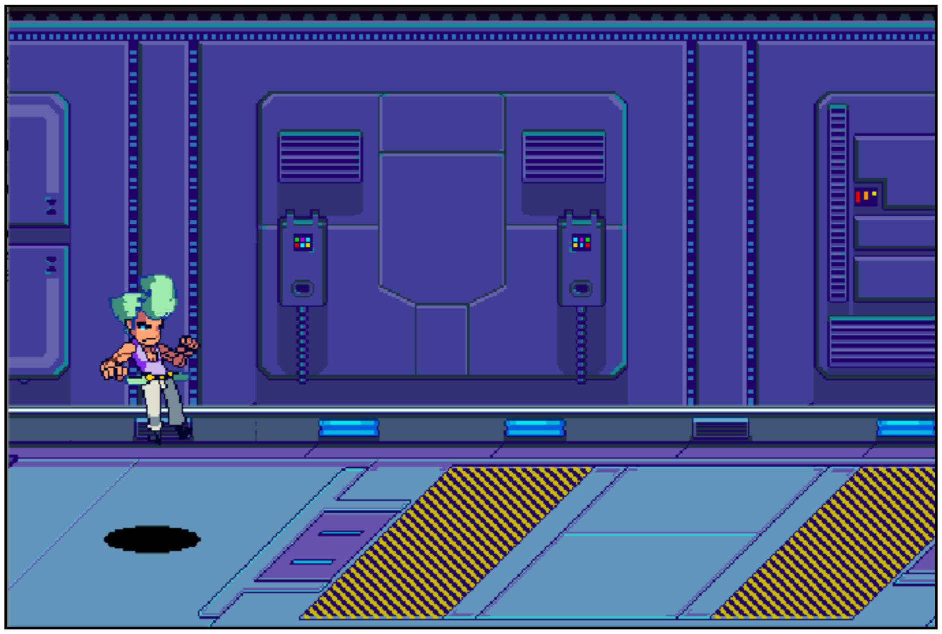
These lines tell the shadow sprite to give the hero some space by staying at a **y** value of `0`.

Save the script and go back to Unity. Select **MyHero** and assign the **Shadow Sprite** property to the **ShadowCharacter**.



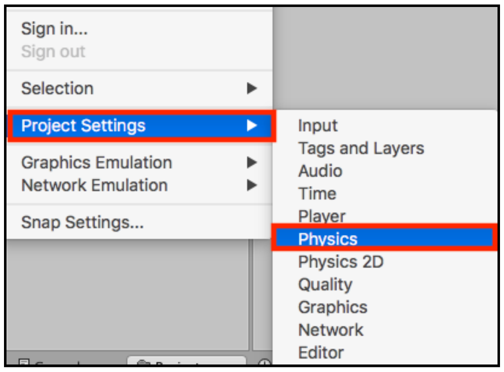
Script: Hero
Base Anim: HeroAnimator (Ani
Body: MyHero (Rigidbody
Shadow Sprite: ShadowCharacter (
Speed: 2
Walk Speed: 2
Run Speed: 5
Can Run: ✓
Input: MyGameManager (I
Jump Force: 1750
Is Grounded: ☐

Run the game again. Ahh...much better. Jump around a bit more to get a feel for your game.

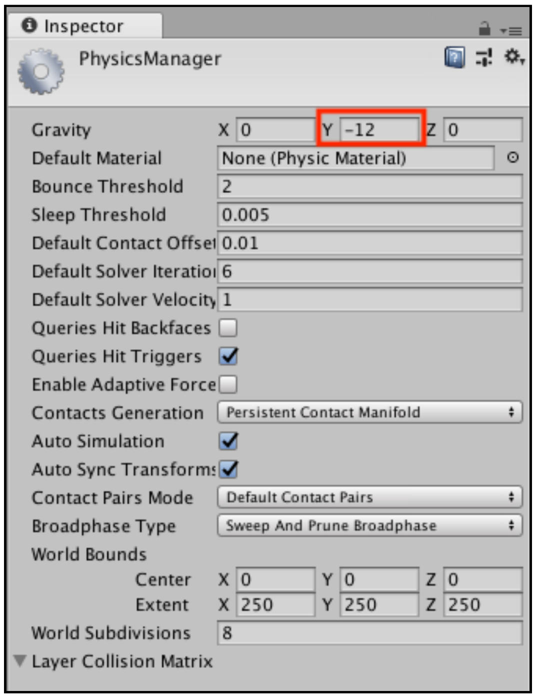


Does it seem a little "floaty" to you? He stays airborne for longer than most people would consider normal, unless they're Martian. The issue isn't that the hero can defy logic, rather it's your gravity settings.

In the top menu, select **Edit** \ **Project Settings** \ **Physics** to open 3D physics settings in the Inspector.



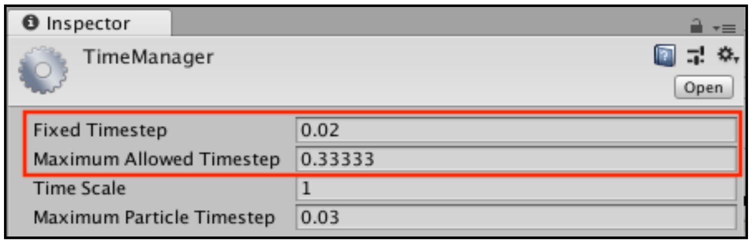
Change **Y Gravity** to –12 to bring the hero back down a little faster.



PhysicsManager

Gravity	X 0 Y -12 Z 0
Default Material	None (Physic Material)
Bounce Threshold	2
Sleep Threshold	0.005
Default Contact Offset	0.01
Default Solver Iterations	6
Default Solver Velocity	1
Queries Hit Backfaces	☐
Queries Hit Triggers	✓
Enable Adaptive Force	☐
Contacts Generation	Persistent Contact Manifold
Auto Simulation	✓
Auto Sync Transforms	✓
Contact Pairs Mode	Default Contact Pairs
Broadphase Type	Sweep And Prune Broadphase
World Bounds	
Center	X 0 Y 0 Z 0
Extent	X 250 Y 250 Z 250
World Subdivisions	8
Layer Collision Matrix	

Next, select **Edit** \ **Project Settings** \ **Time** from the top menu to open the TimeManager settings. Set the **Fixed Timestep** to `0.02` and **Maximum Allowed Timestep** to `0.33333`.



TimeManager

Fixed Timestep	0.02
Maximum Allowed Timestep	0.33333
Time Scale	1
Maximum Particle Timestep	0.03

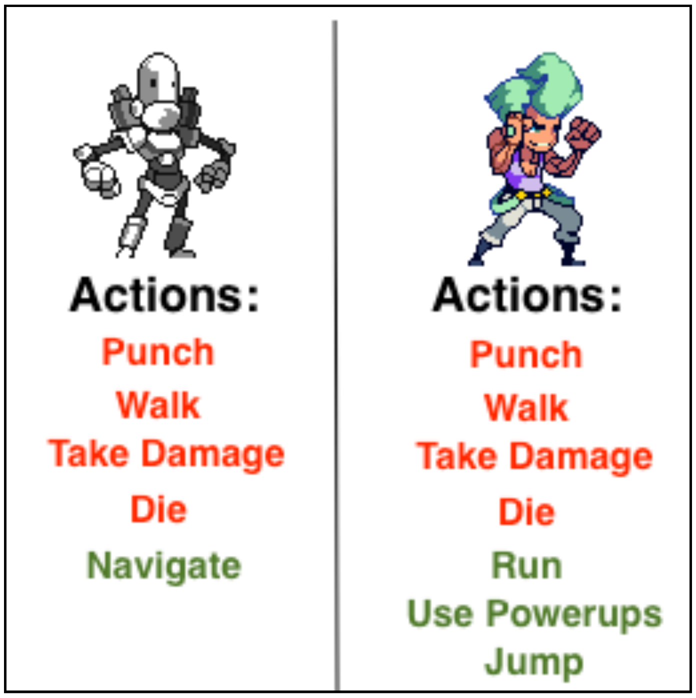
Test it out again. Sure, his jumps are little impressive but far more believable.

Great job! The hero can run and jump his way around the map! You're at the halfway mark for this chapter. Save your work, get up, shake off the cobwebs and get ready to add punching to the hero's repertoire of movement.

A little cleanup

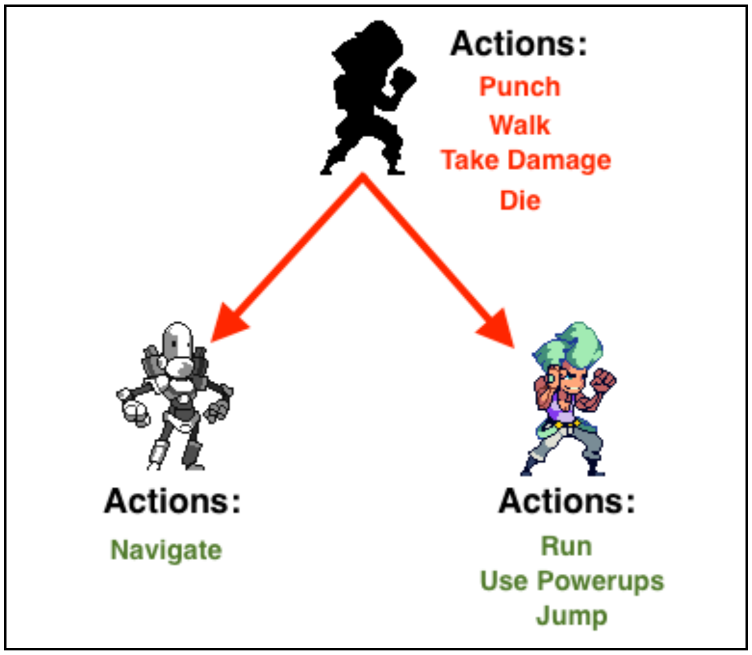
Before you add punches, take a look at the **Hero** script. It's growing quickly! At this rate, it's going to be massive. You should **refactor**, aka restructure, to keep functionality redundancy to a minimum.

Classes are the basic building block of **Object Oriented Programming** (OOP). A class can be based on another class, much like parenting works for GameObjects. A child class can **Inherit** the behavior of its parent without needing its own classes to tell it what to do. Consider the following image:



The protagonist and enemy share several actions: walk, punch, take damage and die. The actions listed in green are not shared: jumping, running and reacting to player controls.

Now is the perfect time to set up a **superclass** from which the Hero and Robot scripts will inherit. You'll call it **Actor** and use it to implement shared methods.



A superclass allows you to retain the functionality of various classes while reducing the amount of code needed to support the game. Less code also means lower maintenance effort. Note that the actor is not necessarily a hero or a robot, but both are always actors.

Create a **new C# script** in the **Scripts** folder, name it **Actor** and open it. Remove the default `Start()` and `Update()` methods.

Now open **Hero.cs** — you're going to switch between these two scripts several times in the next couple of pages. Change the class declaration in **Hero** to:

```
public class Hero : Actor {
```

This makes the `Hero` class a derived class of `Actor`. All public and protected methods and properties of `Actor` will be accessible to the `Hero` class.

Copy the following properties from `Hero` to `Actor` and then delete them from `Hero`:

```
public Animator baseAnim;
public Rigidbody body;
public SpriteRenderer shadowSprite;

public float speed = 2;
protected Vector3 frontVector;

public bool isGrounded;
```

Make sure they're safely in the `Actor` script before you delete them from the **Hero** script.

You'll need to migrate some methods from the `Hero` script to the `Actor` script as well: `OnCollisionEnter`, `OnCollisionExit`, `DidLand` and `FlipSprite`. You'll also need to change the access modifiers for a few methods.

Add the following code inside the **Actor** script:

```
public virtual void Update() {
  Vector3 shadowSpritePosition = shadowSprite.transform.position;
  shadowSpritePosition.y = 0;
  shadowSprite.transform.position = shadowSpritePosition;
}
```

The `virtual` keyword, when used in a method declaration, allows any derived class to override said method. More on this later.

Next, change the `Update()` method declaration in the **Hero** script to the following:

```
//1
public override void Update() {
```

Make this the first line of the `Update()` method:

```
//2
base.Update();
```

Remove these lines from the end of the `Update()` method:

```
//3
Vector3 shadowSpritePosition = shadowSprite.transform.position;
shadowSpritePosition.y = 0;
shadowSprite.transform.position = shadowSpritePosition;
```

Overriding is used when you want a derived class to change the behavior of a parent class' method. In the above case, the **Hero** script overrides and adds its own functionality to the actor's `Update` method. The `Actor` class is now responsible for updating the shadow's position, while the derived class `Hero` now handles all the unique hero-ey actions.

1. The `Update()` method of the **Hero** script now overrides its parent's `Update()` so that an instance of the hero will not use `Update()` from `Actor`.
2. By adding `base.Update();` inside of the Hero's `Update()`, you invoke `Update()` from `Actor`, which lets you update the protagonist's shadow. The keyword `base` refers to the **superclass**, which is the `Actor` class.
3. Since `base.Update()` now handles the shadow's position, you removed this duplicate functionality from the `Hero` class.

Next, add the following methods to the `Actor` class:

```
protected  virtual void OnCollisionEnter(Collision collision) {
  if (collision.collider.name == "Floor") {
    isGrounded = true;
    baseAnim.SetBool("IsGrounded", isGrounded);
    DidLand();
  }
}

protected virtual void OnCollisionExit(Collision collision) {
  if (collision.collider.name == "Floor") {
    isGrounded = false;
    baseAnim.SetBool("IsGrounded", isGrounded);
  }
}

protected virtual void DidLand()
{
}
```

Look familiar? Save for the `virtual` keyword, these are methods you created for the **Hero** script. To refresh your memory, these methods help with detecting and animating ground collision, which is a shared functionality between the hero and all other actors.

Next, open **Hero.cs** and remove the following methods: `OnCollisionEnter`, `OnCollisionExit`.

Still in the same script, replace `DidLand` with the following:

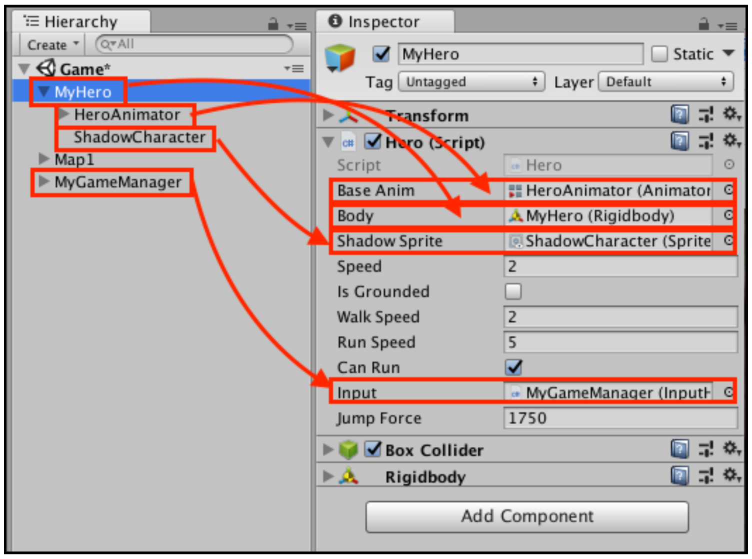
```
protected override void DidLand()
{
  base.DidLand();
  Walk();
}
```

Almost there! Migrate the `FlipSprite` method from the **Hero** script to the **Actor** script, shown below for reference:

```
public void FlipSprite(bool isFacingLeft) {
  if (isFacingLeft) {
    frontVector = new Vector3(-1, 0, 0);
    transform.localScale = new Vector3(-1, 1, 1);
  } else {
    frontVector = new Vector3(1, 0, 0);
    transform.localScale = new Vector3(1, 1, 1);
  }
}
```

You simply moved this method from the `Hero` script to the `Actor` script. Nothing new here.

Save the scripts and return to Unity to check if the references to other components are still correct against the below image:



MyHero	☐ Static
Tag	Untagged
Layer	Default
Hero (Script)	
Script	Hero
Base Anim	HeroAnimator (Animator
Body	MyHero (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite
Speed	2
Is Grounded	☐
Walk Speed	2
Run Speed	5
Can Run	✓
Input	MyGameManager (InputH
Jump Force	1750

If you see duplicate properties in the `Inspector`, then it means you forgot to remove them from the `Hero` script. Go back to the migration steps shown above.

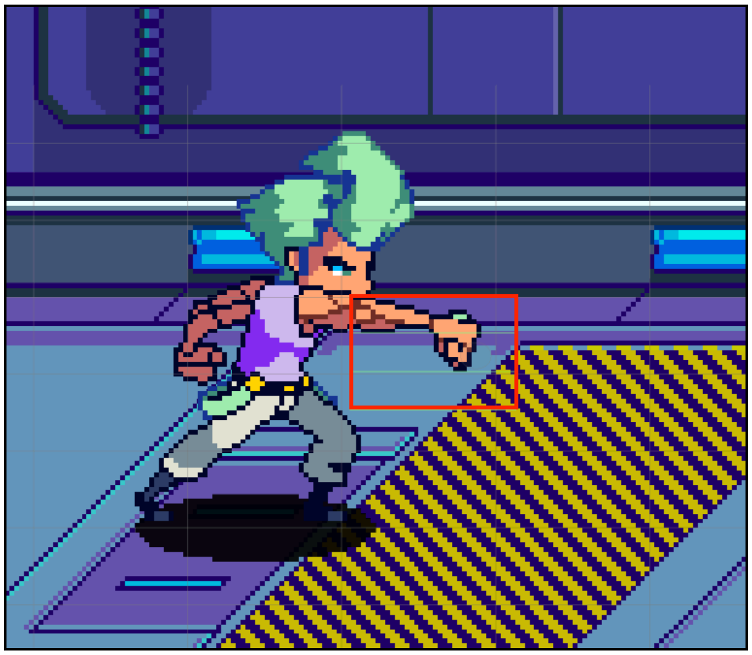
Save both scripts and play your game. It should work exactly like it did before refactoring. If not, retrace some steps or reference the final version of the project for this chapter to see what went wrong before you continue.

Taking a jab at it

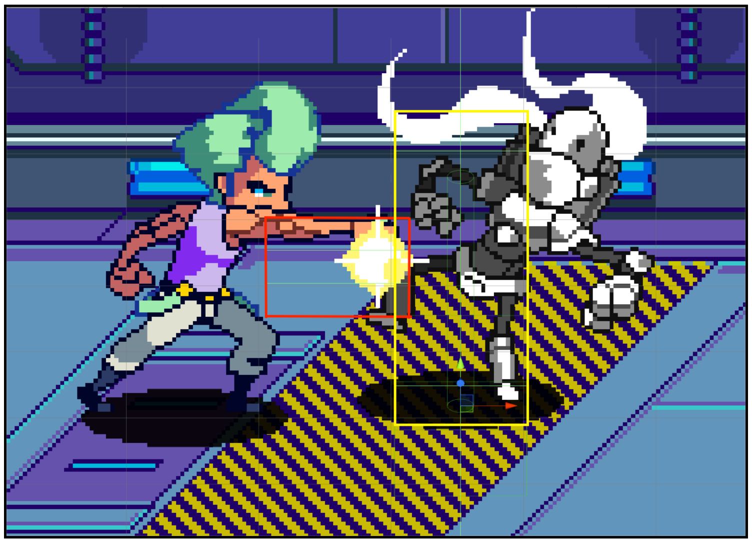
The first order of business before adding any punches is to set up collision detectors so that your game can tell if the hero is just shadow boxing or delivering some poor droid a punishing blow. You’ll use **box collision** to allow the game to detect these collisions between the hero and enemies. In the case of this game, there are boxes around the enemy and the hero’s punch. When they come into contact, you get a collision.

Note: Fancy mesh collisions are more accurate but can slow down the game and require more effort to set up. And with a simple game like this, box colliders are a logical tradeoff.

The hero's punch box is shown in red below:



If it finds a hit, the game registers it and applies damage to the object that sustained the hit. In the image below, the hero's punch box overlaps the robot's hit box (yellow), which triggers a small explosion.

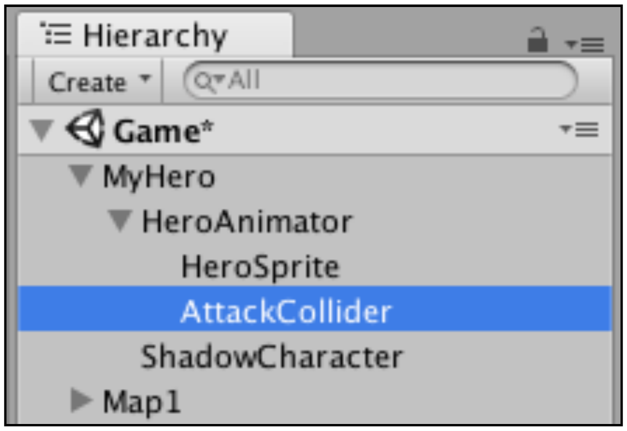


You'll make an animation, box collider and a script to set up the punch.

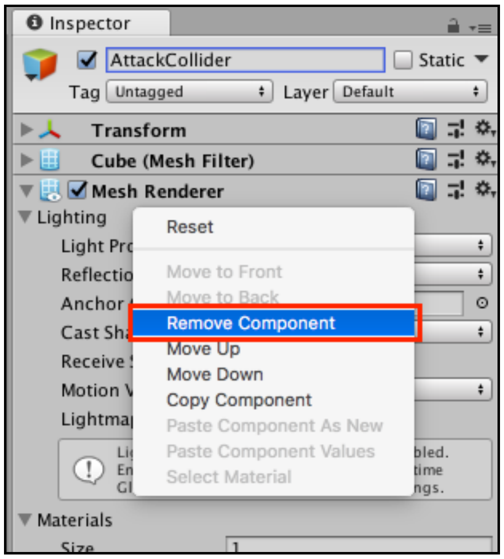
Animating the punch

First, import the **Hero Punch Animation.unitypackage** from the **Unity Packages** folder, which contains the **hero_attack_anim** animation clip along with the sprites.

Add a new **3D Cube GameObject**, rename it **AttackCollider**, set **HeroAnimator** as its parent and reset its **Transform**.

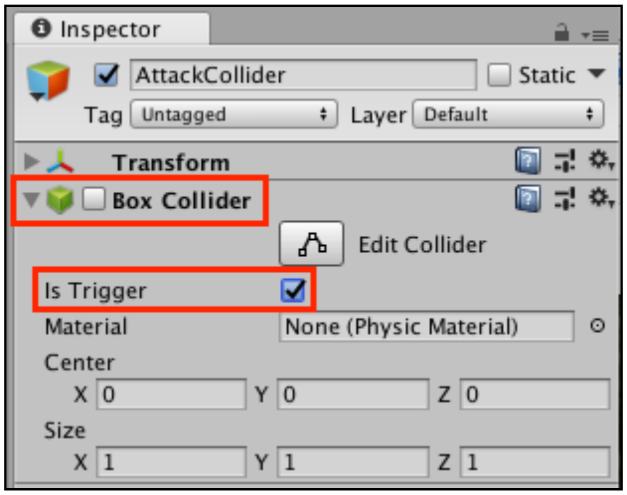


Remove the **Mesh Renderer** component from the **AttackCollider** by right-clicking the **Mesh Renderer** label and selecting **Remove Component** — the box collider is the only component necessary to determine collisions with enemies. Do the same for the **Mesh Filter** component.



In the **Box Collider** component of the **AttackCollider**, check **IsTrigger**. This will convert the box collider to a **Trigger Collider**; it detects and allows other objects to pass through, unlike a rigid collider.

Uncheck the **Box Collider** component to disable it. You'll set up the attack animation to control the collider's enabled state, because the game only needs to check collisions whenever the hero is punching!

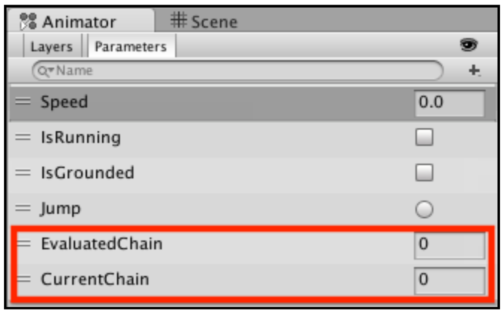


Is Trigger ✓
Material None (Physic Material)
Center X 0 Y 0 Z 0
Size X 1 Y 1 Z 1

Setting up the animation

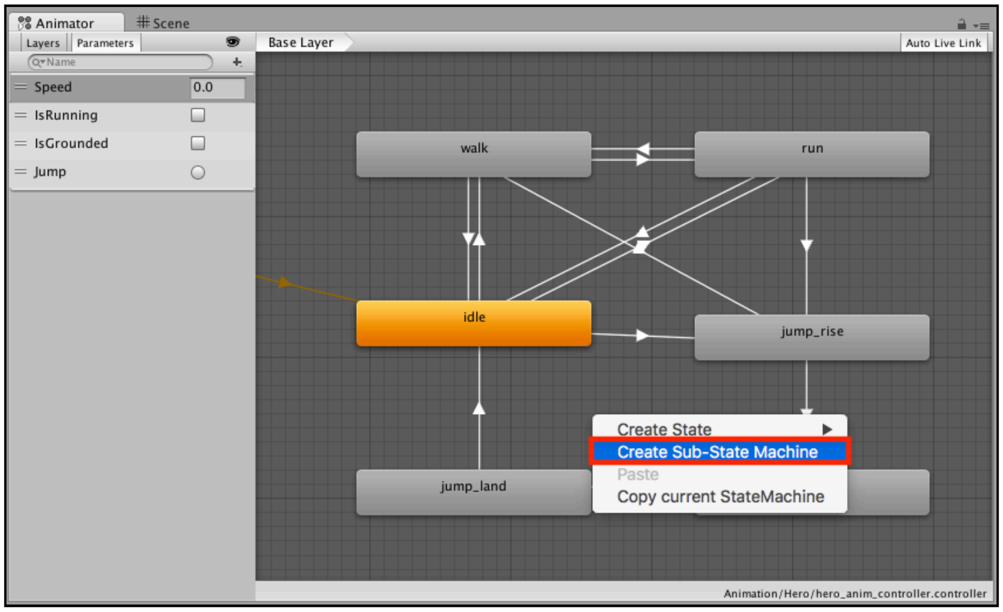
Select the **hero_anim_controller** from the **Animation\Hero** folder.

In the **Animator** window, add two **int event parameters**: **EvaluatedChain** and **CurrentChain**. You'll use these to determine if an attack animation has played. Additionally, since you're using two parameters to store attack states, the game will allow the hero to "queue" attacks, which will be fun for the player and useful for you when you implement **attack chaining**.

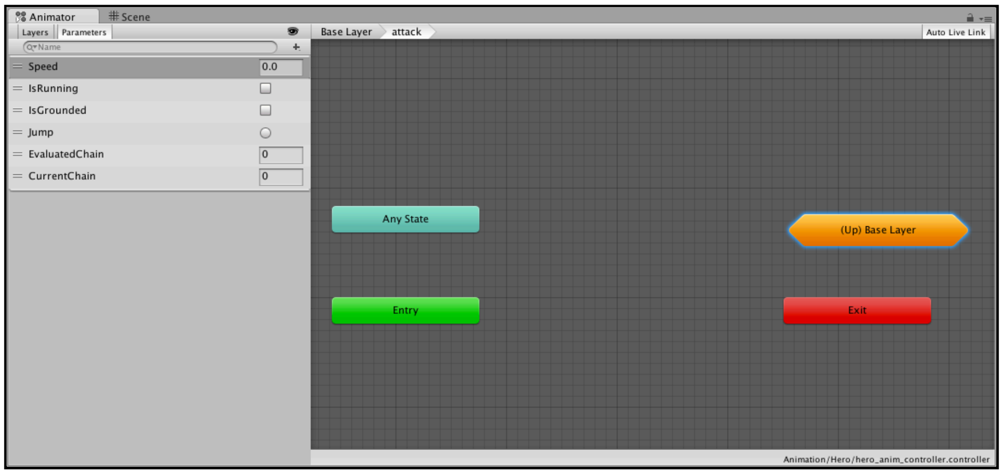


Speed	0.0
IsRunning	☐
IsGrounded	☐
Jump	○
EvaluatedChain	0
CurrentChain	0

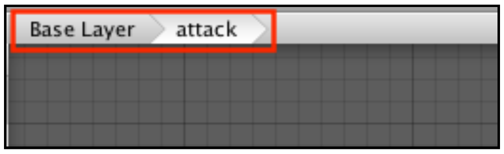
You need to add the **hero_attack1_anim** state to the state machine. Select **hero_anim_controller**, and in the Animator window, right-click and select **Create a Sub-State Machine**. This is how you add a new state machine inside of your base state machine.



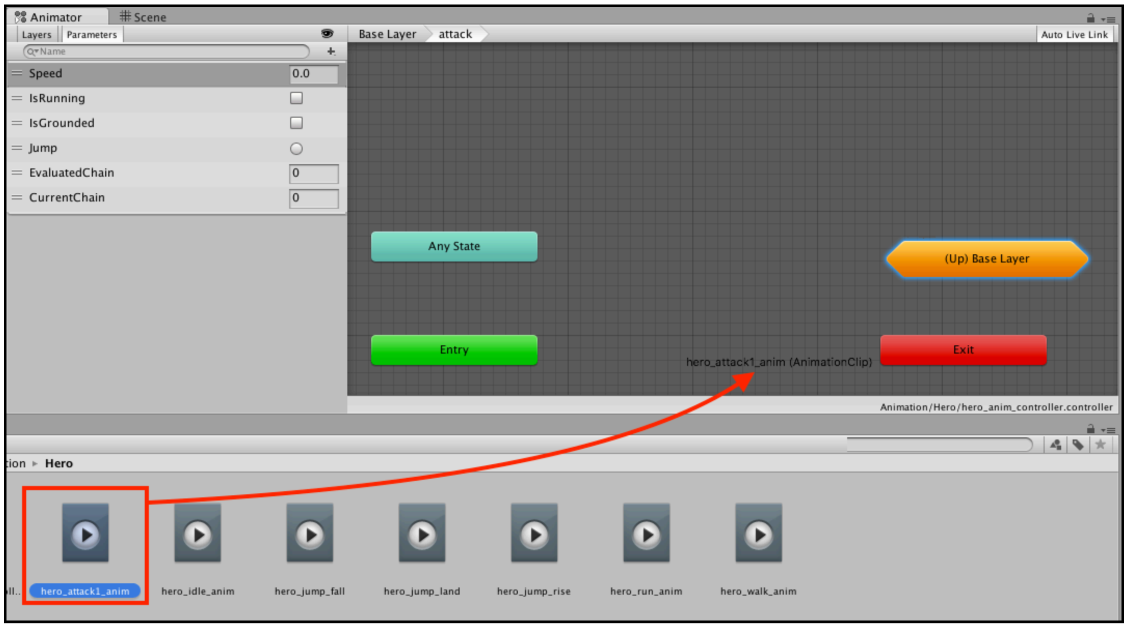
Rename it **attack**, then double-click the **attack state** to view what is inside.



This new sub-state machine is a complete state machine with its own entry and exit methods. A key benefit is that sub-state machines make it easier to manage an animator. Note that the top bar shows when you're inside a sub-state machine, and it lets you navigate back to the parent state machine, **Base Layer**.

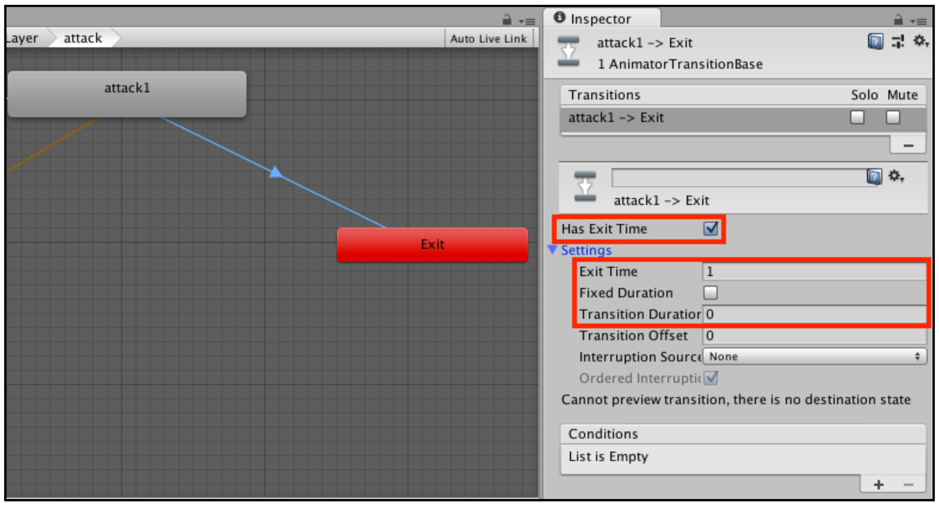


While you're inside the **attack** state machine, drag the **hero_attack1_anim** animation to the grid to create a new attack state then name it **attack1**.



Speed	0.0
IsRunning	☐
IsGrounded	☐
Jump	○
EvaluatedChain	0
CurrentChain	0

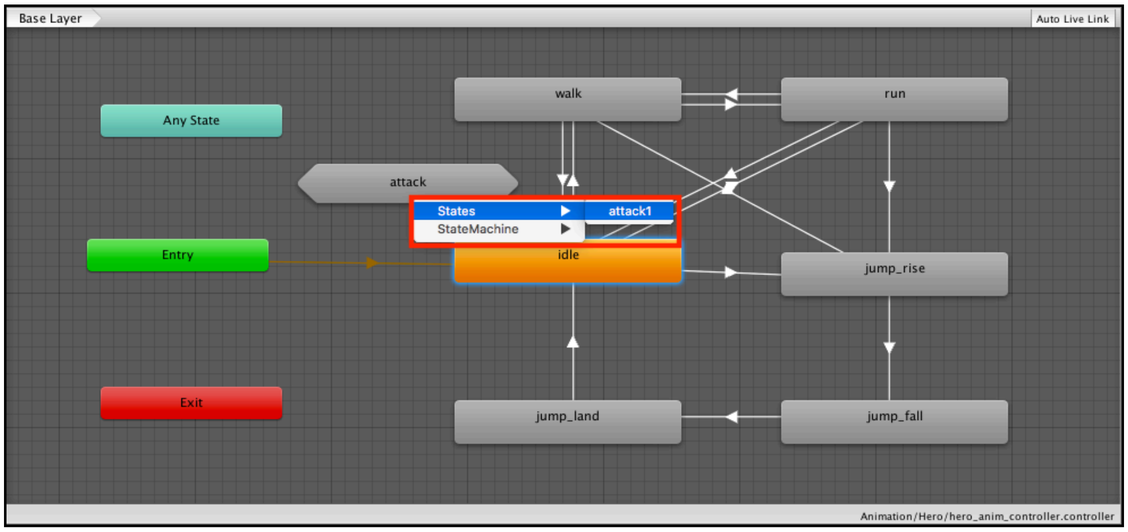
Add a transition from the **attack1** state to **Exit**. Set **Exit Time** to 1, uncheck **Fixed Duration** and set **Transition Duration** to `0`. When attack1 finishes playing the punch animation, the animator will exit the sub-state machine and proceed as normal.



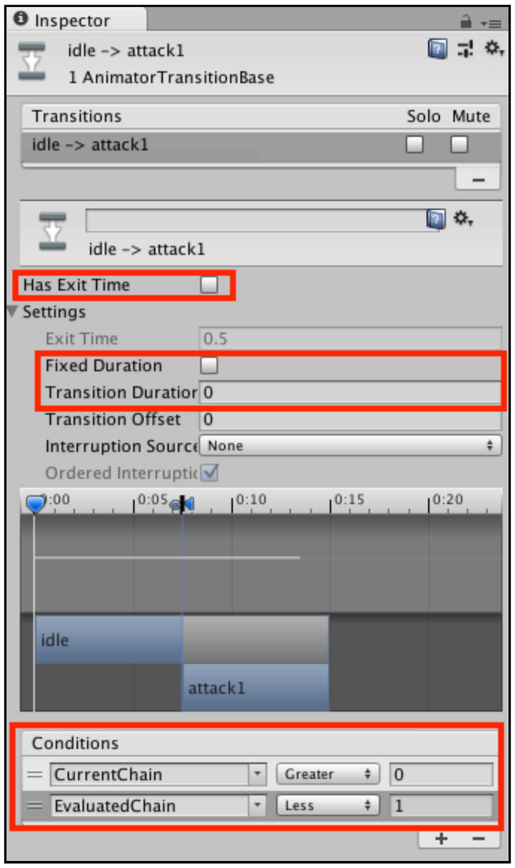
Has Exit Time	✓
Exit Time	1
Fixed Duration	☐
Transition Duration	0
Transition Offset	0
Interruption Source	None
Ordered Interruption	✓

Cannot preview transition, there is no destination state

Return to the **Base Layer** by clicking in the navigation at the top-left corner of the **Animator** window. Add a **new transition** from **idle** to the **attack** sub-state — you'll need to specify the transition is for the **attack1** state. (**States\attack1**)



Select that new transition then uncheck **Has Exit Time** and **Fixed Duration**. Set **Transition Duration** to `0`, and add two conditions: **CurrentChain** must be **Greater** than `0`, and **EvaluatedChain** must be **Less** than `1`.


Inspector

idle -> attack1

1 AnimatorTransitionBase

Transitions	Solo	Mute
idle -> attack1	☐	☐

idle -> attack1

Has Exit Time ☐

Settings

Exit Time 0.5

Fixed Duration ☐

Transition Duration 0

Transition Offset 0

Interruption Source None

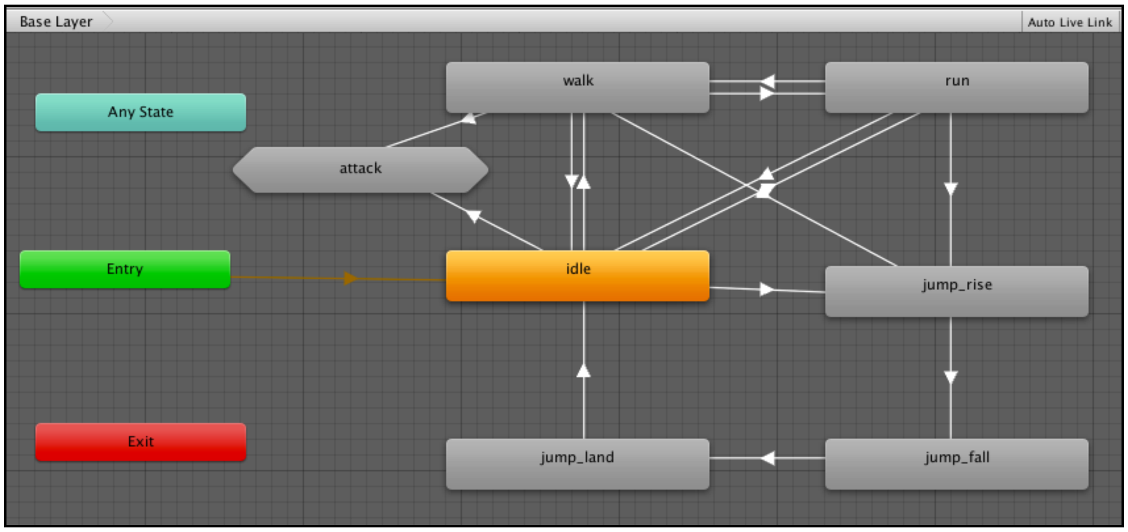
Ordered Interruption ✓

Conditions

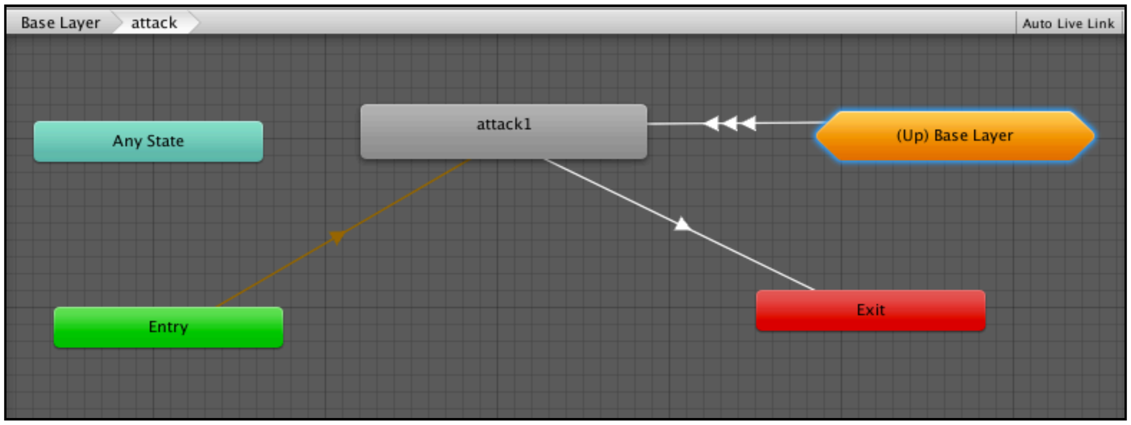
CurrentChain	Greater	0
EvaluatedChain	Less	1

Add a **transition** from **walk** to **attack/attack1** with the same settings and conditions.

Your state machine should look like this when done:



The attack sub-state machine should look like this:



You can now transition to attack from either the idle or walk state. You're all finished creating the animation state for punching!

Coding the punch

Before editing the `Hero` script, you need to include the attack in the `InputHandler` class. These additions are minor and simply pass the input from when the player presses the attack button to the **InputHandler**.

Open the **InputHandler** script, and add the following variable to the top of the class:

```
bool attack;
```

Add the following method beneath the other ones:

```
public bool GetAttackButtonDown() {
  return attack;
}
```

In the `Update()` method, find `vertical = Input.GetAxis("Vertical");` and add the following right below it:

```
attack = Input.GetButtonDown("Attack");
```

That's it for the **InputHandler** script. **Save** and close it. You'll modify the **Actor** and **Hero** scripts next, so open up the **Actor** script and add this method:

```
public virtual void Attack() {
  baseAnim.SetTrigger("Attack");
}
```

Here you've made the generic attack trigger for all `Actor` derived classes. Unless it's overridden, all derived classes will trigger an `Attack` event on the `baseAnim`.

Open the **Hero** script and add the following properties near the top:

```
bool isAttackingAnim;
float lastAttackTime;
float attackLimit = 0.14f;
```

These variables detect when the hero is attacking. You also set an `attackLimit` to limit the player's ability to set up excessive attack queues.

In the `Update()` method, at the top and right below the `base.Update()` line, add the following code;

```
isAttackingAnim =
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1");
```

This detects when the hero is in attack state.

Still in `Update()`, add this right below `bool jump = input.GetJumpButtonDown();`:

```
bool attack = input.GetAttackButtonDown();
```

Here you have your attack input code along with the other input commands.

In `Update()`, wrap the whole movement code block with the `if(!isAttackingAnim)` condition. It should look like this when you're done.

```
if (!isAttackingAnim) {
  if ((v == 0 && h == 0)) {
    Stop ();
```

```
      isMoving = false;
    } else if (!isMoving && (v != 0 || h != 0)) {
      isMoving = true;
      float dotProduct = Vector3.Dot (currentDir, lastWalkVector);
      if (canRun && Time.time < lastWalk + tapAgainToRunTime && dotProduct
  > 0) {
        Run ();
      } else {
        Walk ();
        if (h != 0) {
          lastWalkVector = currentDir;
          lastWalk = Time.time;
        }
      }
    }
  }
```

The `if(!isAttackingAnim)` condition limits the hero's movement when the attack animation is playing. You don't want the hero to move when he's punching — trust me when I say that it just doesn't look smooth in this game! Additionally, the walk animation shouldn't play simultaneously with the attack animation, and it won't now. Our pompadoured protagonist would appear to slide around the map while punching. You can imagine how amateurish that would look to the player!

You need to modify the jump condition too. In the `Update` method, add an `!isAttackingAnim` condition check, as shown below:

```
  if (jump && !isJumpLandAnim && !isAttackingAnim &&
    (isGrounded || (isJumpingAnim && Time.time < lastJumpTime +
  jumpDuration)) ) {
    Jump(currentDir);
  }
```

Add the following lines after the jump code block at the end of the `Update()` method.

```
  if (attack && Time.time >= lastAttackTime + attackLimit) {
    lastAttackTime = Time.time;
    Attack();
  }
```

This is *the* block you've been waiting for. It detects the attack button input and tells the hero to throw that punch!

In the `FixedUpdate()` method, add the `!isAttackingAnim` condition to the `isGrounded` condition as shown below:

```
  if (isGrounded && !isAttackingAnim) {
    body.MovePosition (transform.position + moveVector *
  Time.fixedDeltaTime);
    baseAnim.SetFloat ("Speed", moveVector.magnitude);
  }
```

This modifies the physics movement condition, disabling the hero's movement whenever he attacks.

Finally, add the `Attack()` method to the `Hero` script:

```
public override void Attack() {
  baseAnim.SetInteger("EvaluatedChain", 0);
  baseAnim.SetInteger("CurrentChain", 1);
}
```

This method overrides the `Attack` method from the `Actor`, allowing the hero to perform a different punch logic whenever `Attack` is called. For a simple punch, the hero just sets the integer values of the two `int` animator parameters, `EvaluatedChain` and `CurrentChain`, to `0` and `1` respectively. The animator then transitions the state machine to the attack state.

Save all scripts, return to the editor and run the game. Press the **A** key on your keyboard. Now there's a guy who doesn't know when to let something go!

He's locked in an eternal cycle of violence because there is no condition that stops punching. Specifically, the **Hero** script doesn't know if the hero actually threw a punch!

To end the perpetual punching, you'll employ **Animation Events**, a feature of Mecanim.

Animation Events allow you to trigger any public method of any component or script that's attached to the same GameObject as the Animator to which it is attached. It's an especially helpful feature for specifying when certain events should happen in the script.

In the current scenario, **Hero** needs to know that **attack1** played. Your animation event will calculate the `EvaluatedChain` value of the `Hero` class.

However, the hero script is not attached to the same GameObject as its animator so the animation events won't reach the `Hero` script. To work around that, you need a class to trigger the same behavior towards the hero script.

Create a new **C# script** in the **Assets/Scripts** folder and name it **HeroCallback**. This script will forward animator callback methods to the `Hero` class. Open it up.

Replace the contents of `HeroCallback` with the following:

```
using UnityEngine;

public class HeroCallback : MonoBehaviour {
  //1
  public Hero hero;
```

```
    //2
    public void DidChain(int chain) {
      hero.DidChain(chain);
    }
  }
```

The new `HeroCallback` class does the following:

1. Creates a reference to the hero where the Hero's `DidChain()` method will be called.
2. When called, this calls the `DidChain(int chain)` method to the hero property, which notifies the **Hero** that the punch animation is complete — this is the actual method that's accessible to the animation event.

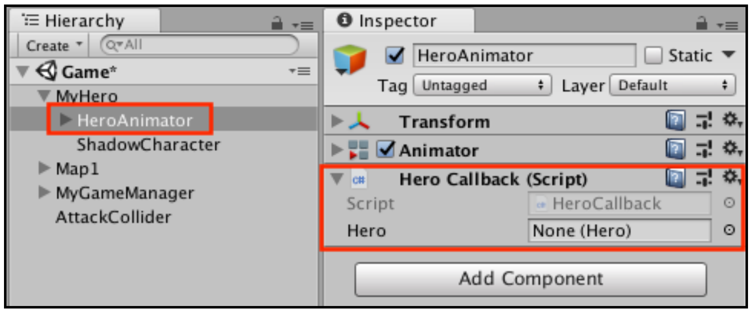
> **Note:** This will cause an error because `DidChain` isn't defined yet in the **Hero** class. Ignore it.

Open the **Hero** script and add the following method:

```
  public void DidChain(int chain) {
    baseAnim.SetInteger("EvaluatedChain", 1);
  }
```

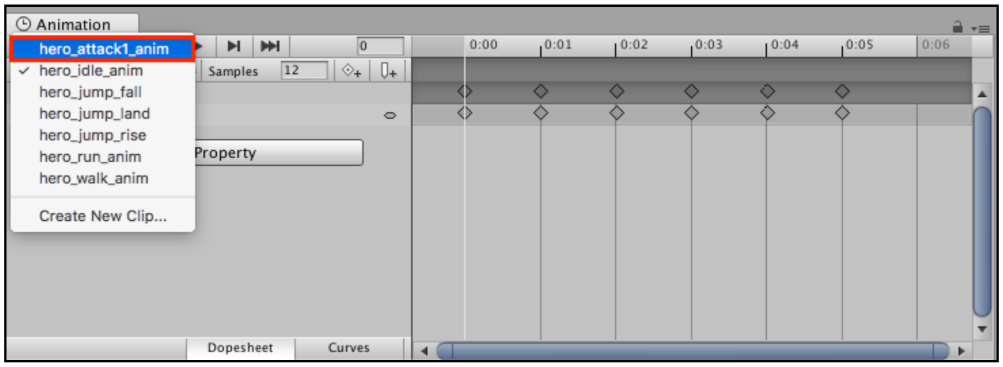
`DidChain` updates the `EvaluatedChain` value of the animator to `1`, meaning that it has evaluated the first attack.

Save your scripts, return to Unity and add the **HeroCallback** component to **HeroAnimator**.

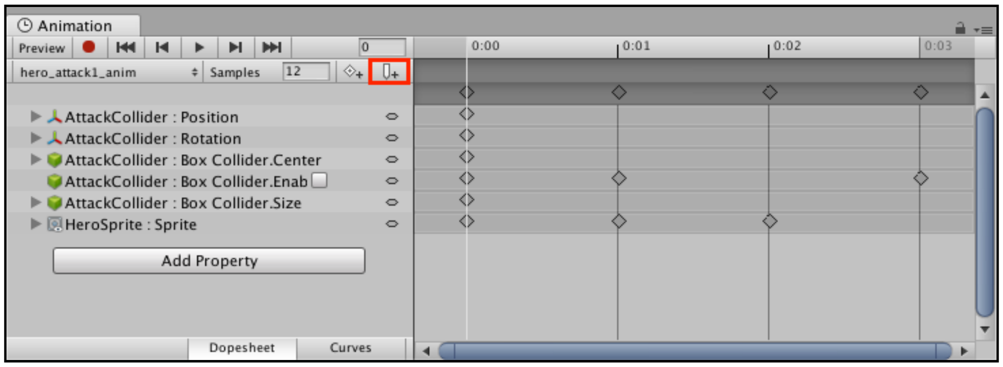


HeroAnimator
Tag Untagged
Layer Default
Hero Callback (Script)
Script HeroCallback
Hero None (Hero)

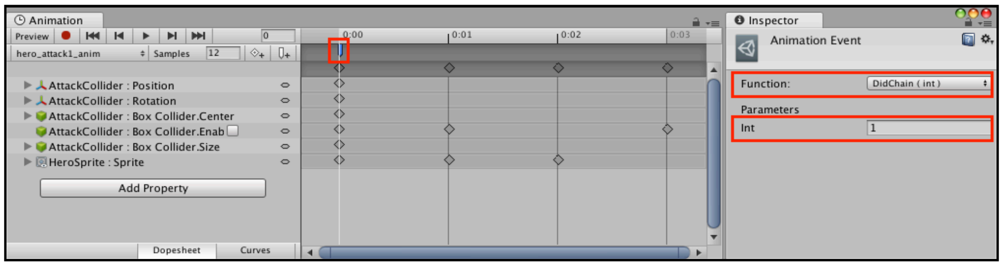
You also need an animation event in the hero_attack1_anim clip. Select the **HeroAnimator**, open the **Animation** window and change the current animation clip to **hero_attack1_anim**.



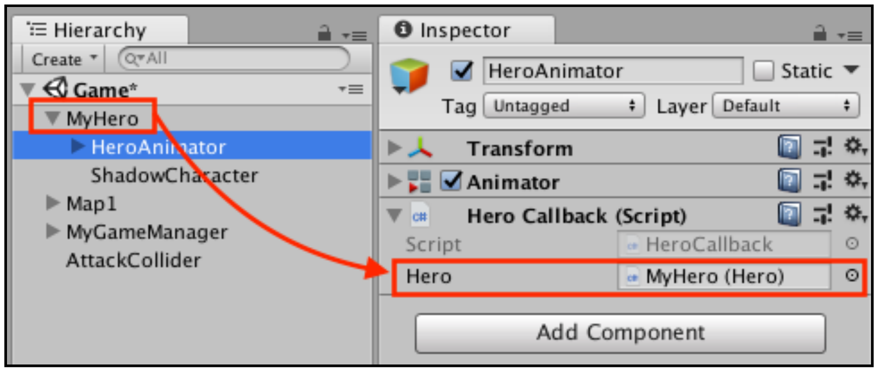
Select the first frame and click the **new animation event** button to add a new animation event marker at the first frame of the animation.


Animation
Preview
hero_attack1_anim
Samples 12
AttackCollider : Position
AttackCollider : Rotation
AttackCollider : Box Collider.Center
AttackCollider : Box Collider.Enab
AttackCollider : Box Collider.Size
HeroSprite : Sprite
Add Property
Dopesheet
Curves

Select the animation event and in the Inspector, change **Function** to `DidChain(int)` and set the **int parameter** value to 1.



Drag **MyHero** to the **Hero** property of the **HeroCallback**.



Now press run and watch him take a quick jab! Take a moment to watch the punch animation and how it activates the punch box too. Neat, isn't it?

Where to go from here?

Wow! Look at you. You started with a character that could only walk around and ended with a lean, mean punching machine. And never will you see that perfectly coifed hairdo fall out of place, no matter how hard you make him punch.

This chapter, you've managed to:

- Make the hero **run**
- Make the hero **jump**
- Create an **Actor** super-class and cleaned the **Hero** class
- Make the hero **punch**!

Along the way, you got a lot of experience working with transitions and animation states. You also set up a couple of input options and an Input Handler so that the player has more control over the jump, and therefore more fun!

You did some refactoring so that your end product is cleaner, lighter and easier to maintain. You also set up collisions so that the hero's punch animation actually does something besides just look cool, and lastly, you spent a lot of time coding to make everything work together.

Although running, jumping and punching are interesting, without a foe to bash to bits, the game is pretty pointless. In the next chapter, you'll start working on the formidable foes to give the hero a purpose. So get ready to bring on the robots!

Chapter 5: Bring on the Droids

At this point, you've got a Pompadoured hero that's full of life and ready to fight. He's walking, idling, jumping and punching on command. Most importantly, his hair is never out of place.

But you're not ready to release this game yet. You still need enemies, and you need to do something about that punch. At present, it's little more than a powerful looking gesture that doesn't connect. You're building a beat 'em up game, so he should be able to crack some robot skulls, right?

You'll address the hero's lack of power in this chapter. Here's a preview of the road ahead:

- Add a robot to the scene, so the hero has something to punch.
- Apply layers and have an understanding of their effects on physics engine collisions.
- Animate the robot, implement a life counter and knock it over while killing it.
- Use coroutines to create time-sensitive effects.
- Add a special animation to show when the robot takes a punch.

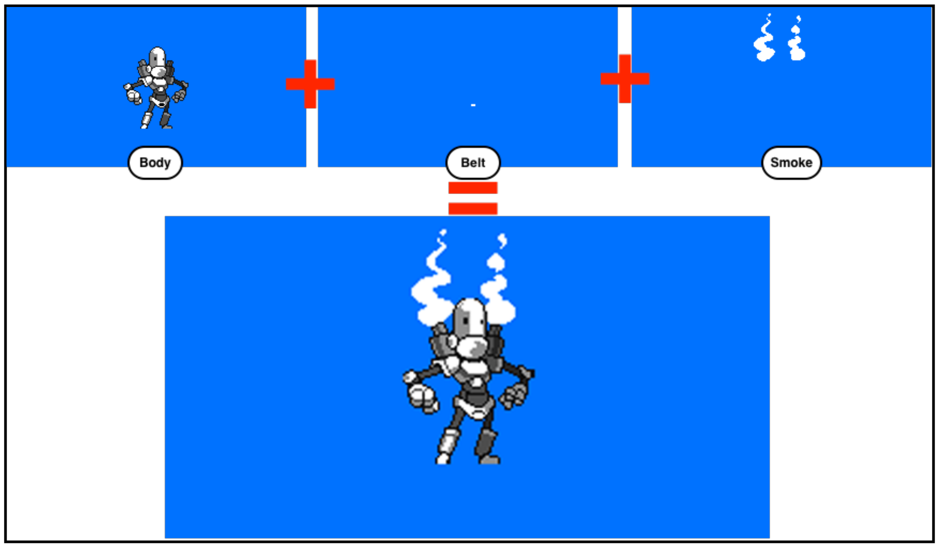
Get ready to lay some smack down!

Robot design 101

First, you'll need to understand a few things about these scrap-metal contraptions. Don't worry — no degree in robot engineering is required!

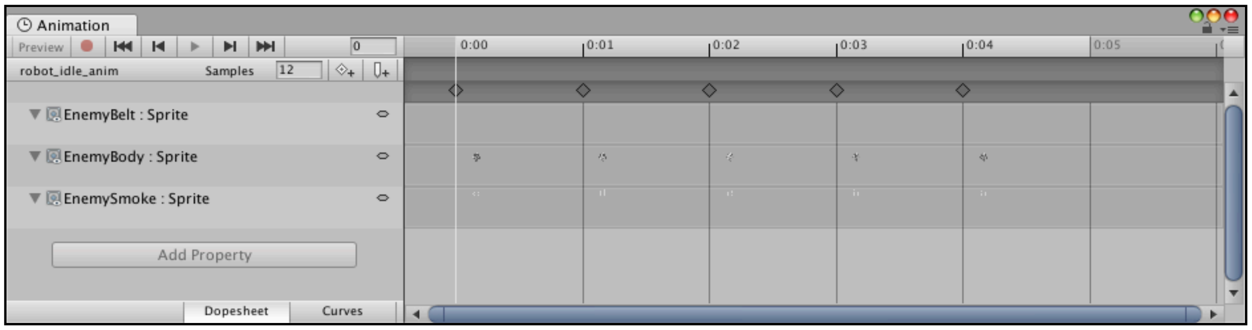
A robot is the basic "bad guy" in Pompadroid. This contraption causes the hero major headache, requiring multiple punches to put it down. As durable as it is, there is nothing remarkable about it. It knows only a few things: punching, chasing, walking and waiting. In this chapter, you'll focus on what happens when it takes a punch and tend to the rest later.

These robots are composed of three overlayed sprites: a body, smoke plume and a belt sprite, as seen below:



You're using separate sprites because you'll eventually create different robot classes, as shown below. To accommodate various actions without resizing, the sprites are also larger than the actual drawing they contain.

Robot animation clips will contain multiple sprite renderers. They'll change at the same time to keep the three sprites in sync as they update during the animation.



In this chapter, you'll only work with three clips: the idle animation for when it's standing about, the hurt animation for when it's taking damage, and the knockout animation for when it's dying.

Import **Robot Animations.unitypackage** from the **Unity Packages** folder, which includes the robot animation states:

- **robot_idle_anim**: The robot's idle animation that has five frames for each robot sprite.
- **robot_knockout_anim**: The death animation that also has five frames for each sprite.
- **robot_hurt_anim clip**: The punch reaction animation that has three frames for each sprite.

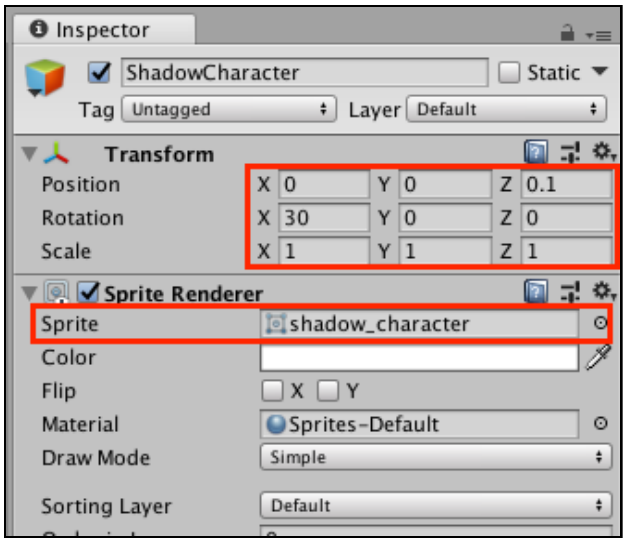
Note the new **Robots** folder in the Assets/Animation folder — this is where you'll store robot animation clips and other assets.

Got everything imported? Good! You're ready to meet these dreadful metal antagonists.

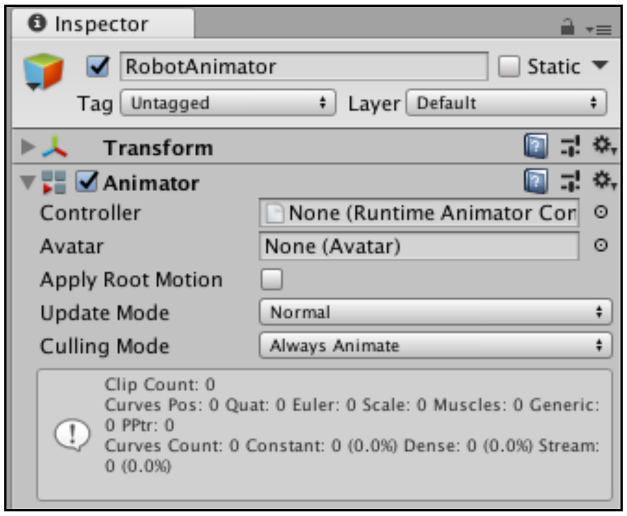
Robot assembly

Create a **new GameObject** named **EnemyRobot** and set its **Position** to `(X:15, Y:0, Z:0)`. This should put the robot right in front of the hero.

Next, add the robot shadow by creating a **new GameObject** named **ShadowCharacter** and make it a **child** of **EnemyRobot**. Set its Transform **Position**, **Rotation** and **Scale** to `(X:0, Y:0, Z:0.1)`, `(X:30, Y:0, Z:0)` and `(X:1, Y:1, Z:1)`, respectively. Add a **SpriteRenderer** to it and set **shadow_character** as its sprite.

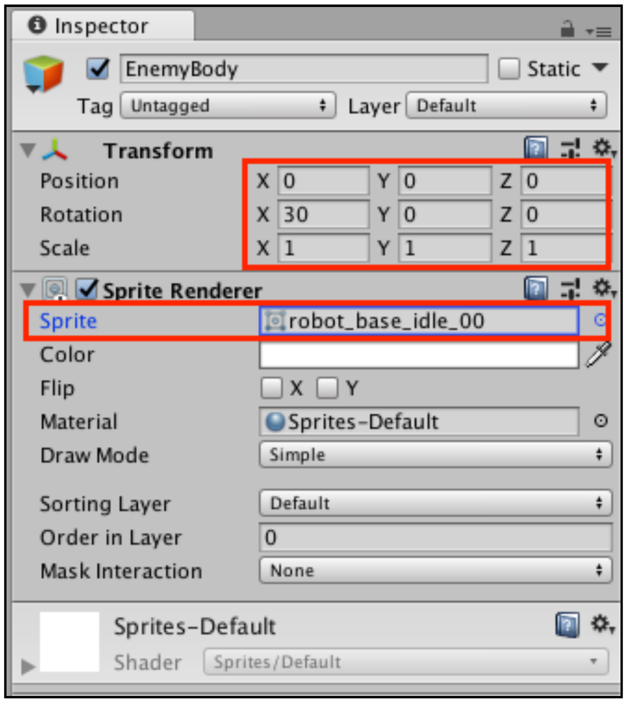

Inspector
ShadowCharacter Static
Tag Untagged Layer Default
Transform
Position X 0 Y 0 Z 0.1
Rotation X 30 Y 0 Z 0
Scale X 1 Y 1 Z 1
Sprite Renderer
Sprite shadow_character
Color
Flip X Y
Material Sprites-Default
Draw Mode Simple
Sorting Layer Default

Create another empty child of the EnemyRobot, and this time name it **RobotAnimator**. **Reset** its Transform and add an **Animator component** to it.


Inspector
RobotAnimator Static
Tag Untagged Layer Default
Transform
Animator
Controller None (Runtime Animator Cor
Avatar None (Avatar)
Apply Root Motion
Update Mode Normal
Culling Mode Always Animate

```
Clip Count: 0
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic:
0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream:
0 (0.0%)
```

Create an **empty GameObject** named **EnemyBody** and make it a child of **RobotAnimator**. **Reset** its Transform and set its **Rotation** to `(X:30, Y:0, Z:0)`. Add a **SpriteRenderer** to it and set **robot_base_idle_00** as its initial sprite value.



Inspector

EnemyBody — Static

Tag: Untagged — Layer: Default

Transform

	X	Y	Z
Position	0	0	0
Rotation	30	0	0
Scale	1	1	1

Sprite Renderer

Sprite	robot_base_idle_00
Color	
Flip	X Y
Material	Sprites-Default
Draw Mode	Simple
Sorting Layer	Default
Order in Layer	0
Mask Interaction	None

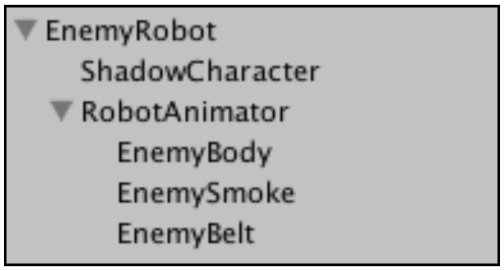
Sprites-Default

Shader: Sprites/Default

Add **two more children** to **RobotAnimator** named **EnemySmoke** and **EnemyBelt**. Use the same settings as with EnemyBody, except set the initial **sprite** values to **robot_smoke_idle_00** and **robot_belt_idle_00**, respectively.

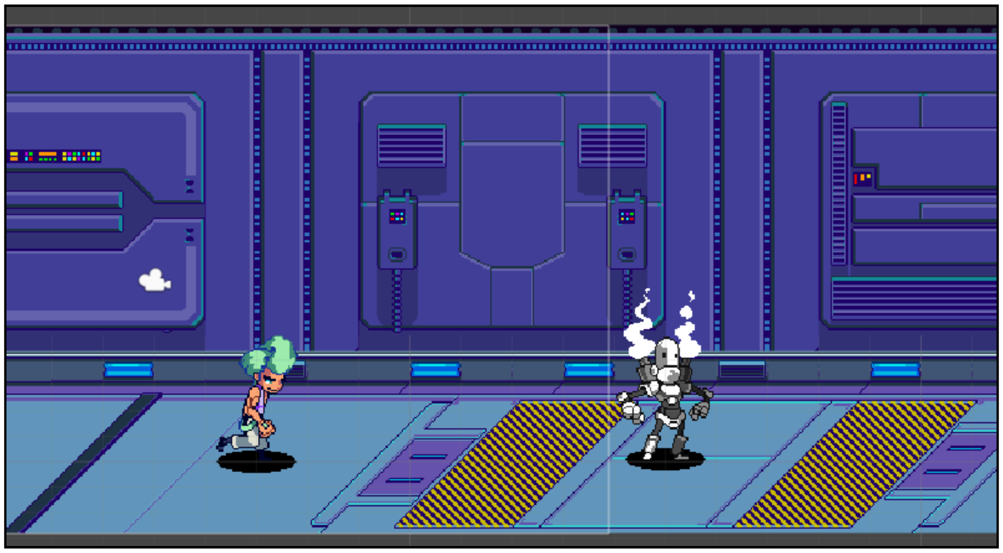
Set the **Position** of EnemyBelt to `(X:0, Y:0.004, Z:-0.006)`.

Your EnemyRobot hierarchy should look like this:



```
▼ EnemyRobot
    ShadowCharacter
  ▼ RobotAnimator
      EnemyBody
      EnemySmoke
      EnemyBelt
```

Check the Scene view to confirm the robot is correctly assembled and standing in front of the hero.



You're done setting up the robot's required GameObjects! It's time to dig into some scripting.

Open the **Actor** script. Add the following variable to the top of the class:

```
public SpriteRenderer baseSprite;
```

This adds a reference to the baseSprite of any instance of the Actor class.

As explained during the refactoring exercise in the last chapter, the robot will inherit from the Actor class, so capabilities available to the hero are also available to the robots.

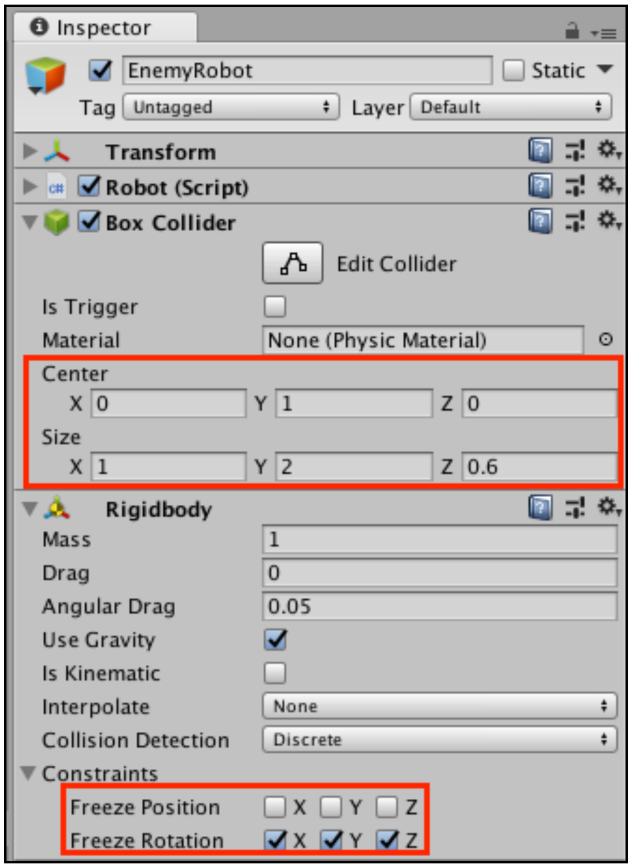
Save the Actor script and return to Unity. Create a **new C# script** in **Assets/Scripts** named **Robot** — this will be the code "heart" of the robot, and ultimately, what differentiates the good guy from a bad guy.

Open the `Robot.cs` and remove the default `Start` and `Update` methods. Set `Actor` as its superclass. It should look like the class below. Note the `public class Robot : Actor` { line:

```
using System.Collections;
using UnityEngine;

public class Robot : Actor {

}
```

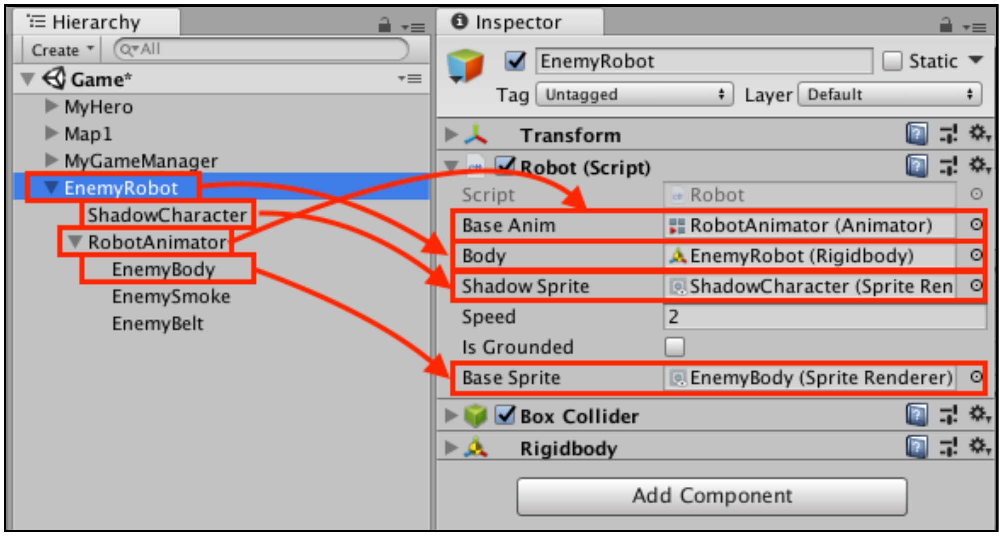
Save the script and return to Unity. For the moment, disregard the fact that the Robot class is empty. You'll come back to it in future chapters to add more.

Add a **Robot**, **Box Collider** and **Rigidbody** component to **EnemyRobot**. For the **Box Collider**, set the **Center** to `(X:0, Y:1, Z:0)` and **Size** to `(X:1, Y:2, Z:0.6)`. In its **RigidBody**, expand **Constraints** and check all the axes under **Freeze Rotation**.



Inspector
EnemyRobot — Static
Tag: Untagged — Layer: Default
Transform
Robot (Script)
Box Collider
Edit Collider
Is Trigger: ☐
Material: None (Physic Material)
Center: X 0, Y 1, Z 0
Size: X 1, Y 2, Z 0.6
Rigidbody
Mass: 1
Drag: 0
Angular Drag: 0.05
Use Gravity: ✓
Is Kinematic: ☐
Interpolate: None
Collision Detection: Discrete
Constraints
Freeze Position: ☐ X ☐ Y ☐ Z
Freeze Rotation: ✓ X ✓ Y ✓ Z

You need to set the property references. Set **RobotAnimator** as the **Base Anim**, **EnemyRobot** as the **Body**, **ShadowCharacter** as the **Shadow Sprite**, and **EnemyBody** as the **Base Sprite**.

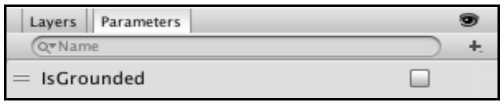


Hierarchy
Game*
MyHero
Map1
MyGameManager
EnemyRobot
ShadowCharacter
RobotAnimator
EnemyBody
EnemySmoke
EnemyBelt

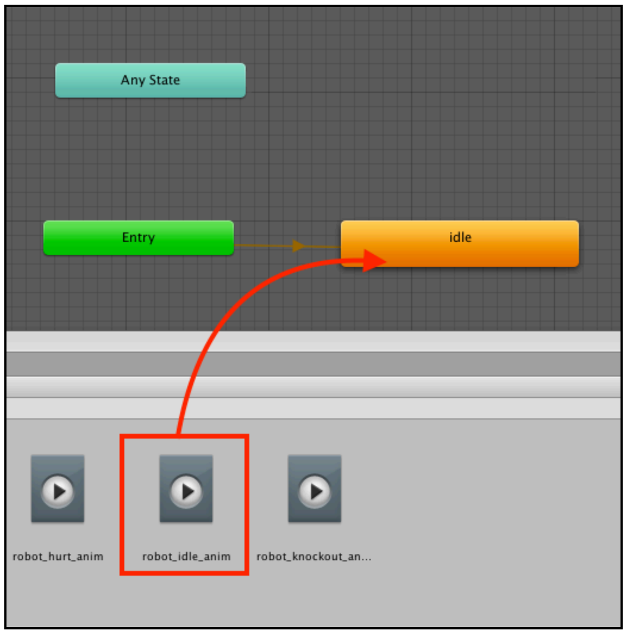
Inspector
EnemyRobot — Static
Tag: Untagged — Layer: Default
Transform
Robot (Script)
Script: Robot
Base Anim: RobotAnimator (Animator)
Body: EnemyRobot (Rigidbody)
Shadow Sprite: ShadowCharacter (Sprite Ren
Speed: 2
Is Grounded: ☐
Base Sprite: EnemyBody (Sprite Renderer)
Box Collider
Rigidbody
Add Component

The animator controller needs to be set. In the **Assets/Animation/Robots** folder, create a new **Animator Controller** named **robot_anim_controller**. Double-click it to open the **Animator** window.

Add an **IsGrounded Boolean** parameter to the animator, which is necessary because the Actor superclass requires IsGrounded to determine if an actor is in the air.



Drag the **robot_idle_anim** clip to the Animator window and **rename** it **idle**. When the robot does absolutely nothing, this animation will play.



Select **RobotAnimator** and assign **robot_anim_controller** to its **Controller** field.

Run the game and look at that smoldering little hunk of metal: That's an animated robot! Try punching. Try harder!!

Just kidding, it doesn't matter how hard you try because you've not set up the logic to make that hit do much. However, the robot doesn't mind if you push it around a bit.

But wait, didn't you put a collider in place to allow for a good, solid punch? You did, but clearly, that wasn't enough. You'll address that later.

There's a more immediate problem. The robot's box collider can interact with the hero's collider, making it possible for the bots to gang up on the hero and box him in. Who knows what they'll do with their powers.

Layers are how you'll correct this suspicious behavior. Broadly speaking, layers set which objects can collide with each other in the physics settings.

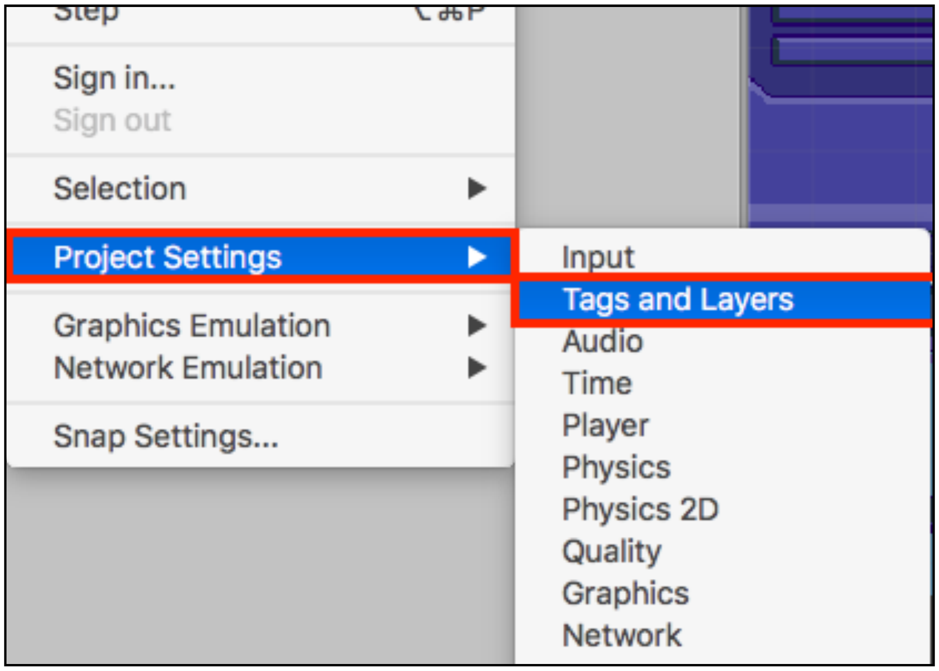
Layers

Layering is how you isolate physics collisions and control exactly what can collide with what. For example, a robot should never harm a fellow robot; it should only be able to damage the hero.

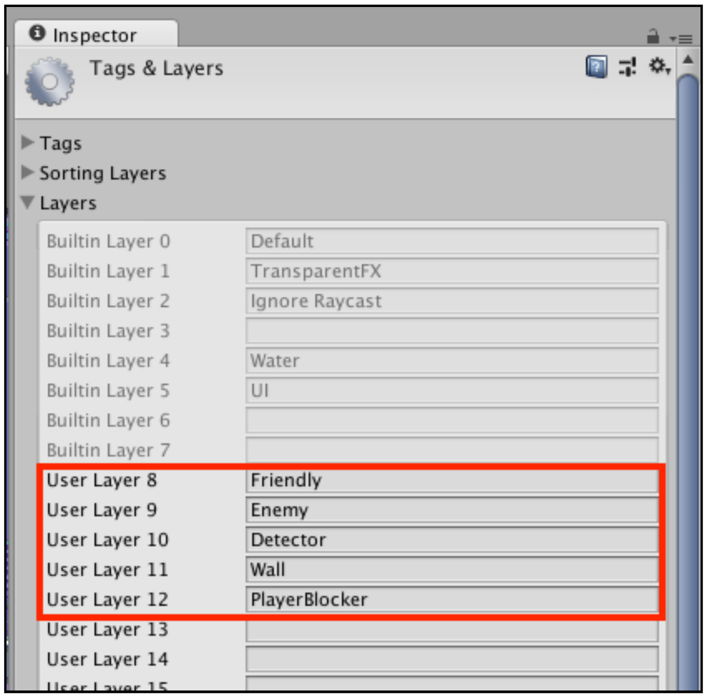
You'll create the following layers:

- **Friendly**: Contains the layers of the hero and his body collider — this layer can't collide with another friendly or an enemy.
- **Enemy**: Contains all the enemy layers ranging from robots to a big baddie boss — enemies can't collide with friendly, enemy or playerblocker layers.
- **Detector**: Contains all the attack colliders and unit detectors — it will collide with the friendly and enemy layers.
- **Wall**: The layer that includes the floors and walls — it detects collisions with the friendly and enemy layers.
- **PlayerBlocker**: A special layer made just for the player — it's similar to the walls layer, except that enemy actors can pass through it.

To create these layers, go to **Edit \ Project Settings \ Tags and Layers** in the top menu.



In the layers drop-down, go down to **Layer 8** and add the **Friendly**, **Enemy**, **Detector**, **Wall** and **PlayerBlocker** layers, like this:



Builtin Layer 0	Default
Builtin Layer 1	TransparentFX
Builtin Layer 2	Ignore Raycast
Builtin Layer 3	
Builtin Layer 4	Water
Builtin Layer 5	UI
Builtin Layer 6	
Builtin Layer 7	
User Layer 8	Friendly
User Layer 9	Enemy
User Layer 10	Detector
User Layer 11	Wall
User Layer 12	PlayerBlocker
User Layer 13	
User Layer 14	

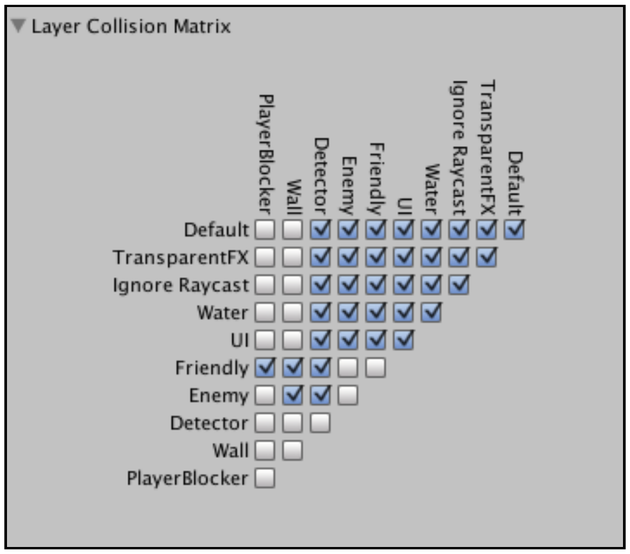
Open the **Physics manager** by selecting **Edit \ Project Settings \ Physics**. You'll be greeted by the **Layer Collision Matrix**.

As you can see, it shows all the layers as rows and columns. When you want a layer to collide with another layer, just check the box where their row and column intersect. If you don't want them to collide, just uncheck the appropriate box.

Set up the collisions according to these rules:

- Friendly can't collide with Friendly and Enemy.
- Enemy can't collide with Friendly, Enemy and PlayerBlocker.
- Detector can't collide with PlayerBlocker, Wall and Detector.
- Walls can *only* collide with Friendly and Enemy.
- PlayerBlockers can *only* collide with Friendly.

Your Collision Matrix should look like this:

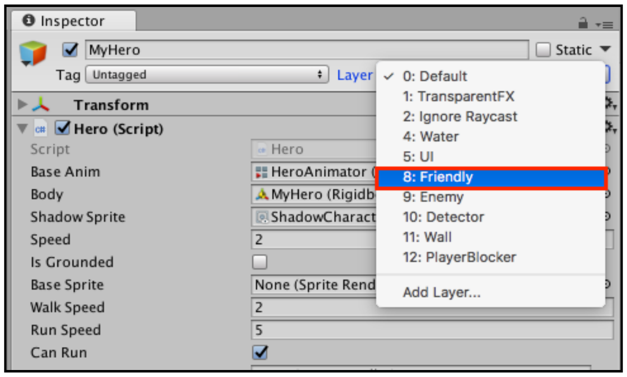


Layer Collision Matrix

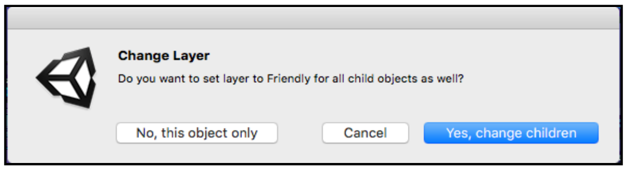
	PlayerBlocker	Wall	Detector	Enemy	Friendly	UI	Water	Ignore Raycast	TransparentFX	Default
Default	☐	☐	✓	✓	✓	✓	✓	✓	✓	✓
TransparentFX	☐	☐	✓	✓	✓	✓	✓	✓	✓	
Ignore Raycast	☐	☐	✓	✓	✓	✓	✓	✓		
Water	☐	☐	✓	✓	✓	✓	✓			
UI	☐	☐	✓	✓	✓	✓				
Friendly	✓	✓	✓	☐	☐					
Enemy	☐	✓	✓	☐						
Detector	☐	☐	☐							
Wall	☐	☐								
PlayerBlocker	☐									

Good work. Close that, select any GameObject and find the **Layer** property at the top-left of the Inspector, next to the Tag property. Here you'll edit the layers each GameObject belongs to.

Select **MyHero** and change the **Layer** to **Friendly**.

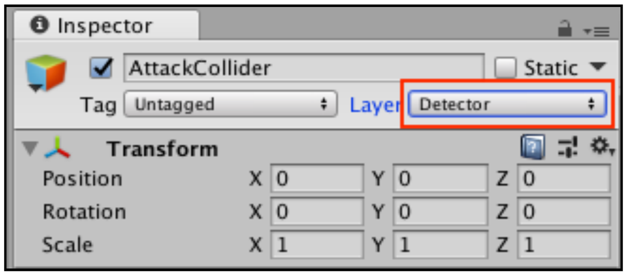


You'll get a prompt that asks if you want to change layers for the child objects. Select **Yes, Change Children**.

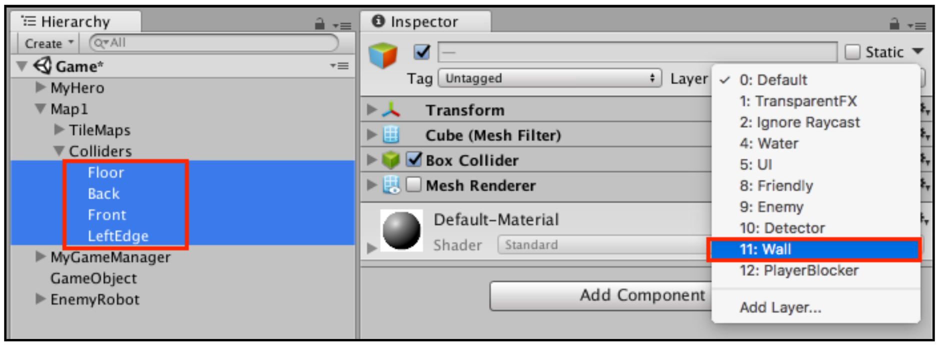

Change Layer

Do you want to set layer to Friendly for all child objects as well?

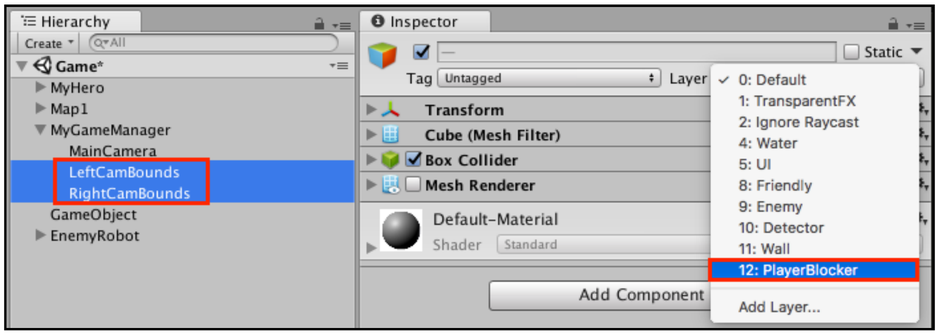
Select **AttackCollider**, the child of **HeroAnimator**, and change its **Layer** to **Detector**.



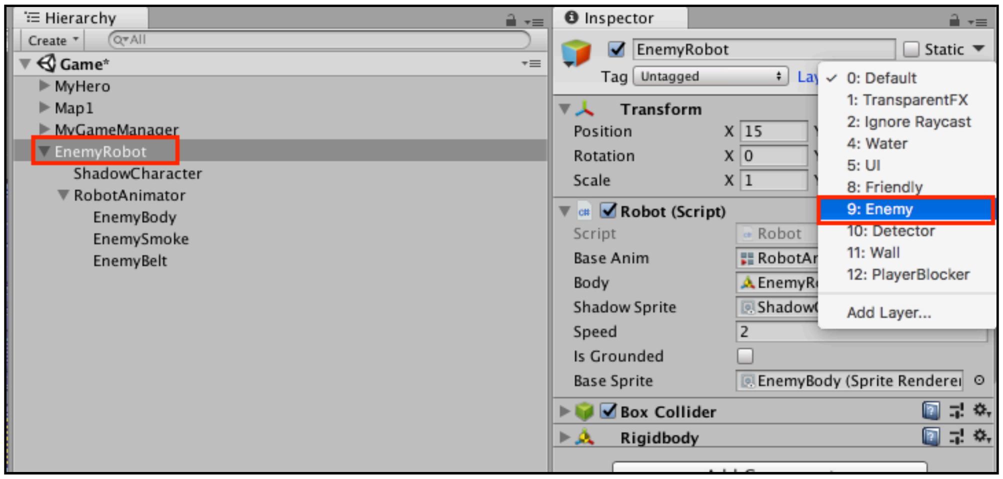
Select all the **children** of **Map1** \ **Colliders** and set their **Layer** to **Wall**.



Select the **LeftCamBounds** and **RightCamBounds** objects under MyGameManager and set their **Layer** to **PlayerBlocker**.



Finally, select **EnemyRobot** and change its **Layer** to **Enemy**. When prompted, select **Yes, Change Children**.



Run the game now and take a run at the enemy to test what you just set up. Like a ghost passes through walls, the hero should be able to pass through the robot. And with that, you've quashed the robot conspiracy to corner and suppress the hero.

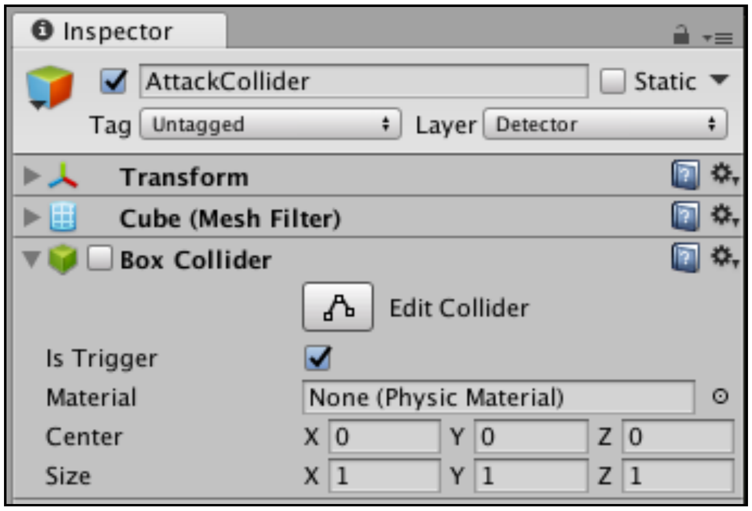
Next up, attack collisions!

Attacking step one: detection

Whenever an actor attacks, be it a hero or a robot, a box collider checks whether it hit anything.

Unity provides callbacks to these methods when there's a collision between two colliders with rigidbodies: `OnCollisionEnter` and `OnCollisionExit`. In your scenario, however, these won't work because colliders automatically push other colliders outside the bounds of their attack boxes.

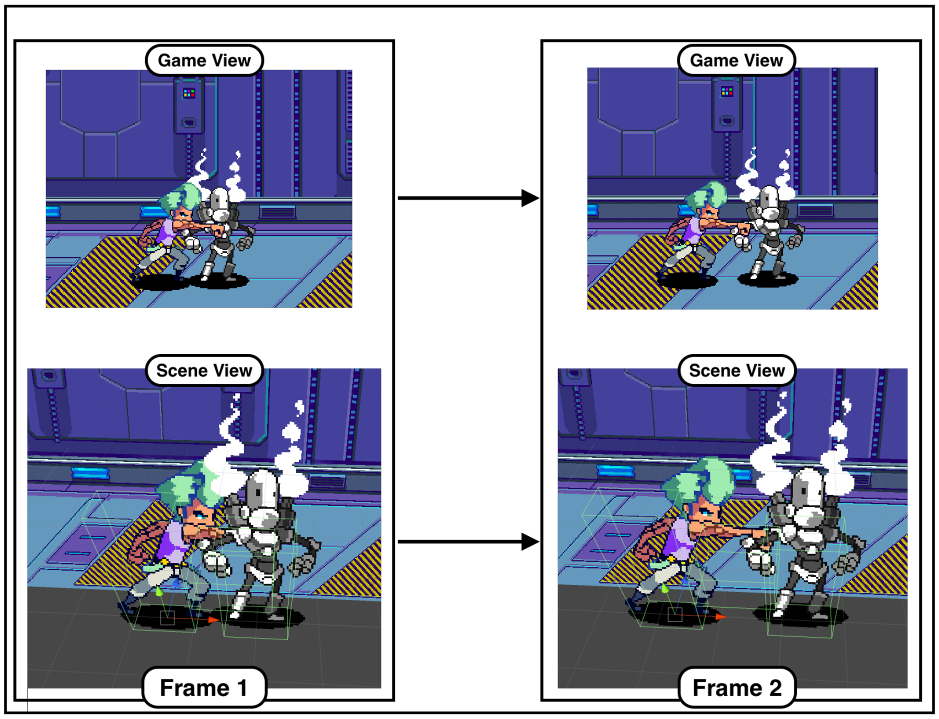
The following image demonstrates how the Inspector looks if the HeroAnimator's box collider, aka AttackCollider, is not set as a trigger. See the effect for yourself by unchecking **Is Trigger** under the Box Collider section. Run the game and observe what happens when you punch the robot.



AttackCollider
Tag: Untagged
Layer: Detector
Is Trigger: ✓
Material: None (Physic Material)
Center: X 0 Y 0 Z 0
Size: X 1 Y 1 Z 1

It's displaced by the box collider. Remember to recheck **Is Trigger** when you're done bullying that hunk of junk!

In the image below, the first frame shows a collision in progress, while the second shows the physics engine doesn't allow colliders to overlap. On the top, you see the game view. Below, the scene view that shows the colliders.



There's another way to detect the collision and apply collision forces — trigger colliders, which help a collider detect when another collider is inside its bounds. These colliders also allow the game to apply custom forces after detecting hits.

The built-in methods for handling triggers events are `OnTriggerEnter()` and `OnTriggerExit()`, but they have a minor limitation.

Similar to the `HeroCallback` class, `AttackCollider` contains attack trigger colliders that receive `OnTrigger` events. Without some help from you, the **Hero** script won't be able to determine if a hit happened. To resolve this, you'll need to implement another "forwarder" class, which is a simple class that informs the **Actor** class of a trigger event.

Create a **new C# script** in the **Scripts** folder named **HitForwarder**. Replace its contents with the following:

```
using UnityEngine;

public class HitForwarder : MonoBehaviour {

  //1
  public Actor actor;
  public Collider triggerCollider;

  //2
  void OnTriggerEnter(Collider hitCollider) {
    Vector3 direction = new Vector3(hitCollider.transform.position.x -
actor.transform.position.x, 0, 0);
    direction.Normalize();

    BoxCollider collider = triggerCollider as BoxCollider;
    Vector3 centerPoint = this.transform.position;
    if (collider) {
      centerPoint = transform.TransformPoint(collider.center);
    }

    Vector3 startPoint = hitCollider.ClosestPointOnBounds(centerPoint);
    actor.DidHitObject(hitCollider, startPoint, direction);
  }
}
```

1. This class starts with declarations for the properties. The script requires a reference to the `Actor` it's checking trigger events for, and the `triggerCollider` that receives the trigger event.

2. `OnTriggerEnter` is a native Unity method for handling trigger events. Because trigger events don't store information about collider overlaps, you use this method to estimate where collisions happen. `ClosestPointOnBounds()` from the overlapping collider calls the actor's `DidHitObject` method with the `hitCollider` that engaged the trigger, estimated `startPoint` of collision, and `direction` of the collision.

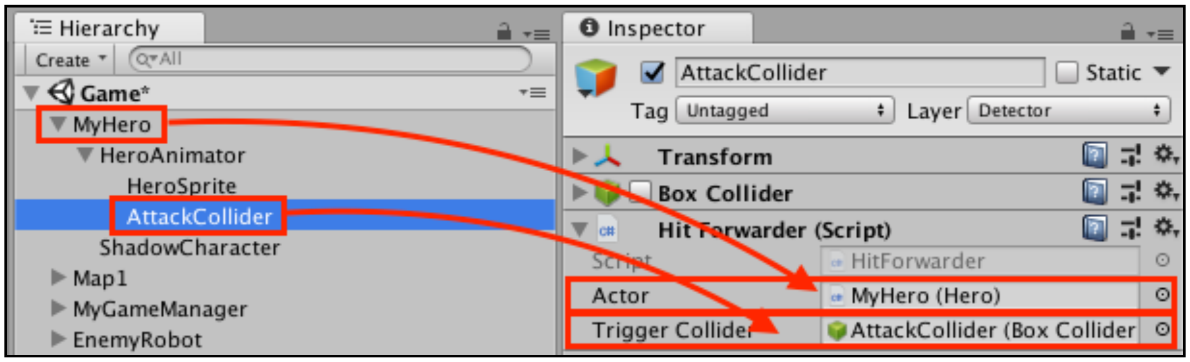
Next, open the **Actor** script and add the following methods inside the `Actor` class:

```
//1
public virtual void DidHitObject(Collider collider, Vector3 hitPoint,
Vector3 hitVector) {
  Actor actor = collider.GetComponent<Actor>();
  if (actor != null) {
    if (collider.attachedRigidbody != null) {
      HitActor(actor, hitPoint, hitVector);
    }
  }
}

//2
protected virtual void HitActor(Actor actor, Vector3 hitPoint, Vector3
hitVector) {
    Debug.Log(gameObject.name + " HIT " + actor.gameObject.name);
}
```

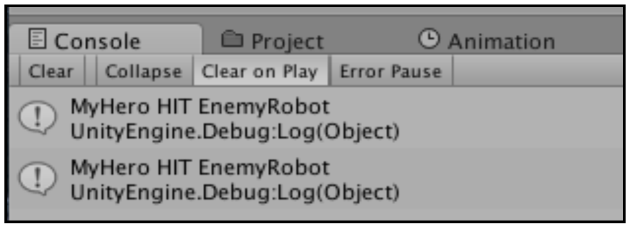
1. `DidHitObject` handles what happens when an actor hits another object that has a collider. First, the method checks if the hit object contains an Actor component. If yes, it registers a hit via `HitActor`.

2. `HitActor` contains instructions for what an actor should do when it hits another actor. For now, it contains placeholder code that writes to the console whenever a hit is detected.

Save the scripts, return to Unity and add a **HitForwarder** component to the HeroAnimator's **AttackCollider**. Set **Actor** to **MyHero** and **Trigger Collider** to **AttackCollider**.



Run the game. Walk up to the robot and start punching. Although it doesn't look like much is happening, the hero is delivering hits. Check the **Console** window for the following output:



```
MyHero HIT EnemyRobot
UnityEngine.Debug:Log(Object)
MyHero HIT EnemyRobot
UnityEngine.Debug:Log(Object)
```

Ok great! Now you've got the hits wired up in the backend. You're ready to add the robot's reaction!

Attacking step two: dying

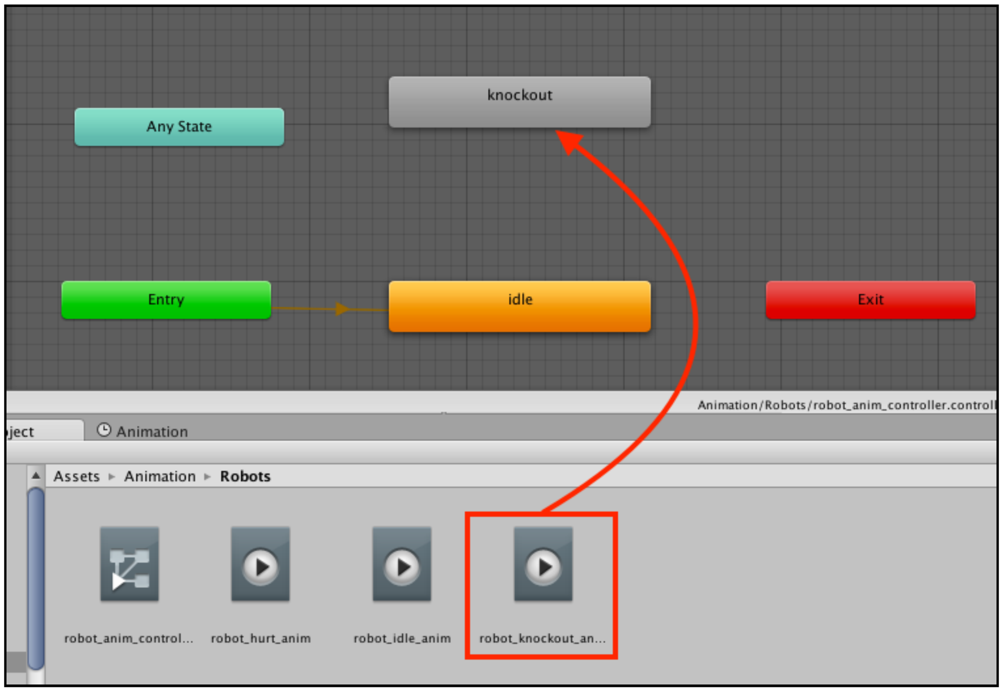
When a robot takes a hit, it should look injured. You can animate this effect in a number of ways, including the old classic fall-over-and-die. Nothing says *"You did something awesome!"* like a one-hit KO.

Hit reaction and robot death

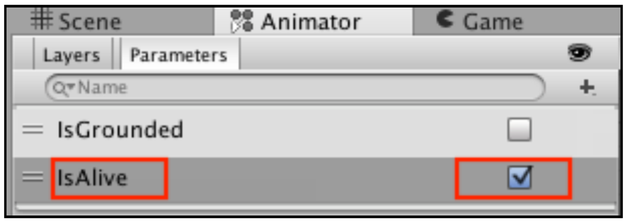
By the time you're done with the game, every actor will have a death animation that's triggered when it's life value drops to zero. For now, however, you'll trigger the robot death sequence whenever it takes a hit.

Another classic beat-em-up death effect is when the dead character repeatedly flickers then vanishes. You'll also implement this effect in Pompadroid.

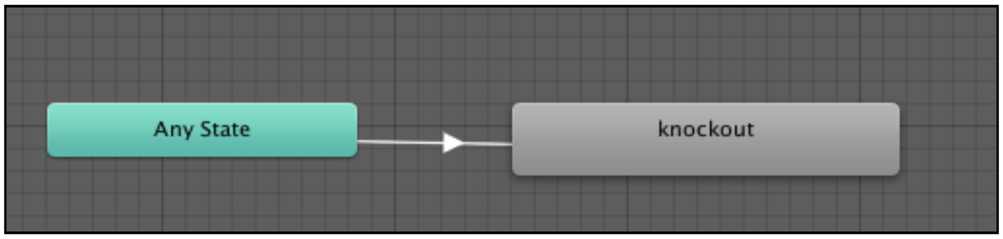
Open the **Animator** window by double-clicking **robot_anim_controller**. Drag **robot_knockout_anim** into the **Animator** and rename it **knockout**.



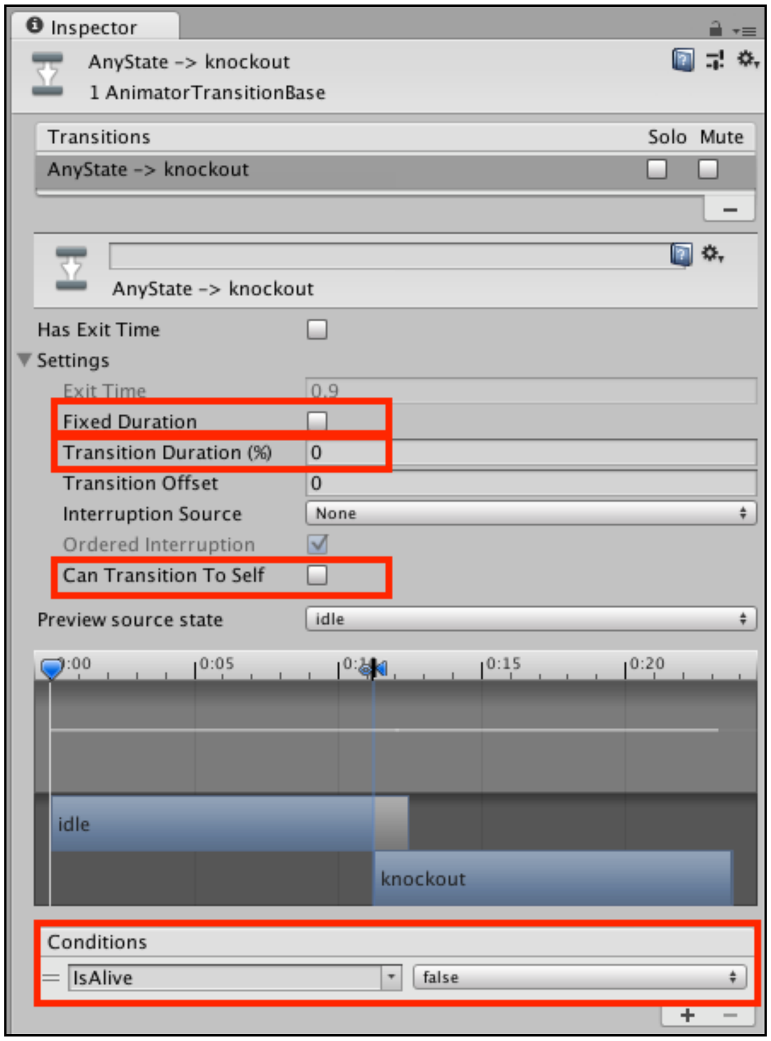
Add a **new Boolean** event parameter named **IsAlive**, which will determine if this robot is still alive and kicking. Check **IsAlive** in the parameters pane to set its default value to true.



Add a new transition from **Any State** to **knockout**.



Select the transition. Disable **Fixed Duration** and set **Transition Duration** to `0`. Uncheck **Can Transition to Self** to disable the animation's ability to call itself whenever the **Actor** is dead. Add a new condition of **IsAlive** and set it to **false**.


Inspector
AnyState -> knockout
1 AnimatorTransitionBase
Transitions Solo Mute
AnyState -> knockout
AnyState -> knockout
Has Exit Time
Settings
Exit Time 0.9
Fixed Duration
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption
Can Transition To Self
Preview source state idle
Conditions
IsAlive false

Any State is a special state that can be transitioned to from other states. It’s a useful feature when you need a state that can interrupt other states, for instance, when there’s a death. In this case, the actor shouldn’t finish its current animation; rather, it should proceed directly to its death state.

Note: You might wonder why you named the death animation **knockout**. You’re using the same base animation for falling down and death. The difference is simple: when the robot takes some damage, it’ll fall and get back up. But when it dies, it never gets back up.

The animation state is now complete. Open **Actor.cs** and add this variable to the top:

```
public bool isAlive = true;
```

This Boolean contains the current status of the actor; this value will be set to false when the actor is dead.

Replace the `Debug.Log` line in `HitActor()` with this:

```
actor.Die();
```

This tells the Actor class that the receiver of the punch should die when hit.

Add the following methods to the class:

```
//1
protected virtual void Die() {
  isAlive = false;
  baseAnim.SetBool("IsAlive", isAlive);
  StartCoroutine(DeathFlicker());
}

//2
protected virtual void SetOpacity(float value) {
  Color color = baseSprite.color;
  color.a = value;
  baseSprite.color = color;
}

//3
private IEnumerator DeathFlicker() {
  int i = 5;
  while (i > 0) {
    SetOpacity(0.5f);
    yield return new WaitForSeconds(0.1f);
    SetOpacity(1.0f);
    yield return new WaitForSeconds(0.1f);
    i--;
  }
}
```

1. `Die()` kills the actor by setting `isAlive` to false and playing the death animation clip by setting the animation clip's `IsAlive` boolean parameter to `false`.

2. `SetOpacity()` is a helper method that changes the `opacity` of the actor's sprite. You use it to create a flickering effect when an actor dies.

3. `DeathFlicker()` toggles the hero's sprite opacity between partially transparent and opaque. It changes the actor's `baseSprite` to 50% opaque for `0.1` seconds, then to 100% opaque for another `0.1` seconds. By repeating this five times, you get a flicker effect.

Save the scripts and return to Unity. Run the game and start punching. One measly punch sends the robot to the floor! The robot's sprite will flicker as it falls to the ground to signify death.

A word on coroutines

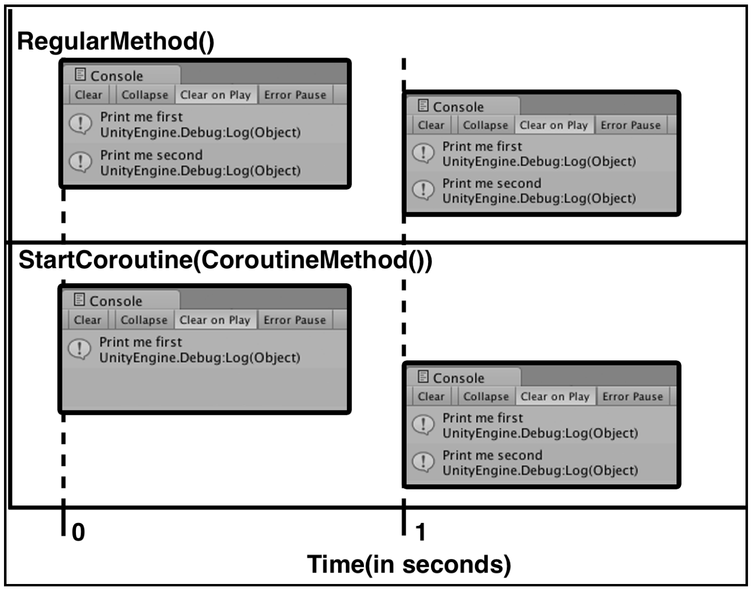
`DeathFlicker()` is a **coroutine**, which is a special kind of method that's created like any other, but it has a return type of `IEnumerator`. It can also contain multiple `yield return` statements. For example, `yield return new WaitForSeconds(0.1f)` that waits `0.1` seconds before continuing the method. Coroutines are invoked from the `StartCoroutine()` method.

Coroutines are helpful when programming time-sensitive events, such as the death flicker effect. They have the ability to pause and resume at a later time. For example, consider these two code blocks:

```
private void RegularMethod()
{
  Debug.Log("Print me first");
  Debug.Log("Print me second");
}

private IEnumerator CoroutineMethod()
{
  Debug.Log("Print me first");
  yield return new WaitForSeconds(1.0f);
  Debug.Log("Print me second");
}
```

If you ran these two methods by calling `RegularMethod()` and `StartCoroutine(CoroutineMethod())`, you'd notice one logs both messages at once, while the other takes a brief pause between.

In `RegularMethod()`, both *Print me first* and *Print me second* appear when the `RegularMethod` is called. But with `CoroutineMethod()`, *Print me first* appears immediately after the `StartCoroutine()` invocation, and *Print me second* appears one second afterward.



```
Print me first
UnityEngine.Debug:Log(Object)
Print me second
UnityEngine.Debug:Log(Object)
```

Coroutines make programming certain game features so much easier!

Robots need life too

Soon, the glee you feel from taking out droids with one hit will wear off — maybe not yet, but soon. As such, the next step is giving the robot a life value, aka health so that it can stand up to the hero's abuse.

The Robot and Hero classes share a common **life** value. Eventually, you'll configure the game so that a hit depletes the value incrementally, and the actor dies when the value is `0`.

Since they share this trait, you can add the life parameter to the `Actor` superclass.

Open **Actor.cs** and add the following lines to the variable declaration:

```
public float maxLife = 100.0f;
public float currentLife = 100.0f;
```

- `maxLife` defines the maximum value of the actor's life.
- `currentLife` is the instance's current life value.

Being that they are public, you can adjust these properties in the Inspector.

Add this `Start` method to **Actor.cs**:

```
protected virtual void Start() {
  currentLife = maxLife;
  isAlive = true;
  baseAnim.SetBool("IsAlive", isAlive);
}
```

When `Start()` is called at the beginning of the game, `currentLife` is set to the `maxLife` value and the `IsAlive` animation parameter is initialized.

Note the presence of the access modifier `protected`, which means only derived classes can access this method. You also have the `virtual` keyword so that derived classes, such as `Robot` and `Hero`, can override this method.

Still in **Actor.cs**, add this method:

```
public virtual void TakeDamage(float value, Vector3 hitVector) {
  //1
  FlipSprite(hitVector.x > 0);
  currentLife -= value;

  //2
  if (isAlive && currentLife <= 0) {
    Die();
  }
}
```

Here you have a shiny, new `TakeDamage` method that handles what happens when an actor sustains damage.

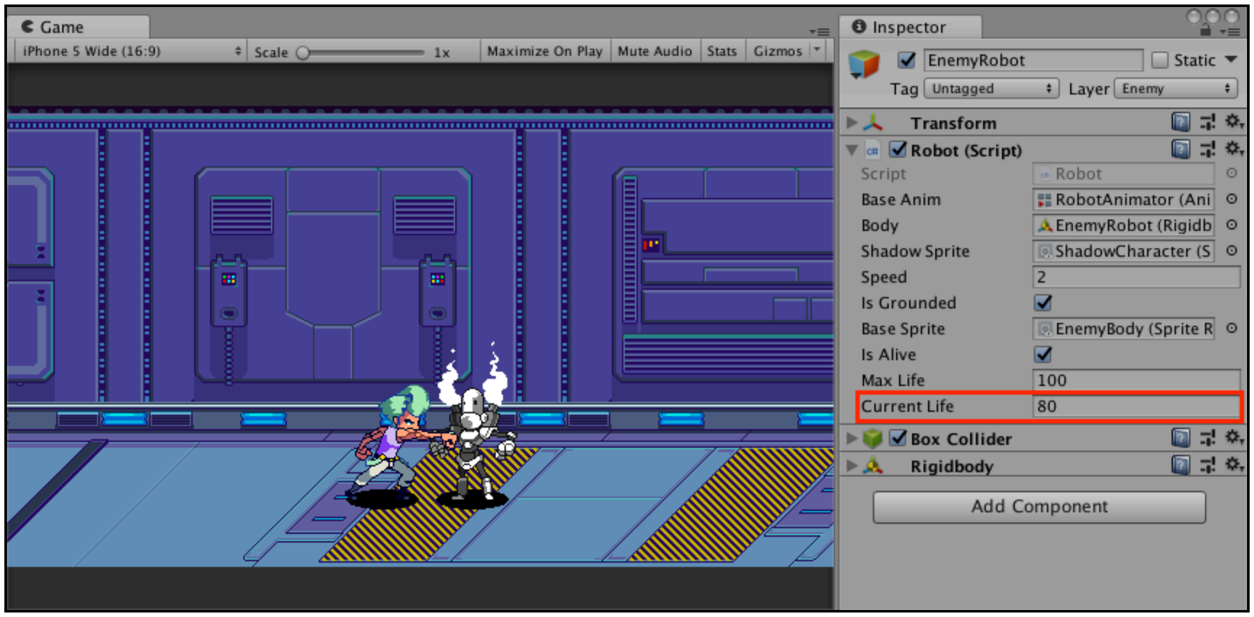
1. `FlipSprite` makes the actor face the direction from whence the damage came.
2. The `if` statement evaluates `currentLife`, and when that value reaches `0`, it triggers the actor's `Die` method.

Open **Actor.cs** again, and replace the contents of `HitActor` with this line:

```
actor.TakeDamage(10, hitVector);
```

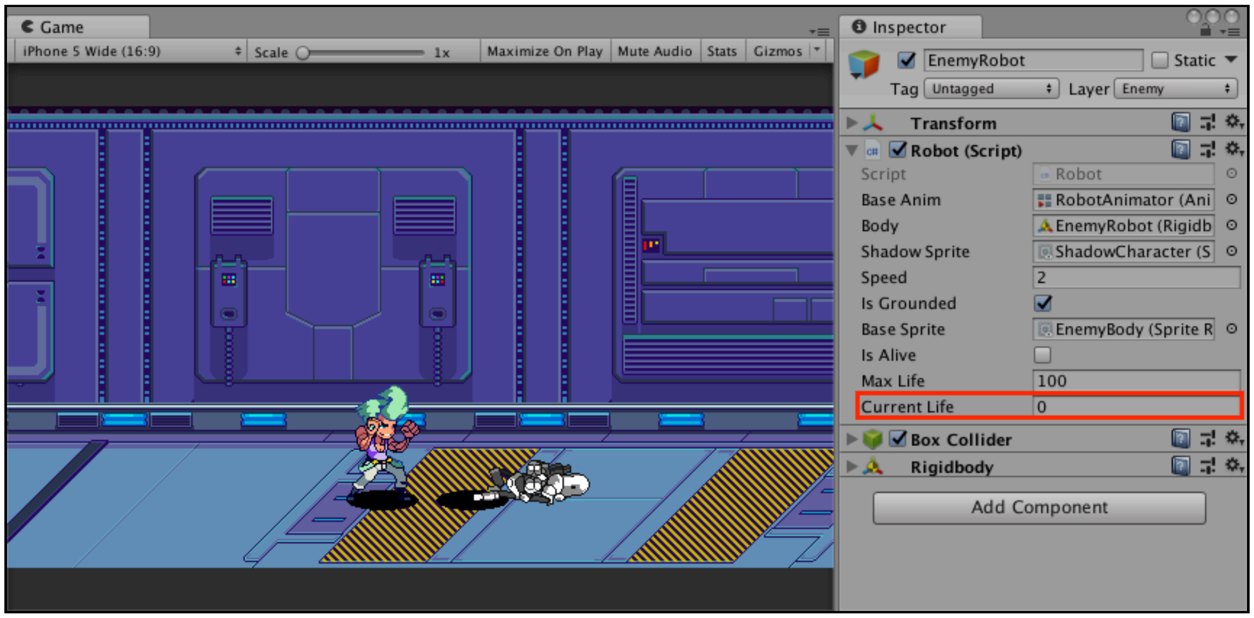
When an actor's collider receives a punch, its `currentLife` property is reduced by 10 points.

Save the script, return to the editor and run the game. Go punch a droid! Watch the EnemyRobot Inspector every time you punch the robot to watch its life go down.



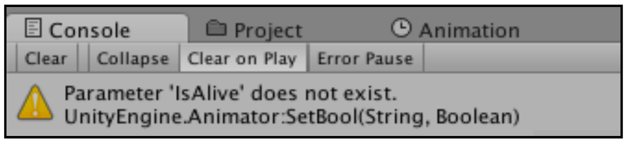
EnemyRobot Static
Tag Untagged Layer Enemy
Robot (Script)
Script: Robot
Base Anim: RobotAnimator (Ani
Body: EnemyRobot (Rigidb
Shadow Sprite: ShadowCharacter (S
Speed: 2
Is Grounded: ✓
Base Sprite: EnemyBody (Sprite R
Is Alive: ✓
Max Life: 100
Current Life: 80

Keep punching till you've reduced its life to `0` and then keep hitting it. That's right, show that droid who's boss!



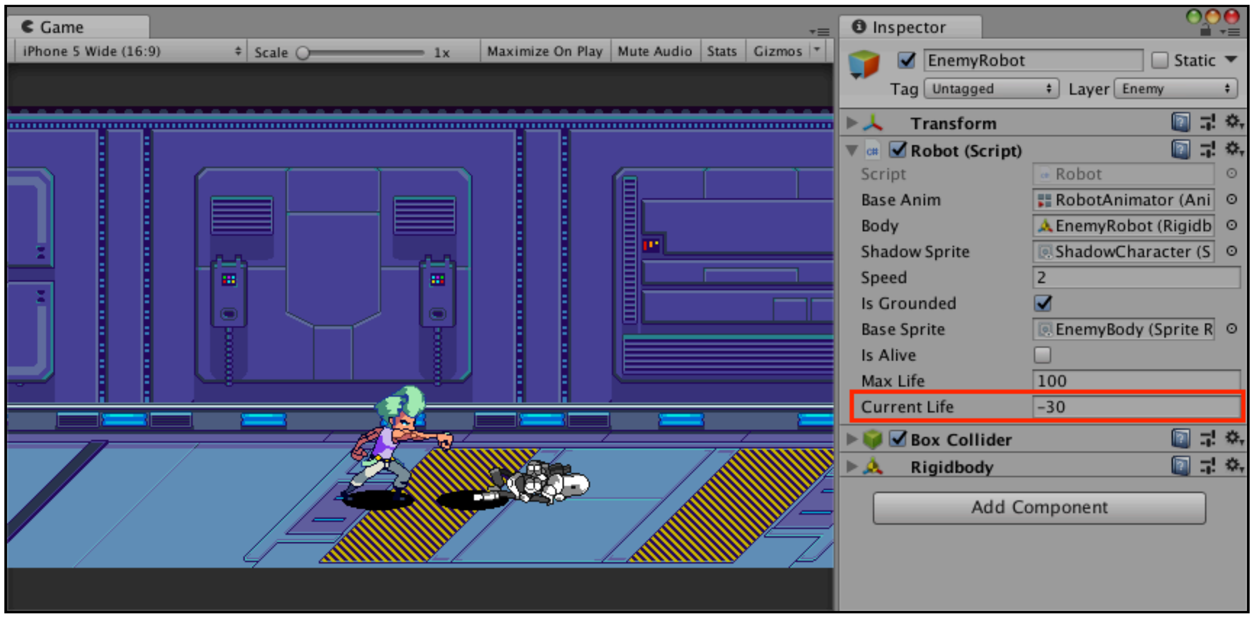
EnemyRobot Static
Tag Untagged Layer Enemy
Robot (Script)
Script: Robot
Base Anim: RobotAnimator (Ani
Body: EnemyRobot (Rigidb
Shadow Sprite: ShadowCharacter (S
Speed: 2
Is Grounded: ✓
Base Sprite: EnemyBody (Sprite R
Is Alive: ☐
Max Life: 100
Current Life: 0

You'll see a warning in the **Console** regarding a missing **IsAlive** parameter in the hero's animator controller. Ignore it — you'll add it in a future chapter.



```
Parameter 'IsAlive' does not exist.
UnityEngine.Animator:SetBool(String, Boolean)
```

While you were punching that poor old metallic carcass, you might have noticed the robot took damage even after it died. Uh-oh! That’s a bug you need to fix.



EnemyRobot ☐ Static
Tag Untagged Layer Enemy
Robot (Script)

Script	Robot
Base Anim	RobotAnimator (Ani
Body	EnemyRobot (Rigidb
Shadow Sprite	ShadowCharacter (S
Speed	2
Is Grounded	✓
Base Sprite	EnemyBody (Sprite R
Is Alive	☐
Max Life	100
Current Life	-30

Add the following method to `Actor.cs`:

```
public bool CanBeHit() {
  return isAlive;
}
```

This method returns a Boolean to determine if the actor can be hit. If the actor isn’t alive, then it can’t get hurt anymore. You’ll see this again in later chapters.

There’s one more line to modify. Find `DidHitObject` and locate `if (actor != null) {` inside of it. Change it to:

```
if (actor != null && actor.CanBeHit()) {
```

This determines if an actor can be hit before applying damage.

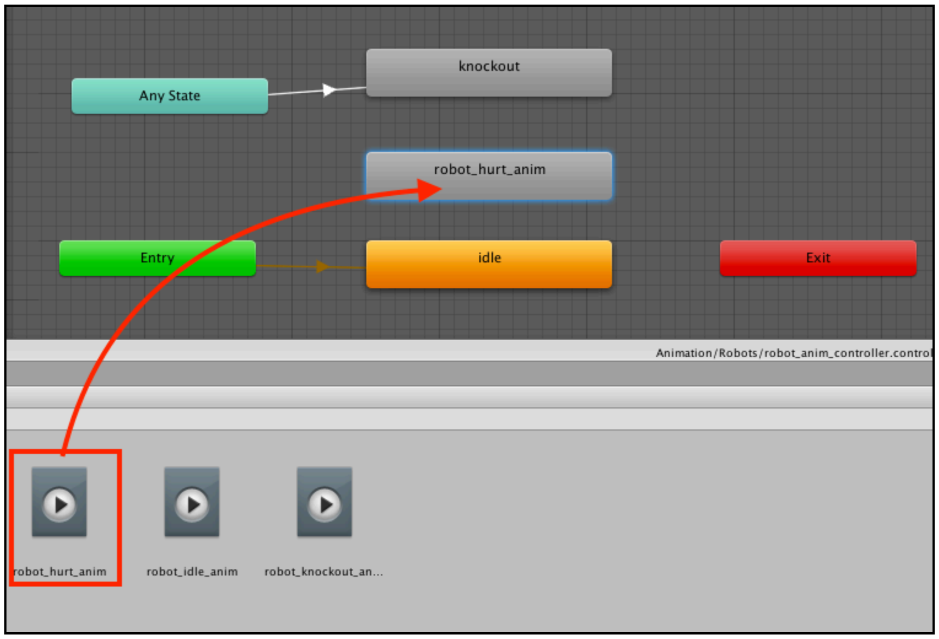
Save the script and return to Unity. **Save** your scene and run the game, again punching that droid into oblivion. Its life should not continue to reduce after death because now it can only take a hit when it’s alive.

Everybody hurts sometimes

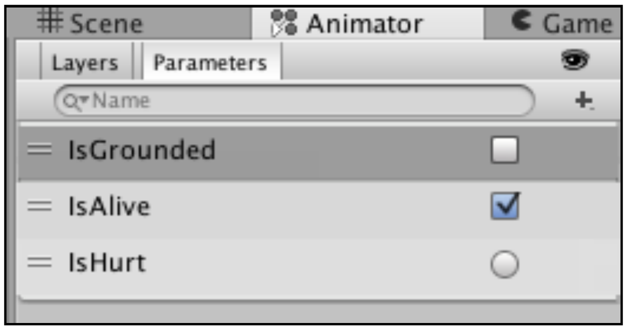
Great! You’ve made a stronger droid that takes multiple punches. However, the robot is remarkably stoic while he’s getting pummeled.

Adding a **flinch** animation will make hitting the actor more satisfying, giving the player a visual cue that they’re doing something.

Open the Animator window that has **robot_anim_controller** loaded. Drag **robot_hurt_anim** in to create a new state and rename it to **hurt**.

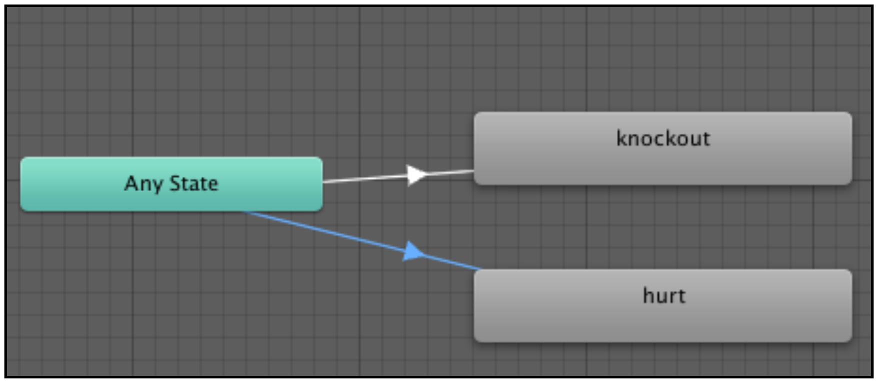


Add a new trigger parameter named **IsHurt**. You’ll use this parameter to trigger the robot’s flinch animation.



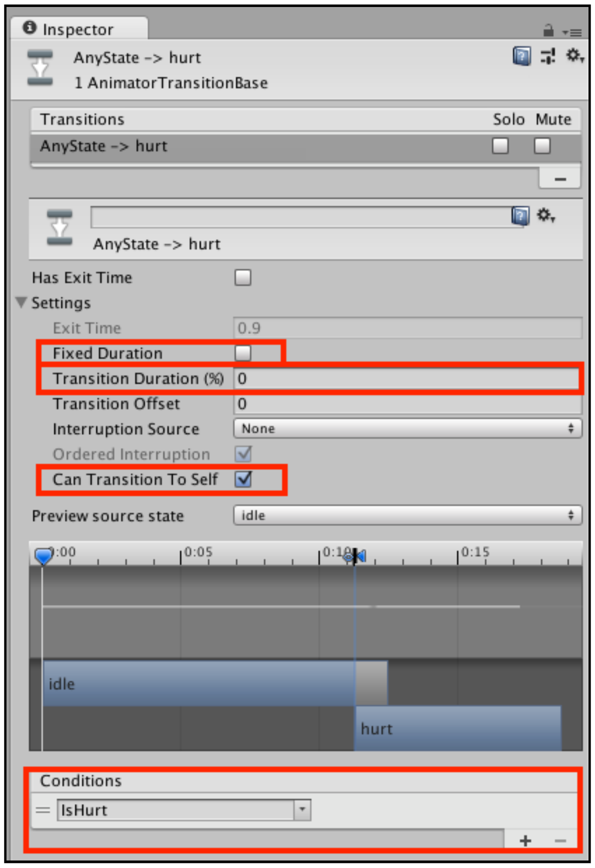
Hurt state is an animation that will show the robot flinching when it takes damage.

Add a transition from **Any State** to the **hurt** state.



Select the transition. Uncheck **Has Exit Time** and **Fixed Duration**. Set **Transition Duration** to `0`. Add a new condition with the **IsHurt** trigger parameter.

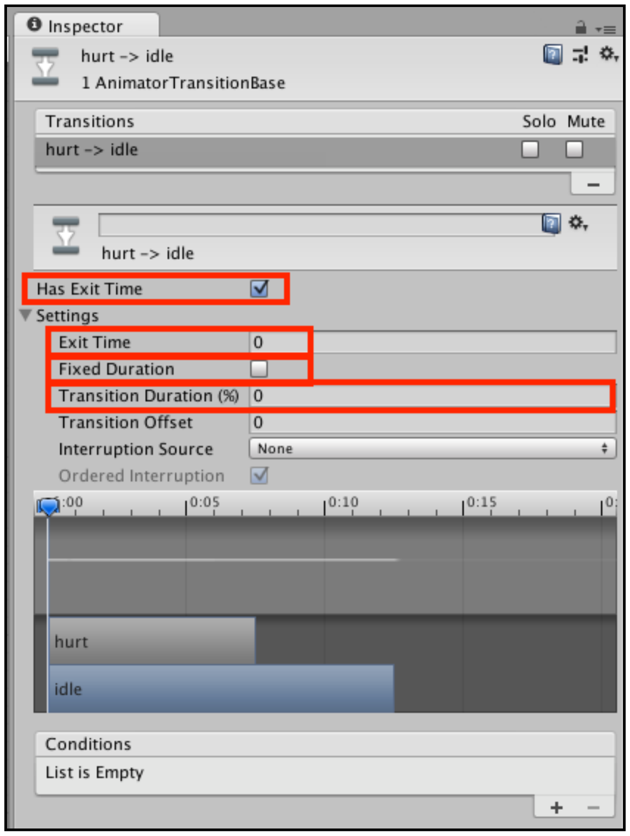
Keeping **Can Transition to Self** checked allows the hurt animation to transition to itself, so the robot can be hurt (again) while being hurt. Make sense?



Has Exit Time ☐
Settings
Exit Time 0.9
Fixed Duration ☐
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓
Can Transition To Self ✓
Preview source state idle

Conditions
IsHurt

Add another transition from hurt to idle. This one should only transition when hurt finishes, so keep **Has Exit Time** selected, uncheck **Fixed Duration** and set **Transition Duration** to `0`.


Inspector

hurt -> idle
1 AnimatorTransitionBase

Transitions	Solo	Mute
hurt -> idle	☐	☐

hurt -> idle

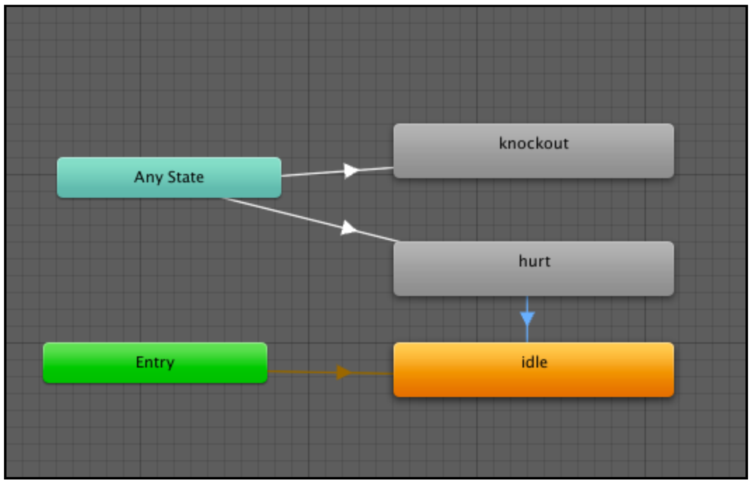
Has Exit Time ✓

Settings

Exit Time	0
Fixed Duration	☐
Transition Duration (%)	0
Transition Offset	0
Interruption Source	None
Ordered Interruption	✓

Conditions
List is Empty

The **robot_anim_controller** state machine should look like this:



Your last act in this chapter is to update the Actor script.

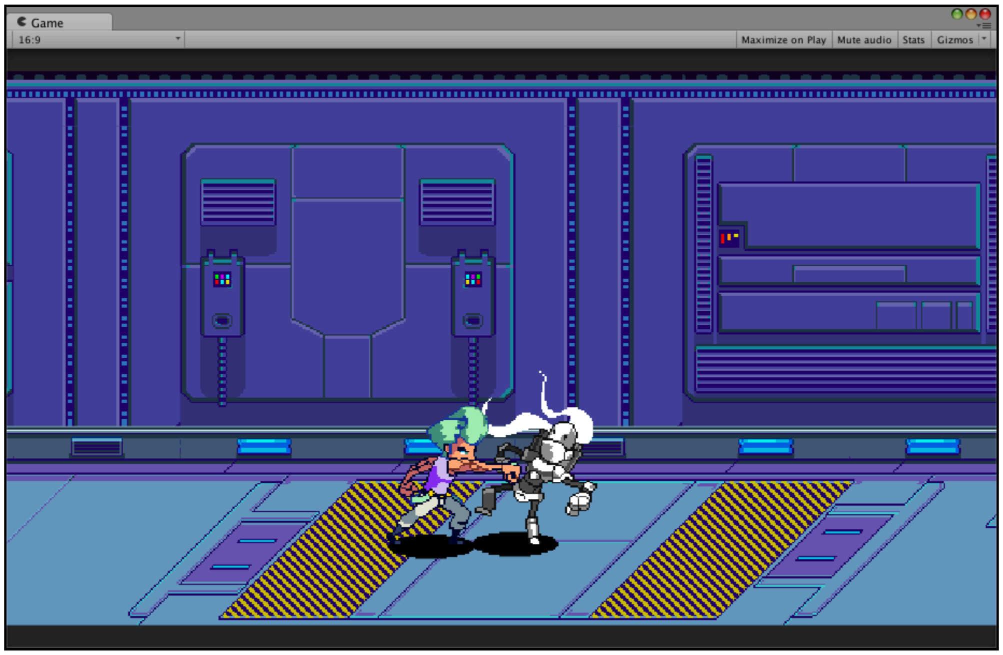
Open **Actor.cs**. Find the `TakeDamage` method and replace it with the following:

```
public virtual void TakeDamage(float value, Vector3 hitVector) {
  FlipSprite(hitVector.x > 0);
  currentLife -= value;
```

```
    if (isAlive && currentLife <= 0) {
      Die();
    } else {
      baseAnim.SetTrigger("IsHurt");
    }
  }
```

You added an `else` condition that tells the **Actor** that if the damage isn't enough to cause death, then it should just play the hurt animation.

Save the script, run the game and punch that droid! Beating metal was never so satisfying — or so easy on your knuckles.



Where to go from here?

Congratulations! You've gotten a lot done in this chapter. When you started, the hero was on a lonesome walkabout in his little world, but now he has a buddy that's begging for a bashing!

Up to this point, you've:

- Added a punching bag — I mean robot — to the scene.
- Created and used layers to manage collisions in the physics engine.
- Gave life to a robot, and then took it back with a few punches.
- Created a timed flickering event to signify death by using Coroutines.

- Added a hurt animation to show when the poor robot has taken a hit.

With the foundation for the robot in place, you're ready to introduce more features to this crucial actor!

In the next chapter, you'll make the robots into sentient beings. They'll learn how to move and attack and be the brainiest bots you have ever faced!

Chapter 6: Brainy Bots

Right now you have a hero with a powerful punch and droids without any smarts — they're just standing there waiting to take a pummeling. By the end of this chapter, you'll have bots with brains that seek and destroy the hero.

This chapter will cover the following topics:

- Animating the robots' walk and attack actions.
- Using navigation mesh pathfinding to teach the robots how to navigate the map.
- Writing decision-based AI to the robots.
- Using tags to identify specific GameObjects.
- Implementing life value for the hero character, thereby making the game a fight for survival!

I, robot — decision-based AI

You can punch that poor robot until it falls down! It never retaliates. Where's the fun in that?

Although punching a helpless robot may bring you some short term joy, a game worth playing is challenging. Usually, you see smart opponents or horrendous obstacles or puzzles — maybe all of the above and then some.

This game is sorely lacking in competitive spirit, so you'll make each robot move similarly to a player, albeit a lesser-skilled player.

In this section, you'll retrofit the robots with brains to help them decide what to do in every situation — decision-based AI. You'll give each robot the ability to choose a course of action at specific time intervals.

Before teaching the robot complex behavior logic, you need to set up the basic actions that the AI will command the robot to do. The robot must be able to:

1. Idle
2. Attack and punch
3. Chase the hero and roam around

You've already made the idle state the robot's default, so that part is done.

The second action resembles the hero's punch, so you'll implement it in a similar way. The last action is the most challenging because it requires pathfinding for the droid to determine how to move towards the hero's position.

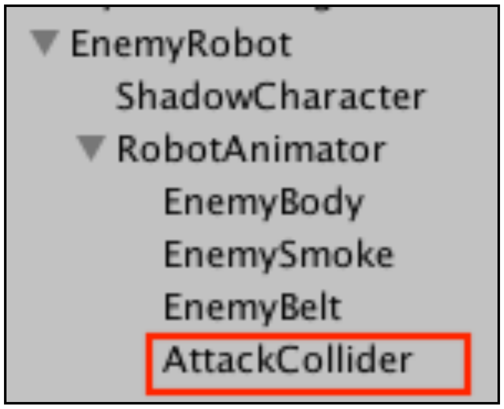
The robot takes a jab at it

First, you must teach the robot to punch.

Start by importing **Robots Attack Animation.unitypackage** from the **Unity Packages** folder. It contains the **robot_attack_anim** clip and sprites.

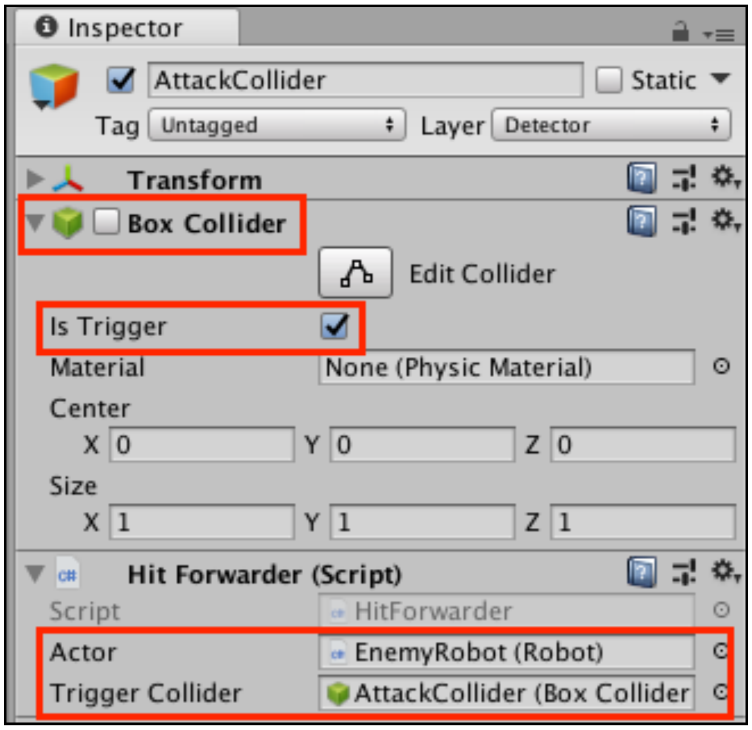
Similarly to the hero, the robot needs a trigger collider to determine when its attack connects. A box collider does this job nicely.

Open the **Game** scene, add a new empty **GameObject** named **AttackCollider**, set its parent to **RobotAnimator**, the child of **EnemyRobot** then reset its **Transform**. Finally, change its layer to **Detector**.



```
▼ EnemyRobot
    ShadowCharacter
  ▼ RobotAnimator
      EnemyBody
      EnemySmoke
      EnemyBelt
      AttackCollider
```

Add a new **Box Collider** to **AttackCollider**. **Disable** it and check **Is Trigger**. Also attach a **HitForwarder** component to it, set **Actor** to **EnemyRobot** and **Trigger Collider** to **AttackCollider**.



Inspector

AttackCollider — Static

Tag: Untagged — Layer: Detector

Transform

Box Collider (unchecked)

Edit Collider

Is Trigger: ✓

Material: None (Physic Material)

Center: X 0, Y 0, Z 0

Size: X 1, Y 1, Z 1

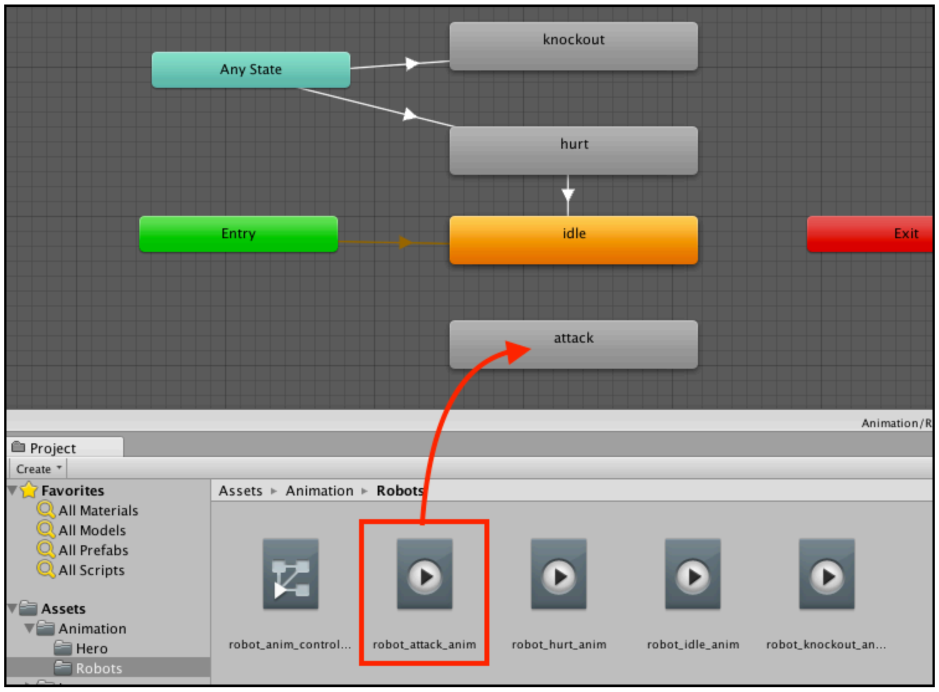
Hit Forwarder (Script)

Script: HitForwarder

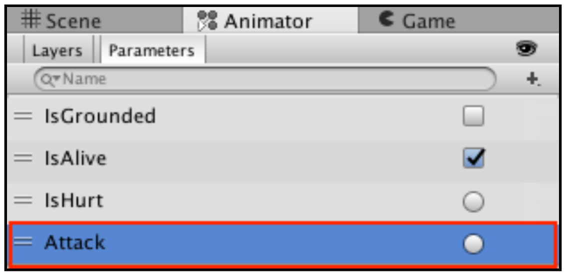
Actor: EnemyRobot (Robot)

Trigger Collider: AttackCollider (Box Collider

Next, double-click **robot_anim_controller** in the **Assets/Animation/Robots** folder to open the **Animator** window. Drag **robot_attack_anim** from the folder to the grid layout to create a new state. Name it **attack**.

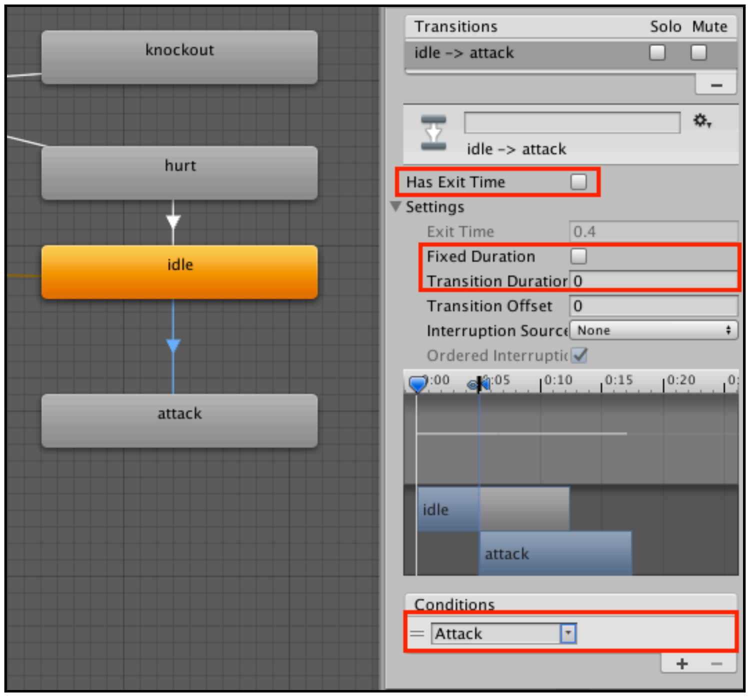


Add a new trigger parameter named **Attack**, which you'll use to trigger the robot's punch animation. This time, you need only one variable for attacking since it only throws a single punch, unlike the hero who later on will be able to deliver a flurry of punch combinations.



IsGrounded	☐
IsAlive	✓
IsHurt	○
Attack	○

Next, add a transition from **idle** to **attack**. Uncheck **Has Exit Time** and **Fixed Duration**, set **Transition Duration** to `0` then add a new condition with the **Attack** trigger being the parameter.



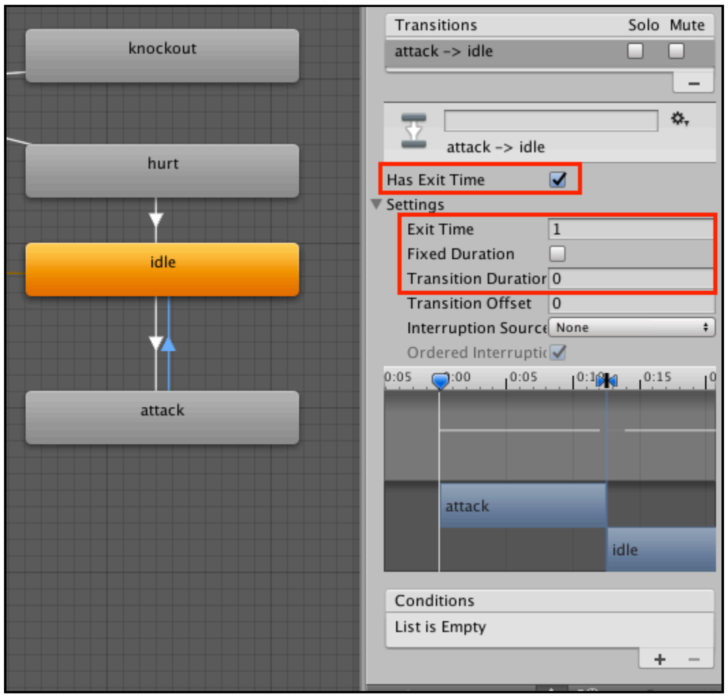
idle -> attack

Has Exit Time ☐

Exit Time 0.4
Fixed Duration ☐
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Conditions: Attack

Next, add another transition from **attack** to **idle**. Keep **Has Exit Time** checked and set **Exit Time** to 1. Uncheck **Fixed Duration** and set **Transition Duration** to `0`.



attack -> idle

Has Exit Time ✓

Exit Time 1
Fixed Duration ☐
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Conditions: List is Empty

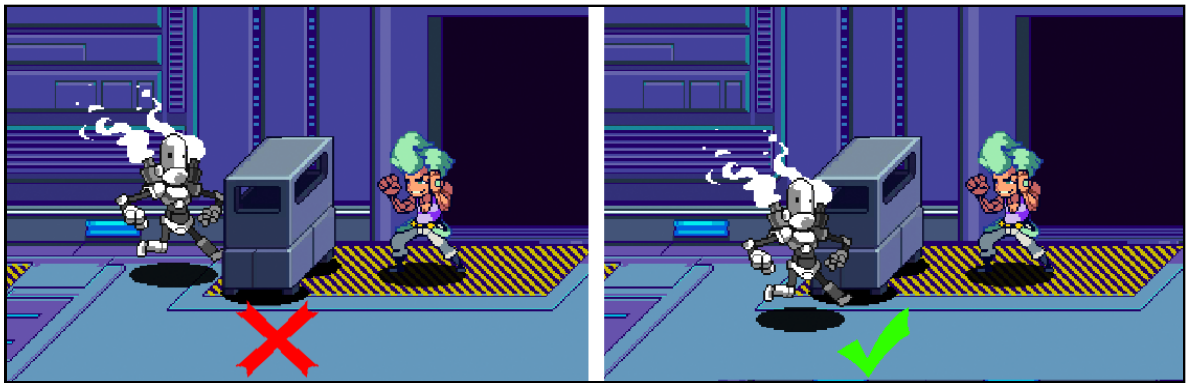
Since the **Actor** script uses the `Attack()` method that sets the attack trigger on its animator, you don’t need to add code to enable the robot’s attack.

If that hunk of metal had controls, you could make it punch the hero and possibly, displace that perfect pompadour. But droids don't get controls because you'll make them sentient soon. I'm sorry to say there will be no pompadour poundings at this point.

Next up, you'll teach the robot how to walk.

Robot baby steps

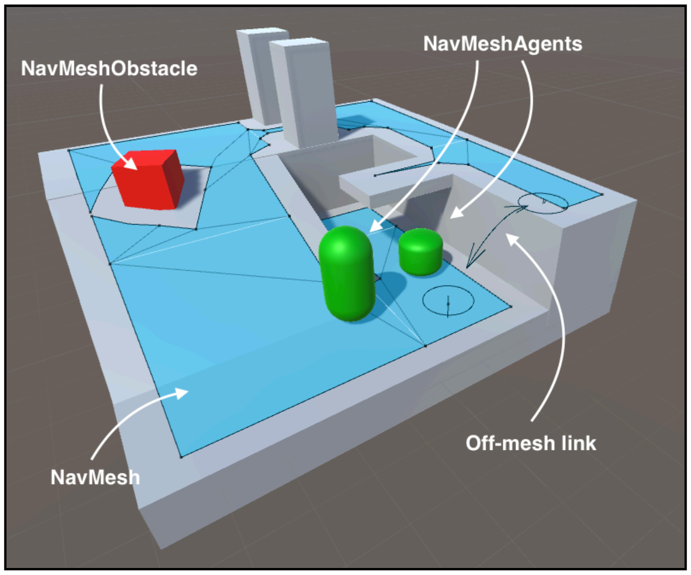
It might seem like a trivial affair, but these droids need to be able to navigate their world. Antagonists should have pathfinding abilities when there are obstacles in a game. Pathfinding allows robots to smartly navigate around obstacles towards the hero instead of getting stuck on walls or behind trash bins.



To achieve this, you'll put Unity's navigation system to good use.

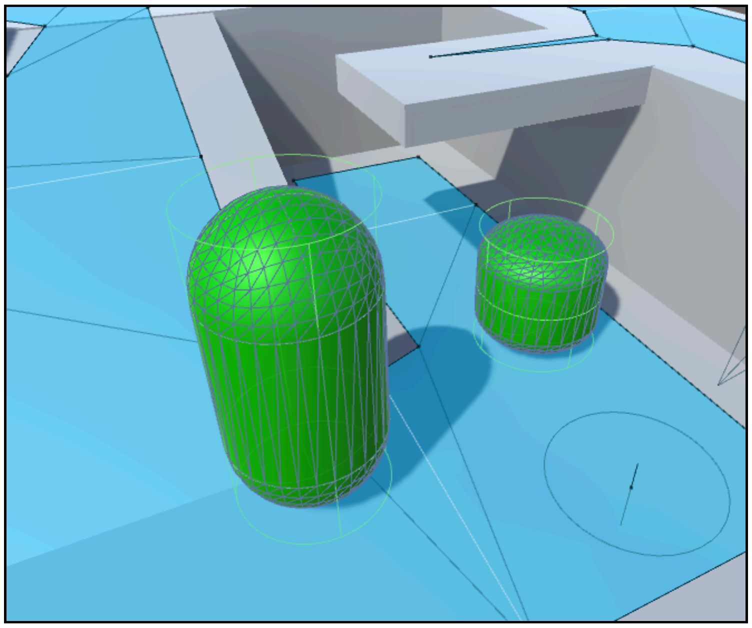
NavMeshes 101

Unity uses **Navigation Meshes** for pathfinding. **Navigation Meshes**, or **NavMeshes**, are data structures that define where entities can go. NavMeshes also determine how an entity gets from point A to point B, and whether B is even reachable. In the image below, the light blue mesh represents the NavMesh on top of the game terrain.



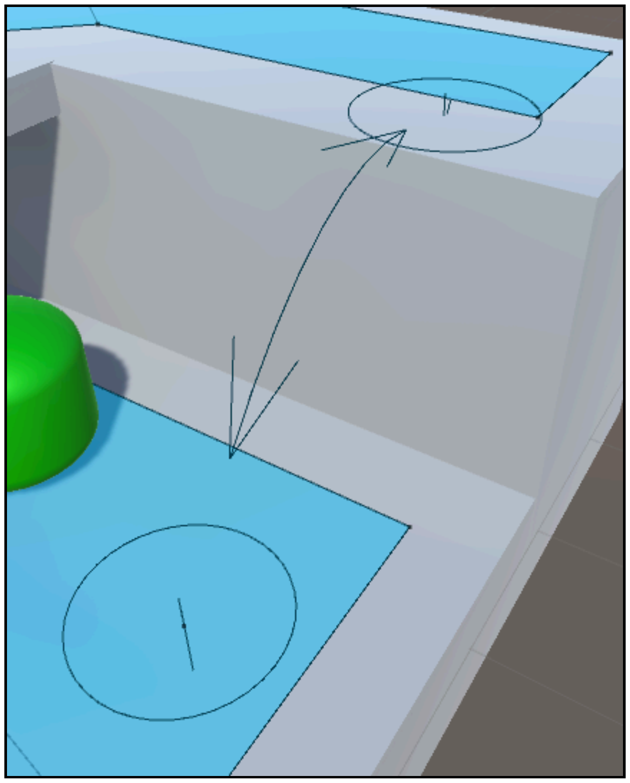
The NavMesh entities are called **NavMeshAgents**. They represent objects that can navigate the NavMesh to get from point A to point B. They also check if they can fit through the path available, avoiding impossible overhangs and narrow doorways where they might encounter clearance problems. NavMeshAgents use cylinders to represent their location on the NavMesh.

In the image below, the agent on the right can pass underneath the overhang, but the agent on the left cannot because it's too tall.

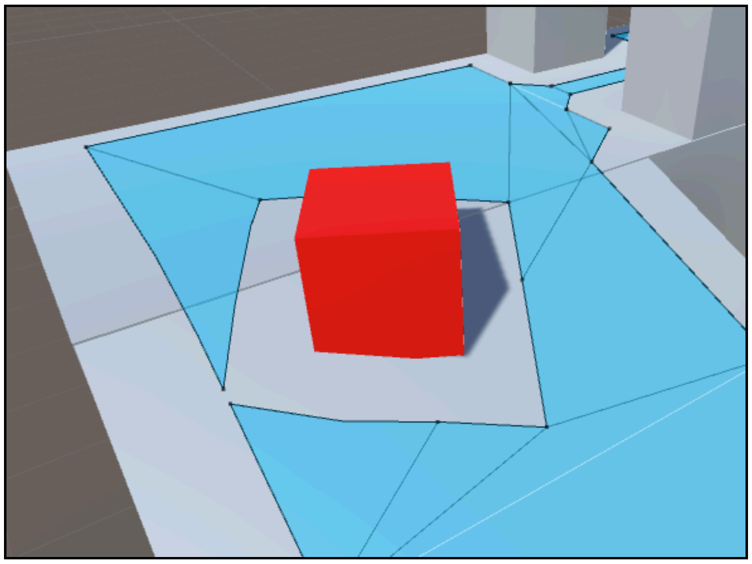


Off-Mesh Links are shortcuts that allow you to specify ways to traverse the mesh faster. A common use case is a bridge gap that's impossible to represent in a NavMesh.

Another is jumping over ridges and down ledges.



Other features of interest are **NavMeshObstacles**. These entities can modify the NavMesh and don't allow pathfinding in the spaces they occupy so that nobody will get stuck on them. The use case for this game is when you add obstacles to the map in a later chapter, such as a powerup-containing garbage can.

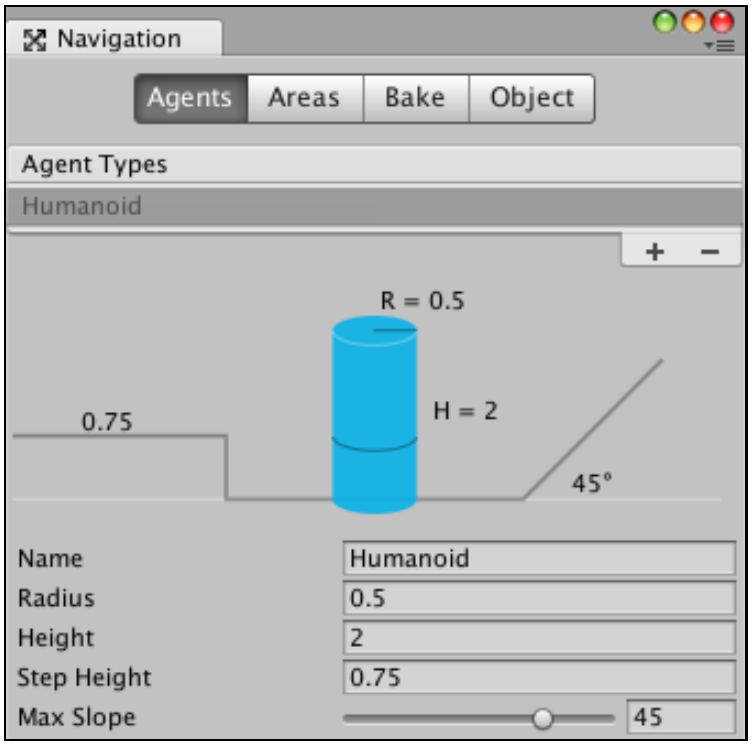


Notice how the red box cuts away the NavMesh, disallowing any pathfinding in its immediate vicinity.

NavMeshes are calculated, or **baked**, on to geometry in the game. Baking creates the necessary data structures on the mesh you'll need for pathfinding.

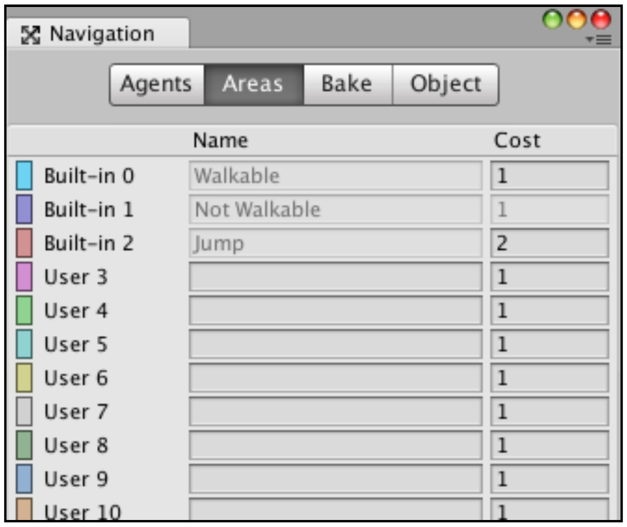
The **Navigation** window is used to generate and modify the game's NavMeshes, and you'll find it in the top menu under **Window** \ **Navigation**. This window has four tabs: **Agents**, **Areas**, **Bake** and **Object**.

The **Agents** tab stores all possible **Agent Types** that you defined. These agents store the radius and height of each agent that will move around the navigation mesh. You can also define the step height and the maximum slope the agent can move around the mesh with.



Name	Humanoid
Radius	0.5
Height	2
Step Height	0.75
Max Slope	45

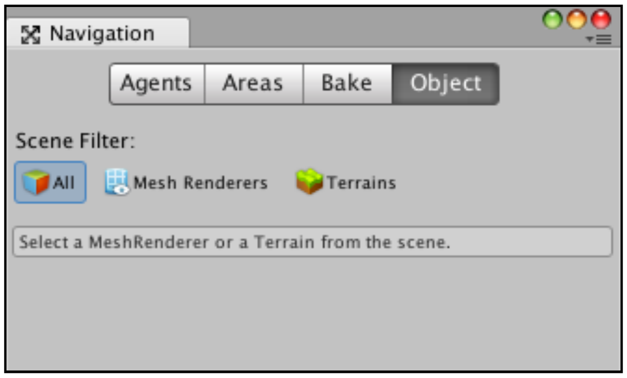
The **Areas** tab allows you to define **Areas**, which are used to specify places that cost more to navigate to. A use case is creating a "Restricted Area" that allows authorized agents in but disallows entry for civilian agents.



	Name	Cost
Built-in 0	Walkable	1
Built-in 1	Not Walkable	1
Built-in 2	Jump	2
User 3		1
User 4		1
User 5		1
User 6		1
User 7		1
User 8		1
User 9		1
User 10		1

The **Objects** tab allows you to select which meshes or objects will be used to generate the NavMesh. When you select a GameObject with a MeshRenderer or Terrain component added to it, you'll see a few parameters associated with the object.

- **Navigation Static**: Determines whether to generate a NavMesh for this GameObject.
- **Generate OffMeshLinks**: Allows Unity to generate off-mesh links by itself.
- **Navigation Area**: Lets you specify which Navigation Area to assign this mesh to — more about this in the Areas tab.

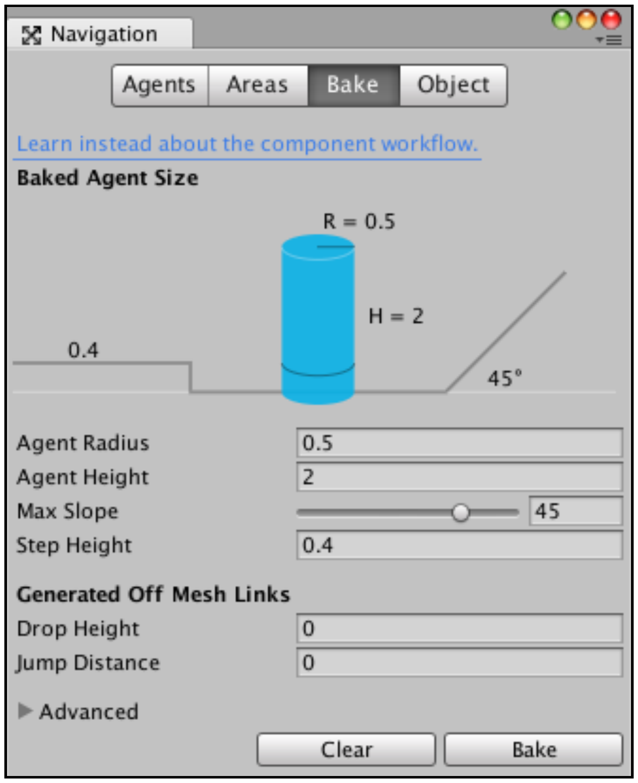


Select a MeshRenderer or a Terrain from the scene.

The **Bake** tab allows you to define the parameters for the agents of your current NavMesh.

- **Radius**: Specifies an agent's width, which could restrict its access to narrow passages on your NavMesh.
- **Height**: Sets an agent's height, which could determine whether it can sneak under ledges and get through doors.

- **Max Slope**: Determines the steepest angle, expressed in degrees, the agent can handle.
- **Step Height**: Defines the height the agent can automatically walk up to, for example, going up a flight of stairs.
- **Generated Off Mesh Links**: Sets the **drop height** at which the mesh will generate an off-mesh link.
- **Jump Distance**: Defines the maximum distance an actor can jump.



Learn instead about the component workflow.

Baked Agent Size

Agent Radius	0.5
Agent Height	2
Max Slope	45
Step Height	0.4

Generated Off Mesh Links

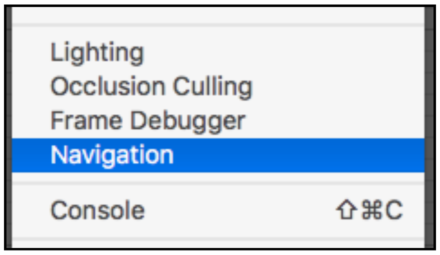
Drop Height	0
Jump Distance	0

At the bottom of the **Bake** tab are two buttons. **Clear** removes the current NavMesh on the selected MeshRenderer or Terrain, and **Bake** generates the NavMesh that will be used by agents in that mesh.

Okay, now it's time to make your very own **NavMesh**!

NavMesh for the AI

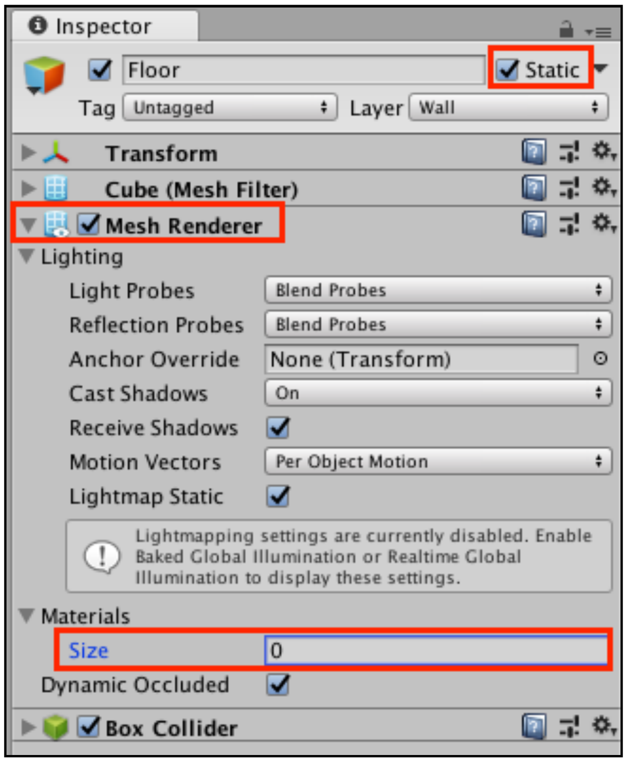
Open the **Navigation** window — it's in the top menu under **Window \ Navigation**.



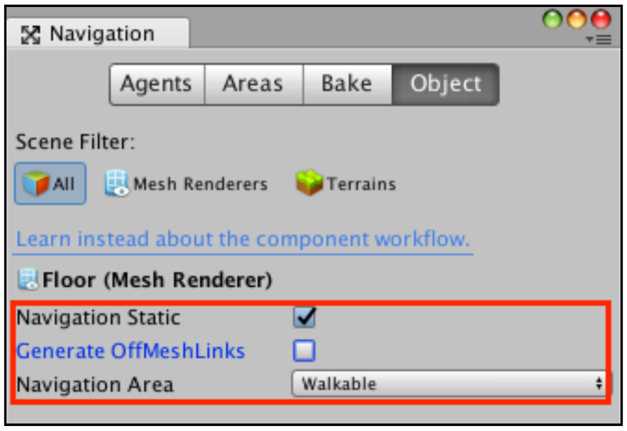
You'll create the NavMesh for Pompadroid with the Floor's MeshRenderer under **Map1 \ Colliders GameObject**.

However, you have a small problem to overcome: The navigation mesh requires an active mesh renderer to calculate the mesh. The Floor has a disabled MeshRenderer because the game doesn't need to display the 3D floor; it only uses it for physics calculations.

Enable the Floor's **MeshRenderer** in the Inspector and set its **Materials** to `0` to activate the MeshRenderer without drawing it on the screen. Also check **Static** at the top of the Inspector, which is a requirement because your calculations are based on static objects in the scene.

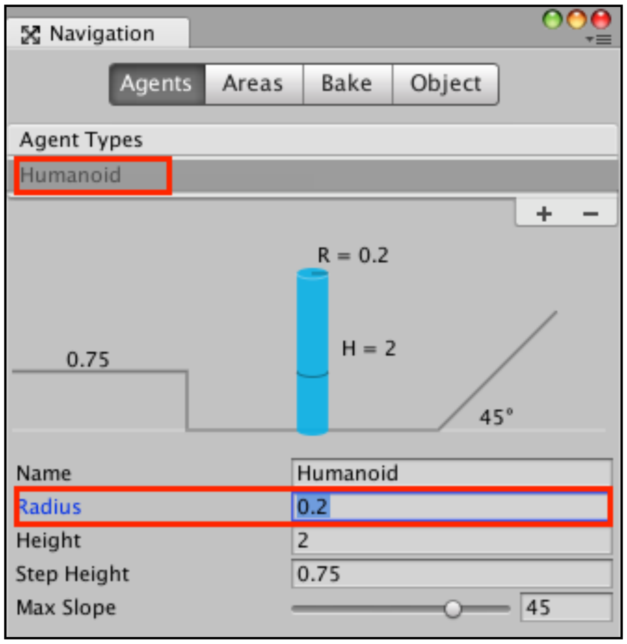

Inspector
Floor ✓ Static
Tag Untagged Layer Wall
Transform
Cube (Mesh Filter)
✓ Mesh Renderer
Lighting
Light Probes: Blend Probes
Reflection Probes: Blend Probes
Anchor Override: None (Transform)
Cast Shadows: On
Receive Shadows: ✓
Motion Vectors: Per Object Motion
Lightmap Static: ✓
Lightmapping settings are currently disabled. Enable Baked Global Illumination or Realtime Global Illumination to display these settings.
Materials
Size: 0
Dynamic Occluded: ✓
✓ Box Collider

Go back to the Navigation window and select the **Floor** GameObject. In the **Object** tab, check **Navigation Static**. Keep **Generate OffMeshLinks** unchecked and make sure **NavigationArea** is **Walkable**.



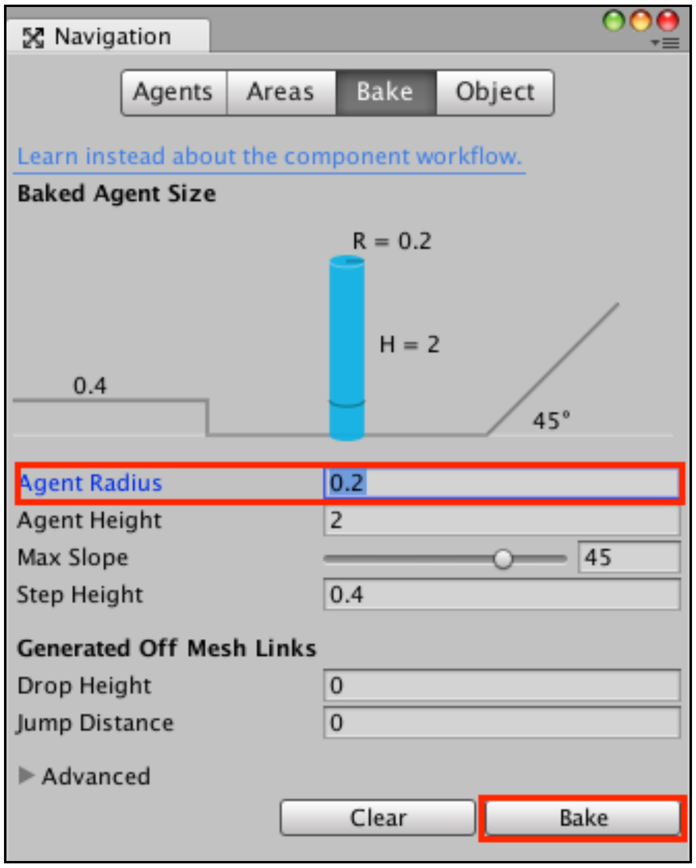
Floor (Mesh Renderer)
Navigation Static ✓
Generate OffMeshLinks ☐
Navigation Area Walkable

Go to the **Agents** Tab, and change the **Humanoid Radius** value to 0.2.



Name Humanoid
Radius 0.2
Height 2
Step Height 0.75
Max Slope 45

Go to the **Bake** tab, set **Agent Radius** to `0.2` and keep the default values shown below:



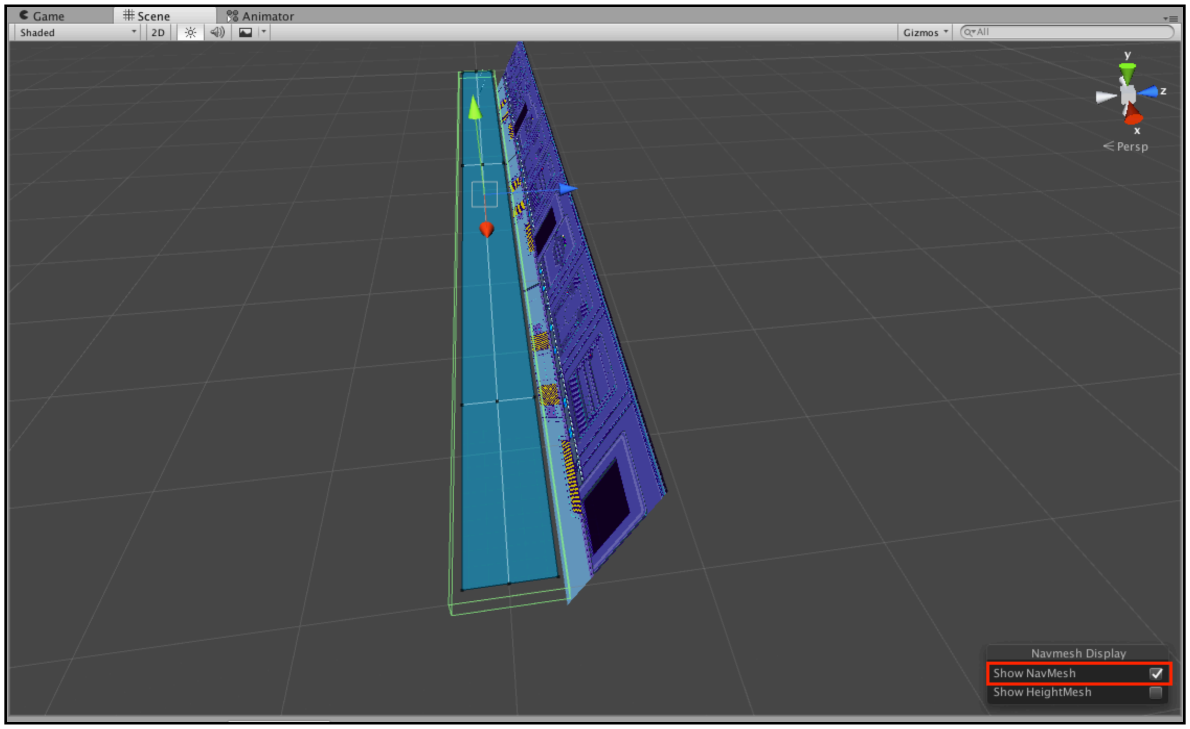
Baked Agent Size

Agent Radius	0.2
Agent Height	2
Max Slope	45
Step Height	0.4

Generated Off Mesh Links

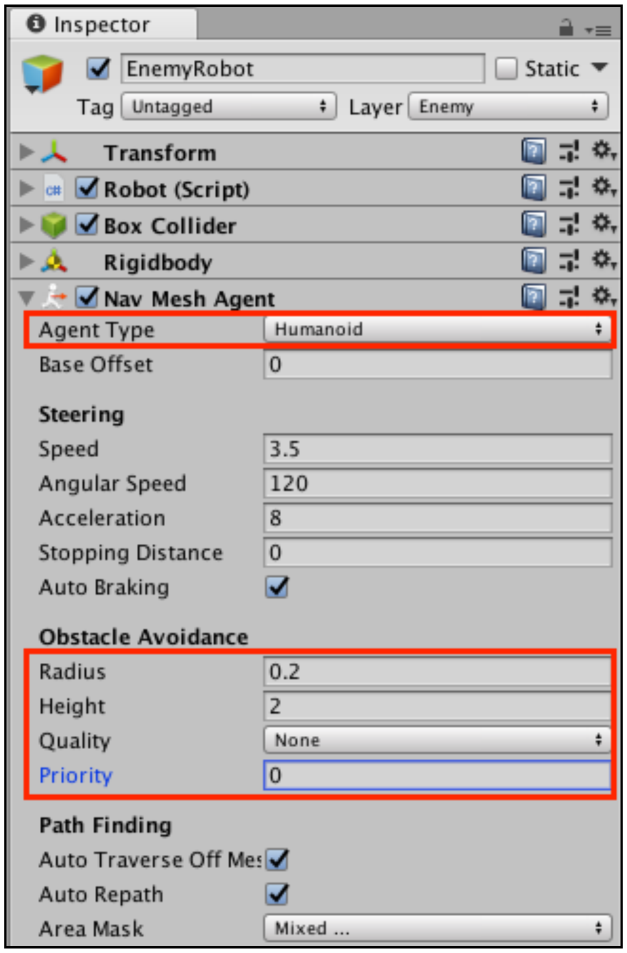
Drop Height	0
Jump Distance	0

Click the **Bake** button at the bottom-right of the Navigation window. It'll create a ready-to-use NavMesh for the agents of the game. If you have **Show NavMesh** enabled, look at the Scene view to see the NavMesh showing as a light blue mesh on top of the floor.



You’re done creating the NavMesh itself. Both the hero and droids will use it to guide themselves around obstacles.

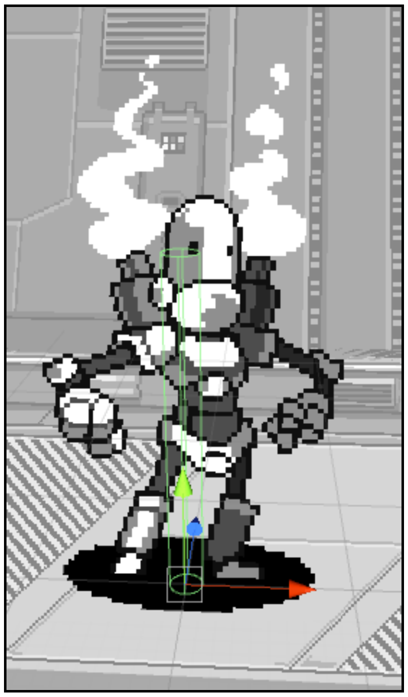
Before the droid can make use of the NavMesh, you must attach a **NavMeshAgent** to it. Select **EnemyRobot** and add a **NavMeshAgent** component to it. Set the **Agent Type** to `Humanoid`, and under the Obstacle Avoidance header, set **Radius** to `0.2`, **Height** to `2`, **Quality** to **None** and **Priority** to `0`.



EnemyRobot ☐ Static
Tag: Untagged
Layer: Enemy

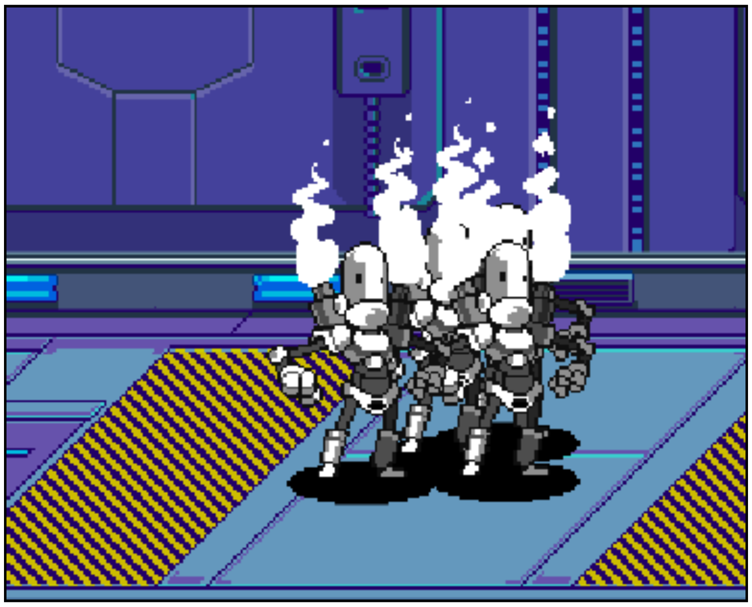
Agent Type	Humanoid
Base Offset	0
Speed	3.5
Angular Speed	120
Acceleration	8
Stopping Distance	0
Auto Braking	✓
Radius	0.2
Height	2
Quality	None
Priority	0
Auto Traverse Off Mes	✓
Auto Repath	✓
Area Mask	Mixed ...

Your new NavMeshAgent will draw a cylinder "collider" of sorts around the EnemyRobot, which serves as the actor's personal marker in the NavMesh.



Obstacle Avoidance should be off because it allows robots to walk on top of each other when navigating the NavMesh. It's an essential behavior because it enables multiple enemies to rush the hero.

You tell me what's more challenging: picking off robots while they stand in line, or punching your way out of a robot gang. Obviously, the answer is the latter, and that's why you turn off Obstacle Avoidance.

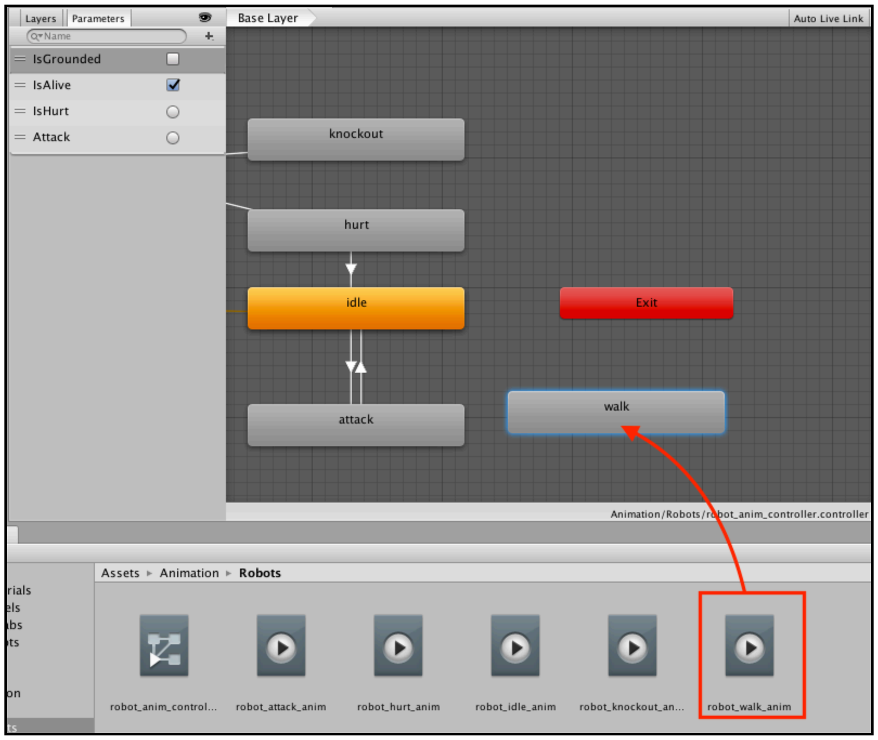


Making the robots walk

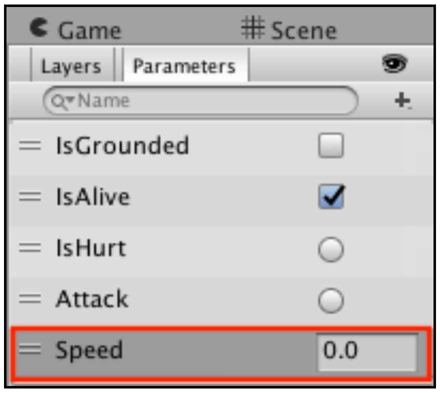
With the NavMesh in place, the droids will be able to navigate the map. Hence, they’re ready for a walk animation.

Import **Robots Walk Animation.unitypackage**, which contains the **robot_walk_anim** clip and all the necessary sprites.

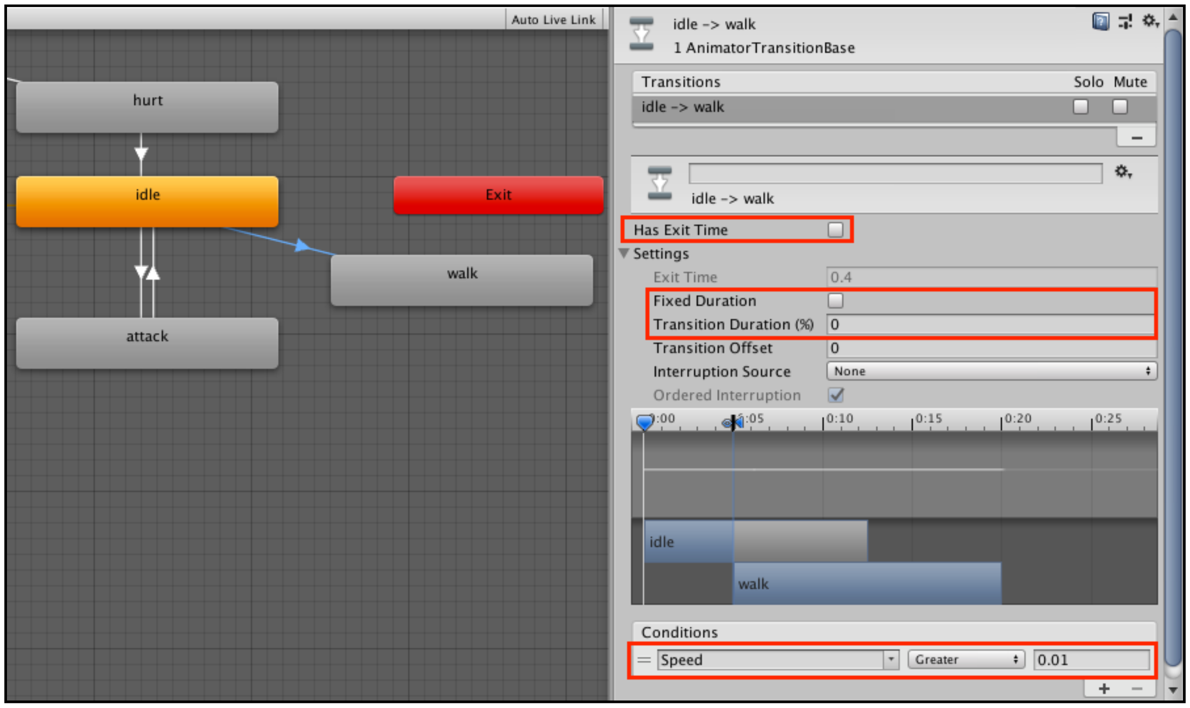
Open the **Animator** window by double-clicking **robot_anim_controller** in the **Assets \ Animation \ Robots** folder. Drag the **robot_walk_anim** clip to the layout area to create a new animation state and rename it **walk**.



Add a new **Float** event parameter named **Speed**.

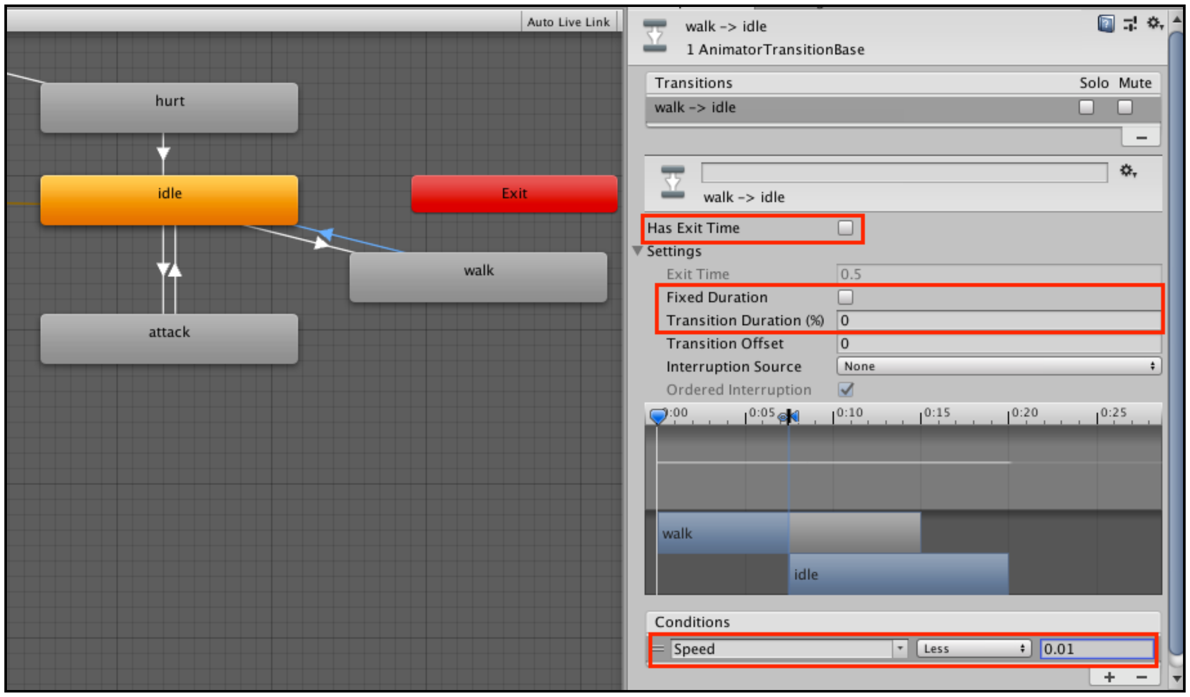


Add a transition from **idle** to **walk**. Uncheck **Has Exit Time** and **Fixed Duration**, set **Transition Duration** to `0`, and add a condition where **Speed** is **Greater than** `0.01`. This state will be responsible for triggering the robot's walk animation.



Has Exit Time ☐
Settings
Exit Time 0.4
Fixed Duration ☐
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓
Conditions
Speed Greater 0.01

Add another transition from **walk** to **idle**. Set the values the same as the previous transition, but set **Speed** at **Less than** `0.01`.



Has Exit Time ☐
Settings
Exit Time 0.5
Fixed Duration ☐
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓
Conditions
Speed Less 0.01

Now you need a script to handle how the robot walks and navigates to a particular point on the map. It’ll handle the actor’s decisions, such as determining if walking is possible. Initially, this **Walker** script will be used by the robot. Eventually, the hero will use it too.

Before creating the script, you’ll append the **Actor** script.

Open `Actor.cs` and add the following method somewhere:

```
public virtual bool CanWalk() {
  return true;
}
```

`CanWalk()` is a method that determines if an actor’s current state allows walking. All actors can walk by default, but derived classes might override this method and `return false`. For instance, the hurt state doesn’t allow walking.

Save the Actor script and create a new **C# script** named **Walker** in the **Assets**\ **Scripts** folder.

Open it and replace the contents of the file with the following code:

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;
using UnityEngine.AI;

//1
[RequireComponent(typeof(Actor))]
public class Walker : MonoBehaviour {
  //2
  public NavMeshAgent navMeshAgent;
  private NavMeshPath navPath;
  private List<Vector3> corners;

  //3
  float currentSpeed;
  float speed;

  //4
  private Actor actor;
  private System.Action didFinishWalk;

  //5
  void Start() {
    navMeshAgent.updatePosition = false;
    navMeshAgent.updateRotation = false;
    actor = GetComponent<Actor> ();
  }
}
```

Here's a breakdown of what you're doing in there:

1. The `Walker` script requires an `Actor` reference. This `Actor` will be moved by the script.
2. These properties are references for handling navigation. First, a public reference to a NavMeshAgent that you'll set in the Inspector. You'll use the next two properties to calculate the pathing.
3. The `speed` and `currentSpeed` parameters are set based on how fast the `Walker` should walk towards the target position.
4. This references the `Actor` that will walk and a `didFinishWalk` callback, which is called when the `Walker` reaches its destination.
5. In `Start()`, you set the initial `speed`, and then set the `actor` by finding the `Actor` via the `GetComponent<Actor>()` method.

The first two lines of `Start()` prevent the NavMeshAgent from updating this GameObject's transform, because the Actor's NavMeshAgent and Rigidbody can modify the transform and create a conflict.

You're going to manually compute the `Rigidbody` position using the NavMeshAgent's pathing and set the Rigidbody's position manually instead of letting the NavMeshAgent automatically change the transform of the Rigidbody.

Add this method to the **Walker** script:

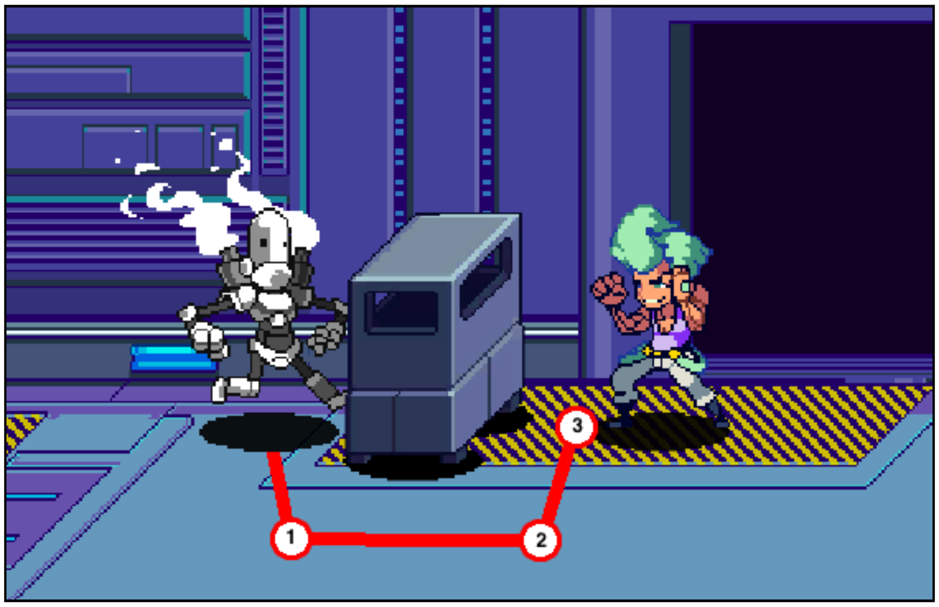
```
public bool MoveTo(Vector3 targetPosition, System.Action callback = null)
{
  navMeshAgent.Warp(transform.position);
  didFinishWalk = callback;
  speed = actor.speed;

  navPath = new NavMeshPath();
  bool pathFound = navMeshAgent.CalculatePath(targetPosition, navPath);

  if (pathFound) {
    corners = navPath.corners.ToList();
    return true;
  }
  return false;
}
```

`MoveTo()` is a method that must be accessible from other classes, so it is public. It calculates how the walker should move to the requested `targetPosition` and returns whether it found a possible path or not.

It teleports the `navMeshAgent` to its current position because the NavMeshAgent and Rigidbody often don’t agree on the actor’s location. Then it calculates how to get to `targetPosition` using the `CalculatePath` method. If a path is found, it stores the path and returns `true`, otherwise, it returns `false`.

The calculated path, in the form of a list of positions, is stored in the `corners` property. These positions represent the “corners” of the path towards the target position.

When the robot wants to get in the hero’s face, it’ll get the following `corners` labeled from 1 to 3. The robot that’s “being walked” by your logic will navigate towards the first corner before moving to the second and third corners.



The `Actor` will use `FixedUpdate()` to actually walk the path so that physics calculations are accurate.

Next, still in the **Walker** script, add these methods:

```
//1
public void StopMovement() {
  navPath = null;
  corners = null;

  currentSpeed = 0;
}

protected void FixedUpdate() {
  //2
  bool canWalk = actor.CanWalk();
  if (canWalk && corners != null && corners.Count > 0) {
    currentSpeed = speed;
    actor.body.MovePosition(
      Vector3.MoveTowards(transform.position, corners[0],
      Time.fixedDeltaTime * speed));

    //3
    if (Vector3.SqrMagnitude(
        transform.position - corners[0]) < 0.6f) {
          corners.RemoveAt(0);
    }

    if (corners.Count > 0) {
      currentSpeed = speed;
      //4
      Vector3 direction = transform.position - corners[0];
      actor.FlipSprite(direction.x >= 0);
    } else {
      //5
      currentSpeed = 0.0f;
      if (didFinishWalk != null) {
        didFinishWalk.Invoke();
        didFinishWalk = null;
      }
    }
  }
  actor.baseAnim.SetFloat("Speed", currentSpeed);
}
```

In here, you've created the `StopMovement()` and `FixedUpdate()` methods the droid needs to get around.

1. `StopMovement` clears the path and stops any movement from the `Walker` script by nullifying `navPath` and `corners` and reducing speed to `0`.

2. `FixedUpdate` manually moves the `Walker` toward the `targetPosition`. First, it checks if `CanWalk()` returns `true` and if a path exists. When both conditions are met, the method moves the Actor's position towards the first of the `corners`.

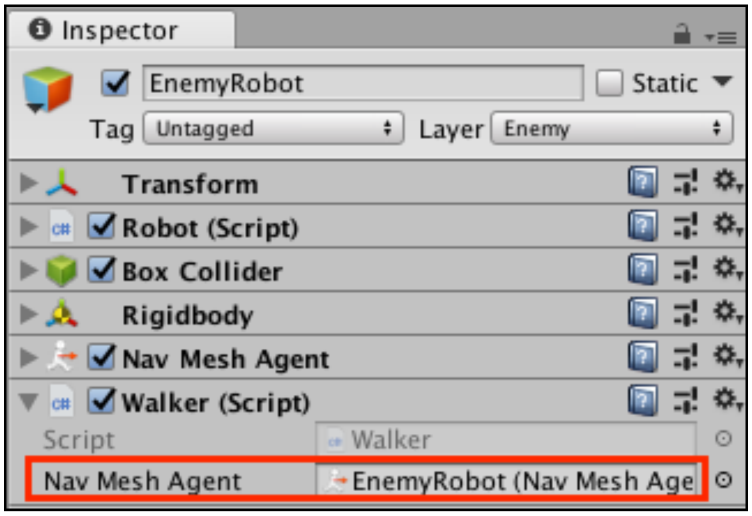
3. Once the walker reaches a corner, you remove that position from the list.

4. You flip the droid around when necessary based on the direction it's headed towards.

5. When `corners` runs out of entries, the `didFinishWalk` callback is triggered. The actor's animator is also updated to the walker's current speed.

Save the Walker script and return to Unity.

Add the **Walker** component to the EnemyRobot and set the **NavMeshAgent** property to the EnemyRobot.



Congratulations, you've just taught your droid to walk. However, it may not seem like you've done a thing! There's no way to test your work because the robot has no controls. You'll work on that next.

If bots only had a brain

You've reached a critical point in this chapter. Soon, you'll give a brain to that mindless, smoldering hunk of metal. But before you dive into it, consider the behavior you want to see in your protégé.

First, think about the states a robot can be in when it sees the hero:

- Stand still
- Wander around the area
- Approach the hero
- Attack the hero

If the robot can reach the hero, then it can take advantage of proximity and launch an attack. However, if the robot is not close and punches the air, it looks stupid — yes, even robots get embarrassed!

Each state should have a different selection probability, and furthermore, these probabilities should differ from one situation to the next.

Consider this scenario:

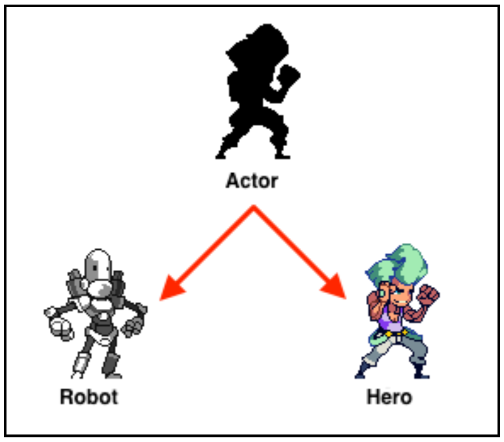
Beep...Beep..
Enemy near.....
70% to attack
30% to stare
Decision: Attack!
Beep...Beep..
Enemy far.....
30% to stare
70% to chase
Decision: Stare.
.....doh!

The robot close to the hero chooses to attack 70 percent of the time. On the other hand, the more distant robot chooses to move closer 70 percent of the time, but in this scenario, it chooses to remain idle.

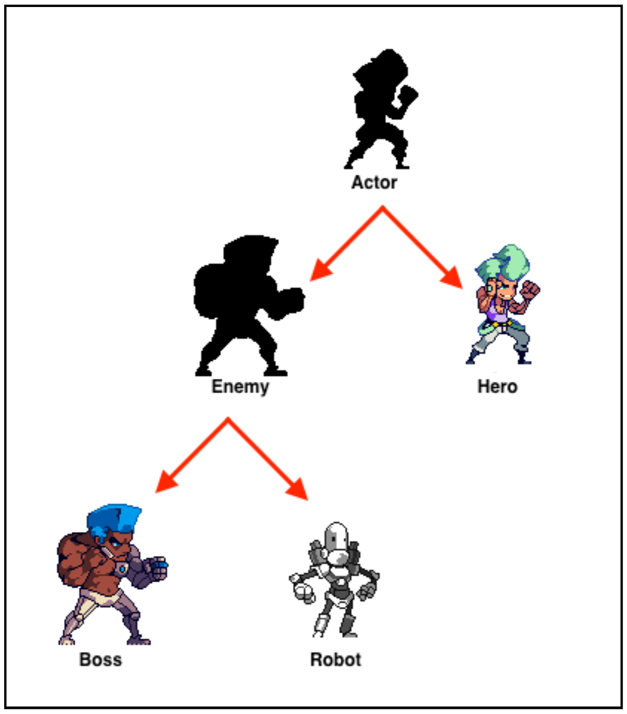
In every situation, the robot should weigh its options and make a choice. The higher the probability of an option, the more likely the robot is to choose it.

You'll create a new script to handle all this decision making. Based on the input of proximity with the hero, the robot should choose whether to attack, approach the hero, wander around or stare blankly into space.

Before you can work on the AI script, you need to modify the robot's hierarchy. Remember how the **Robot** and **Hero** classes derive from a common superclass named **Actor**?



You'll make a new subclass of **Actor** that will be the superclass of the **Robot**, and you'll name it **Enemy**. You new subclass will handle code that powers all the enemies, including the big bad bosses you haven't made yet!



Laying the groundwork for AI

Create a **new C# script** named **Enemy** in the **Assets / Scripts** folder, open it and replace its contents with the following:

```
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class Enemy : Actor {
  //1
  public static int TotalEnemies;
  public Walker walker;

  public bool stopMovementWhenHit = true;

  //2
  public void RegisterEnemy() {
    TotalEnemies++;
  }

  //3
  protected override void Die() {
    base.Die();
    walker.enabled = false;
    TotalEnemies--;
    }
  }
```

1. Enemy has three properties: TotalEnemies holds the amount of live, on-screen enemies, walker is a reference to the Walker script, and stopMovementWhenHit determines if this Enemy should stop moving when it takes damage.

2. Every time an enemy spawns, it calls RegisterEnemy() so the enemy count is correct at all times.

3. In Die(), the Actor's Die() method is called with base.Die() and Walker is disabled. Lastly, TotalEnemies is reduced by one.

You declareTotalEnemies as static so that it becomes a class variable instead of an instance variable. The effect is that all instances of the Enemy share the same value for the integer TotalEnemies.

Add the following methods to the **Enemy** script:

```
  //1
  public void MoveTo(Vector3 targetPosition) {
    walker.MoveTo (targetPosition);
  }

  //2
  public void MoveToOffset(Vector3 targetPosition, Vector3 offset) {
    if (!walker.MoveTo (targetPosition + offset)) {
      walker.MoveTo (targetPosition - offset);
    }
  }

  //3
  public void Wait() {
    walker.StopMovement ();
  }

  //4
  public override void TakeDamage(float value, Vector3 hitVector) {
    if (stopMovementWhenHit) {
      walker.StopMovement ();
    }
    base.TakeDamage(value, hitVector);
  }

  //5
  public override bool CanWalk () {
    return
      !baseAnim.GetCurrentAnimatorStateInfo(0).IsName("hurt");
  }
```

The above methods integrate the walker into the Enemy script:

1. The enemy's MoveTo calls the walker's MoveTo method.

2. `MoveToOffset` is similar to the `MoveTo` method, but it determines whether the enemy can walk to the right (positive offset) or the left (negative offset) of the `targetPosition`. This allows the enemy to move to a free space next to the hero.

3. `Wait()` stops the walker's movement.

4. `TakeDamage` overrides calls to the Actor's `TakeDamage` with the `base.TakeDamage` line. However, it checks if the enemy's walk should be interrupted when it takes damage. Setting `stopMovementWhenHit` to **true** in the Inspector will stop the enemy's movement when it takes a punch.

5. `CanWalk()` is an override that checks whether the hurt animation is currently playing since the enemy should not be able to walk when it's taking damage.

Save the Enemy script and open **Robot.cs** to change its class declaration from this:

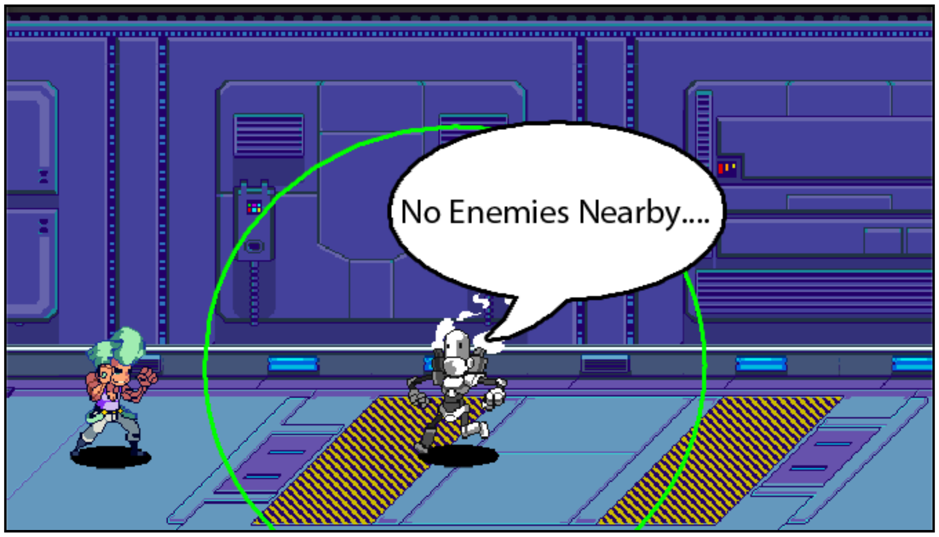
```
public class Robot : Actor {
```

To this:

```
public class Robot : Enemy {
```

Now the robot is a derived class of enemy, which enables movement via the Walker script whenever its AI demands movement.

As previously discussed, in order to decide next steps the robot has to know if the hero is within its proximity. Think of it as the robot's "detection bubble" — the hero officially invades the droid's personal space when he enters the bubble.



This can be accomplished by creating another script that detects whenever the hero overlaps a Sphere collider. This script will be the **HeroDetector** script.

Create a new **C# Script** in the **Assets / Scripts** folder named **HeroDetector**. Replace the contents with the following code block:

```
using UnityEngine;

//1
[RequireComponent(typeof(Collider))]
public class HeroDetector : MonoBehaviour {
  //2
  public bool heroIsNearby;

  //3
  public void OnTriggerEnter(Collider collider) {
    if (collider.tag == "Hero") {
      heroIsNearby = true;
    }
  }

  public void OnTriggerExit(Collider collider) {
    if (collider.tag == "Hero") {
      heroIsNearby = false;
    }
  }
}
```

1. The `HeroDetector` script requires a collider component that detects when the hero overlaps it. Note that it is a generic collider type that allows you to use any subclass of collider.

2. The Boolean `heroIsNearby` serves as a flag when the collider detects the hero. External classes can access this property because it's `public`.

3. Both instances of `OnTrigger` detect colliders marked with the tag **Hero**. These methods also update `heroIsNearby` whenever the hero is detected or lost.

Save HeroDetector, open the **Actor** script and add this method:

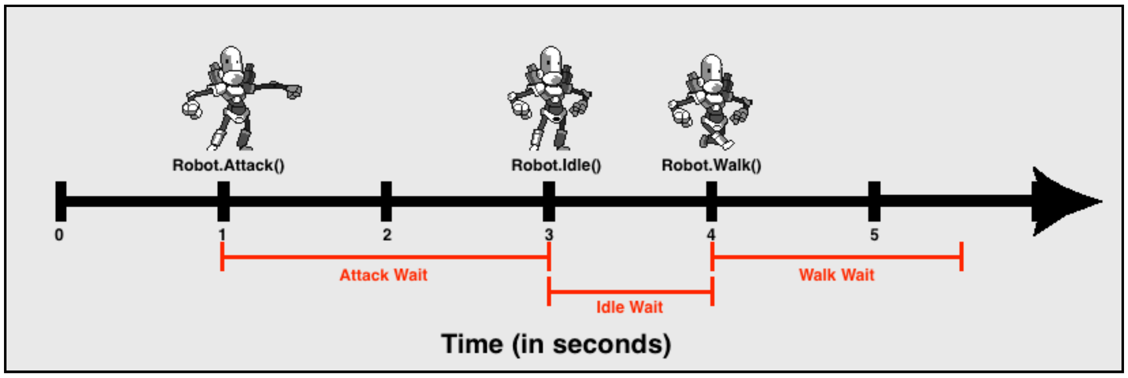
```
public virtual void FaceTarget(Vector3 targetPoint) {
  FlipSprite(transform.position.x - targetPoint.x > 0);
}
```

`FaceTarget` changes the direction an actor faces based on its target point, preventing the droids from the crushing embarrassment of punching the air when standing next to the hero. It calls `FlipSprite` to calculate which direction to face based on whether the target is to the actor's left or right.

Okay, the game is ready for you to write the **EnemyAI** script. Don't forget to **save** the Actor script before proceeding.

Writing the AI

The AI script you're about to write uses timing and probabilities to give the robots limited decision making abilities. Waiting periods between decisions will vary, for instance, the AI will wait longer after it decides to punch than when it decides to do nothing.

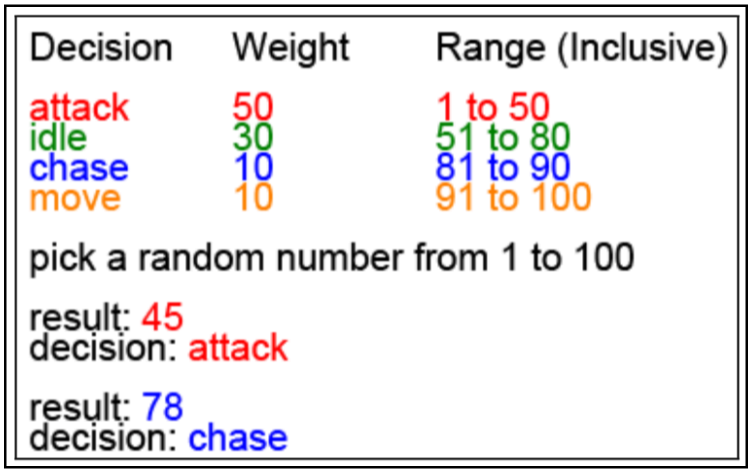


You'll implement weighting system that considers the distance between the robot and hero to help the droid decide what to do. For example, when the robot and hero are close to each other, the robot is more likely to attack than chase. Conversely, if the hero is out of reach, the robot is more likely to chase than attack.

The process is simple:

1. List the actions. Assign a weight to each option according to its probability.
2. Get the total of the weights, and then assign a range to each option.
3. Choose a random number between `1` and the sum of the weights.
4. If the random number is within the action's assigned range, then carry out that action.

In the example below, given the weights of `50` attack, `30` idle, `10` chase and `10` move, a result of `45` means an attack is imminent. A result of `78` means the droid will chase the hero.



Decision	Weight	Range (Inclusive)
attack	50	1 to 50
idle	30	51 to 80
chase	10	81 to 90
move	10	91 to 100

pick a random number from 1 to 100

result: 45
decision: attack

result: 78
decision: chase

Got that? Good. Now on with the script!

Writing the AI script

Create a **new C# script** named **EnemyAI** in the **Assets** / **Scripts** folder. Open it and replace its contents with the following code block:

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

//1
[RequireComponent(typeof(Enemy))]
public class EnemyAI : MonoBehaviour {
  //2
  public enum EnemyAction {
    None,
    Wait,
    Attack,
    Chase,
    Roam
  }

  //3
  public class DecisionWeight {
    public int weight;
    public EnemyAction action;
    public DecisionWeight(int weight, EnemyAction action) {
      this.weight = weight;
      this.action = action;
    }
  }

  //4
  Enemy enemy;
  GameObject heroObj;

  public float attackReachMin;
  public float attackReachMax;
  public float personalSpace;
```

```
    //5
    public HeroDetector detector;

    List<DecisionWeight> weights;

    public EnemyAction currentAction = EnemyAction.None;

    //6
    private float decisionDuration;
  }
```

Here's what you've got in there:

1. The required reference to `Enemy` — the EnemyAI depends on the Enemy script for specific actions, such as attacking or moving towards the hero.
2. A declaration of the `EnemyAction` enumeration — this is a complete list of the AI's potential actions.
3. `DecisionWeight` is a class that stores data for the weighted decision randomizer. It contains a `weight` property and the corresponding `action` property.
4. These variables reference the hero and enemy objects. The float variables will be used to calculate if the enemy will hit the hero if it chooses to attack.
5. `HeroDetector` is a reference to the Hero script that checks if the hero is nearby, and `weights` lists all possible actions when a decision is made. `EnemyAction` is the action that the AI is currently performing.
6. `decisionDuration` is the time the AI must wait between decisions. It's determined by the most recent decision.

Next, also add the following method:

```
  //1
  void Start() {
    weights = new List<DecisionWeight>();
    enemy = GetComponent<Enemy>();
    //2
    heroObj = GameObject.FindGameObjectWithTag("Hero");
  }
```

This is a `Start()` method that:

1. Instantiates the `weights` list and finds the `Enemy` component on the GameObject to which this script is attached.
2. Searches `heroObj` using the static `FindGameObjectWithTag` method — you'll play around with tags and this method later in this chapter.

Add the first of the AI action methods, the `Chase()` method:

```
private void Chase() {
  //1
  Vector3 directionVector = heroObj.transform.position -
    transform.position;
  directionVector.z = directionVector.y = 0;
  directionVector.Normalize();

  //2
  directionVector *= -1f;
  directionVector *= personalSpace;

  //3
  directionVector.z += Random.Range(-0.4f, 0.4f);

  //4
  enemy.MoveToOffset(heroObj.transform.position,
    directionVector);
  decisionDuration = Random.Range(0.2f, 0.4f);
}
```

`Chase()` moves the enemy into attack position. The AI will order the robot to go to the hero's position plus an offset value, moving that metal machine to a point that allows for head-on attack. It's calculated like this:

1. Set an offset vector to the normalized direction vector from the enemy towards the hero. You set `z` and `y`-values to `0` since you only need the `x`-value to determine if the hero is to the robot's left or right.

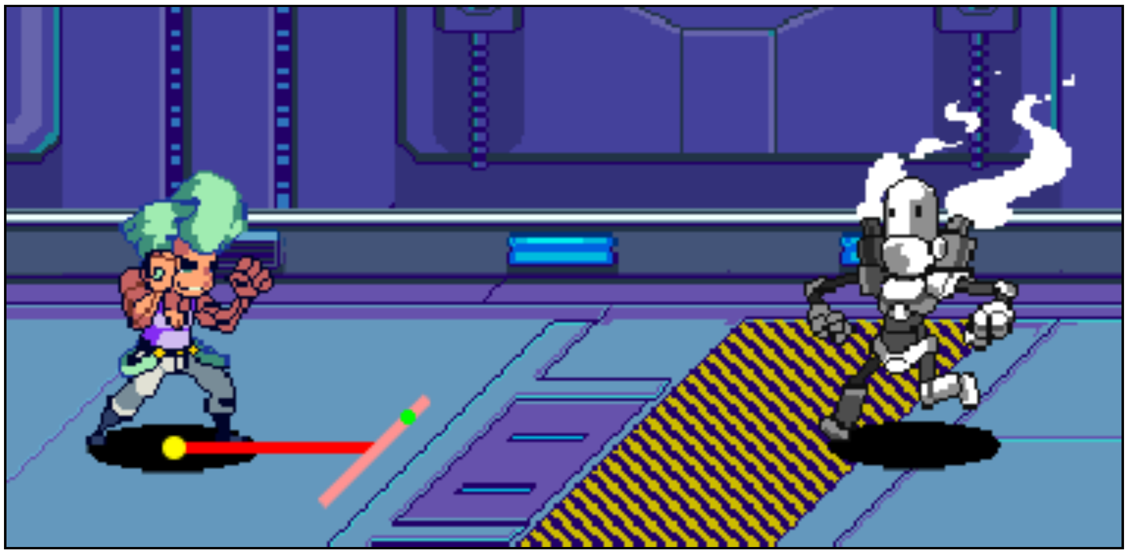
2. Negate the direction vector and multiply this with the `personalSpace` property, which sets the robot's destination to a point in front of the hero — not on top — so that the robot's punches land on the hero.

3. Generate a random value between `-0.4` and `0.4` and set it as the offset's `z`-value. This affords the robot a more natural attack position, instead of always placing it all at the exact same point in front of the hero.

4. Determine the hero's position plus a calculated offset and move the robot there. After that, the AI waits for a random duration between `0.2` and `0.4` seconds before making another decision.

The image below demonstrates the calculation. The **red** line represents `personalSpace`. The higher it is, the farther the robot's position will be from the hero. The **pink** line represents the possible randomized `z`-offset, given the red line's calculations. The **green** point represents a potential offset value based on the hero's position, which is the **yellow** point.

The AI will tell the robot to move to any point on the pink line when chasing the hero, for example, the green point.



Add the remaining AI methods:

```
//1
private void Wait() {
  decisionDuration = Random.Range(0.2f, 0.5f);
  enemy.Wait();
}

//2
private void Attack() {
  enemy.FaceTarget(heroObj.transform.position);
  enemy.Attack();
  decisionDuration = Random.Range(1.0f, 1.5f);
}

//3
private void Roam() {
  float randomDegree = Random.Range(0, 360);
  Vector2 offset = new Vector2(Mathf.Sin(randomDegree),
    Mathf.Cos(randomDegree));
  float distance = Random.Range(1, 3);
  offset *= distance;

  Vector3 directionVector = new Vector3(offset.x, 0, offset.y);
  enemy.MoveTo(enemy.transform.position + directionVector);

  decisionDuration = Random.Range(0.3f, 0.6f);
}
```

`Wait()`, `Roam()` and `Attack()` are methods that represent the remaining actions the enemy can take.

1. `Wait()` calls `enemy.Wait()` to stop the droid's movement. In turn, it reverts to the idle state. This method also randomly sets the value of `decisionDuration` between `0.2` to `0.5` seconds.

2. `Attack()` performs actions that are necessary before the enemy can punch; specifically, it faces the robot toward the hero and calls `enemy.Attack()`. Lastly, `decisionDuration` is longer with this action than any other, giving the hero a bit of time to throw the first punch or evade.

3. `Roam()` generates a random `offset` vector that the robot should move toward. `Vector2(Mathf.Sin(randomDegree), Mathf.Cos(randomDegree))` provides a random vector. Once it's obtained, it's multiplied by a random value between `1` and `3`. The AI makes the robot move to the offset. This method's decision duration is a little longer than `Wait()`, but shorter than `Attack()`.

Add the actual decision-making methods:

```
private void DecideWithWeights(int attack, int wait, int chase, int move)
{
  weights.Clear();

  //1
  if (attack > 0) {
    weights.Add(new DecisionWeight(attack, EnemyAction.Attack));
  }
  if (chase > 0) {
    weights.Add(new DecisionWeight(chase, EnemyAction.Chase));
  }
  if (wait > 0) {
    weights.Add(new DecisionWeight(wait, EnemyAction.Wait));
  }
  if (move > 0) {
    weights.Add(new DecisionWeight(move, EnemyAction.Roam));
  }

  //2
  int total = attack + chase + wait + move;
  int intDecision = Random.Range(0, total - 1);

  //3
  foreach (DecisionWeight weight in weights) {
    intDecision -= weight.weight;
    if (intDecision <= 0) {
      SetDecision(weight.action);
      break;
    }
  }
}

//4
private void SetDecision(EnemyAction action) {
  currentAction = action;
```

```
    if (action == EnemyAction.Attack) {
      Attack();
    } else if (action == EnemyAction.Chase) {
      Chase();
    } else if (action == EnemyAction.Roam) {
      Roam();
    } else if (action == EnemyAction.Wait) {
      Wait();
    }
  }
```

1. `DecideWithWeights` is a helper method that calculates and calls the next `EnemyAction` based on the input parameters. It takes integer weight values for attack, wait, chase and move. If a parameter is greater than `0`, it adds that action and its corresponding weight to the `weights` list.

2. Here you calculate the weighting and obtain a random index between `0` and the `total-1`.

3. Subtract the value of each possible `EnemyAction` weight in the `weights` list from the random index value, until the value is less than or equal to zero. Then choose the last `EnemyAction` and call the `SetDecision` method.

4. `SetDecision` method takes in an `EnemyAction` parameter and calls the respective methods to perform it. For example, if the action variable is set to `EnemyAction.Chase`, the `Chase()` method is called. It also sets the `currentAction` property with the chosen `EnemyAction` value.

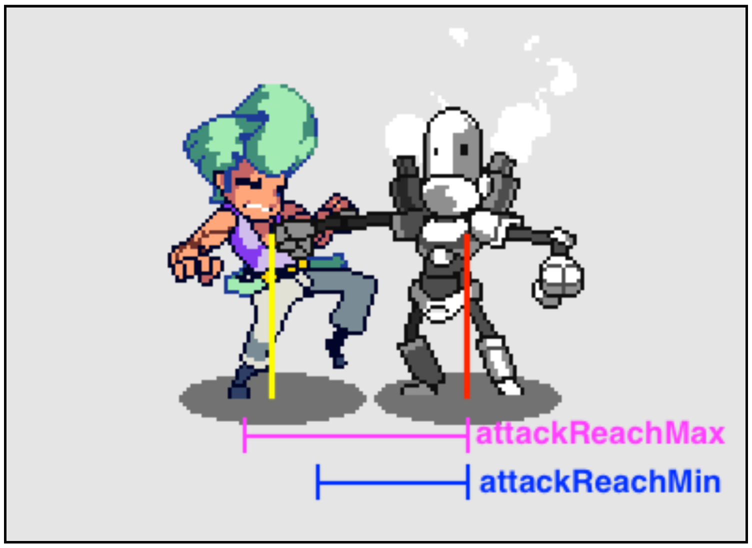
Next, add this `Update()` method:

```
  void Update() {
    //1
    float sqrDistance = Vector3.SqrMagnitude(
      heroObj.transform.position - transform.position);

    //2
    bool canReach = attackReachMin * attackReachMin < sqrDistance
      && sqrDistance < attackReachMax * attackReachMax;

    //3
    bool samePlane = Mathf.Abs(heroObj.transform.position.z -
      transform.position.z) < 0.5f;

    //4
    if (canReach && currentAction == EnemyAction.Chase) {
      SetDecision(EnemyAction.Wait);
    }

    //5
    if (decisionDuration > 0.0f) {
      decisionDuration -= Time.deltaTime;
    } else {
      if (!detector.heroIsNearby) {
        DecideWithWeights(0, 20, 80, 0);
      } else {
        if (samePlane) {
          if (canReach) {
            DecideWithWeights(70, 15, 0, 15);
          } else {
            DecideWithWeights(0, 10, 80, 10);
          }
        } else {
          DecideWithWeights(0, 20, 60, 20);
        }
      }
    }
  }
```

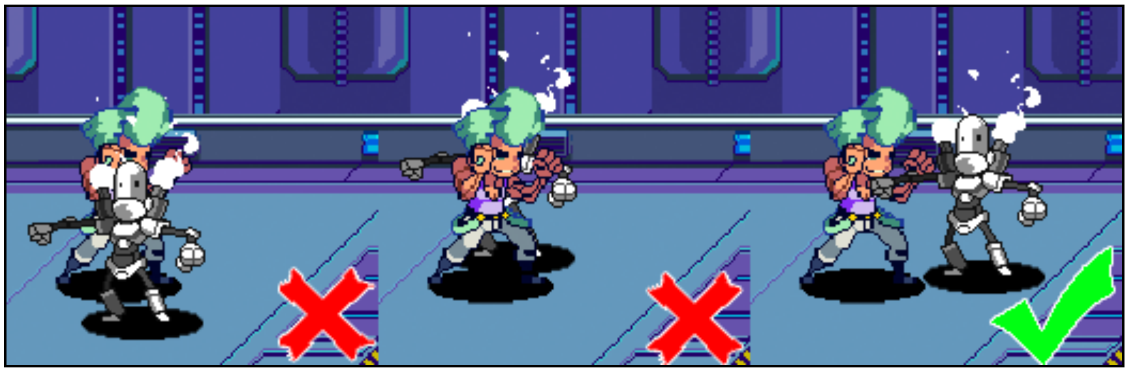
The `Update()` method handles the AI's logic. It tells the robot when to make decisions and how to weigh them. Warning: This explanation is lengthy and detailed but comes with pictures!

1. First, it calculates the distance between the hero and enemy. Note that it only calculates the squared distance, because the square root operation is expensive and unnecessary for distance comparisons. When you don't need the actual distance, it's fine to compare the squared distance or square magnitudes — it's a little optimization that helps Pompadroid run buttery-smooth on slower devices. Next, it checks the distance from the enemy to the hero. The diagram below illustrates this: The red and yellow lines represent the positions of the hero and the robot.

2. Then the script sets the `canReach` Boolean when the distance between the hero and robot falls between `attackReachMin` and `attackReachMax`, as shown by the pink and blue lines below. This is important. When the hero is outside of the minimum and maximum, the robot's attacks won't connect.



3. The `samePlane` Boolean variable is set based on whether the two `z`-positions are within `0.5` units of each other. This variable addresses cases when the hero is technically close enough to punch, but its position on the `z`-axis makes it impossible to do anything but punch the air. Consider the below image — there's no way the droid can punch the hero in the first two cases because it can only throw sideways punches.



4. If the droid could reach the hero but it's currently in a chase, this stops the chase and initiates a wait action.
5. Whenever `decisionDuration` is less than zero, the AI makes a decision based on the variables calculated before. If the hero is not close, the AI will only decide between doing nothing and chasing. When the hero is close enough to strike and on the same plane, this logic gives more weight to *attack* than the other options. When the hero is reachable but on a different plane, then the logic will move the robot into a position where it *can* hit the hero.

Save your work, open the **Enemy** script and add this to the top:

```
public EnemyAI ai;
```

This variable declaration adds a reference to the EnemyAI.

Find the `Die()` override method, and insert the following code after the `base.Die()` line:

```
ai.enabled = false;
```

This disables the EnemyAI when hero puts the droid out of its misery.

Whew!! You just finished scripting the AI. **Save** all your scripts and return to Unity.

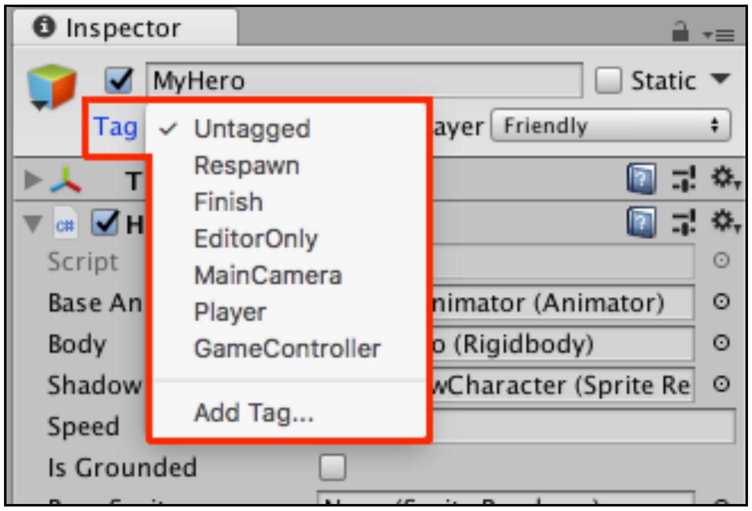
Tag: It's just a nickname

You're close to assembling your new smart robot with its EnemyAI. But before you do, you need to give the robot a way to find the hero at runtime. You'll use **tags**, which are words you associate with specific GameObjects to make it easy to find them in a scene.

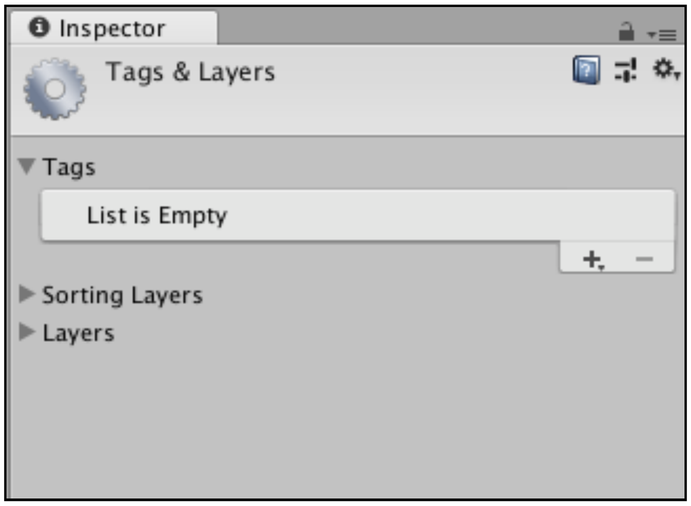
Consider the following line in the `Start()` method of the EnemyAI script; it looks for a GameObject tagged with `"Hero"`.

```
heroObj = GameObject.FindGameObjectWithTag("Hero");
```

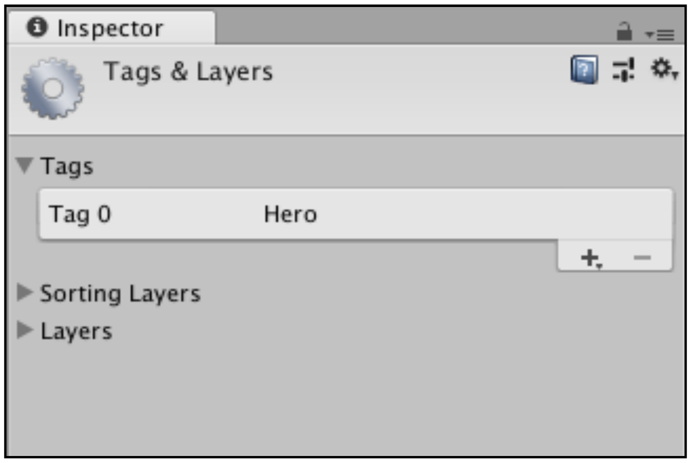
Select **MyHero**, open the **Tag** drop-down under the GameObject name in the Inspector and select **Add Tag**. You can also find these settings via **Edit** \ **Project Settings** \ **Tags & Layers** in the top menu.



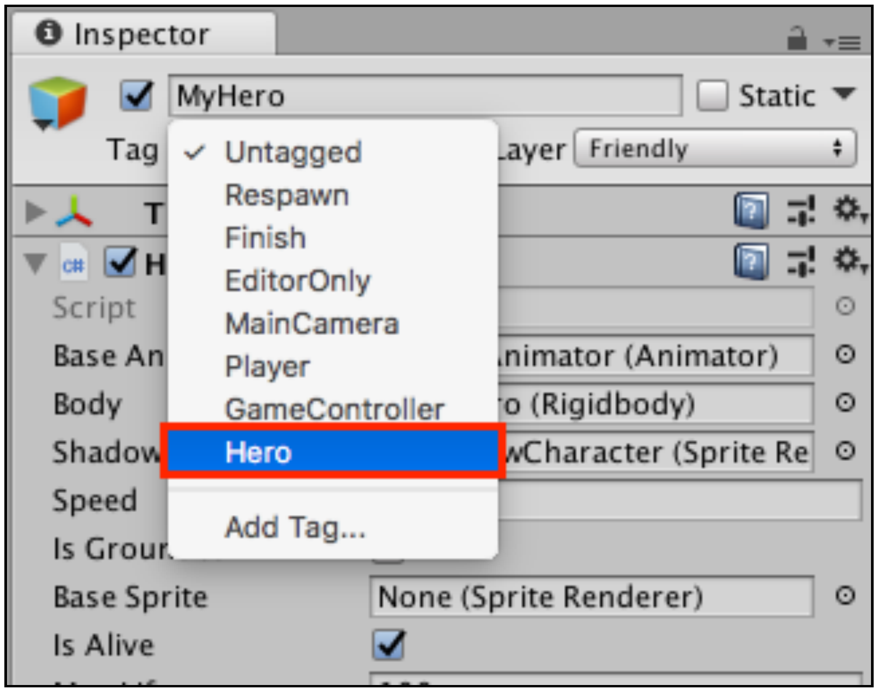
You'll see the **Tags & Layer** settings in the Inspector:



Click the **+ icon** and add a tag named **Hero**. Voila!



Open the tag drop-down again for MyHero to confirm the tag is there. Select **Hero** to apply the tag to **MyHero**.



Now components can locate MyHero by searching for the Hero tag!

Stop the robot infighting

One last detail before you assemble your droid: You need to make sure that robots can't hurt each other. Your fancy new smartdroid should know better than to beat up on its own kind.

Open **Actor.cs** and find the `DidHitObject` method. Replace this:

```
if (actor != null && actor.CanBeHit()) {
```

With this:

```
if (actor != null && actor.CanBeHit() &&
  collider.tag != gameObject.tag) {
```

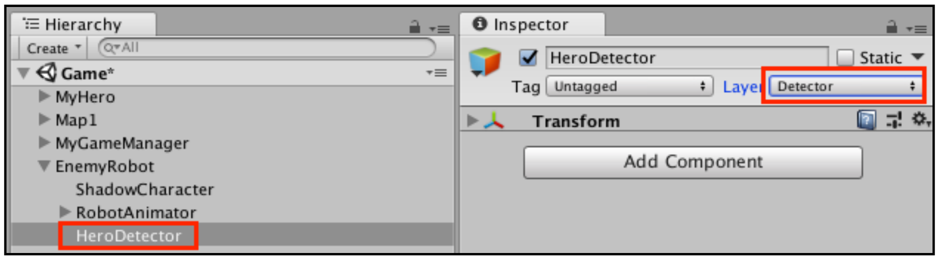
Actors with the same tag can no longer punch each other.

With all the scripts done, it's time to add sentience to our friend.

Assemble all the robots!

First, you need a **HeroDetector** that allows the robot to feel the presence of its enemy, the hero.

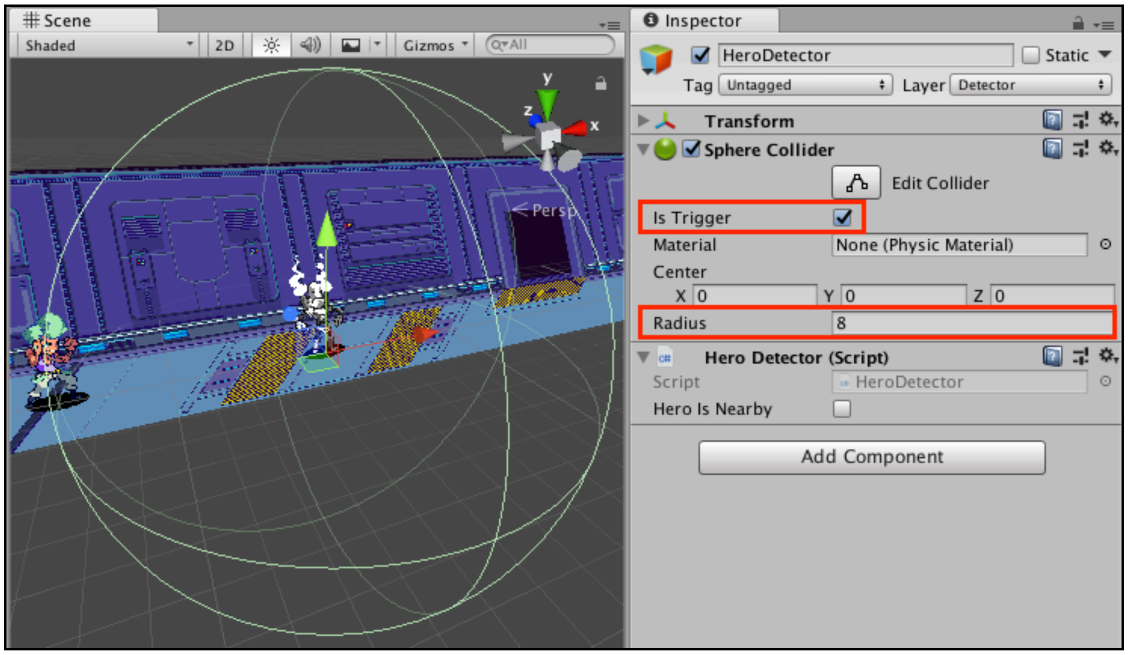
Create a new **Empty GameObject** named **HeroDetector**. Make it a child of **EnemyRobot** and reset its **Transform**. Set its **Layer** to **Detector**.



Add a **Sphere Collider** and a **HeroDetector** component to it. Check **Is Trigger** in the Sphere Collider and set its **Radius** to `8`.

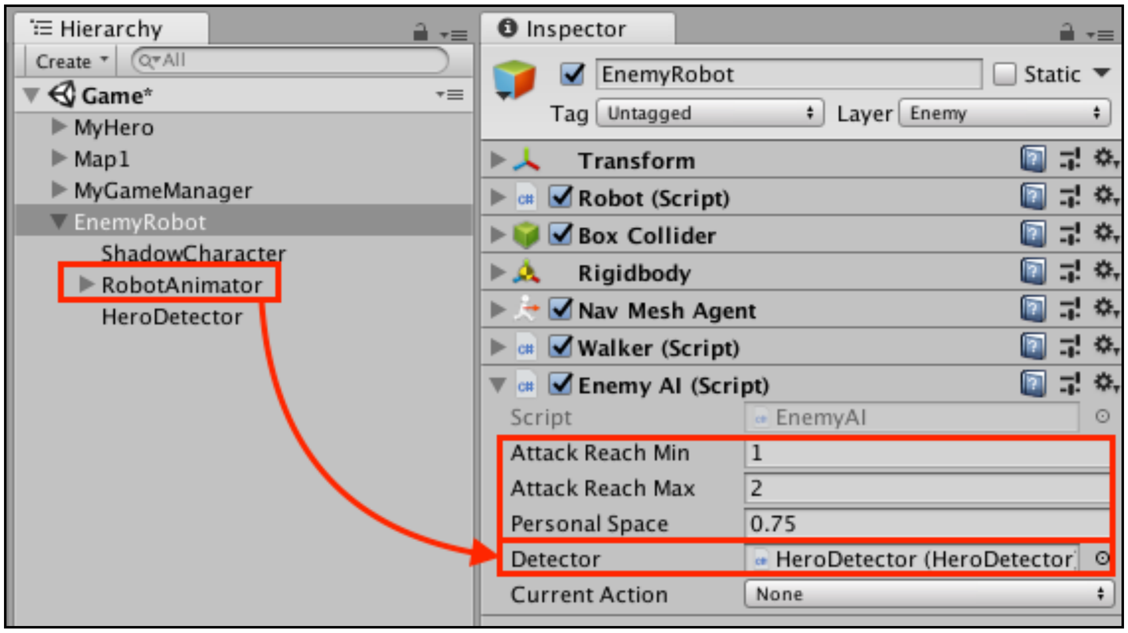
> **Note**: Add the Sphere Collider first because it's a required component of the HeroDetector. Although Unity adds required components automatically, it doesn't know whether you want a box, sphere or capsule when you add a collider.

Look at the Scene view. The collider is big enough to detect the hero even when he's pretty far away.



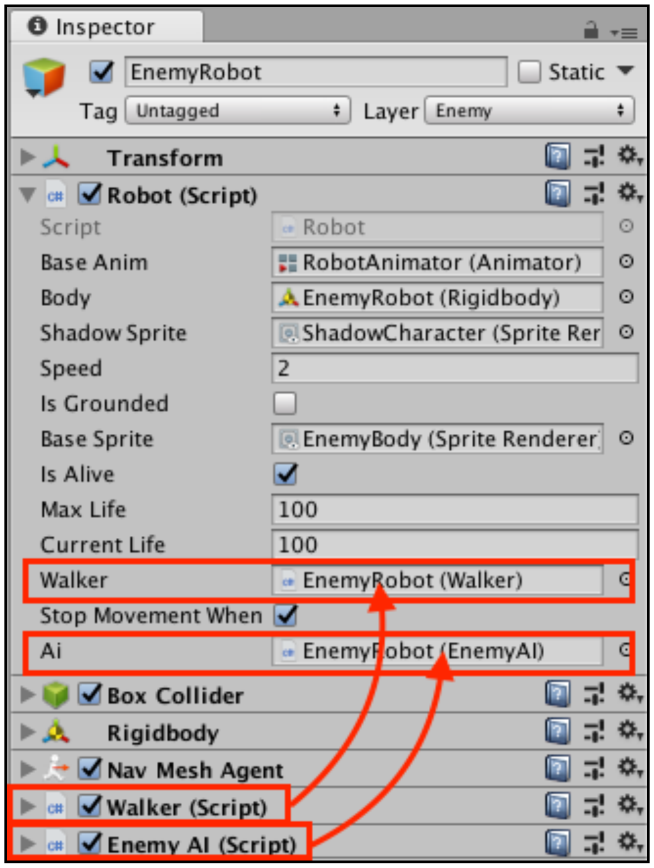
HeroDetector
Tag: Untagged
Layer: Detector
Is Trigger: ✓
Material: None (Physic Material)
Center: X 0, Y 0, Z 0
Radius: 8
Hero Detector (Script)
Script: HeroDetector
Hero Is Nearby: ☐

Add the **EnemyAI** script to **EnemyRobot**. On the newly added EnemyAI component, set **Attack Reach Min** to 1, **Attack Reach Max** to 2, **Personal Space** to `0.75`, and then assign the **HeroDetector** child as the **Detector**.



EnemyRobot
Tag: Untagged
Layer: Enemy
Script: EnemyAI
Attack Reach Min: 1
Attack Reach Max: 2
Personal Space: 0.75
Detector: HeroDetector (HeroDetector)
Current Action: None

Select **EnemyRobot** and look at the Robot component in the Inspector. Because you changed its superclass to **Enemy**, you now need a few references to the component. Set **Ai** and the **Walker** properties by dragging the **EnemyAI** and **Walker** components into their respective fields.


Inspector

EnemyRobot ☐ Static
Tag Untagged Layer Enemy

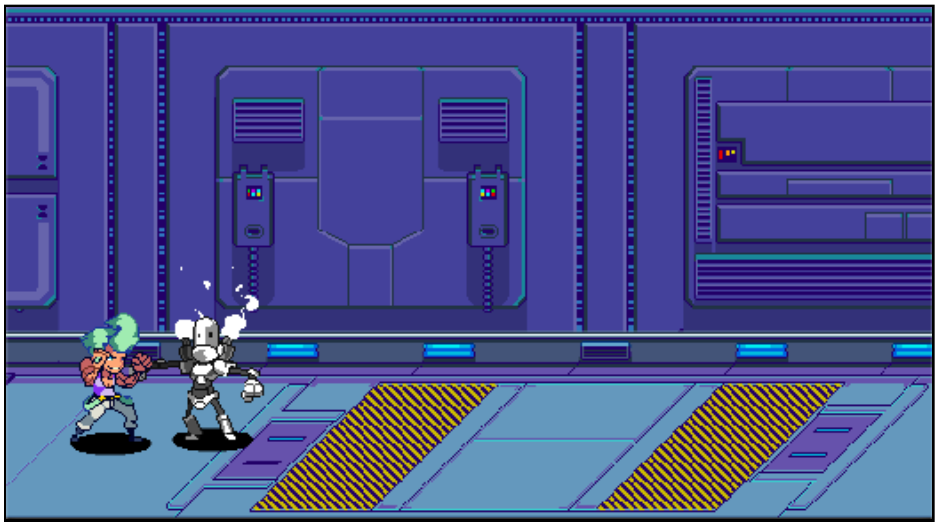
Transform

Robot (Script)

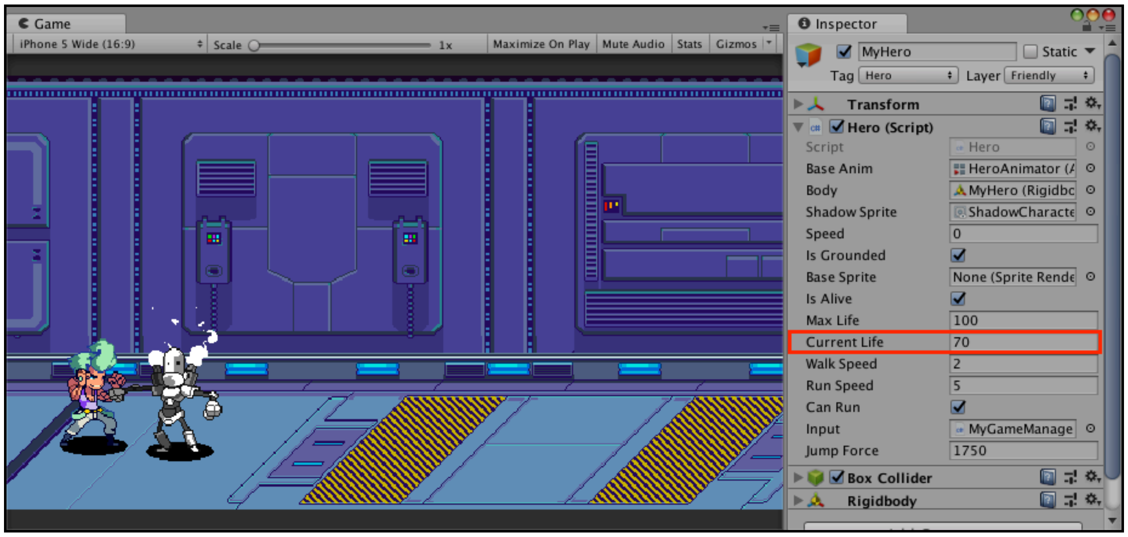
Script	Robot
Base Anim	RobotAnimator (Animator)
Body	EnemyRobot (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Rer
Speed	2
Is Grounded	☐
Base Sprite	EnemyBody (Sprite Renderer
Is Alive	✓
Max Life	100
Current Life	100
Walker	EnemyRobot (Walker)
Stop Movement When	✓
Ai	EnemyRobot (EnemyAI)

Box Collider
Rigidbody
Nav Mesh Agent
Walker (Script)
Enemy AI (Script)

Save the project and run the game. You'll notice the robot moves around and tries to engage you (the hero) in combat! However, you can't tell if the hero is hit because he doesn't flinch when punched. What a badass!

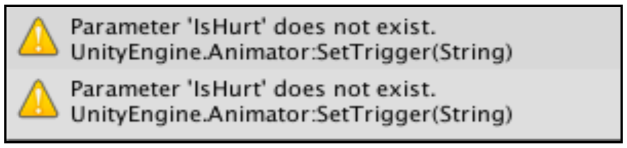


You can tell that the punches are landing because the hero's `currentLife` goes down whenever he gets hit.



```
MyHero                 Static
Tag Hero     Layer Friendly
Transform
Hero (Script)
Script            Hero
Base Anim         HeroAnimator (A
Body              MyHero (Rigidbc
Shadow Sprite     ShadowCharacte
Speed             0
Is Grounded       ✓
Base Sprite       None (Sprite Rende
Is Alive          ✓
Max Life          100
Current Life      70
Walk Speed        2
Run Speed         5
Can Run           ✓
Input             MyGameManage
Jump Force        1750
Box Collider
Rigidbody
```

Also, you get this warning whenever that droid lands a punch:

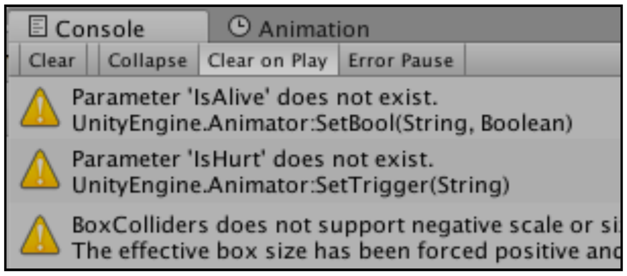


```
Parameter 'IsHurt' does not exist.
UnityEngine.Animator:SetTrigger(String)
Parameter 'IsHurt' does not exist.
UnityEngine.Animator:SetTrigger(String)
```

Now, why is that? Is the hero impervious to pain? Not so much. The hero's animator just doesn't have any support for a hurt animation.

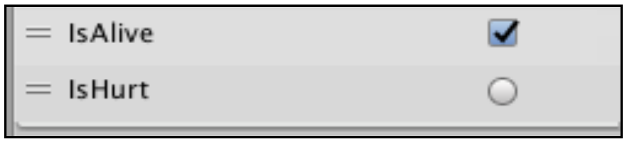
Heroes feel pain too

You're almost done with this chapter! In this final section, you'll make the hero react appropriately when punched. First, you'll clear up these annoying warnings:



```
Parameter 'IsAlive' does not exist.
UnityEngine.Animator:SetBool(String, Boolean)
Parameter 'IsHurt' does not exist.
UnityEngine.Animator:SetTrigger(String)
BoxColliders does not support negative scale or si
The effective box size has been forced positive and
```

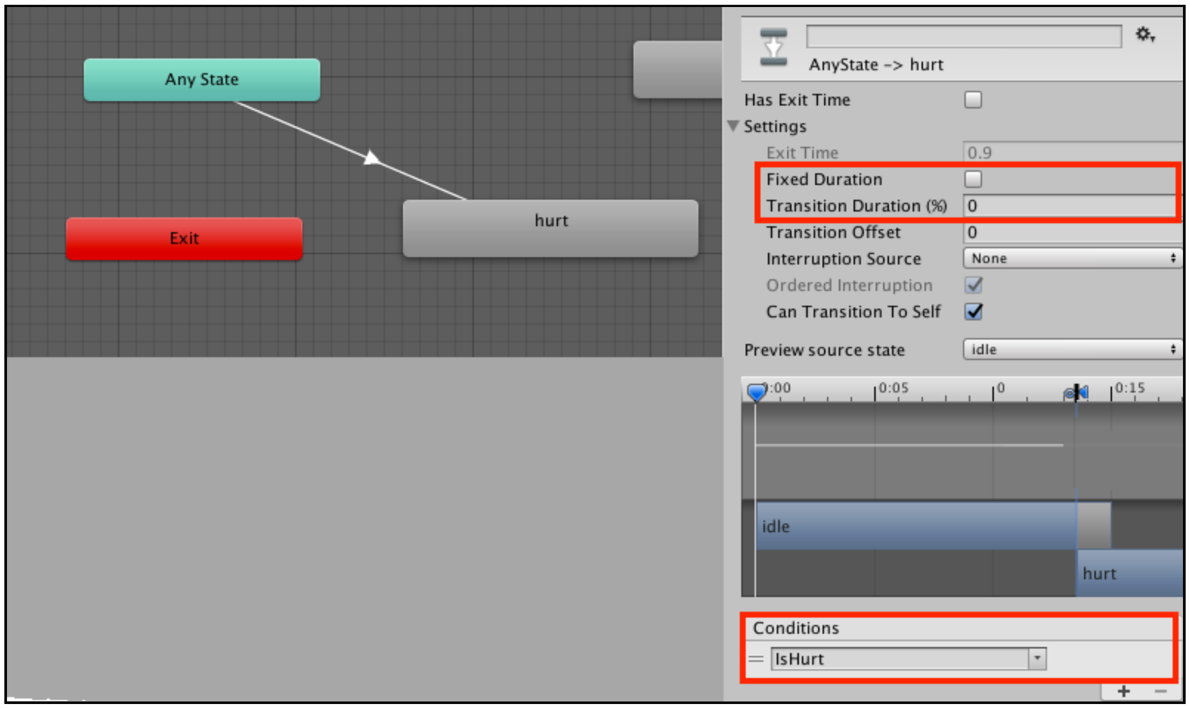
Double-click **hero_anim_controller** in the **Animation \ Hero** folder to open the **Animator** window. Add the two missing parameters: an **IsAlive** Boolean and an **IsHurt** trigger. Set the default value of **IsAlive** to **true** — you don't want to play as an undead hero, right? That should quell the warnings.


IsAlive ✓
IsHurt

Next, import **Hero Hurt and Death Animation.unitypackage** which contains the animation clips and sprites for the hero's hurt and death states.

In the Animator window for the **hero_anim_controller**, drag the **hero_hurt_anim** clip to the state machine near **Any State** and rename it **hurt**.

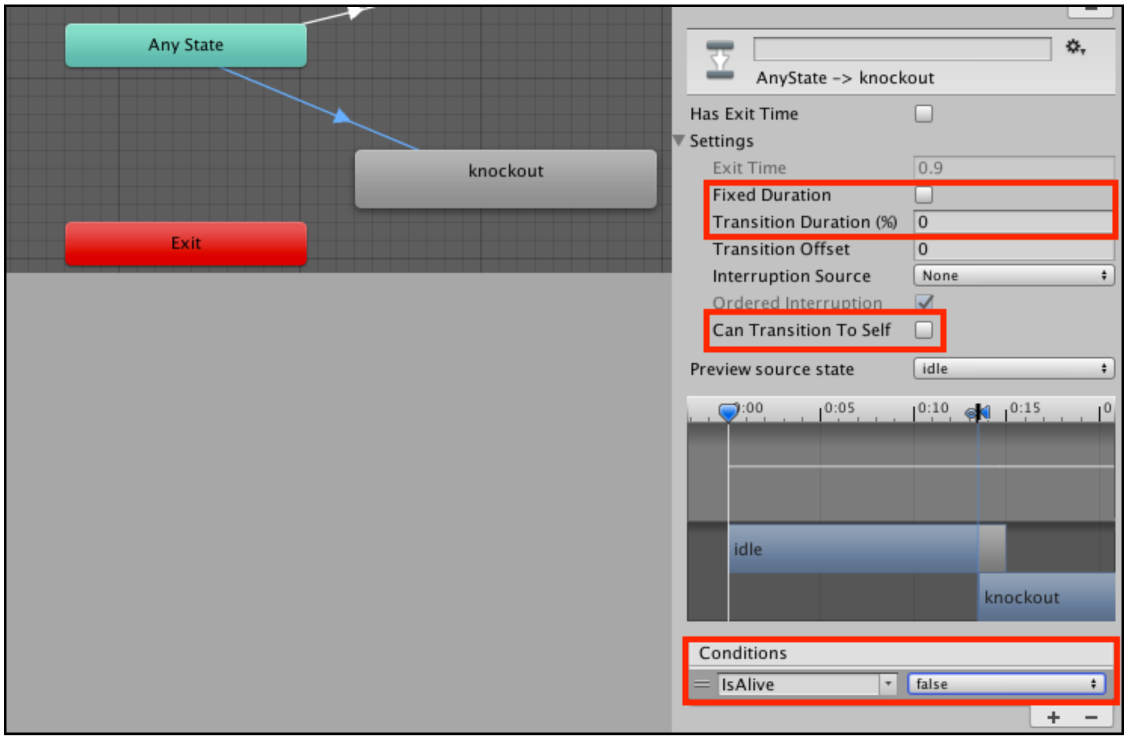
Add a transition from **Any State** to **hurt**. Uncheck **Fixed Duration**, set **Transition Duration** to `0` and add a new condition using the **IsHurt** trigger.



AnyState -> hurt
Has Exit Time ☐
Settings
Exit Time 0.9
Fixed Duration ☐
Transition Duration (%) 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓
Can Transition To Self ✓
Preview source state idle
Conditions
IsHurt

Note: Feel free to drag the animation states around so you can visualize the state machine. The transitions will not change when you rearrange things.

Add **hero_knockout_anim** and rename it **knockout**. Add a transition from **Any State** to **knockout**. **Uncheck Fixed Duration**, set **Transition Duration** to `0` and add a new condition with **IsAlive** set to **false**. Also **uncheck** the **Can Transition To Self** parameter.



AnyState -> knockout

Has Exit Time ☐

Settings

Exit Time 0.9

Fixed Duration ☐

Transition Duration (%) 0

Transition Offset 0

Interruption Source None

Ordered Interruption ✓

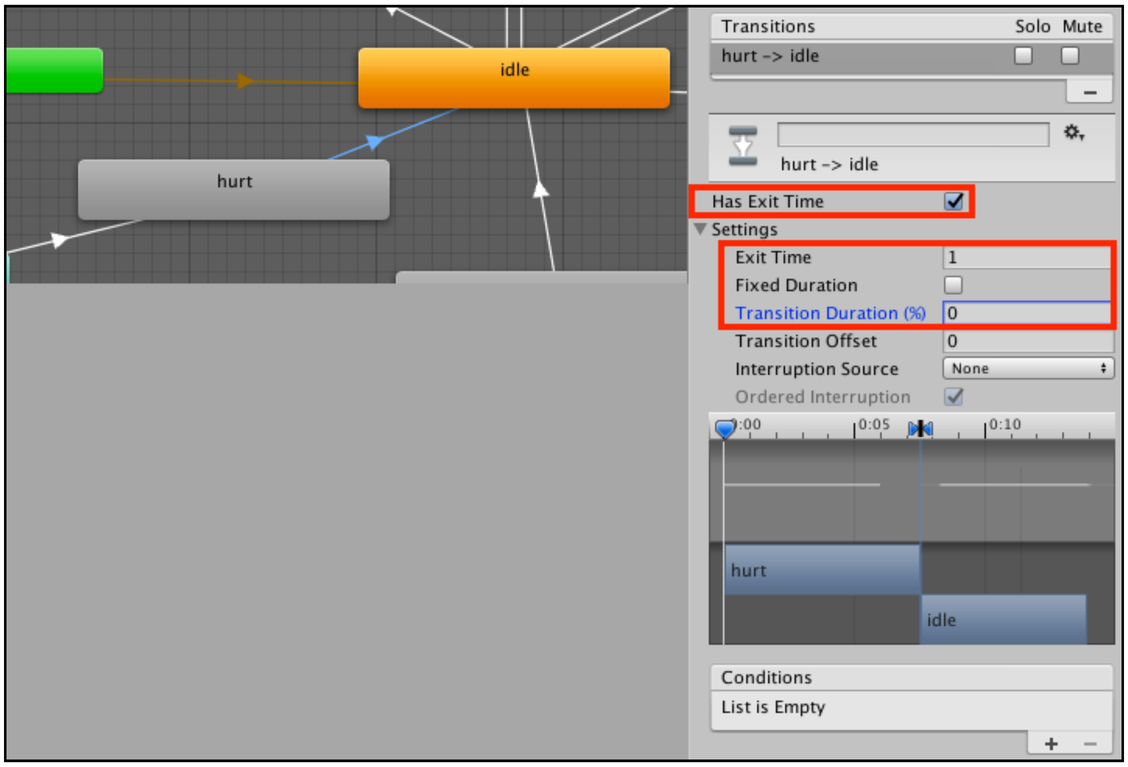
Can Transition To Self ☐

Preview source state idle

Conditions

IsAlive false

Lastly, add a transition from **hurt** to **idle** to return the hero to idle when the hurt animation finishes. Keep **Has Exit Time** checked, set **Exit Time** to `1`, uncheck **FixedDuration** and set **Transition Duration** to `0`.



Transitions	Solo	Mute
hurt -> idle	☐	☐

hurt -> idle

Has Exit Time ✓

Settings

Exit Time 1

Fixed Duration ☐

Transition Duration (%) 0

Transition Offset 0

Interruption Source None

Ordered Interruption ✓

Conditions

List is Empty

You just completed the Hero’s Animator. Open **Hero.cs** to add a few lines of code that will prevent any further action when the hero dies.

In the `Update()` method, insert the following code after the `base.Update();` statement:

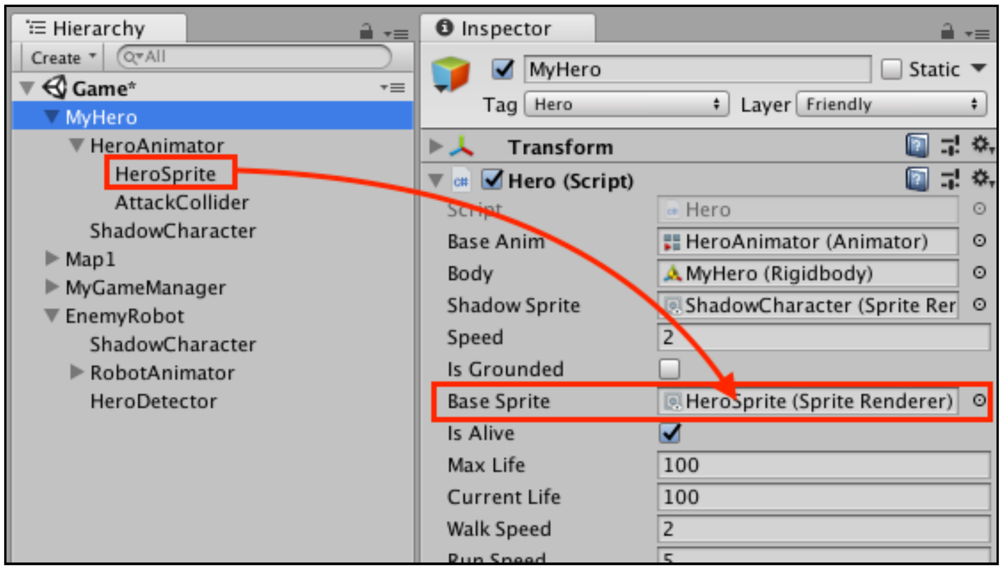
```
if (!isAlive) {
  return;
}
```

This prevents the hero from handling button presses when he’s dead.

Next, **insert** the same `if` statement at the start of the `FixedUpdate()` method to disable horizontal movement when the hero is dead.

Save the script and return to Unity. The only thing missing is a reference in MyHero.

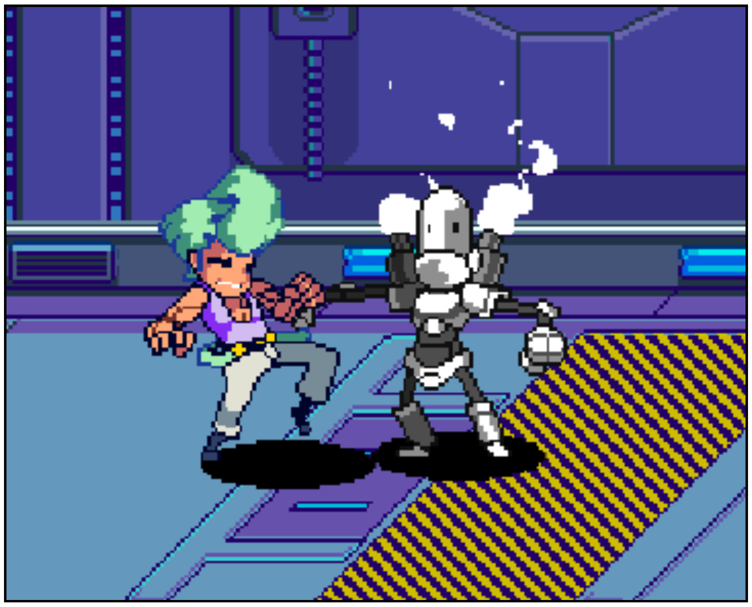
Set the **Base Sprite** parameter of MyHero’s **Hero** component to the **HeroSprite’s SpriteRenderer**:



MyHero
Tag Hero
Layer Friendly

Script	Hero
Base Anim	HeroAnimator (Animator)
Body	MyHero (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Rer
Speed	2
Is Grounded	☐
Base Sprite	HeroSprite (Sprite Renderer)
Is Alive	✓
Max Life	100
Current Life	100
Walk Speed	2
Run Speed	5

Save the scene and the project and try it out! That pompadoured protagonist has lost his poise. He flinches just like the robot when he takes a punch.



Without doing any additional scripting, the hero follows the same rules of life as the droid. Each punch reduces the hero's `Current Life` parameter by `10` and when it reaches `0`, he dies. He also flickers to signify his passing.

Where to go from here?

Great job! You've added the AI to the droid and now it can attack and make other decisions with ease. The hero became more realistic with wincing, flickering and a diminishing life value when he's under attack. Now you've got a game!

Reflecting on what you've done in this chapter:

- Implemented animations for a variety of actions for both the hero and enemy.
- Used pathfinding to let your droids navigate the map intelligently without human intervention.
- Created decision-based AI that lets the droids choose to attack, approach or do nothing based on their proximity to the hero and some nifty weighting logic.
- Used tags to enable easy searches for specific GameObjects, like MyHero.
- Made the hero a mortal with animation and the same code as you did for the robot — clever!

That was a long chapter but look at what you did. You wrote AI! Furthermore, you took the game from barely interesting to challenging. That wandering droid can defend itself and the hero isn’t impervious to attack.

But something feels lacking. What’s that? You want more metal to shred? You got it. Soon you’ll have an army of enemies!

In the next chapter, you’ll learn how to create a more interesting game by mixing combat scenarios with walking segments. Up next, game progression and creating the game’s playlist!

Chapter 7: Pompadroid's Playbill

Up to this point, Pompadroid hasn't lived up to basic expectations you'd have for any game. Sure, you have a main character and an opponent, but there's nothing to do outside of beating that sad, lonely robot into oblivion. From this point forward, that's all going to change.

In this chapter, you'll create data that you'll use to make levels where you can do cool stuff, such as specify where the hero can run and where he needs to fight. Also, you'll animate the hero's entry and exit in the scene. Moreover, you'll add a transition to a more challenging (but fun!) second level.

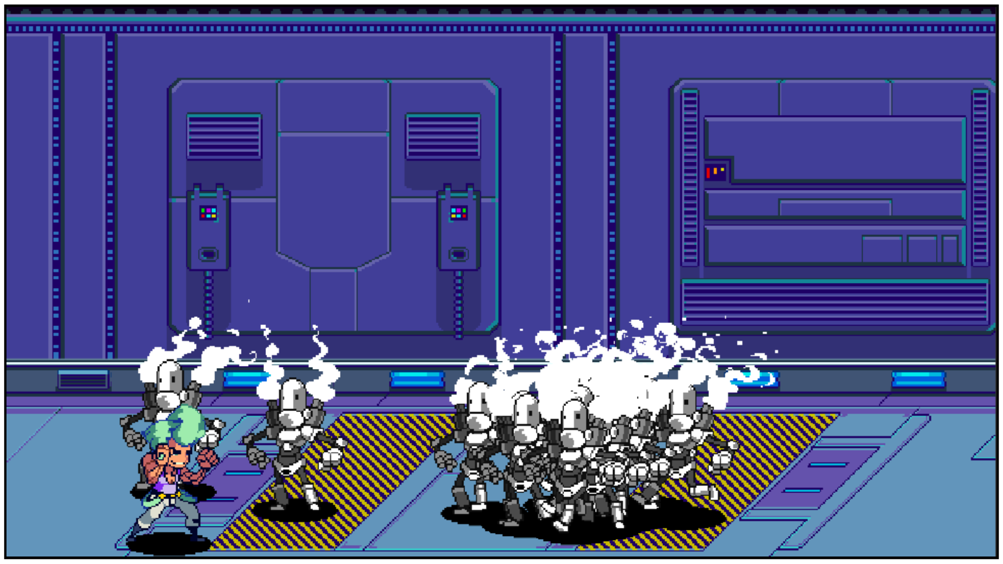
In this chapter, you'll:

- Tint the robot sprites to differentiate classes
- Store level data by using ScriptableObjects
- Understand Pompadroid's battle events
- Load the level's ScriptableObject and spawn enemies whenever necessary
- Animate the hero's entrance and exit animations
- Implement multi-level support for Pompadroid, allowing the hero to play through multiple levels

So, what are you waiting for? On to making Pompadroid a fun game!

Robot coloring 101

Pompadroid’s basic “baddie”, the robot, comprises three sprites: the base (or body), the smoke plumes and the belt. Normally they are white with a few highlights and shadows. A screen full of these white metal beasts would look pretty monotonous.



Your robots deserve to have some flair in the form of colors — lots of them. You’ll achieve this effect through **tinting**.

In the old days, when developers carved their code into clay tablets, you would have used a technique called palette swapping, which involves drawing the same sprite using different color sets or palettes. Today, however, there’s an easy alternative: adjust one color property to alter the color of your sprite.

Did you wonder why the robot is white? White is easy to tint!

Color tinting explained

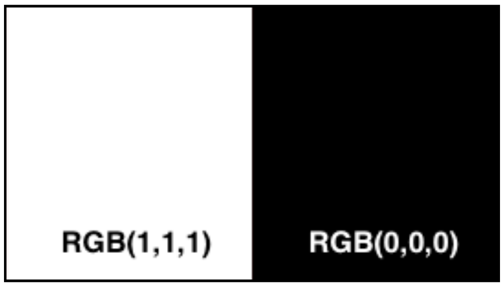
Unity uses the **RGB color model**, meaning that each color comprises three color components: **red**, **green** and **blue** that have values from 0 to 1, where 0 means no color and 1 means full color. Each color component is “added” together to get the resulting RGB color.

Note: Many applications that use the RGB model use a range from 0 to 255 so that each color component fits into one byte. To convert an RGB color from another application to Unity’s scale, simply divide each value by 255.

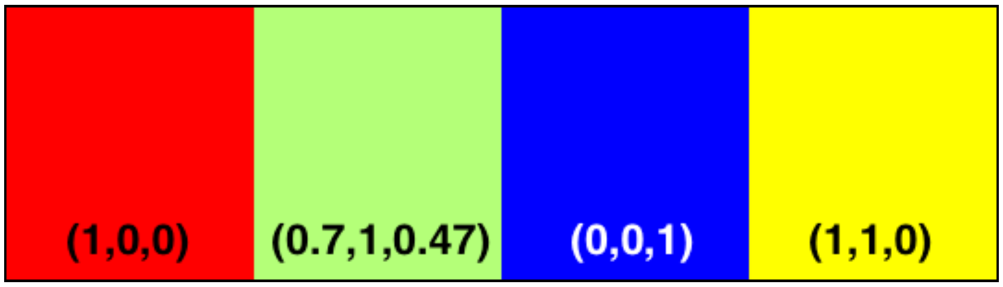
The RGB model is an **additive** model, meaning that it's a representation of mixing colors of light. It's very different from the subtractive models that are common in everyday life — think paint mixing. Most computer applications mix light colors. It's highly likely that you're looking at mixed light colors on your screen right now.

Colors are usually written like a Vector3, with the first element being the red component, the second being green and the third being blue. A color of (1,0.5,0) means red is 1, green is 0.5, and blue is 0.

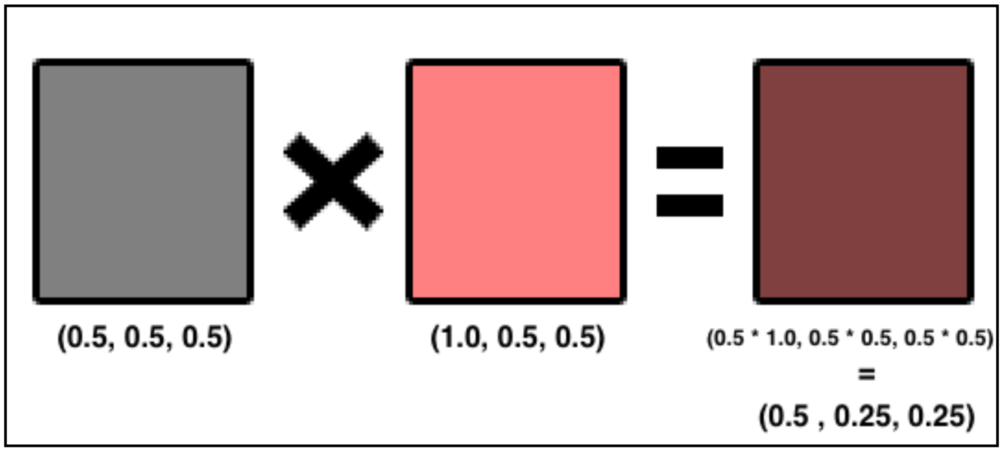
White is represented as a color with a red-green-blue (RGB) value of **(R:1, G:1, B:1)**, while **black** is shown as the absence of all three colors, with an RGB value of **(R:0, G:0, B:0)**.



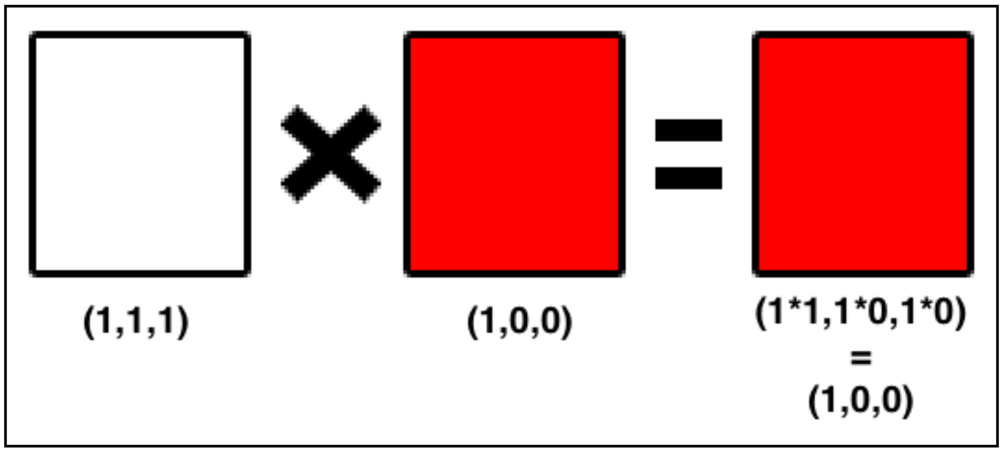
To make other colors, you vary the RGB values.



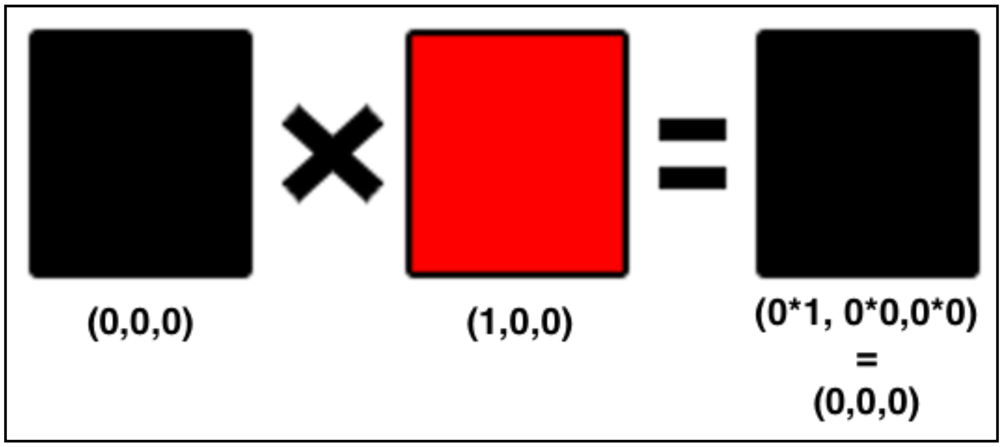
Tinting works by **multiplying** a color with a texture or another color. The two sets of components are multiplied to make another color.



For example, take white and tint it with red.

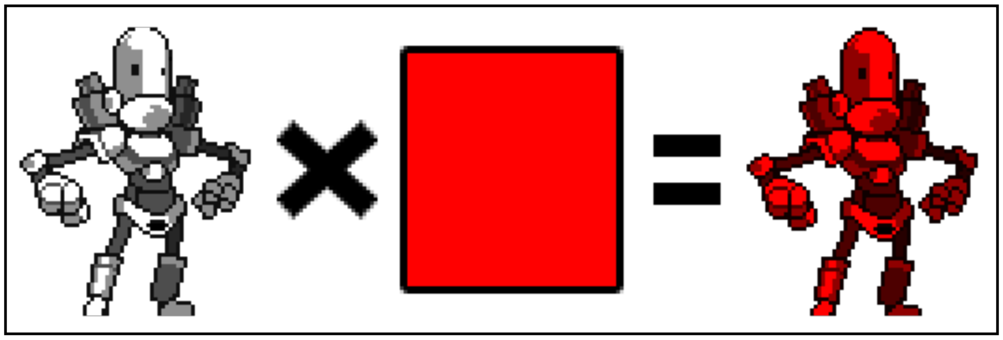


Now take black and tint it with red.



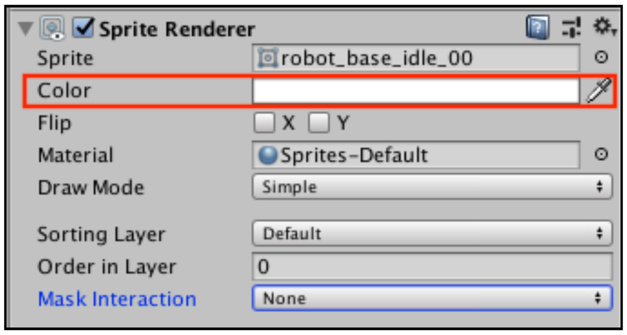
Bright colors are easier to affect with tinting since their values are greater than zero. Darker colors are not affected as much due to lower component values. Multiply anything by zero, and you still get…zero.

Consider the following robot sprite. Tinting it with the color **(R:1.0, G:0, B:0)** results in the following output:



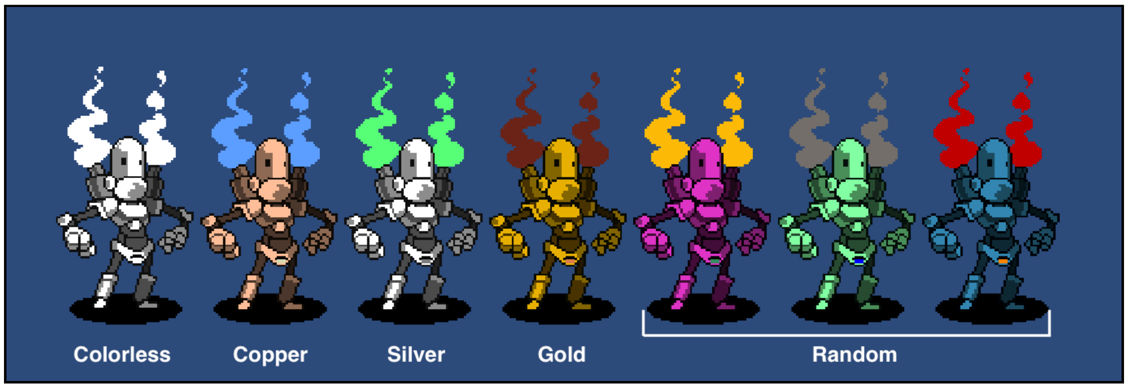
Now you see why the robot sprite is white with black lines. Tinting preserves the black lines, but the white parts take on the target color.

Unity makes tinting available to all SpriteRenderer components through the color property.



Sprite Renderer
Sprite: robot_base_idle_00
Color
Flip: X Y
Material: Sprites-Default
Draw Mode: Simple
Sorting Layer: Default
Order in Layer: 0
Mask Interaction: None

You'll use different color tints to differentiate between robot classes.



The classes are **Colorless**, **Copper**, **Silver**, **Gold** and **Random**, and each class has varying amounts of damage and hitpoints. Colorless is the weakest while Gold and Random are the strongest.

Adding robot classes

As you now know, you'll use robot classes to "treat" the pompadoured protagonist to a barrage of enemies with varying strength and constitution. In this section, you'll add them to the game.

Open **Actor.cs** and add the following variable declaration at the top:

```
public float attackDamage = 10;
```

You've created the `attackDamage` variable, which will decide how much damage a robot's punch should deliver.

Find the `HitActor` method and replace this:

```
actor.TakeDamage(10, hitVector);
```

With this:

```
actor.TakeDamage(attackDamage, hitVector);
```

`HitActor` can now use the `attackDamage` variable to determine how much to deplete the life value of other actors. It's more flexible than the damage of `-10` that you set up previously.

Save Actor.cs and open **Robot.cs**. Add this enumeration at the end of Robot.cs, **outside** of the Robot class definition:

```
public enum RobotColor {
  Colorless = 0,
  Copper,
  Silver,
  Gold,
  Random
}
```

The `RobotColor` enum contains the robot's possible classes.

Add these variable declarations to the top of the class:

```
//1
public RobotColor color;

//2
public SpriteRenderer smokeSprite;
public SpriteRenderer beltSprite;
```

1. `color` will store the current `RobotColor` value.

2. These two variables will reference the robot's smoke and belt sprites. These two SpriteRenderers, along with the baseSprite, will be tinted to the RobotColor's color.

Next, add the following `SetColor` method.

```
public void SetColor(RobotColor color) {
  this.color = color;

  switch (color) {
    case RobotColor.Colorless:
      baseSprite.color = Color.white;
      smokeSprite.color = Color.white;
      beltSprite.color = Color.white;
      maxLife = 50.0f;
      attackDamage = 2;
      break;
    case RobotColor.Copper:
      baseSprite.color = new Color(1.0f, 0.75f, 0.62f);
      smokeSprite.color = new Color(0.38f, 0.63f, 1.0f);
      beltSprite.color = new Color(0.86f, 0.85f, 0.71f);
      maxLife = 100.0f;
      attackDamage = 4;
      break;
    case RobotColor.Silver:
      baseSprite.color = Color.white;
      smokeSprite.color = new Color(0.38f, 1.0f, 0.5f);
      beltSprite.color = new Color(0.5f, 0.5f, 0.5f);
      maxLife = 125.0f;
      attackDamage = 5;
      break;
    case RobotColor.Gold:
      baseSprite.color = new Color(0.91f, 0.7f, 0.0f);
      smokeSprite.color = new Color(0.42f, 0.15f, 0.10f);
      beltSprite.color = new Color(0.86f, 0.5f, 0.32f);
      maxLife = 150.0f;
      attackDamage = 6;
      break;
    case RobotColor.Random:
      baseSprite.color = new Color(Random.Range(0, 1.0f), Random.Range(0,
1.0f), Random.Range(0, 1.0f));
      smokeSprite.color = new Color(Random.Range(0, 1.0f),
Random.Range(0, 1.0f), Random.Range(0, 1.0f));
      beltSprite.color = new Color(Random.Range(0, 1.0f), Random.Range(0,
1.0f), Random.Range(0, 1.0f));
      maxLife = Random.Range(100, 250);
      attackDamage = Random.Range(4, 10);
      break;
  }

  currentLife = maxLife;
}
```

This method takes a parameter named `RobotColor` and tints the sprites according to your code.

You set the SpriteRenderer's `color` to the desired tint for the robot's base, smoke and belt sprites at each `RobotColor` switch condition. `Colorless`, `Copper`, `Silver` and `Gold` sprites are tinted to their respective colors. As you'd expect for `Random`, you randomly obtain the color for each sprite.

You also set `maxLife` and `attackDamage` for each robot class. Generally speaking, `RobotColor.Gold` is the strongest but `RobotColor.Random` has the potential to be even stronger as its attack and life values are, well, random. They can go much higher than gold.

This next set of methods is for testing tinting. Add these to **Robot.cs**:

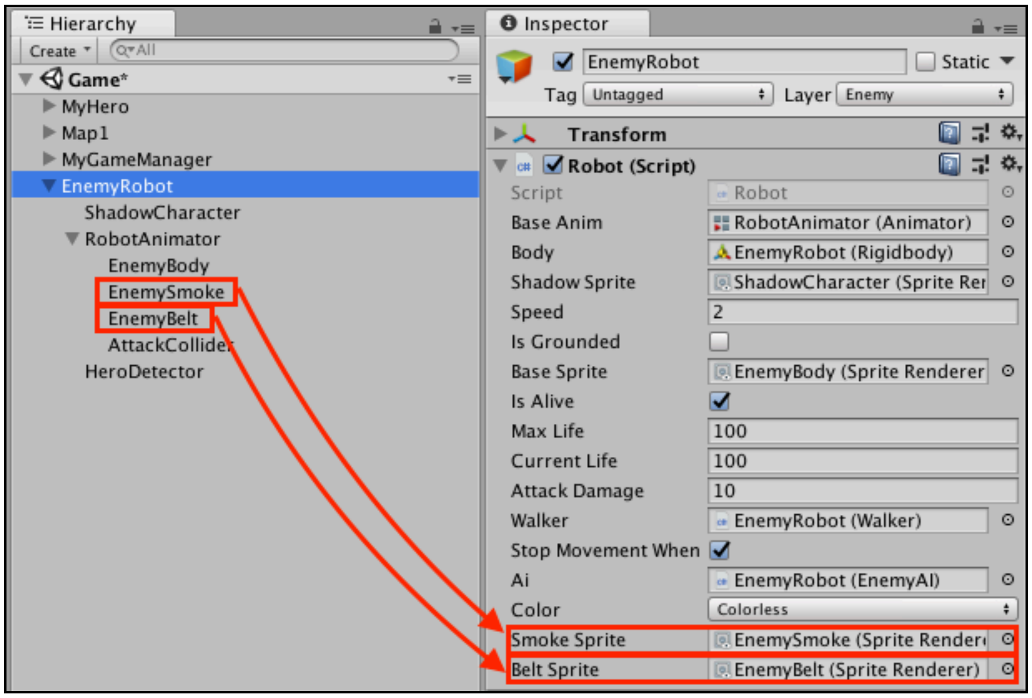
```
//1
[ContextMenu("Color: Copper")]
void SetToCopper() {
  SetColor (RobotColor.Copper);
}

//2
[ContextMenu("Color: Silver")]
void SetToSilver() {
  SetColor (RobotColor.Silver);
}

//3
[ContextMenu("Color: Gold")]
void SetToGold() {
  SetColor (RobotColor.Gold);
}

//4
[ContextMenu("Color: Random")]
void SetToRandom() {
  SetColor (RobotColor.Random);
}
```

These methods have the `ContextMenu` attribute, which adds a command to the context menu that uses the string parameter for the command's title. You access them by clicking the **cog** icon on the right side of the Inspector.

You created `Color: Silver`, `Color: Copper`, `Color: Gold` and `Color: Random` commands that you'll now see in the context menu of any Robot component. When you select one of these commands, Unity automatically calls the associated method.

Save this script and return to Unity. You need to add a few references before you test tinting.

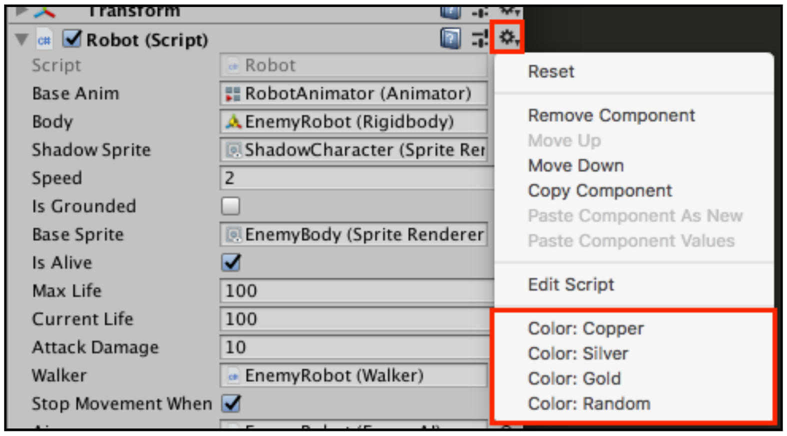
Select **EnemyRobot** and set up the references on the newly added **SmokeSprite** and **BeltSprite** fields to **EnemySmoke** and **EnemyBelt**, as shown below:



EnemyRobot
Tag Untagged Layer Enemy

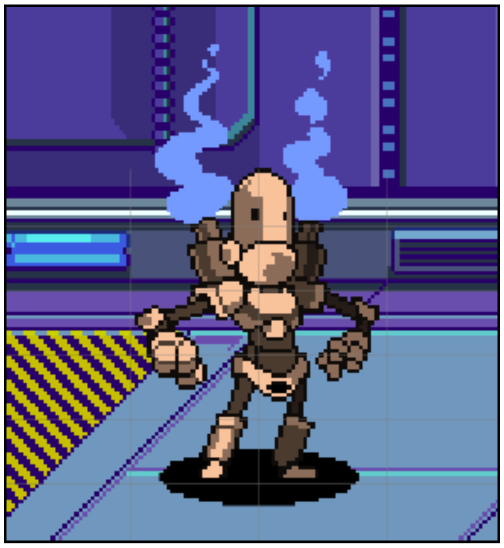
Script Robot
Base Anim RobotAnimator (Animator)
Body EnemyRobot (Rigidbody)
Shadow Sprite ShadowCharacter (Sprite Rer
Speed 2
Is Grounded ☐
Base Sprite EnemyBody (Sprite Renderer
Is Alive ✓
Max Life 100
Current Life 100
Attack Damage 10
Walker EnemyRobot (Walker)
Stop Movement When ✓
Ai EnemyRobot (EnemyAI)
Color Colorless
Smoke Sprite EnemySmoke (Sprite Render
Belt Sprite EnemyBelt (Sprite Renderer)

Test the robot tinting by selecting **EnemyRobot** in the Hierarchy. Click the **cog** icon to open the context menu for the Robot script, and then select one of the commands to change the robot's tint.

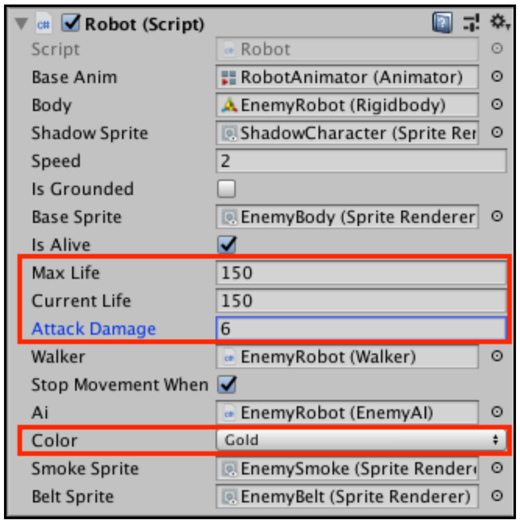


Script Robot
Base Anim RobotAnimator (Animator)
Body EnemyRobot (Rigidbody)
Shadow Sprite ShadowCharacter (Sprite Rer
Speed 2
Is Grounded ☐
Base Sprite EnemyBody (Sprite Renderer
Is Alive ✓
Max Life 100
Current Life 100
Attack Damage 10
Walker EnemyRobot (Walker)
Stop Movement When ✓

That's a droid of a different color!



Take note of how the robot's attack, maxLife, and currentLife change when you switch RobotColors.



Robot (Script)	
Script	Robot
Base Anim	RobotAnimator (Animator)
Body	EnemyRobot (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Rer
Speed	2
Is Grounded	☐
Base Sprite	EnemyBody (Sprite Renderer
Is Alive	✓
Max Life	150
Current Life	150
Attack Damage	6
Walker	EnemyRobot (Walker)
Stop Movement When	✓
Ai	EnemyRobot (EnemyAI)
Color	Gold
Smoke Sprite	EnemySmoke (Sprite Render
Belt Sprite	EnemyBelt (Sprite Renderer)

Remember that these commands are just for testing. Later in the chapter, you'll make the game set the robots' color automatically when they spawn.

And with that, you're done playing around with colors.

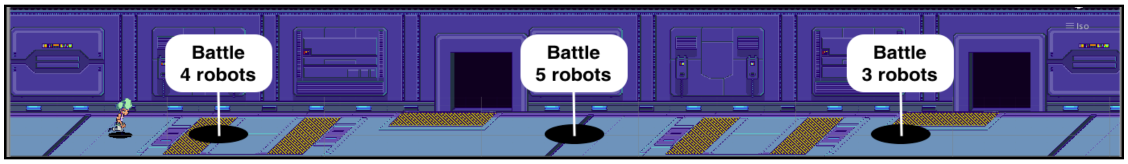
Make it interesting with game pacing

What's next? The hero's life is far too leisurely because he lacks goals and faces no obstacles. That's going to change, starting right now. Pacing is how you'll manage the flow of the game. It's how you'll determine whether this game is going to be boring, hard or fun. Through pacing, you'll dictate whether the player should be ready to fight or stroll leisurely along.

Pompadroid allows the hero to freely walk across the map to reach the other side. But that's boring. The first pacing element you'll implement is **battle events**. During one of these, the game will spawn a set amount of robots from either side of the screen, and you can probably guess what happens next.

To make it interesting for the player, you'll disable the camera's movement during battle events, preventing the hero's escape from the droids' wrath. Once the hero defeats all of the combatants, he may continue onward.

Here's how the game will play out: a set of battle events will litter the map, and the hero must fight through them to complete the level. Once the hero defeats all the levels of the game, he'll be victorious!



The strength of enemy forces must be stored at each battle event to enable you, the game designer, to create a balanced map. You don't want to make the map impossible to beat with an insurmountable amount of opponents. On the other hand, you don't want to give the hero too few opponents either. Striking a perfect balance between difficulty and pacing is the key to making a fun game.

Scriptable Objects are Unity game assets that you can script and use to store any type of data. You'll use them to store information about battle events. Think of them as components that behave as **assets** rather than GameObjects.

Note: While a ScriptableObject is nice and all, it's important to remember it has a caveat. Unity assets, unlike GameObjects, retain changes made to them even when the game is in play mode. Be careful when editing these assets — you could easily make a mistake you can't undo!

Each battle event will store its own set of information. First, you'll store the position that will trigger a battle event and use it to determine if the hero is close enough to initiate a battle. Second, you'll store enemy types to determine what kind of robot mob should spawn for each event.

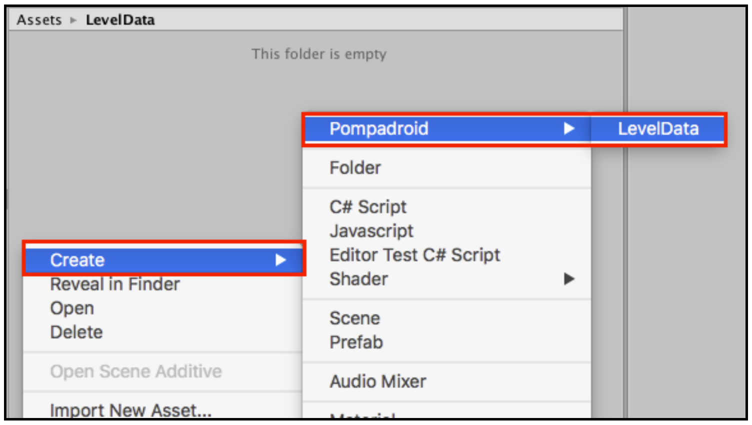
Currently, the game only supports robots, but in a later chapter, you'll build in support for a boss. When storing robots, you'll also store `RobotColors`. Finally, you'll also store each robot spawn position. Sounds simple enough, right?

Open the Unity Packages folder, and import **Level1Data.unitypackage**. It contains an asset named **Level1Data** and a **C#** script named `LevelData.cs`. Open the script and take a closer look at its code.

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

//1
[CreateAssetMenu(fileName ="LevelData", menuName = "Pompadroid/
LevelData")]
public class LevelData : ScriptableObject {
//2
  public List<BattleEvent> battleData;
  public GameObject levelPrefab;
  public string levelName;
}

//3
[Serializable]
public class BattleEvent {
  //4
  public int column;
  public List<EnemyData> enemies;
}

//5
[Serializable]
public class EnemyData {
  public EnemyType type;
  public RobotColor color;
  //6
  public int row;
  public float offset;
}

public enum EnemyType {
  Robot = 0,
  Boss
}
```

The script contains three different classes and an enumeration type. Here's a deeper explanation:

1. The LevelData class has the `CreateAssetMenu` attribute, which allows you to create a LevelData ScriptableObject from within the **Assets** / **Create** submenu found in Unity's menu bar. This attribute also creates a shortcut when you right-click the contents of a folder. From there, you can select **Create** / **Pompadroid** / **LevelData** from the Project view.



2. The `LevelData` class stores information about a single level. It contains `battleData`, which is simply a list of battle events in a level. It also has a `levelPrefab` variable that references the level's tiled map, and finally, `levelName`, which as the name suggests, is the title of the level.

3. The `BattleEvent` class is given the `Serializable` attribute so that the contents of the class display in the Inspector.

4. The `column` variable determines the X position at which to trigger the battle event. Think of it like a booby-trapped column of tiles that, when stepped upon, will send a horde of `enemies` the hero's way.

5. The `EnemyData` class stores information about enemy spawns; `type` determines whether the enemy will be a robot or a boss, and in the case of the former, you'll use `color` to specify the robot's class.

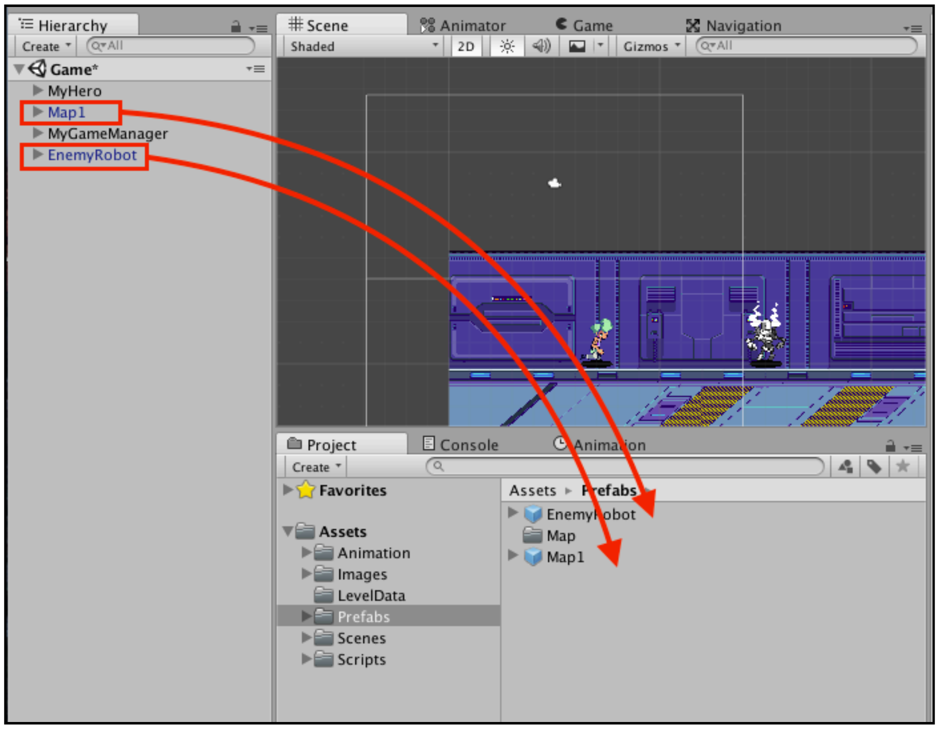
6. The `row` variable determines the Y position at which the enemy will spawn, while `offset` specifies how far from the center of the screen it will spawn.

That's how the `LevelData` stores the game events. Neat isn't it?Return to Unity. You'll need to set up a few things before you use any level data files.

Creating prefabs and LevelData

You'll need to create two new prefabs: EnemyRobot and Map1.

Drag the **EnemyRobot** towards the **Assets** / **Prefabs** folder in the Project view to create an **EnemyRobot** prefab. This will automatically change the EnemyRobot label's color to blue in the Hierarchy. Drag **Map1** to the same folder to create a **Map1** prefab. After you've created both prefabs, delete the original GameObjects from the Hierarchy.

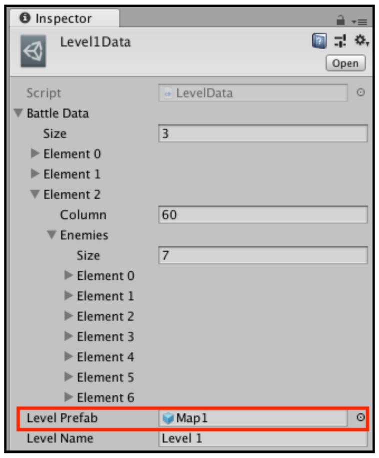


Now you have the necessary prefabs for your first level. At this point, you can create a **LevelData** asset for your first level. You have the power to design levels!

Creating a level from scratch may sound daunting — there are many options! While it’s true that level creation can be a big task, you’ll find the LevelData included in this starter kit makes it manageable and allows you to learn the process pretty quickly.

The **Level1Data.unitypackage** you've imported earlier contains a Level1Data asset in the **Assets / LevelData** folder. Select it and check its contents in the Inspector.

Drag the **Map1** prefab into the **LevelPrefab** field to create a prefab reference for that asset, effectively setting Map1 as the tiled map for the first level.



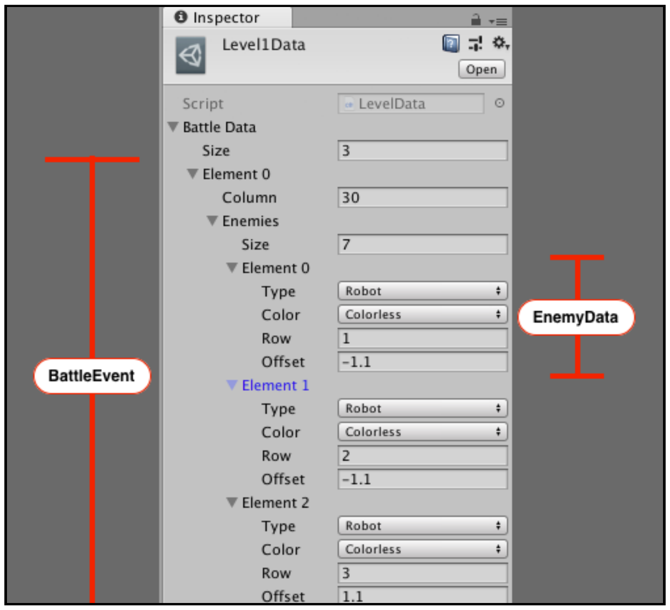
Inspector
Level1Data
Open

```
Script                LevelData
▼ Battle Data
    Size              3
  ▶ Element 0
  ▶ Element 1
  ▼ Element 2
      Column          60
    ▼ Enemies
        Size          7
      ▶ Element 0
      ▶ Element 1
      ▶ Element 2
      ▶ Element 3
      ▶ Element 4
      ▶ Element 5
      ▶ Element 6
Level Prefab          Map1
Level Name            Level 1
```

With that, you have functional LevelData for the first level. Now let's step through how exactly your LevelData works.

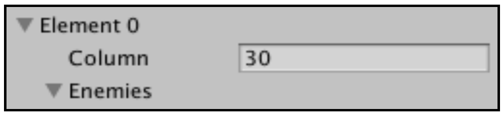
LevelData's inner workings

Take a closer look at **Level1Data** in the Inspector, and expand Battle Data to see what's inside.



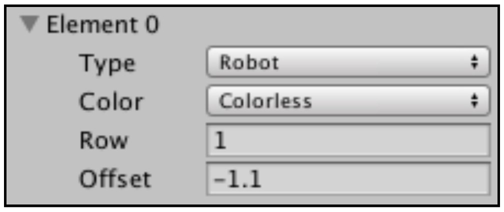
Script	LevelData
▼ Battle Data	
Size	3
▼ Element 0	
Column	30
▼ Enemies	
Size	7
▼ Element 0	
Type	Robot
Color	Colorless
Row	1
Offset	-1.1
▼ Element 1	
Type	Robot
Color	Colorless
Row	2
Offset	-1.1
▼ Element 2	
Type	Robot
Color	Colorless
Row	3
Offset	1.1

Remember that the game's events are stored in an array of **BattleEvents**, as seen in the **Battle Data** list.



▼ Element 0	
Column	30
▼ Enemies	

For each battle event, `Column` represents the X position that triggers the event. For example, the first battle event begins when the hero's X position gets close to `30`. Exactly which kind of robots will mob the player are specified in the `Enemies` list.



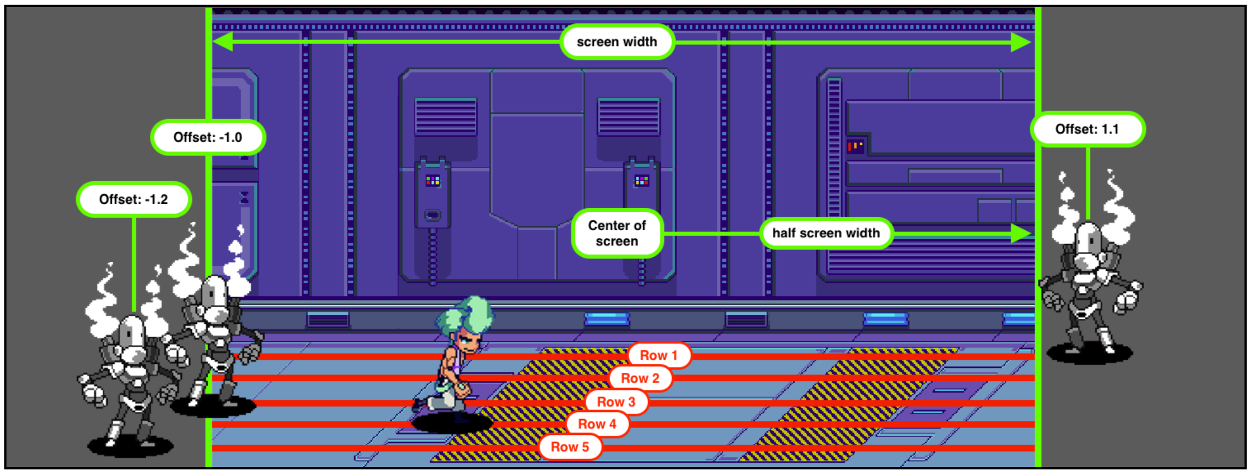
▼ Element 0	
Type	Robot
Color	Colorless
Row	1
Offset	-1.1

Each instance of **EnemyData** contains the following information:

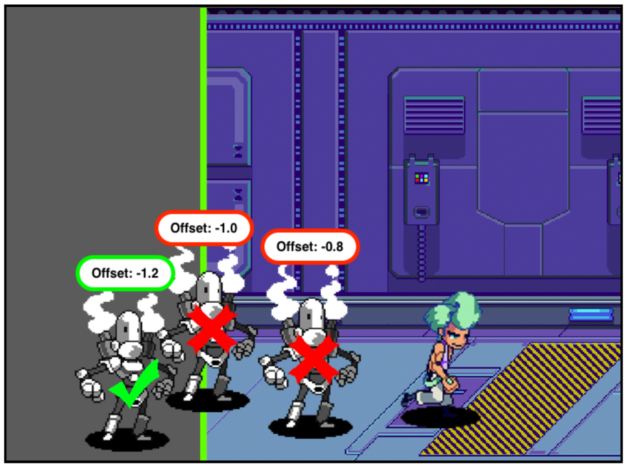
- `Type` determines if the enemy will be a robot or a boss.
- `Color` sets the class when the enemy is a robot.
- `Row` and `Offset` define where the enemy spawns.

You can think of the map as a series of rows. When a robot spawns, its EnemyData's `row` variable determines at which row the spawn will occur. The `offset` value's magnitude depicts how far from the center the enemy will spawn, with `1.0` being half the width of the screen.

For example, the robot will spawn to the right when the value is *greater* than `0` and to the *left* when the value is less than `0`.



Note that most spawn points are greater than `1.0` or less than `–1.0`. This is so enemies spawn outside the visible area and walk into the scene rather than "magically" appearing in the middle of it.



And now you know how LevelData is structured.

Looking at Level1Data, you can see that difficulty for each BattleEvent increases. Initially, you get colorless robots and a couple of random strong ones, while the next event spawns coppers, silvers and a few randoms. The final event spawns heavy-hitting silvers, golds and randoms.

Arguably, the coolest thing here is that you can modify this ScriptableObject to provide the difficulty and pace you want for the game; you can add and remove events and robots to your heart's desire.

Using LevelData in the game

Even though you have a perfectly functional LevelData asset, your game has no idea how to use it. You need to modify some scripts to enable the reading and translating of this data into commands that the game understands.

Open the **CameraBounds** script and find this line:

```
private Camera activeCamera;
```

Replace it with this:

```
public Camera activeCamera;
```

You changed the access modifier from `private` to `public` to allow other classes to access the camera used by `CameraBounds`. This will come in handy later.

Save the script and open the **GameManager** script. It's still bare at this point, but you'll soon add support for complex behavior such as enemy spawns and battle events.

Add the following to the top of `GameManager.cs`:

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

Nothing fancy here. Both libraries contain classes that you'll use later.

Next, add the following variables above `Start()`:

```
//1
public LevelData currentLevelData;
private BattleEvent currentBattleEvent;
private int nextEventIndex;
public bool hasRemainingEvents;

//2
public List<GameObject> activeEnemies;
public Transform[] spawnPositions;

//3
public GameObject currentLevelBackground;

//4
public GameObject robotPrefab;
```

These variables contain data about the current level.

1. The first four variables are close collaborators. `currentLevelData` references the data for the current level, while `currentBattleEvent` references the active battle event from that data. `nextEventIndex` is an index of all battle events you'll use to fetch the next event. Lastly, `hasRemainingIndex` determines whether there are available events in the level.

2. After loading a battle event, the `activeEnemies` list stores all enemies of that event, while `spawnPositions` stores the position, in rows, where these enemies will spawn.

3. `currentlevelBackground` references the tile map for the level. You can't have a level without it, otherwise, everyone will float in space!

4. Finally, `robotPrefab`, as its name suggests, references the robot's prefab. You will need this to create copies of robots later.

Next, still in **GameManager.cs**, add the following method:

```
private GameObject SpawnEnemy(EnemyData data) {
  //1
  GameObject enemyObj = Instantiate(robotPrefab);

  //2
  Vector3 position = spawnPositions[data.row].position;
  position.x = cameraBounds.activeCamera.transform.position.x +
(data.offset * (cameraBounds.cameraHalfWidth + 1));
  enemyObj.transform.position = position;

  //3
  if (data.type == EnemyType.Robot) {
    enemyObj.GetComponent<Robot>().SetColor(data.color);
  }

  //4
  enemyObj.GetComponent<Enemy>().RegisterEnemy();

  return enemyObj;
}
```

This method creates an enemy according to the specifications of the `EnemyData` parameter.

1. First, you create an enemy using `robotPrefab`. Remember that prefabs are like templates for objects. By calling `Instantiate`, you create a new `GameObject` that has all the data from that prefab.

2. Next, you calculate the spawn position using the `row` and `offset` values from the `EnemyData`. You have to make sure the spawn point is outside the visible area so that the enemy spawns offscreen. To do this, you multiply the offset by a little over half the screen width, and add the resulting value to the current position of the camera.

3. You check if the enemy is a robot, and if so, you set its color accordingly. This check will be more important later on, when you have more than one type of enemy in the level.

4. Lastly, you call the enemy's `RegisterEnemy` method — you wrote this method in `Enemy.cs` to keep track of the current number of active enemies.

Still in the same script, add the following method that loads a battle event:

```
private void PlayBattleEvent(BattleEvent battleEventData) {
  //1
  currentBattleEvent = battleEventData;
  nextEventIndex++;

  //2
  cameraFollows = false;
  cameraBounds.SetXPosition(battleEventData.column);

  //3
  foreach (GameObject enemy in activeEnemies) {
    Destroy(enemy);
  }
  activeEnemies.Clear();
  Enemy.TotalEnemies = 0;

  //4
  foreach (EnemyData enemyData in currentBattleEvent.enemies) {
    activeEnemies.Add(SpawnEnemy(enemyData));
  }
}
```

You'll use the`PlayBattleEvent` method to load a battle event, using the data from its `battleEventData` parameter.

1. Assigns the battle event to the `currentBattleEvent` variable to preserve `battleEventData` so you can use it elsewhere, and then increments `nextEventIndex` since the current event was just loaded.

2. Prevents the hero from escaping the battle event by setting the camera's X position to the event's column.

3. Destroys remnants of prior battle events by obliterating`GameObjects` in `activeEnemies` then it resetting `TotalEnemies` to `0` to specify that there are no enemies in the scene.

4. Assembles a mechanical army of doom! First, it goes through the list of enemies in `currentBattleEvent` and plugs each `EnemyData` from it into the `SpawnEnemy` method, which you created earlier. Next, it repopulates the `activeEnemies` list with the newly-created enemies.

You have the means to load new battle events, but what happens when the last droid dies? Even if the hero beats up and destroys all enemies in the scene, the camera is locked in to position, trapping the hero forever! Thankfully, it doesn't take much to free the hero from this invisible prison.

Add the following method to `GameManager.cs`:

```
private void CompleteCurrentEvent() {
  currentBattleEvent = null;

  cameraFollows = true;
  hasRemainingEvents = currentLevelData.battleData.Count >
nextEventIndex;
}
```

Here you complete the current battle event by clearing the `currentBattleEvent` variable and allowing the camera to follow the hero again. You also update the Boolean `hasRemainingEvents` to dictate if there are more battle events ahead.

Next up, you'll create the mechanism that automates the flow of battle events by making use of `EnemyData` and `BattleEvent`.

Insert the following code block at the start of the `Update()` method:

```
//1
if (currentBattleEvent == null && hasRemainingEvents) {
  if (Mathf.Abs(currentLevelData.battleData[nextEventIndex].column -
cameraBounds.activeCamera.transform.position.x) < 0.2f) {
    PlayBattleEvent(currentLevelData.battleData[nextEventIndex]);
  }
}

//2
if (currentBattleEvent != null) {
  //has event, check if enemies are alive!
  if (Robot.TotalEnemies == 0) {
    //no more enemies;
    CompleteCurrentEvent();
  }
}
```

The above code handles the current state of the game at every frame. More specifically, it:

1. Checks if the game isn't running a battle event and if there are still events to play through. If there are, it checks if the next BattleEvent's `column` value is close enough to the hero to trigger it. When it's close enough, it loads the next event with the `PlayBattleEvent` method.
2. Checks if all robots are dead when there's an actively loaded battle event. If all are dead, it calls `CompleteCurrentEvent`.

Both `EnemyData` and `BattleEvent` need to be loaded into the game before they can be used, so add this method to the same script:

```
private IEnumerator LoadLevelData(LevelData data) {
  //1
  cameraFollows = false;
  currentLevelData = data;

  //2
  hasRemainingEvents = currentLevelData.battleData.Count > 0;
  activeEnemies = new List<GameObject>();

  //3
  yield return null;
  cameraBounds.SetXPosition(cameraBounds.minVisibleX);

  //4
  currentLevelBackground = Instantiate(currentLevelData.levelPrefab);
  yield return new WaitForSeconds(0.1f);

  //5
  cameraFollows = true;
}
```

The `LoadLevelData` method essentially loads a `LevelData` asset. Note that it's an `IEnumerator` that will be used in a coroutine. The reason is that it allows for easier code sequence checking and makes it simpler to add a loading screen in the future. Here's a section-by-section explanation:

1. You disable the camera's movement and store a reference to the current level data.
2. Next, you check if there are battle events in the level, and store the result in `hasRemainingEvents`. You also initialize the `activeEnemies` list so it can be used later.

3. `yield return null` pauses the method for one frame, allowing other scripts to run before executing the next line. In this case, you use it to allow the `CameraBounds` script to perform its `Start` method before setting its X position to the minimum visible value.
4. You create the tile map referenced by `currentLevelData.levelPrefab` and store it in the `currentLevelBackground` variable. To give it time to initialize, you pause the coroutine again for a few milliseconds.
5. With the level completely loaded, you allow the camera to move once again.

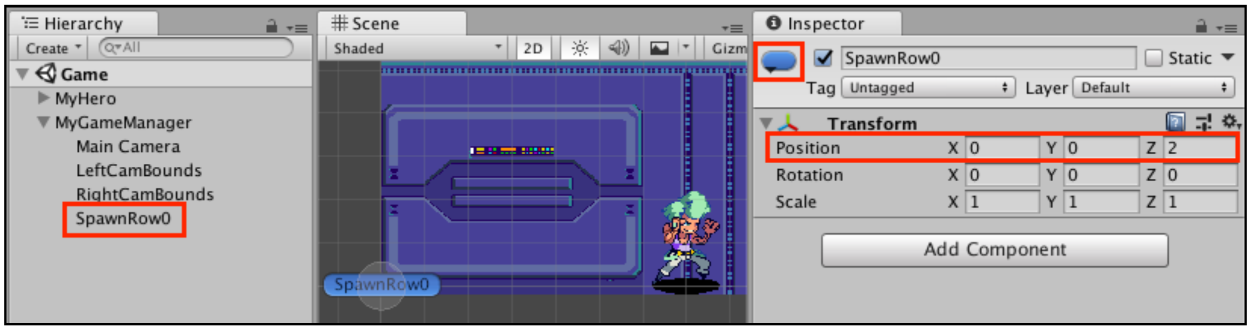
Next, add the following lines to the end of the `Start` method:

```
nextEventIndex = 0;
StartCoroutine(LoadLevelData(currentLevelData));
```

Here you place loading the level near the top of the GameManager's to-do list when the game begins: When GameManager executes its `Start` method, you reset `nextEventIndex` to `0` and load the LevelData asset referenced by the `currentLevelData` variable.

Save the script and return to Unity. There's one last thing to do before everything works correctly. You need to add the spawn rows — these markers identify the rows where the enemies can spawn.

Create a new **empty GameObject** and make it a child of **MyGameManager**. Rename it **SpawnRow0**. In the Inspector, click the multi-colored **cube** icon next to its name and select the **blue** oval-shaped label icon. Reset its Transform and set its **Position** to `(X:0, Y:0, Z:2)`.



SpawnRow0 — Static
Tag: Untagged — Layer: Default

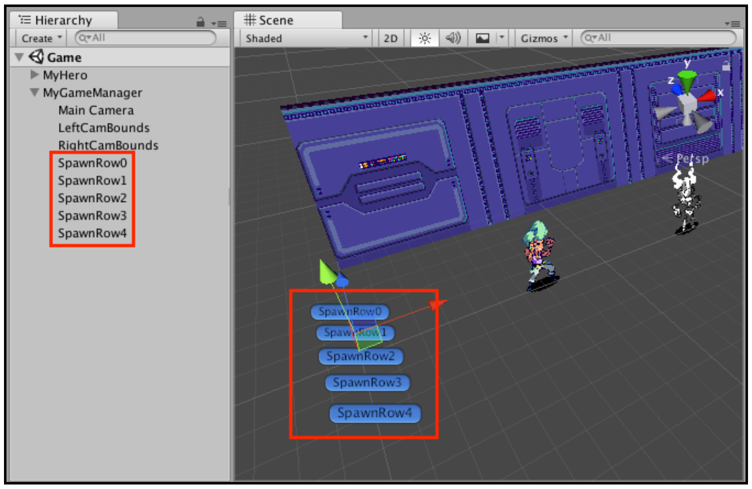
Transform
Position X 0 Y 0 Z 2
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

You've just created the first marker. Next, duplicate **SpawnRow0**, rename it to **SpawnRow1** and set its **Position** to `(X:0, Y:0, Z:1)`. Repeat these steps and set as per below:

- **SpawnRow2** at **Position** `(X:0, Y:0, Z:0)`

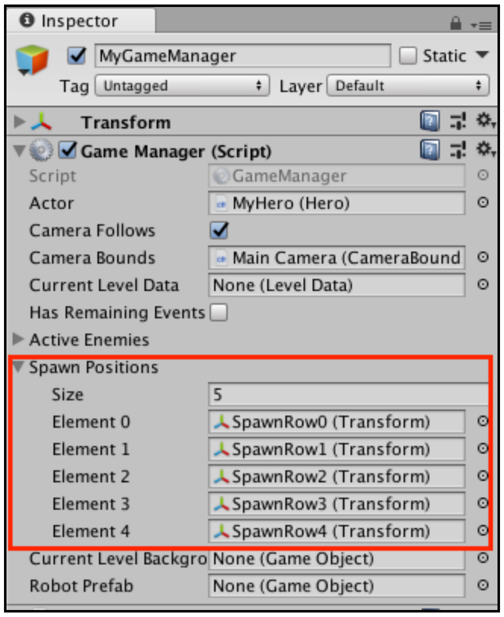
- **SpawnRow3** at **Position** `(X:0, Y:0, Z:-1)`
- **SpawnRow4** at **Position** `(X:0, Y:0, Z:-2)`

The SpawnRows should be visible in the Scene view as blue label icons.



The markers are bound to the camera because the spawn points should always be at the center of the camera's view.

Now to configure the GameManger to actually use these rows. In the Inspector for the GameManager script, set the **Size** of the **SpawnPositions** list to 5. Assign **SpawnRow0** to **SpawnRow4** as the **Elements** of SpawnPositions, as shown below:


Inspector

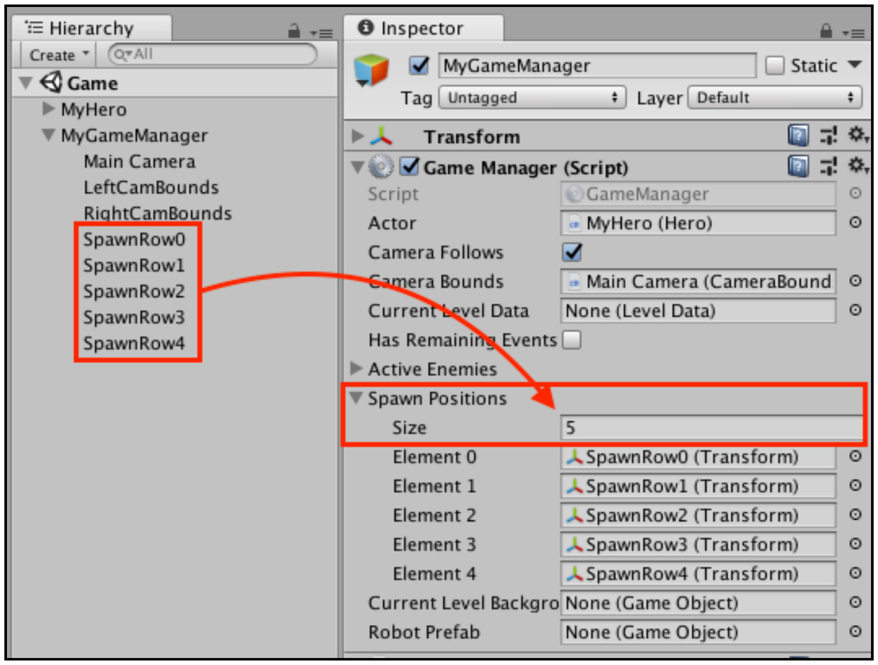
MyGameManager Static
Tag Untagged Layer Default

Transform

Game Manager (Script)

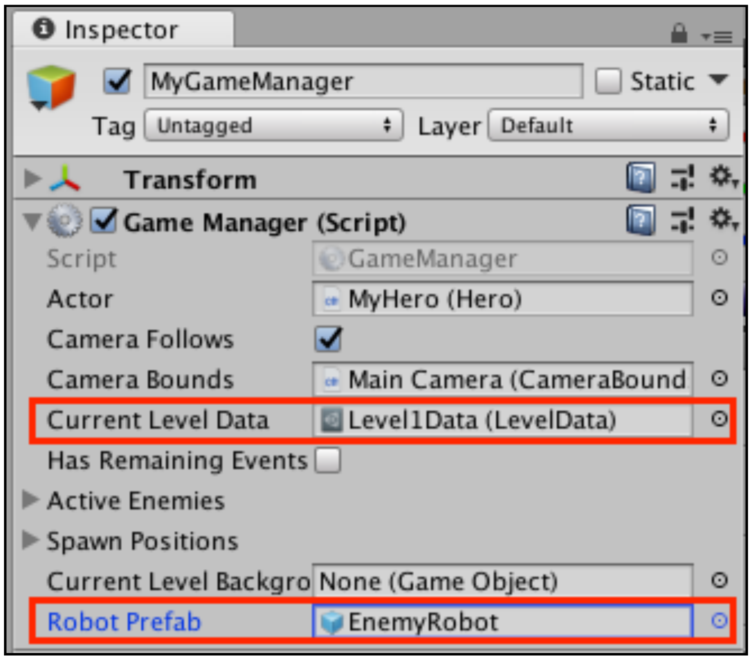
Script	GameManager
Actor	MyHero (Hero)
Camera Follows	✓
Camera Bounds	Main Camera (CameraBound
Current Level Data	None (Level Data)
Has Remaining Events	☐
Active Enemies	
Spawn Positions	
Size	5
Element 0	SpawnRow0 (Transform)
Element 1	SpawnRow1 (Transform)
Element 2	SpawnRow2 (Transform)
Element 3	SpawnRow3 (Transform)
Element 4	SpawnRow4 (Transform)
Current Level Backgro	None (Game Object)
Robot Prefab	None (Game Object)

There are multiple ways to assign an array in the Inspector. One of my favorites is to select all the objects you want then drag the group to the list's label.



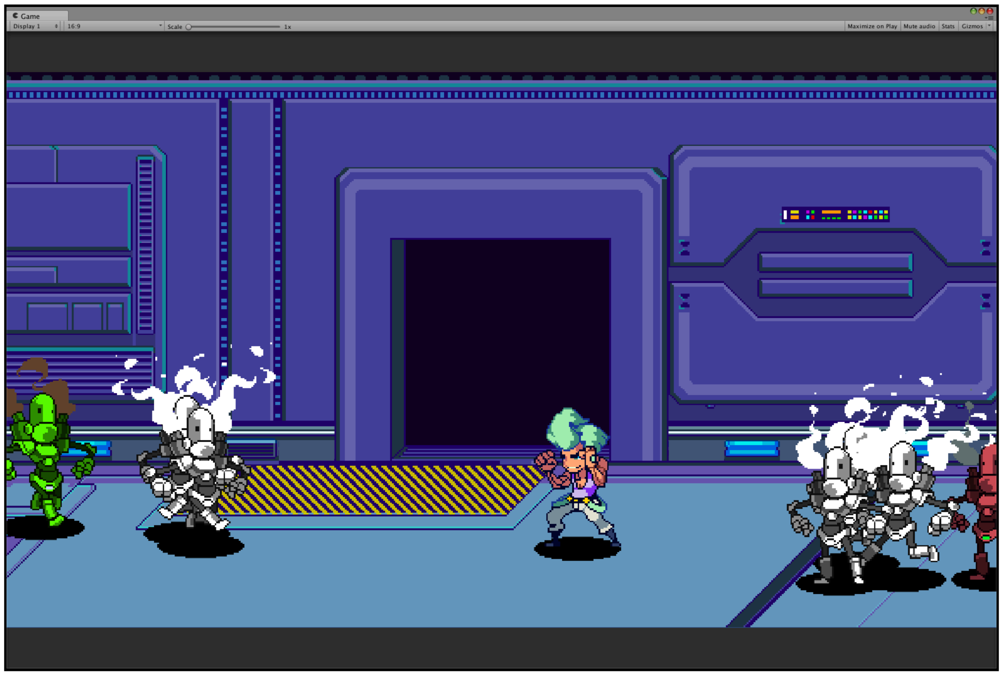
Script	GameManager
Actor	MyHero (Hero)
Camera Follows	✓
Camera Bounds	Main Camera (CameraBound
Current Level Data	None (Level Data)
Has Remaining Events	
Active Enemies	
Spawn Positions	
Size	5
Element 0	SpawnRow0 (Transform)
Element 1	SpawnRow1 (Transform)
Element 2	SpawnRow2 (Transform)
Element 3	SpawnRow3 (Transform)
Element 4	SpawnRow4 (Transform)
Current Level Backgro	None (Game Object)
Robot Prefab	None (Game Object)

Next, set the GameManager's **Robot Prefab** field to the **EnemyRobot** prefab you created and set **Current Level Data** to **Level1Data.asset** in **Assets/LevelData**.

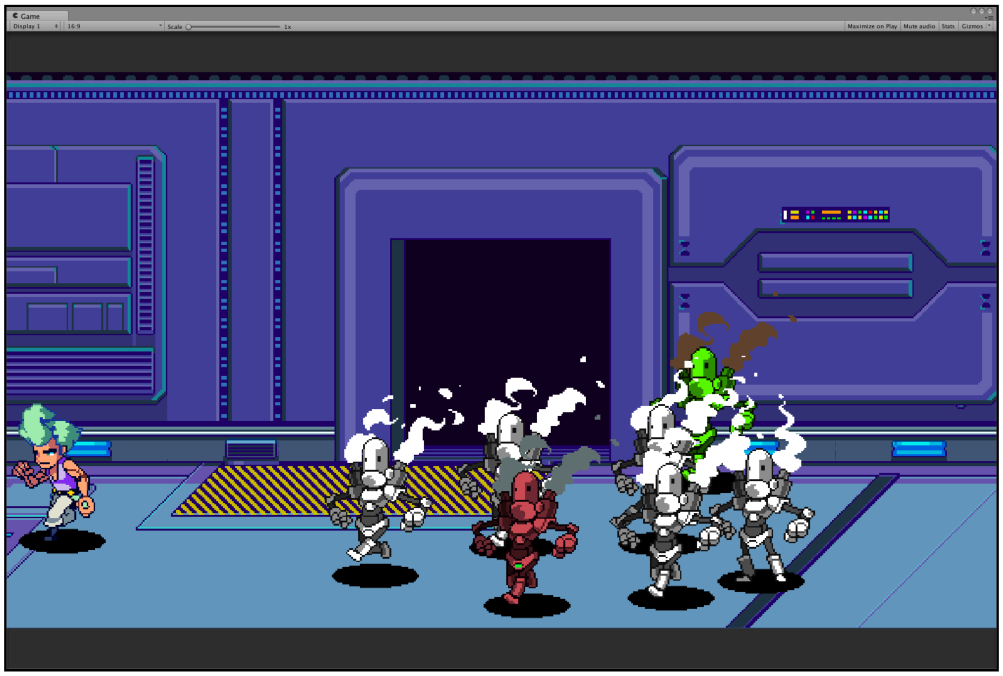


Script	GameManager
Actor	MyHero (Hero)
Camera Follows	✓
Camera Bounds	Main Camera (CameraBound
Current Level Data	Level1Data (LevelData)
Has Remaining Events	
Active Enemies	
Spawn Positions	
Current Level Backgro	None (Game Object)
Robot Prefab	EnemyRobot

Save the **scene** and **project** then play the game. Walk far enough to the right to trigger a battle event and watch as those nasty droids spawn on both sides. They look eager to terminate you!

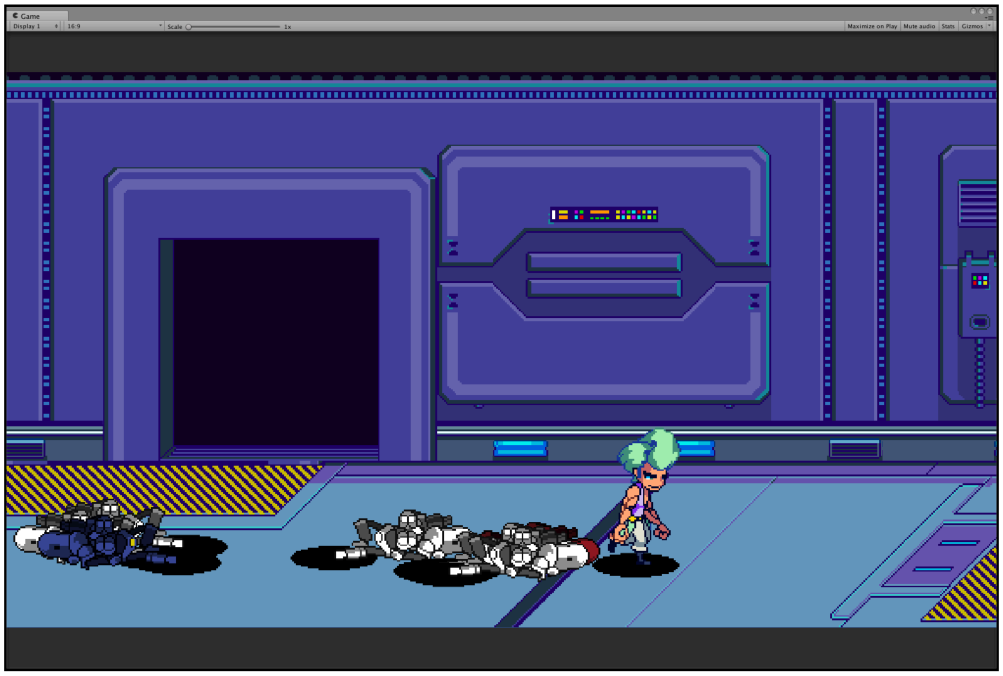


Try to escape. The camera won’t budge! You have nowhere to go because you can’t move past the edge of the camera’s frame, at least not until you lay waste to those miserable hunks of metal and circuitry.



You should allow yourself to fall prey to the droids at least once so that you see what happens. You’ll spend your time being dead on the floor — you haven’t added a “game over” screen yet. You’ll do that later, by the way.

And also make sure you emerge victorious against these rust buckets at least once, so you can see how you're free to wander again — at least until you walk into the next battle event.



There's a subtle bug when you finish a battle event: The camera instantly centers over the hero's position. It could cause a problem for the player's visual orientation and at the very least hurts the eyes!

The best fix is to smoothly ease the camera back to center on the hero's position rather than snapping to it. This approach is friendlier to the eyes and looks cleaner.

Open **CameraBounds.cs** and add the following variable to the top:

```
public float offset;
```

This is the offset from the center of the camera that you'll use to smooth out the camera's change in position. An `offset` value of `0` means that the camera is centered on the hero.

Still in the same script, add the following methods:

```
//1
public void CalculateOffset(float actorPosition) {
  offset = cameraRoot.position.x - actorPosition;
  SetXPosition(actorPosition);
  StartCoroutine(EaseOffset());
}

//2
```

```
private IEnumerator EaseOffset() {
  while (offset != 0) {
    offset = Mathf.Lerp(offset, 0, 0.1f);
    if (Mathf.Abs(offset) < 0.05f) {
      offset = 0;
    }
    yield return new WaitForFixedUpdate();
  }
}
```

1. `CalculateOffset` computes for the horizontal distance between the `actorPosition` parameter and the camera's X position. This distance becomes the value of the offset. Next, you move the camera to the `actorPosition` value and start a coroutine with the `EaseOffset` method.

2. `EaseOffset` gradually reduces the value of `offset`. Every loop decreases the offset until it reaches a value less than `0.05`, in which cases, it forces the offset to become `0`. The line `yield return WaitForFixedUpdate();` forces this coroutine to pause and continue only after every call to `FixedUpdate`, which Unity does regularly.

Next, edit the `SetXPosition` method by replacing this:

```
trans.x = Mathf.Clamp(x, minValue, maxValue);
```

With this:

```
trans.x = Mathf.Clamp(x + offset, minValue, maxValue);
```

When setting the camera's X position, you now use the value of `offset` to compute for its final position. If the offset is `0`, then the camera will behave as it did before, when it centered on and followed the hero. Otherwise, it will use the value of offset to displace the camera by a certain distance from the hero's position. The camera will then gradually move towards the player as the offset decreases.

Save the CameraBounds script and open **GameManager.cs**. Find the `CompleteCurrentEvent` method then find `cameraFollows = true;` and add this line after it:

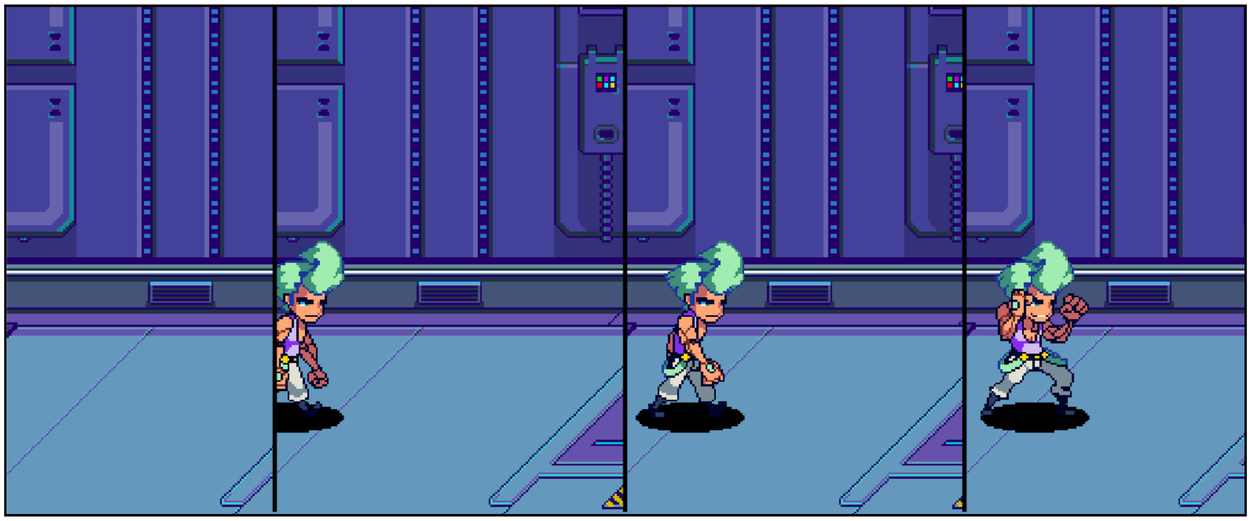
```
cameraBounds.CalculateOffset(actor.transform.position.x);
```

This calculates the offset at the end of every battle event — this is the point where the camera should resume its default of following the hero.

Save the script, return to Unity, press **play** and take out some aggression on those droids. Notice how the camera eases back to the hero's position after the last enemy falls.

You have a functional LevelData script. What's next? Well, how about giving the hero some heroic fanfare when he enters the scene?

Heros deserve heroic entrances



To make a dramatic entrance, the hero will use the Walker script to navigate the map. His entrance will be automatic and scripted, so the player won't be able to interrupt or stop the animation.

First, you need to modify the Hero script so the hero can be "walked" by the Walker script. Open **Hero.cs** and add the following variables to the top:

```
public Walker walker;
public bool isAutoPiloting;
public bool controllable = true;
```

The `walker` variable is a reference to a Walker script that enables the hero to be "walked" around automatically. `isAutoPiloting` serves as a flag that disables the processing of character movement when the hero is under the walker's control, and `controllable` is a variable that flags the controls and disables player input when the condition is false.

Add the following namespace usage at the top of the class, along with the other `using` statements, if the script doesn't already have it:

```
using System;
```

This will enable the class to use the System.Action callbacks.

Add the following methods to the bottom of the class:

```
//1
public void AnimateTo(Vector3 position, bool shouldRun, Action callback)
{
  if(shouldRun) {
    Run();
  }
  else {
    Walk();
  }
  walker.MoveTo (position, callback);
}

//2
public void UseAutopilot(bool useAutopilot) {
  isAutoPiloting = useAutopilot;
  walker.enabled = useAutopilot;
}
```

1. `AnimateTo` is a method that first checks whether the hero should walk or run towards its target by using the `shouldRun` parameter, and then it calls the Walker's `MoveTo` by using the target `position` where the hero should move to. The `callback` parameter is a method that is invoked when the walking is done.

2. `UseAutopilot` is a method that enables or disables the Walker with the `useAutopilot` input parameter. It also sets the `isAutoPiloting` Boolean.

Find the `FixedUpdate` method then find this block at the top:

```
if (!isAlive) {
  return;
}
```

Surround everything underneath (in `FixedUpdate`) with this `if` statement:

```
if (!isAutoPiloting) {
  //Old code goes here
}
```

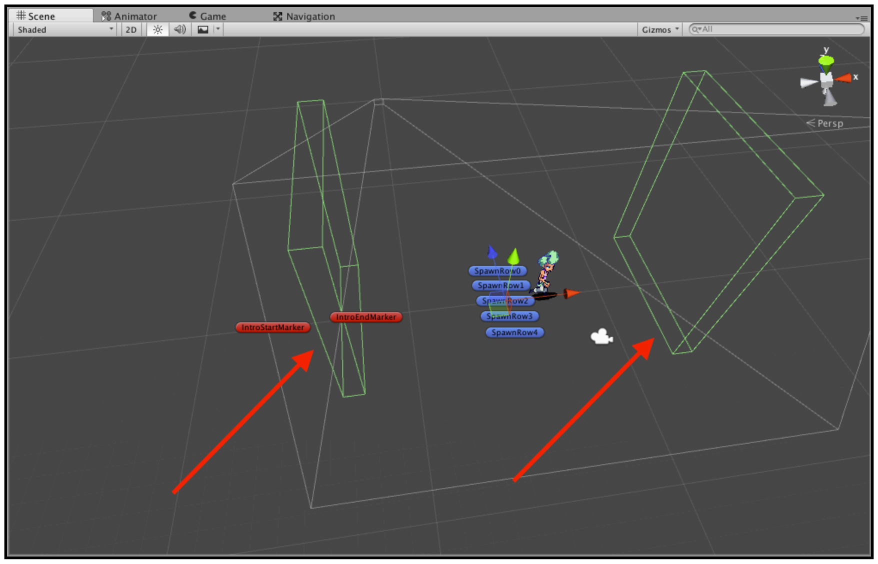
Remember the closing brace for the `if` statement! This is a check that prevents any movement calculations from happening when the hero is under Walker control.

Add the following above `float h = input.GetHorizontalAxis ();` in the `Update()`method:

```
if (isAutoPiloting) {
  return;
}
```

This prevents the script from performing actions such as jumping or punching during the hero's dramatic entrance! **Save** Hero.cs and open **CameraBounds.cs**.

Remember how the camera has a collider at each edge of the screen to prevent the hero from walking out of the camera's view? You need to disable these colliders to allow the dramatic entrance.



Add the following method in `CameraBounds.cs`:

```
public void EnableBounds(bool isEnabled) {
  rightBounds.GetComponent<Collider>().enabled = isEnabled;
  leftBounds.GetComponent<Collider>().enabled = isEnabled;
}
```

This method toggles the enabled state on both bound colliders and turns them off when the hero's dramatic entrance is triggered. They are enabled once the hero's entrance is done.

You also need to make the hero's walk-in position relative to the left edge of the camera's frame, regardless of the camera's current aspect ratio. To achieve this, you'll need to compute the walk-in position every time the game runs.

Still in `CameraBounds.cs`, add the following variables:

```
public Transform introWalkStart;
public Transform introWalkEnd;
```

These reference the walk-in start and end transforms associated with the walk-in animation.

Insert the following at the end of the `Start` method:

```
//1
position = introWalkStart.transform.localPosition;
position.x = transform.localPosition.x - cameraHalfWidth - 2.0f;
introWalkStart.transform.localPosition = position;

//2
position = introWalkEnd.transform.localPosition;
position.x = transform.localPosition.x - cameraHalfWidth + 2.0f;
introWalkEnd.transform.localPosition = position;
```

1. First, you move the walk-in start marker two units to the left of the camera's left edge.
2. Second, you move the walk-in end marker two units to the right of the camera's left edge.

Save CameraBounds.cs and open **GameManager.cs**, which is where you'll handle the hero's entrance and all its logic.

Add the following variables:

```
public Transform walkInStartTarget;
public Transform walkInTarget;
```

You added two transform references that you'll use to mark the hero's starting and ending points.

Still in `GameManager.cs`, add the following method:

```
private void DidFinishIntro() {
  actor.UseAutopilot (false);
  actor.controllable = true;
  cameraBounds.EnableBounds(true);
}
```

This method will serve as the callback parameter for `AnimateTo`, meaning that you'll trigger this method when the entrance animation finishes. It turns off the hero's autopilot and reinstates the colliders at the edge of the camera's view.

Find the `LoadLevelData` method. Add the following code before the `yield return new WaitForSeconds(0.1f);` line:

```
cameraBounds.EnableBounds(false);
actor.transform.position = walkInStartTarget.transform.position;
```

You disable the camera's edge colliders and move the hero to the `walkInStartTarget` position.

Find `yield return new WaitForSeconds(0.1f);` and add these lines just below it:

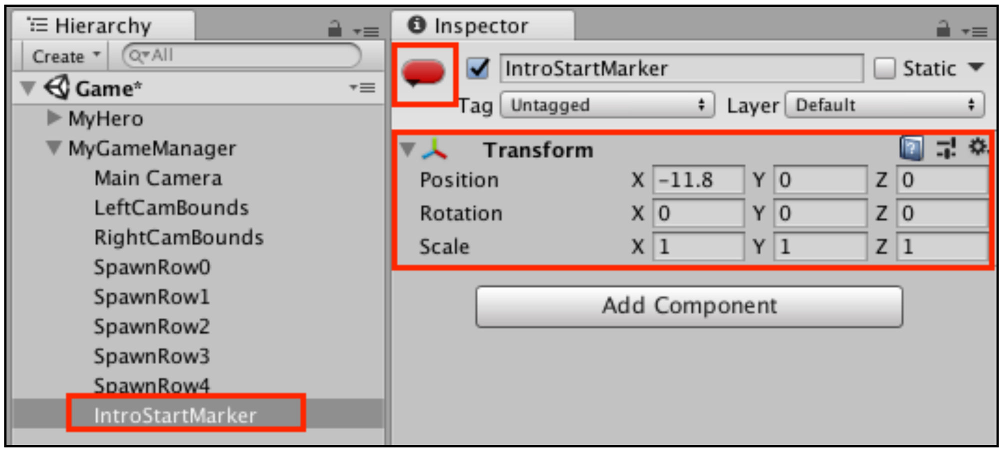
```
actor.UseAutopilot (true);
actor.AnimateTo(walkInTarget.transform.position,false, DidFinishIntro);
```

After the brief pause, you engage the hero's autopilot and walk him toward the target position.

Save the GameManager script and return to Unity to assign references that will make the walk-in animation work.

First, you need start and end markers.

Create a new **empty GameObject** named **IntroStartMarker** and make it a child of **MyGameManager**. Set its icon as the **red oval**, reset its Transform and set its **Position** to `(X:-11.8, Y:0, Z:0)`.



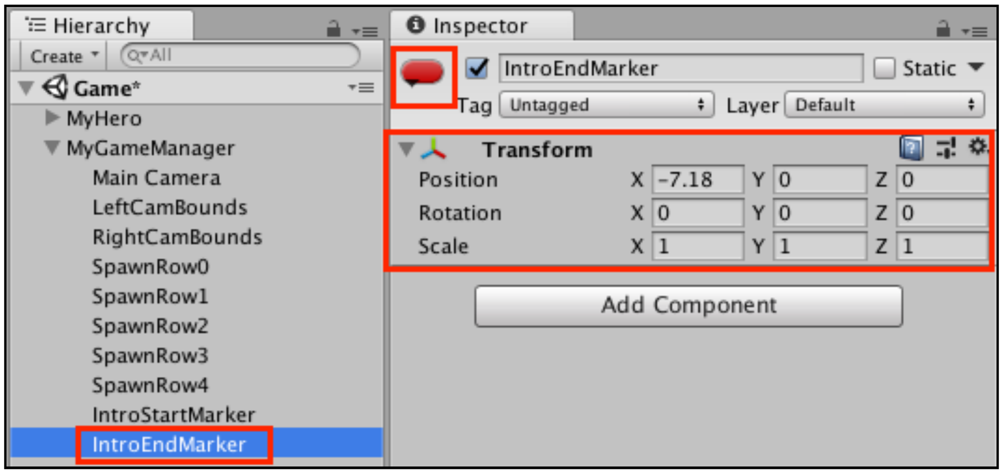
Hierarchy
Game*
- MyHero
- MyGameManager
  - Main Camera
  - LeftCamBounds
  - RightCamBounds
  - SpawnRow0
  - SpawnRow1
  - SpawnRow2
  - SpawnRow3
  - SpawnRow4
  - IntroStartMarker

Inspector
IntroStartMarker — Static
Tag Untagged — Layer Default

Transform

Position	X	-11.8	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

Duplicate **IntroStartMarker** and rename the duplicate to **IntroEndMarker**. Reset its Transform and set its **Position** to `(X:-7.18, Y:0, Z:0)`.



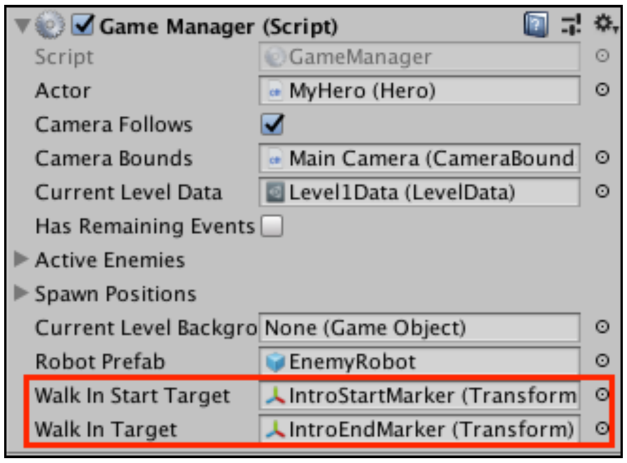
Hierarchy
Game*
- MyHero
- MyGameManager
  - Main Camera
  - LeftCamBounds
  - RightCamBounds
  - SpawnRow0
  - SpawnRow1
  - SpawnRow2
  - SpawnRow3
  - SpawnRow4
  - IntroStartMarker
  - IntroEndMarker

Inspector
IntroEndMarker — Static
Tag Untagged — Layer Default

Transform

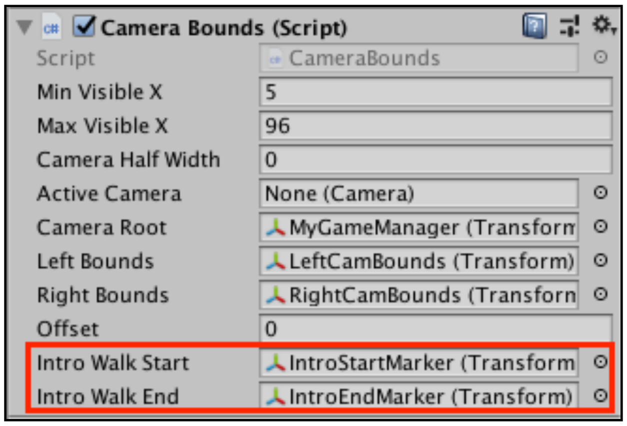
Position	X	-7.18	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

Go to **GameManager** in the Inspector. Assign **IntroStartMarker** to the **Walk In Start Target** field and **IntroEndMarker** to the **Walk In Target** field.



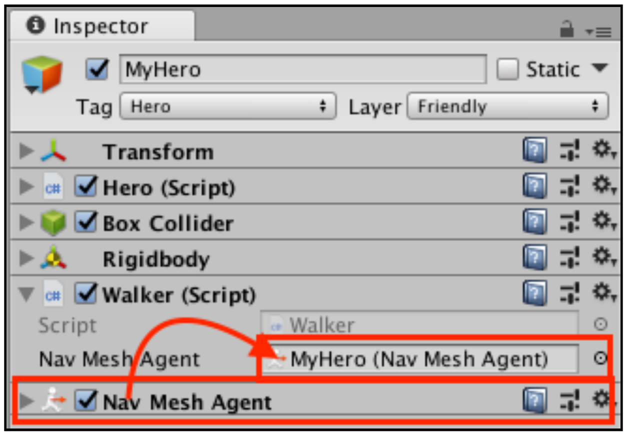
Game Manager (Script)	
Script	GameManager
Actor	MyHero (Hero)
Camera Follows	✓
Camera Bounds	Main Camera (CameraBound
Current Level Data	Level1Data (LevelData)
Has Remaining Events	
Active Enemies	
Spawn Positions	
Current Level Backgro	None (Game Object)
Robot Prefab	EnemyRobot
Walk In Start Target	IntroStartMarker (Transform
Walk In Target	IntroEndMarker (Transform)

In the **CameraBounds** component of the MainCamera, set the value of the **Intro Walk Start** and the **Intro Walk End** fields to **IntroStartMarker** and **IntroEndMarker**, respectively.

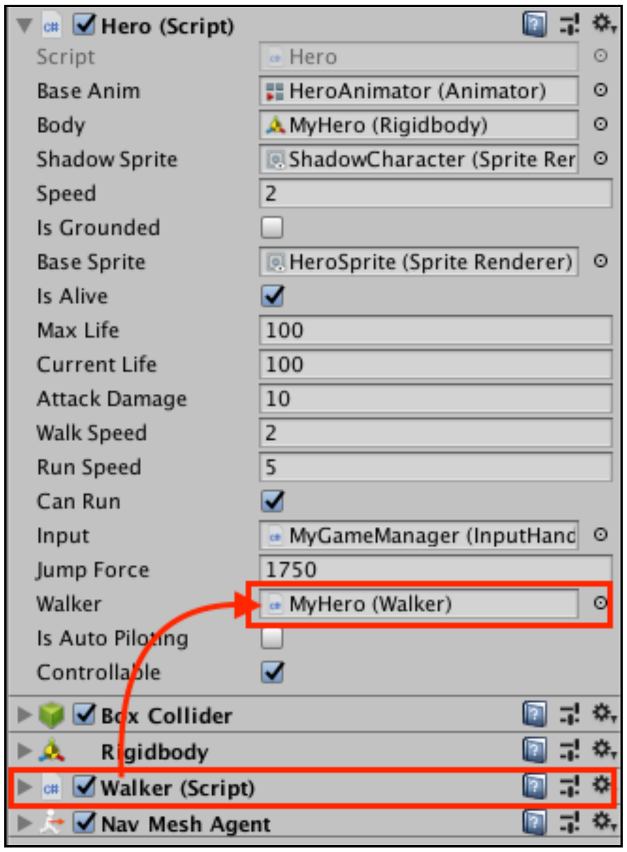


Camera Bounds (Script)	
Script	CameraBounds
Min Visible X	5
Max Visible X	96
Camera Half Width	0
Active Camera	None (Camera)
Camera Root	MyGameManager (Transform
Left Bounds	LeftCamBounds (Transform)
Right Bounds	RightCamBounds (Transforn
Offset	0
Intro Walk Start	IntroStartMarker (Transform
Intro Walk End	IntroEndMarker (Transform)

Select MyHero in the Hierarchy. Add a **Walker** component and a **NavMeshAgent** component to **MyHero**. Set the **Nav Mesh Agent** field of the Walker component to the **NavMeshAgent** component.


Inspector
MyHero Static
Tag Hero Layer Friendly
Transform
Hero (Script)
Box Collider
Rigidbody
Walker (Script)
Script Walker
Nav Mesh Agent MyHero (Nav Mesh Agent)
Nav Mesh Agent

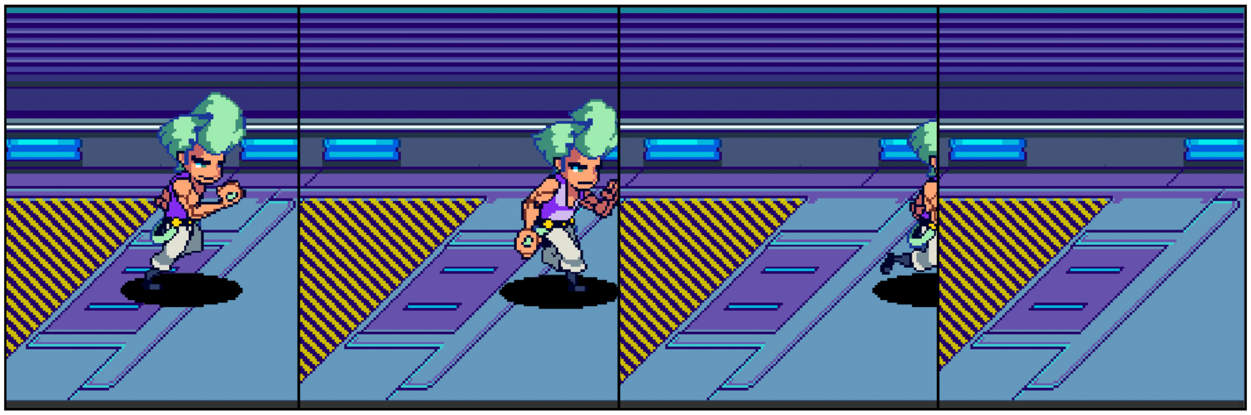
Assign the **Walker** component to the **Hero** component's **Walker** field.


Hero (Script)
Script Hero
Base Anim HeroAnimator (Animator)
Body MyHero (Rigidbody)
Shadow Sprite ShadowCharacter (Sprite Rer
Speed 2
Is Grounded ☐
Base Sprite HeroSprite (Sprite Renderer)
Is Alive ✓
Max Life 100
Current Life 100
Attack Damage 10
Walk Speed 2
Run Speed 5
Can Run ✓
Input MyGameManager (InputHanc
Jump Force 1750
Walker MyHero (Walker)
Is Auto Piloting ☐
Controllable ✓
Box Collider
Rigidbody
Walker (Script)
Nav Mesh Agent

Save the scene and **run** the game. Before you start, the hero marches into the scene from outside the camera's frame to a position inside the camera's view. Then he returns to the idle state and should be under your control.

Play the game until you destroy all the droids in the third battle event. Oddly enough, the camera continues to follow the hero. The game gives no credence to the fact that you just kicked some robot butt and therefore completed the level.

Cue the dramatic exit



You need balance in your life, and games are no different. When there's a dramatic entrance, there should be some kind of fanfare to celebrate your victorious exit.

Similarly to how you scripted the hero's grand entrance, you'll set up a dramatic exit when the player finishes a level. The main difference is that the hero will be in a hurry to leave.

Open **GameManager.cs** and add the following variable:

```
public Transform walkOutTarget;
```

This references the hero's exit point. Whenever the hero completes a level, he'll run to this transform's position.

Add these methods to `GameManager.cs`:

```
//1
private IEnumerator HeroWalkout() {
  cameraBounds.EnableBounds(false);
  cameraFollows = false;

  actor.UseAutopilot (true);
  actor.controllable = false;
  actor.AnimateTo(walkOutTarget.transform.position, true,
DidFinishWalkout);
  yield return null;
}

//2
private void DidFinishWalkout() {
  Debug.Log("Level Completed!");

  cameraBounds.EnableBounds(true);
  cameraFollows = false;
  actor.UseAutopilot (false);
  actor.controllable = false;
}
```

1. The `HeroWalkout` method is an IEnumerator that will be used in a coroutine. Similarly to the `LoadLevelData` method, it disables the camera's edge colliders and movement and puts the hero on autopilot. It then walks the hero to a point outside the camera's frame.

2. The `DidFinishWalkout` method will be the callback whenever the hero has exited the frame. It re-enables the edge colliders and puts the hero back under the player's control. Then it prints *Level Completed* to the console.

Still in the same script, insert the following code at the end of the `CompleteCurrentEvent` method:

```
if (!hasRemainingEvents) {
  StartCoroutine(HeroWalkout());
}
```

You call `HeroWalkout` when the hero emerges victorious after completing all the battle events.

Save the script and open **CameraBounds.cs**. You'll change the exit marker's transform so it's relative to the right edge of the camera's frame.

Add the following variable:

```
public Transform exitWalkEnd;
```

This references the transform of the hero's exit marker.

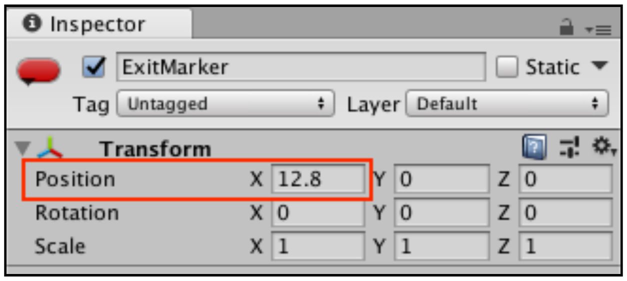
Still in `CameraBounds.cs`, insert the following lines at the end of the `Start()` method.

```
position = exitWalkEnd.transform.localPosition;
position.x = transform.localPosition.x + cameraHalfWidth + 2.0f;
exitWalkEnd.transform.localPosition = position;
```

You simply move the exit marker two units to the right of camera's right border.

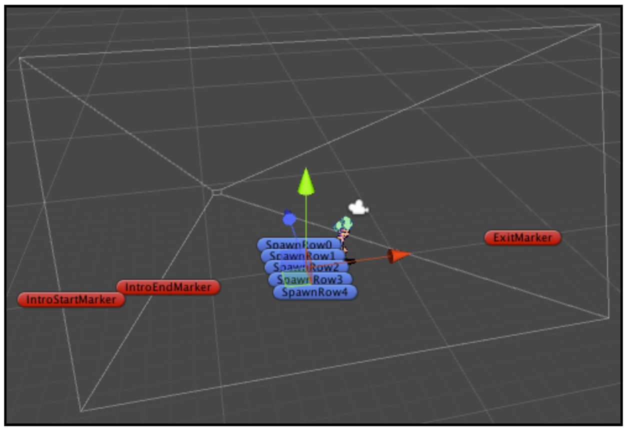
Save the CameraBounds script and return to Unity to assign the `Walk Out Target` field.

Duplicate `IntroEndMarker` and rename the duplicate to `ExitMarker`. Set its **Position** to `(X:12.8, Y:0, Z:0)`.

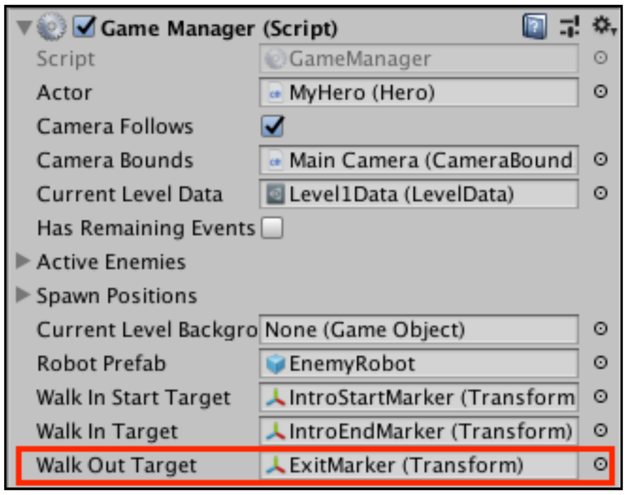


ExitMarker
Tag Untagged Layer Default
Position X 12.8 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Select **MyGameManager** in the Hierarchy. Notice the intro markers on the left side of the scene and the exit marker on the right.



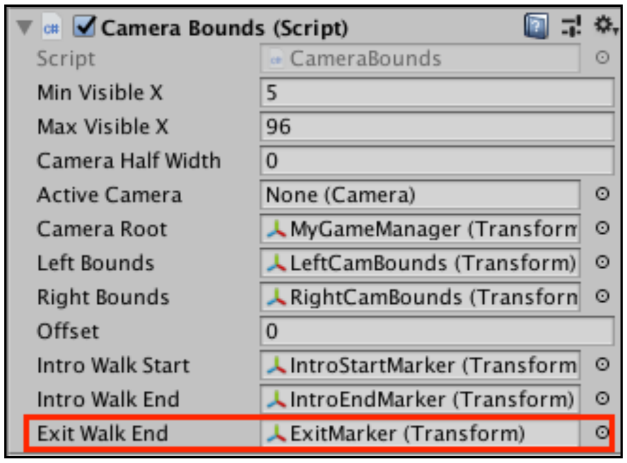
Assign **ExitMarker** to the **Walk Out Target** field of GameManager.



Game Manager (Script)

Script	GameManager
Actor	MyHero (Hero)
Camera Follows	✓
Camera Bounds	Main Camera (CameraBound
Current Level Data	Level1Data (LevelData)
Has Remaining Events	☐
Active Enemies	
Spawn Positions	
Current Level Backgro	None (Game Object)
Robot Prefab	EnemyRobot
Walk In Start Target	IntroStartMarker (Transform
Walk In Target	IntroEndMarker (Transform)
Walk Out Target	ExitMarker (Transform)

Now assign it to the **Exit Walk End** field on the Main Camera's CameraBounds component, like this:



Camera Bounds (Script)

Script	CameraBounds
Min Visible X	5
Max Visible X	96
Camera Half Width	0
Active Camera	None (Camera)
Camera Root	MyGameManager (Transforn
Left Bounds	LeftCamBounds (Transform)
Right Bounds	RightCamBounds (Transforn
Offset	0
Intro Walk Start	IntroStartMarker (Transform
Intro Walk End	IntroEndMarker (Transform)
Exit Walk End	ExitMarker (Transform)

Save the **scene** and run the game. Punch your way through the battle events and watch how the hero runs out of frame when you're done! Check the console to confirm your awesomeness.

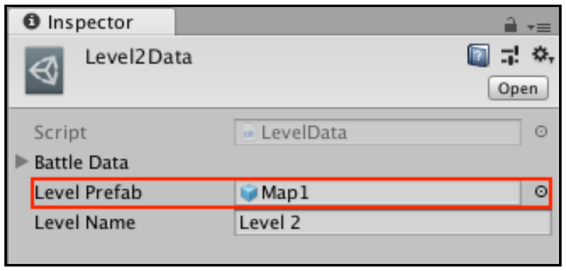
Congratulations, you've just finished creating a level for Pompadroid! What are you going to do now? Play that level again and again until you've had your fill of metal bashing? No. You're going to add more levels.

More than one level to shred a droid

The intent of the design of the **LevelData** class is that each instance represents a single level. So, you can create more levels simply by creating more LevelData ScriptableObjects.

Import **Level2Data.unitypackage** from the **Unity Packages** folder, which contains LevelData for the second level. It also contains another copy of LevelData.cs, which is the same script you've already imported.

Assign **Level Prefab** of **Level2Data** to the **Map1** prefab located in **Assets** / **Prefabs**.



Script	LevelData
▶ Battle Data	
Level Prefab	Map1
Level Name	Level 2

You'll use the same prefabs as you did for the first level.

Now that you have the data for the second level ready to go, the GameManager needs a way to start it. Open **GameManager.cs** and add the following line to the top of the script:

```
using UnityEngine.SceneManagement;
```

`SceneManagement` contains methods to handle scene loading and unloading.

Still in the same script, add the following variables:

```
public LevelData[] levels;
public static int CurrentLevel = 0;
```

The `levels` array will contain all the possible levels of the game. Levels will load based on this array, one by one. The second variable, `CurrentLevel`, will store the player's current level. The next level will load based on this value.

Still in `GameManager.cs`, in the `Start` method, find this line:

```
StartCoroutine(LoadLevelData(currentLevelData));
```

And change it to this:

```
StartCoroutine(LoadLevelData(levels[CurrentLevel]));
```

`LoadlevelData()` will use the `levels` array as its source for `LevelData` instead of using `currentLevelData`.

Find the `LoadLevelData(LevelData data)` method. Then find the line `currentLevelBackground = Instantiate(currentLevelData.levelPrefab);` and insert the following before it:

```
if (currentLevelBackground != null) {
  Destroy(currentLevelBackground);
}
```

This `if` statement destroys the `currentLevelBackground` whenever a level loads to prevent multiple instances of the map from being stacked atop one another, which could eventually crash the game.

You're almost done! Still in the same script, add this method:

```
private IEnumerator AnimateNextLevel() {
  yield return null;
  SceneManager.LoadScene("Game");
}
```

This coroutine will simulate a temporary loading period. It'll pause for a frame before loading the game scene. It's important because you'll add animations that tell the player they've completed a level in the future.

Next, find the `DidFinishWalkout` method. Replace this:

```
Debug.Log("Level Completed")
```

With this:

```
CurrentLevel++;
if (CurrentLevel >= levels.Length) {
  Debug.Log("Game Completed!");
  SceneManager.LoadScene("MainMenu");
} else {
  StartCoroutine(AnimateNextLevel());
}
```

When the hero's walkout animation ends, you increment `CurrentLevel` and check for remaining levels in the `levels` array. If there are more levels, you reload the game using the `AnimateNextLevel()` coroutine. Otherwise, it prints *Game Completed* and loads the MainMenu scene.

Save this script and open **MainMenu.cs**.

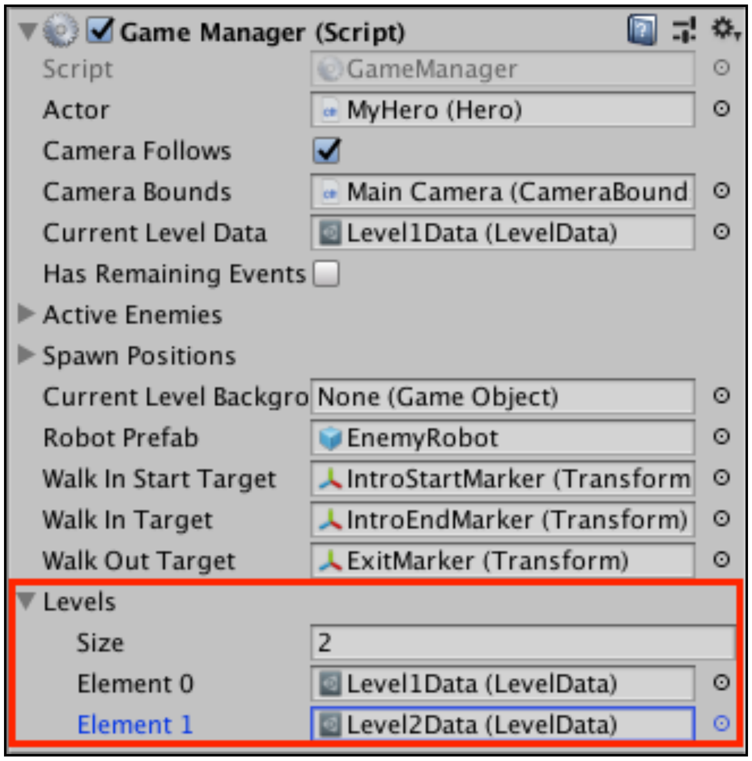
Find the `GoToGame()` method, then find the line `SceneManager.LoadScene("Game");` and insert the following above it:

```
GameManager.CurrentLevel = 0;
```

This resets the `CurrentLevel` value in the GameManager script when the player presses the play button in the main menu.

Save your scripts and return to Unity to add the two levels.

In **MyGameManager**, set the **Levels** array to a **Size** of `2` and drag **Level1Data** to **Element 0** and **Level2Data** to **Element 1**.



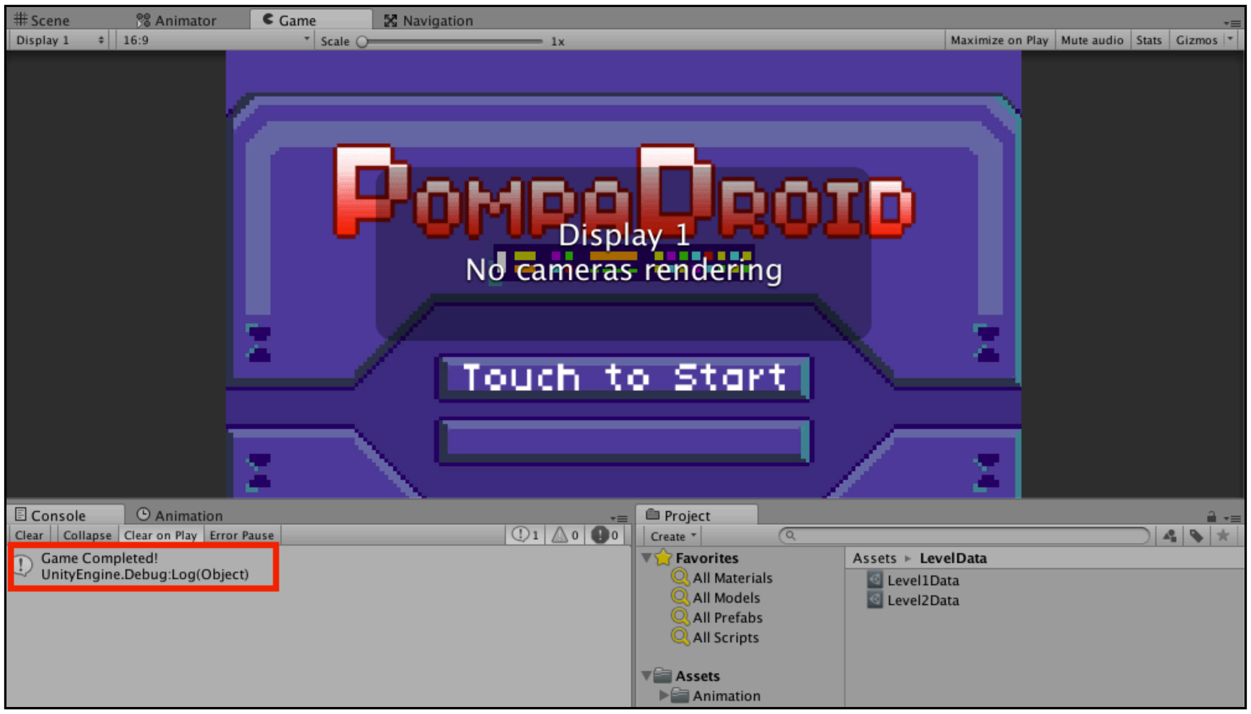
Game Manager (Script)

Script	GameManager
Actor	MyHero (Hero)
Camera Follows	✓
Camera Bounds	Main Camera (CameraBound
Current Level Data	Level1Data (LevelData)
Has Remaining Events	☐
Active Enemies	
Spawn Positions	
Current Level Backgro	None (Game Object)
Robot Prefab	EnemyRobot
Walk In Start Target	IntroStartMarker (Transform
Walk In Target	IntroEndMarker (Transform)
Walk Out Target	ExitMarker (Transform)
Levels	
Size	2
Element 0	Level1Data (LevelData)
Element 1	Level2Data (LevelData)

Run the scene again and finish the level. When you're done, you should get a new level and the hero should strut in as he did before.

You can tell it's a new level because the first battle event spawns different enemies than you saw before. Also, the second level only has two battle events.

Once you clean those droids' clocks — by winning both battle events — the game should print *Game Completed* to the console and load the **MainMenu** scene.



```
Game Completed!
UnityEngine.Debug:Log(Object)
```

Where to go from here?

Congratulations, you took Pompadroid from an interesting setting to a real game! It gives the hero suitable entrances and exits, and there are now objectives and various waves of enemies.

It takes a lot of work to create levels through ScriptableObjects, but the result is dynamic and it's easy to make new levels by making a few adjustments. It's a flexible system and easier to manage than hand coding every.single.level.

To take it a step further, you could create more LevelData assets and add their references to the GameManager to make more levels, changing up the battle events as you do.

To recap what you learned:

- Tinting to visually differentiate enemy classes
- Writing methods that tint *and* modify enemies' attributes
- Storing level data with ScriptableObjects

- Creating and implementing battle events
- Loading a level's ScriptableObject and spawning enemies as needed
- Animating the hero's entrance and exit animations
- Implementing multi-level support for Pompadroid so you can play through multiple levels

In the next chapter, you'll add more complex animations to the hero so he isn't a one-trick pony — you'll be able to perform a flurry of different attacks, such as jump attacks, run attacks and a serious three-punch combo! Bots beware.

Chapter 8: Power Attacks

You know what makes thrashing robots more satisfying? When you can do it with some style. I'm talking about combination attacks and knockouts and sparks flying.

This chapter is all about adding features to the game, so you can expect a lot of jumping between scripts, the editor and animator windows, and game play. Nothing unfamiliar lies ahead, but there are many tasks to do. You should be able to move through things pretty quickly now!

In this chapter, you'll:

- Define a new AttackData class to process attacks
- Implement the hero's droid-bashing combo attacks: jump, run, and triple punch.
- Implement knockdown and get up behavior for hero and enemy.
- Tighten up your code and fix some bugs.
- Add a hit effect and hurt threshold to help the hero overcome an embarrassing weakness.

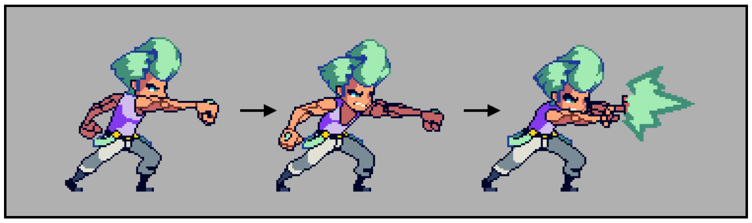
Set yourself up for a bit of time at your desk and get to it!

The rules of engagement

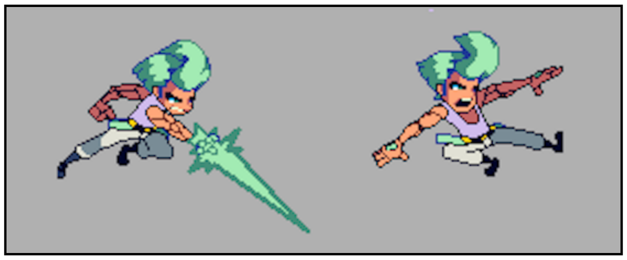
The pompadoured protagonist isn't exactly what you'd call formidable. Well, maybe if you were a can of hair spray.

And let's face it. He's got a weak punch!

Normal Attacks will comprise the basic punches. They are weaker than their special counterparts, but can be sequenced together to create a string of attacks.



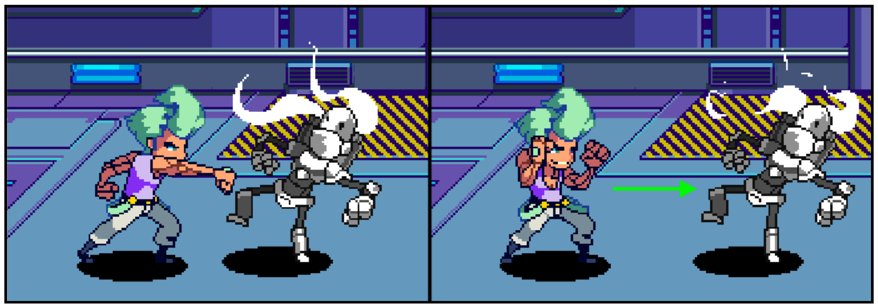
Special Attacks comprise the run and jump attacks, and they require special conditions to activate. They are stronger than normal attacks but also one-hit wonders that cannot be sequenced.



Before adding the complex attacks, you'll need to modify the attack code in the actor classes. Punching is about to get more intricate!

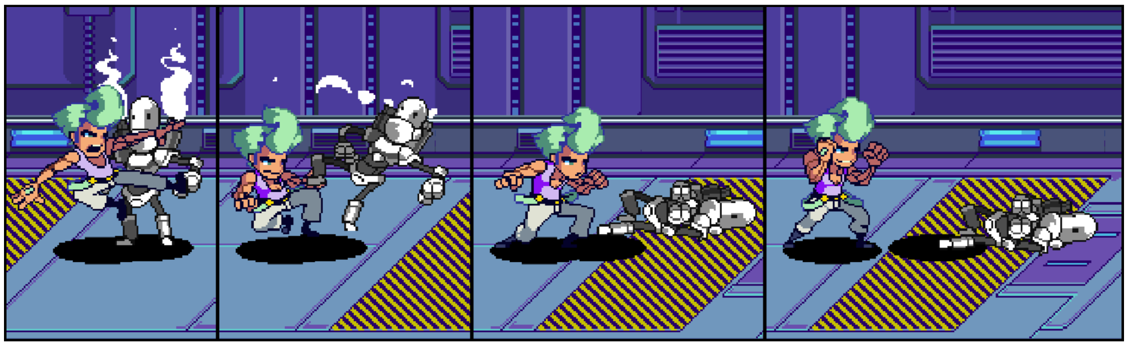
Attack overview

The first component is **pushback force**, which nudges the actor that receives a punch backward.



How far an actor moves back depends on how much pushback force is applied. Higher pushback means the hero gets more space to breath after punching a droid — useful when multiple enemies attack.

The next component is **knockdown**, which you'll use to guarantee certain punches result in a KO.



Of course, there will be variables to store damage dealt, pushback force and whether the punch returns a knockout.

If you accounted for all of the hero's attacks, you'd end up with 15 variables to track. Sounds messy. Hence, you'll create a new AttackData class to store these values, rather than storing them in the Hero class.

Amping up the attack

Open **Actor.cs** and add the following class declaration at the bottom of the file, **outside** the `Actor` class, underneath the last closing bracket:

```
[System.Serializable]
public class AttackData {
  public float attackDamage = 10;
  public float force = 50;
  public bool knockdown = false;
}
```

AttackData contains the attack's damage value, its pushback `force` value, and a `knockdown` flag. This class also has a `Serializable` attribute to make its variables visible in the Inspector.

Add the below to the top of the `Actor` class, beneath the other variables:

```
public AttackData normalAttack;
```

`normalAttack` will store the basic attack values for all actors.

Currently, the game relies on `HitActor` from the Actor class to calculate damage. You'll move this function to another method and add pushback to the attack calculation.

Add the following method to the `Actor` class:

```
public virtual void EvaluateAttackData(AttackData data, Vector3
hitVector, Vector3 hitPoint) {
  body.AddForce(data.force * hitVector);
  TakeDamage(data.attackDamage, hitVector);
}
```

`EvaluateAttackData` will process the attack. It takes three parameters: the `data` of the attack, hit direction as `hitVector`, and `hitPoint`, which is the position where the hit took place.

This method applies the pushback `force` variable of the AttackData to the Rigidbody of this Actor. Then it multiplies `data.force` by the direction of the hit, because the actor needs to lunge backward in the opposite direction of the hero's punch. Lastly, it executes `TakeDamage` using the AttackData's `attackDamage` and `hitVector` values.

Still in **Actor.cs**, find this line in the `HitActor` method:

```
actor.TakeDamage(attackDamage, hitVector);
```

Replace it with the following:

```
actor.EvaluateAttackData(normalAttack, hitVector, hitPoint);
```

`HitActor` will evaluate an attack by running this newly created `EvaluateAttackData` method on the receiving actor.

Next, remove this variable from the Actor class:

```
public float attackDamage = 10;
```

You'll use `normalAttack` to store the attack values for this class, making `attackDamage` redundant.

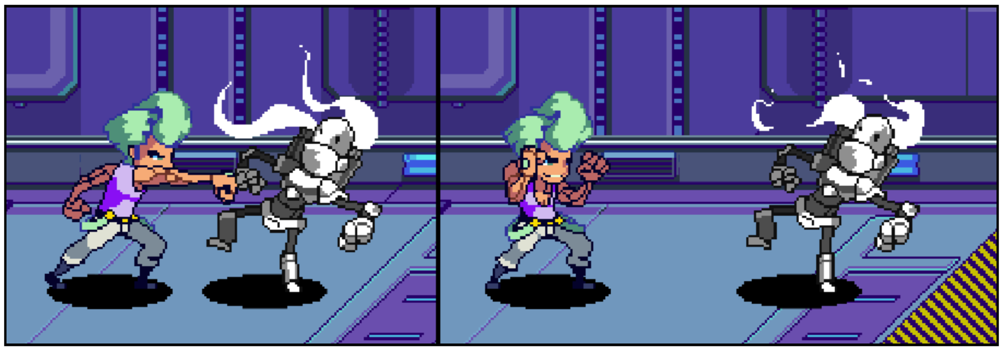
Note: Removing this variable will result in bugs, as you should expect when you mess with your code. If you play the game now, you'll see Unity's complaints — ignore them because you'll fix them next.

Save this script and open **Robot.cs**. In the `SetColor` method, replace all references to `attackDamage` with `normalAttack.attackDamage`, for example, `attackDamage = 2;` should become `normalAttack.attackDamage = 2;`.

And that should stop the complaints about the missing `attackDamage` variable.

Nice job! You just reworked the mechanics of punching in PompaDroid. **Save** Robot.cs and return to Unity. Open up the **Hero** component. Looks like the initial attack values were reset.

In the Hero component's **Normal Attack** field, set **Attack Damage** to `10` and **Force** to `500`. Click **Play** and test out that fancy new pushback feature by destroying a droid. Wow! Just a little (ok a lot) too far.



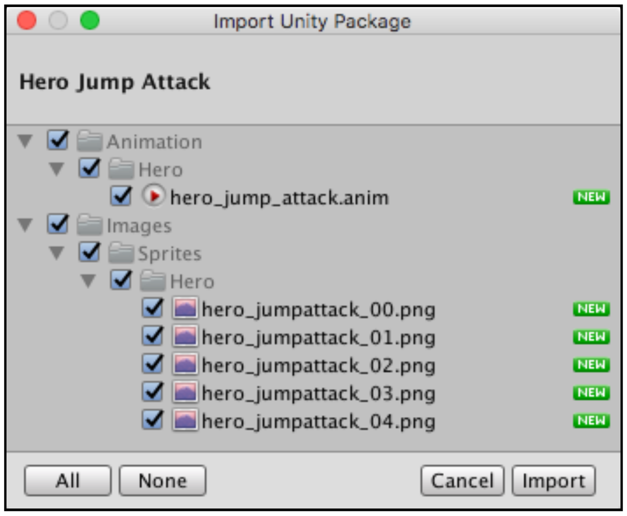
The greater the force value, the further backwards the actor goes. 500 is obviously excessive, so set **Normal Attack** to the Hero's default values of **Attack Damage** to `5` and **Force** to `15`.

Awesome work! You just finished `AttackData` and you're about to give the hero some sweet new moves, starting with a jump attack.

Creating the jump attack

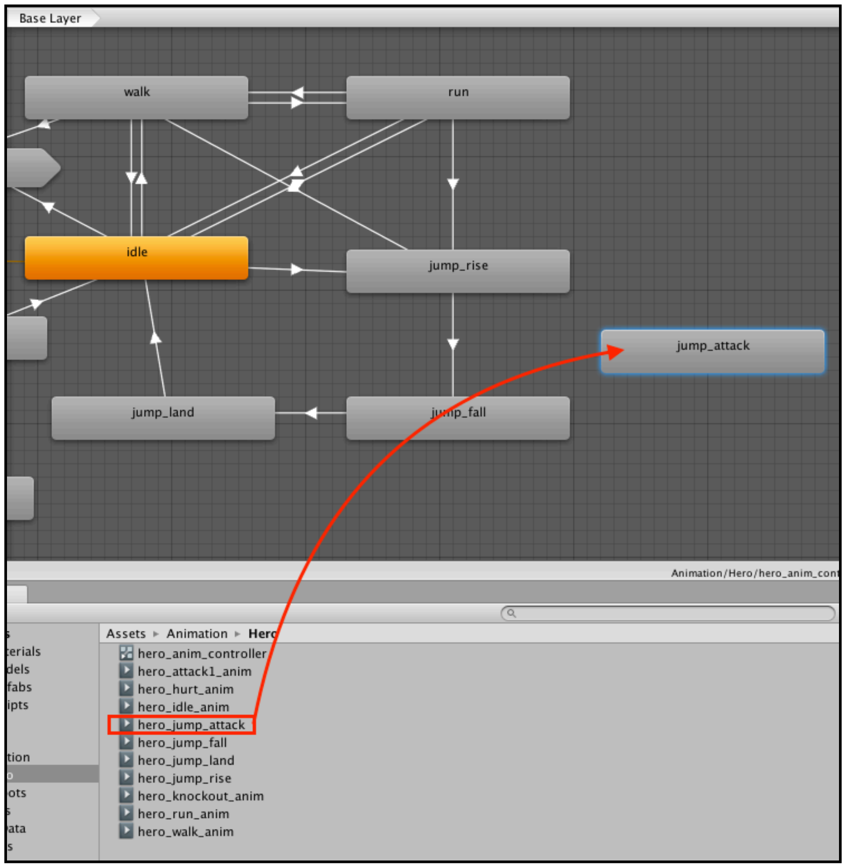
During the attack, which is triggered by pressing the jump and attack buttons, the hero will be unaffected by gravity until the animation ends. He'll also knock out enemies in front of him with a vicious punch attack.

Import **Hero Jump Attack.unitypackage** from the **Unity Packages** folder (in the Git for this chapter). In here, you have assets for implementing the jump attack animation.



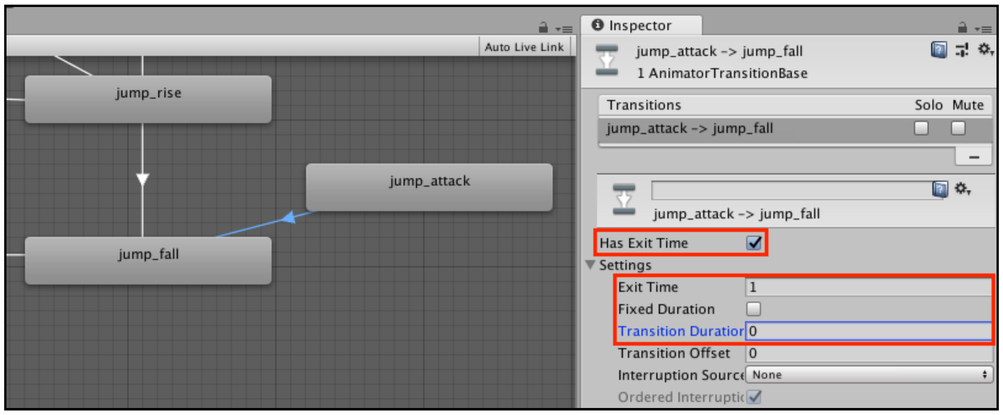
Time to add this new state to the HeroAnimator's state machine.

Double-click **hero_anim_controller** inside **Assets / Animation / Hero** to open the animator window. Drag the **hero_jump_attack** clip from the same folder to the grid layout to create a new state. Rename this state to **jump_attack**.



Add a **transition** from **jump_attack** to **jump_fall** — this is the transition when the jump_attack completes, and then the hero returns to the falling state.

Keep **Has Exit Time** checked and set **Exit Time** to `1.0`, uncheck **Fixed Duration** and set **Transition Duration** to `0`.



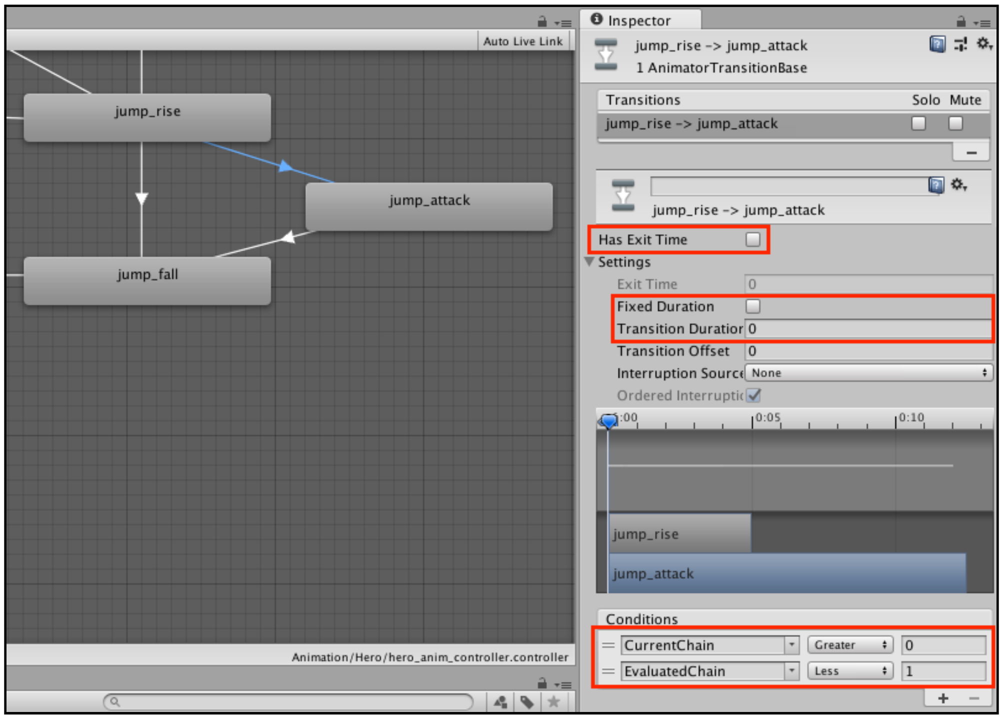
jump_attack -> jump_fall
1 AnimatorTransitionBase

Transitions	Solo	Mute
jump_attack -> jump_fall	☐	☐

jump_attack -> jump_fall
Has Exit Time ✓
Settings
Exit Time 1
Fixed Duration ☐
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Add a transition from **jump_rise** to **jump_attack**. Uncheck **Has Exit Time** and **Fixed Duration** then set **Transition Duration** to `0`.

Also add conditions with **CurrentChain Greater** than `0` and **EvaluatedChain Less** than `1`.



jump_rise -> jump_attack
1 AnimatorTransitionBase

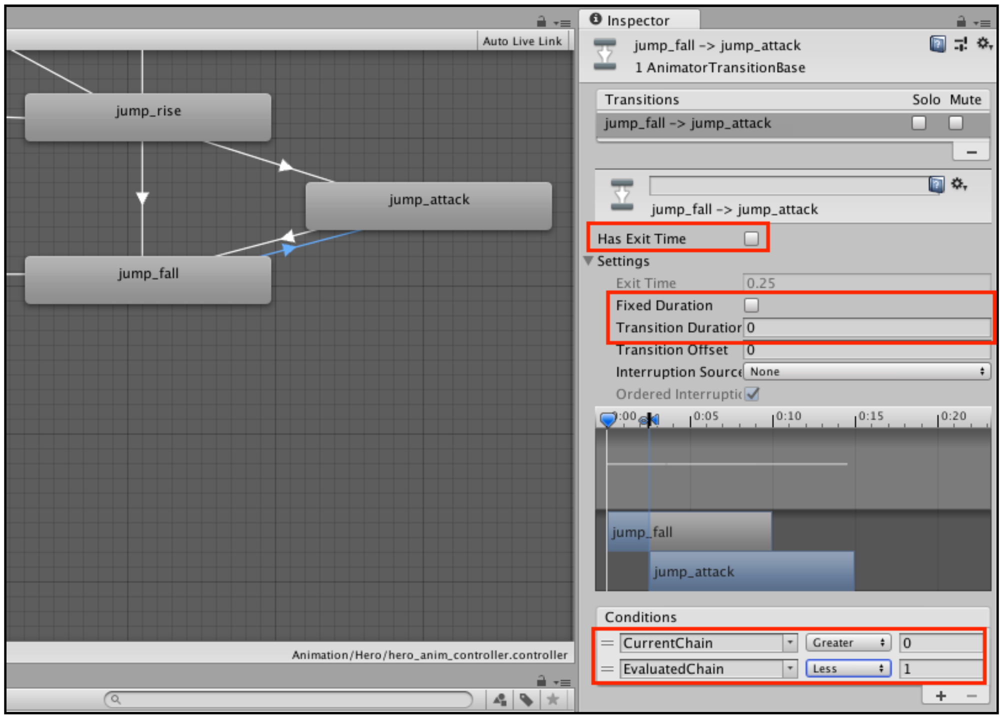
Transitions	Solo	Mute
jump_rise -> jump_attack	☐	☐

jump_rise -> jump_attack
Has Exit Time ☐
Settings
Exit Time 0
Fixed Duration ☐
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Conditions

CurrentChain	Greater	0
EvaluatedChain	Less	1

Create another transition from **jump_fall** to **jump_attack**. Set its parameters to the same values you set from **jump_rise** to **jump_attack**.



jump_fall -> jump_attack
1 AnimatorTransitionBase

Transitions	Solo	Mute
jump_fall -> jump_attack	☐	☐

jump_fall -> jump_attack

Has Exit Time	☐
Exit Time	0.25
Fixed Duration	☐
Transition Duratior	0
Transition Offset	0
Interruption Source	None
Ordered Interruptio	☑

Conditions		
CurrentChain	Greater	0
EvaluatedChain	Less	1

The transitions for the jump attack are done.

Next up, you’ll create the logic for the attack, so open **Hero.cs** and add the following variables:

```
public bool canJumpAttack = true;
private int currentAttackChain = 1;
public int evaluatedAttackChain = 0;
public AttackData jumpAttack;
```

A player can only use the jump attack once per jump, and the `canJumpAttack` variable tracks this. `currentAttackChain` and `evaluatedAttackChain` will trigger attacks for the hero. The new `jumpAttack` variable contains the hit data for the jump attack.

You need to override `OnCollisionEnter` in the Hero class, so add this method:

```
protected override void OnCollisionEnter(Collision collision) {
  base.OnCollisionEnter(collision);
  if (collision.collider.name == "Floor") {
    canJumpAttack = true;
  }
}
```

Here you call the Actor’s `OnCollisionEnter` method using the `base.OnCollisionEnter(collision);` line. When the object hit is named `Floor`, the `canJumpAttack` value is reset, allowing the player to jump attack again.

Still in **Hero.cs**, add this method:

```
public void DidJumpAttack() {
  body.useGravity = true;
}
```

`DidJumpAttack` handles what happens after a jump attack — the hero’s rigidbody is affected by gravity once again.

You also need to replace the `Attack()` method with this:

```
public override void Attack() {
  //1
  if (!isGrounded) {
    //2
    if (isJumpingAnim && canJumpAttack) {
      //3
      canJumpAttack = false;

      //4
      currentAttackChain = 1;
      evaluatedAttackChain = 0;
      baseAnim.SetInteger ("EvaluatedChain", evaluatedAttackChain);
      baseAnim.SetInteger ("CurrentChain", currentAttackChain);

      //5
      body.velocity = Vector3.zero;
      body.useGravity = false;
    }
  } else {
    //6
    currentAttackChain = 1;
    evaluatedAttackChain = 0;
    baseAnim.SetInteger ("EvaluatedChain", evaluatedAttackChain);
    baseAnim.SetInteger ("CurrentChain", currentAttackChain);

  }
}
```

The `Attack` method is responsible for determining which attack the hero will perform.

1. Checks if the hero is in the air — in other words, if he is currently jumping.
2. Checks more conditions. The hero must be in a jump animation but not have previously performed a jump attack in this particular jump.
3. Performs the jump attack if section 1 and 2 pass. Here you set `canJumpAttack` to `false` to prevent the hero from doing a second jump attack in this particular jump.
4. Sets `currentAttackChain` and `evaluatedAttackChain` to `1` and `0`, respectively, to prevent chaining with other attacks.

5. Pauses the hero's rigidbody in the air until the animation ends by setting `body.velocity` to `Vector3.zero` and `body.useGravity` to `false`.

Okay, that wraps up what happens with the jump attack for the hero.

Now think about what you've done to the enemies' reactions to the new attack. Oh, that's right — nothing. You should modify how those walking scrap heaps sustain damage. Jump attacks should carry more weight than your basic punch.

Add this method to the script:

```
private void AnalyzeSpecialAttack(AttackData attackData, Actor actor,
Vector3 hitPoint, Vector3 hitVector) {
  actor.EvaluateAttackData(attackData, hitVector, hitPoint);
}
```

Here you have a helper method that facilitates processing of special attacks. It calls the actor's `EvaluateAttackData` with parameters that give details about the hit that just occurred.

Add this override method for `HitActor` to the Hero script:

```
protected override void HitActor(Actor actor, Vector3 hitPoint, Vector3
hitVector) {
  if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1")) {
    base.HitActor (actor, hitPoint, hitVector);
  } else if
(baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_attack")) {
    AnalyzeSpecialAttack (jumpAttack, actor, hitPoint, hitVector);
  }
}
```

With this, you override Actor's `HitActor` method. It executes after somebody gets punched.

First, you check if the current Animator is in the "attack1" state. If so, the Actor's `HitActor` handles the hit. Otherwise, it checks if the current animator is in the "jump_attack" state. If so, it processes the hit with `AnalyzeSpecialAttack`, taking `jumpAttack` as a parameter.

Save Hero.cs and open **HeroCallback.cs**.

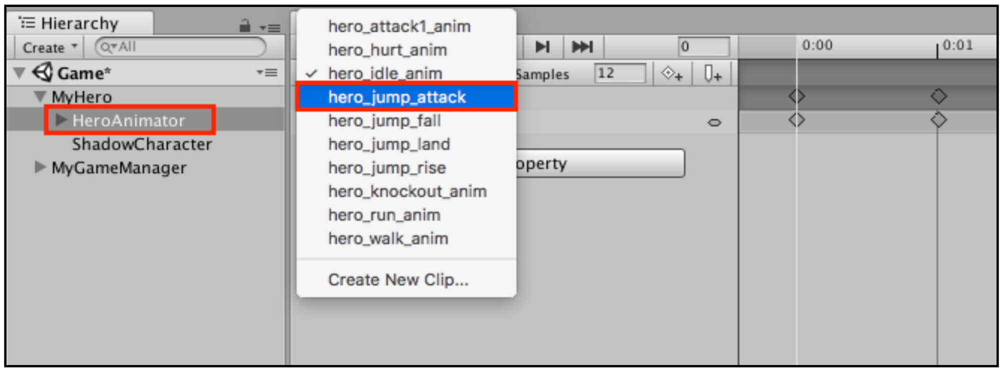
Add the following:

```
public void DidJumpAttack() {
  hero.DidJumpAttack();
}
```

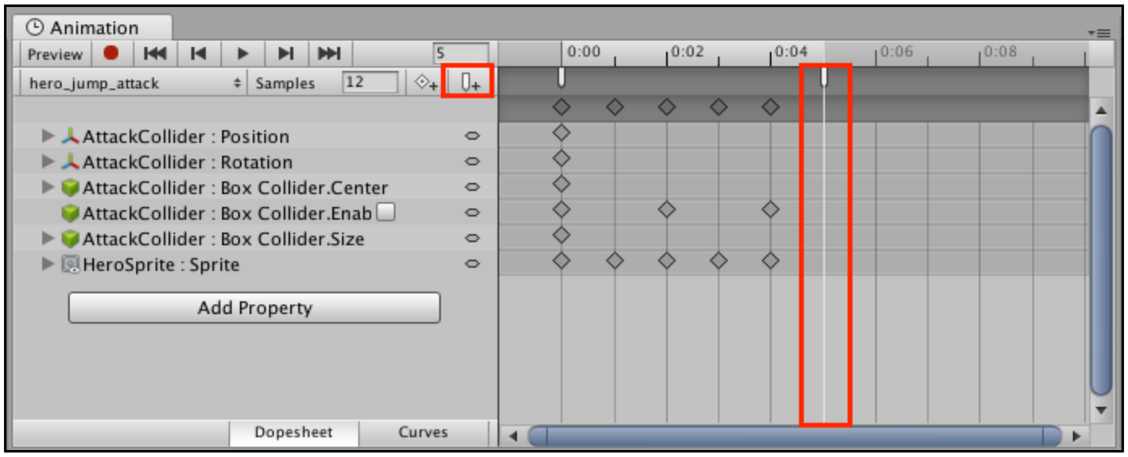
Here you simply add a way for the `HeroCallback` class to trigger the hero's `DidJumpAttack` method. This is necessary to allow `HeroAnimator`, which has the `HeroCallback` component, to trigger this method when it is done playing the jump_attack animation.

Save the script and return to Unity. Time to set some references!

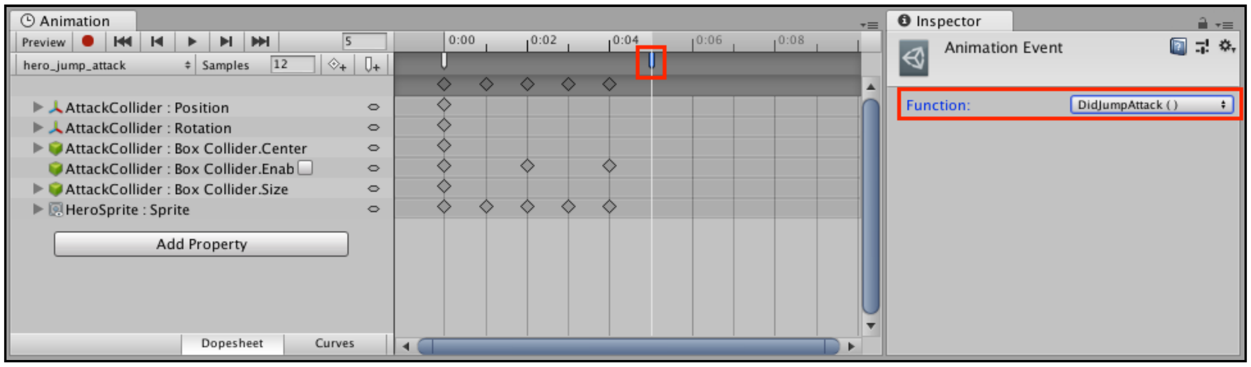
Open the **Animation window** and select the **HeroAnimator** in the Hierarchy to load all the animations for the HeroAnimator GameObject. Set the animation to **hero_jump_attack**.



Select the last frame of the animation and click the **Animation Event** button.

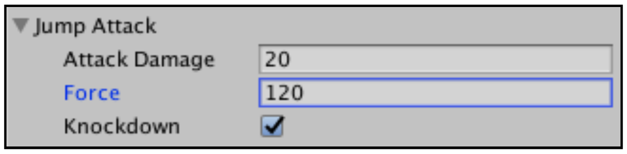

Animation
Preview
5
hero_jump_attack
Samples 12
0:00
0:02
0:04
0:06
0:08
AttackCollider : Position
AttackCollider : Rotation
AttackCollider : Box Collider.Center
AttackCollider : Box Collider.Enab
AttackCollider : Box Collider.Size
HeroSprite : Sprite
Add Property
Dopesheet
Curves

Set **DidJumpAttack** as this event's **Function**.


Animation
Preview
hero_jump_attack
Samples 12
AttackCollider : Position
AttackCollider : Rotation
AttackCollider : Box Collider.Center
AttackCollider : Box Collider.Enab
AttackCollider : Box Collider.Size
HeroSprite : Sprite
Add Property
Dopesheet
Curves
Inspector
Animation Event

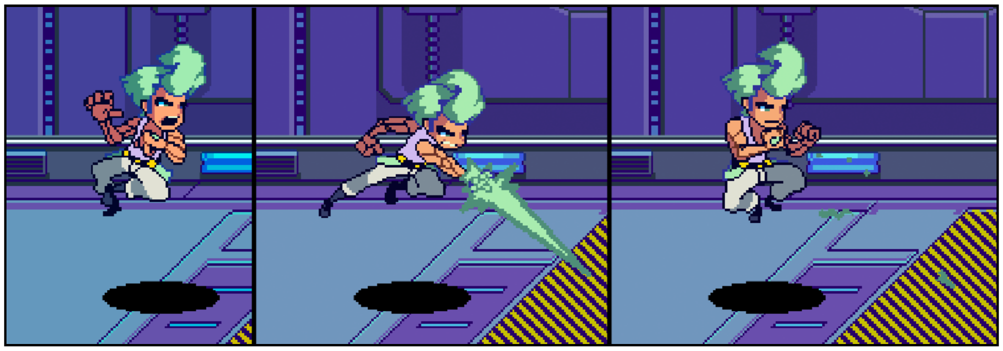
Function: DidJumpAttack ()

In the Hero component of MyHero, set the **Attack Damage** and **Force** of the **Jump Attack** variable to `20` and `120`. Also, check the **Knockdown** checkbox.

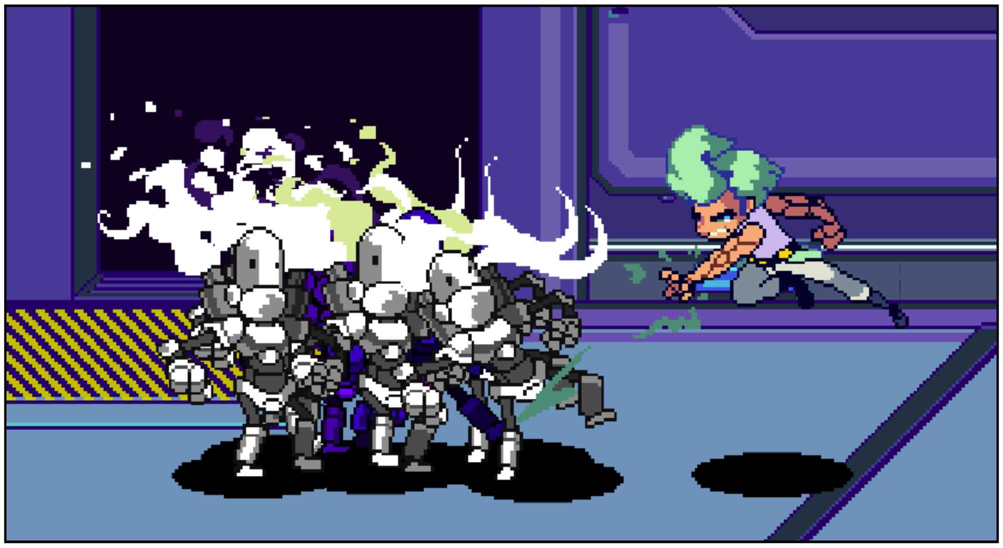


Jump Attack	
Attack Damage	20
Force	120
Knockdown	✓

Save the scene and project then click **Play**. You can now perform a jump attack! Just press jump, and while in the air, press attack!



Looks like the pushback force is working, but you haven't implemented the knockdown yet — you'll get to that in a bit.

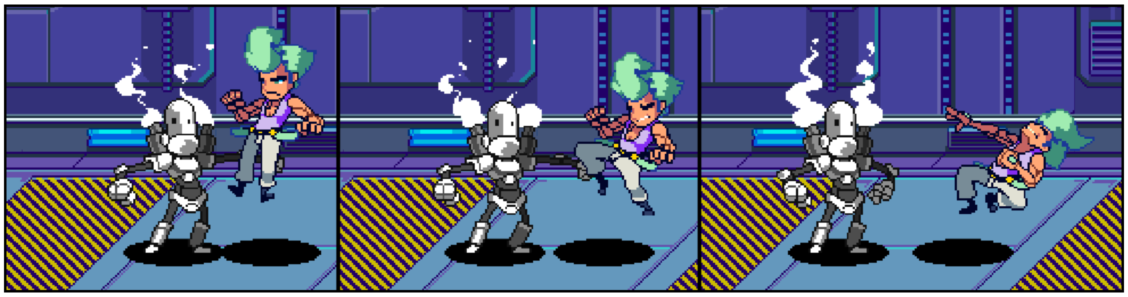


Make sure you let yourself take a punch from a droid during a jump attack. The hero hangs in the air. Amusing and quite fixable. What's happening is that the game did not call `DidJumpAttack`, so gravity remains disabled for the hero.

You'll fix this next as you add the knockout feature. Soon, all actors will be prone to knockdowns, but they'll be able to get back up and keep fighting too.

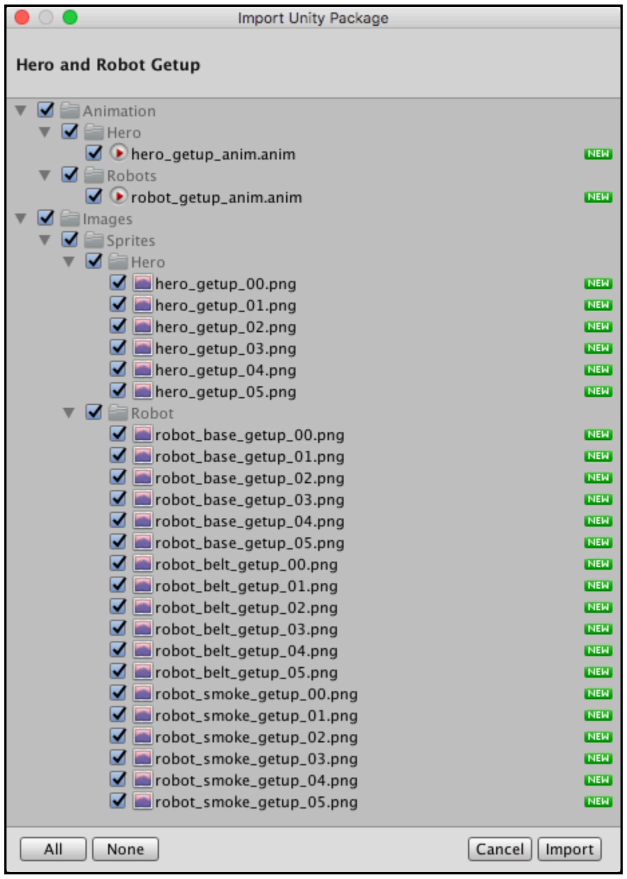
Rise of the machines! (and the hero too)

Think about what should trigger the knockdown event for the hero. Obviously, he is vulnerable when airborne (isn't everybody?), so it makes sense that he should crash to the ground when he takes a hit in the air.



You already have the falling down part — you'll duplicate the death animations for that. Get-up animations are the part that you'll need to create.

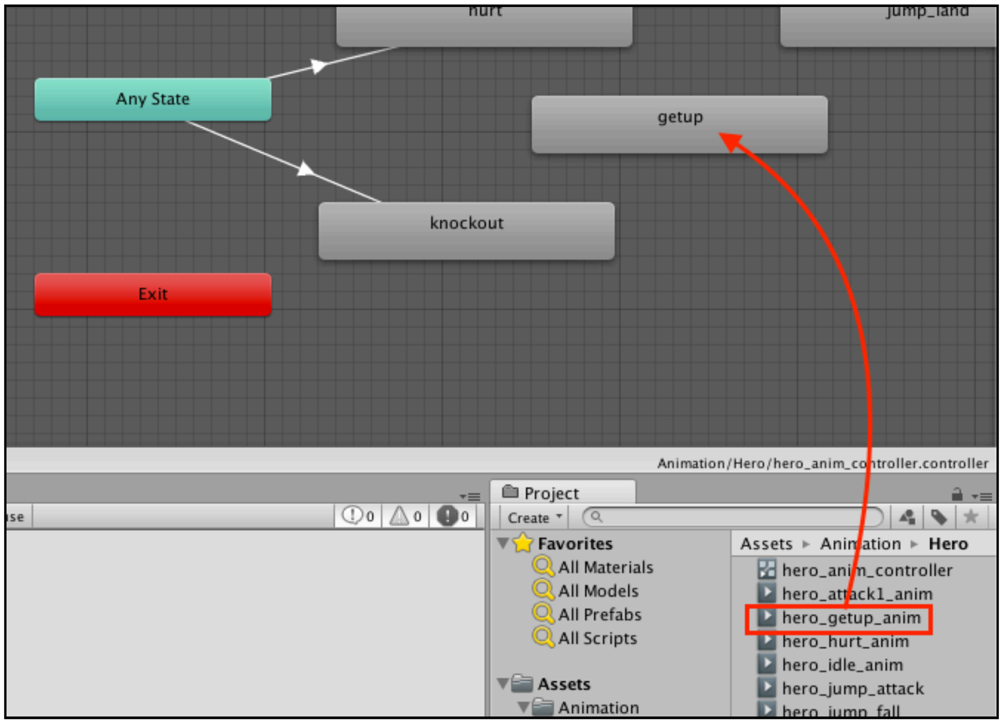
Import the **Robot and Hero Getup.unitypackage** from the **Unity Packages** folder.



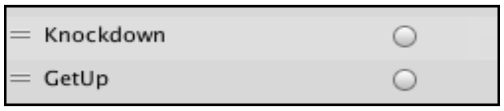
Hero and Robot Getup

```
▼ ☑ Animation
  ▼ ☑ Hero
      ☑ hero_getup_anim.anim                NEW
  ▼ ☑ Robots
      ☑ robot_getup_anim.anim               NEW
▼ ☑ Images
  ▼ ☑ Sprites
    ▼ ☑ Hero
        ☑ hero_getup_00.png                 NEW
        ☑ hero_getup_01.png                 NEW
        ☑ hero_getup_02.png                 NEW
        ☑ hero_getup_03.png                 NEW
        ☑ hero_getup_04.png                 NEW
        ☑ hero_getup_05.png                 NEW
    ▼ ☑ Robot
        ☑ robot_base_getup_00.png           NEW
        ☑ robot_base_getup_01.png           NEW
        ☑ robot_base_getup_02.png           NEW
        ☑ robot_base_getup_03.png           NEW
        ☑ robot_base_getup_04.png           NEW
        ☑ robot_base_getup_05.png           NEW
        ☑ robot_belt_getup_00.png           NEW
        ☑ robot_belt_getup_01.png           NEW
        ☑ robot_belt_getup_02.png           NEW
        ☑ robot_belt_getup_03.png           NEW
        ☑ robot_belt_getup_04.png           NEW
        ☑ robot_belt_getup_05.png           NEW
        ☑ robot_smoke_getup_00.png          NEW
        ☑ robot_smoke_getup_01.png          NEW
        ☑ robot_smoke_getup_02.png          NEW
        ☑ robot_smoke_getup_03.png          NEW
        ☑ robot_smoke_getup_04.png          NEW
        ☑ robot_smoke_getup_05.png          NEW
```

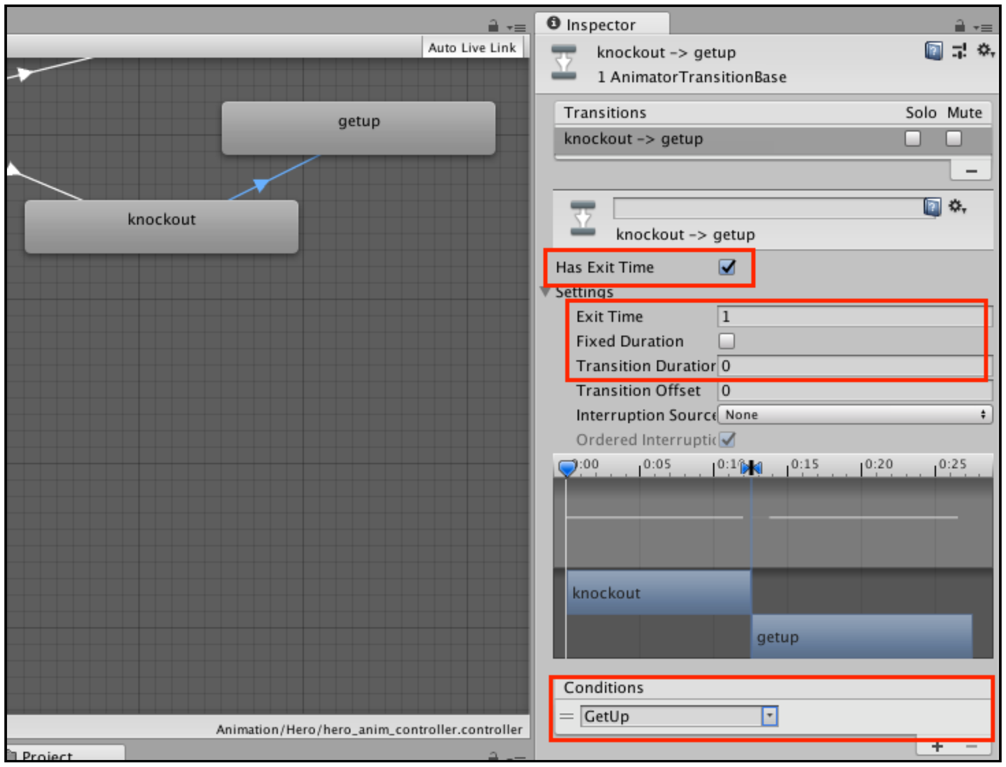
Double-click **hero_anim_controller** from the **Assets / Animation / Hero** folder to open the Animator window. Drag the **hero_getup_anim** into the grid layout, and rename the newly created state **getup**.



Next, add two trigger parameters to this Animator, and name them **Knockdown** and **GetUp**. These two will serve as the event triggers for the knockdown and get up animations of the hero.



Add a transition from **knockout** to **getup**. Keep **Has Exit Time** checked, set **Exit Time** to `1`, uncheck **Fixed Duration** and set **Transition Duration** to `0`. Add a condition with the trigger **GetUp**. Now the get-up animation will play after the knockout animation finishes.



knockout -> getup
1 AnimatorTransitionBase

Transitions	Solo	Mute
knockout -> getup	☐	☐

knockout -> getup

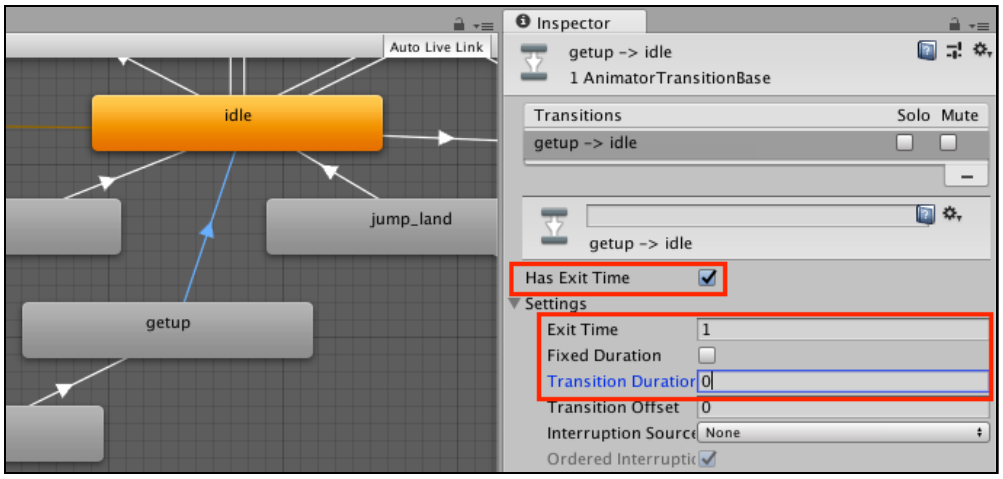
Has Exit Time ✓

Settings

Exit Time 1
Fixed Duration ☐
Transition Duratior 0
Transition Offset 0
Interruption Source None
Ordered Interruptic ✓

Conditions
GetUp

Add another transition, this time from **getup** to **idle**. Use the same parameters as the knockout to getup animation, without adding any conditions.



getup -> idle
1 AnimatorTransitionBase

Transitions	Solo	Mute
getup -> idle	☐	☐

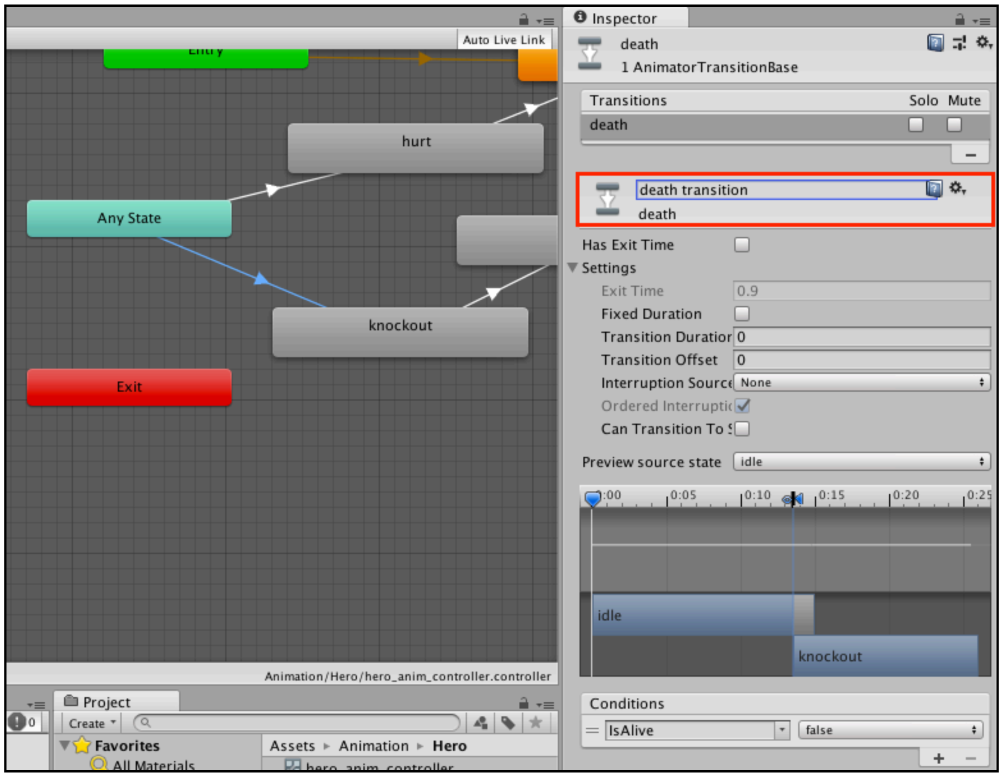
getup -> idle

Has Exit Time ✓

Settings

Exit Time 1
Fixed Duration ☐
Transition Duratior 0
Transition Offset 0
Interruption Source None
Ordered Interruptic ✓

Select the transition from **Any State** to **knockout** and rename it **death transition**. This name change is purely for convenience.



death
1 AnimatorTransitionBase

Transitions	Solo	Mute
death	☐	☐

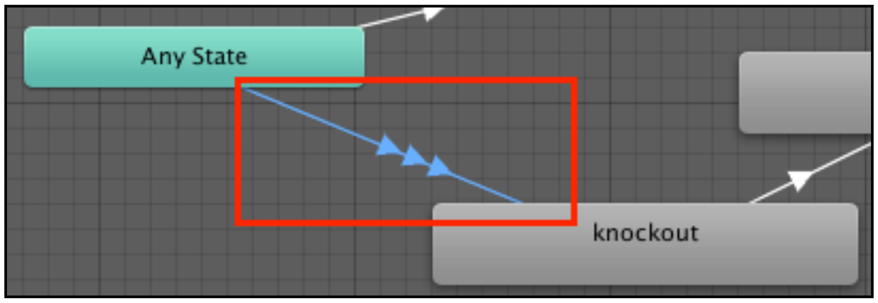
death transition
death

Has Exit Time ☐
Settings
Exit Time 0.9
Fixed Duration ☐
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓
Can Transition To Self ☐

Preview source state idle

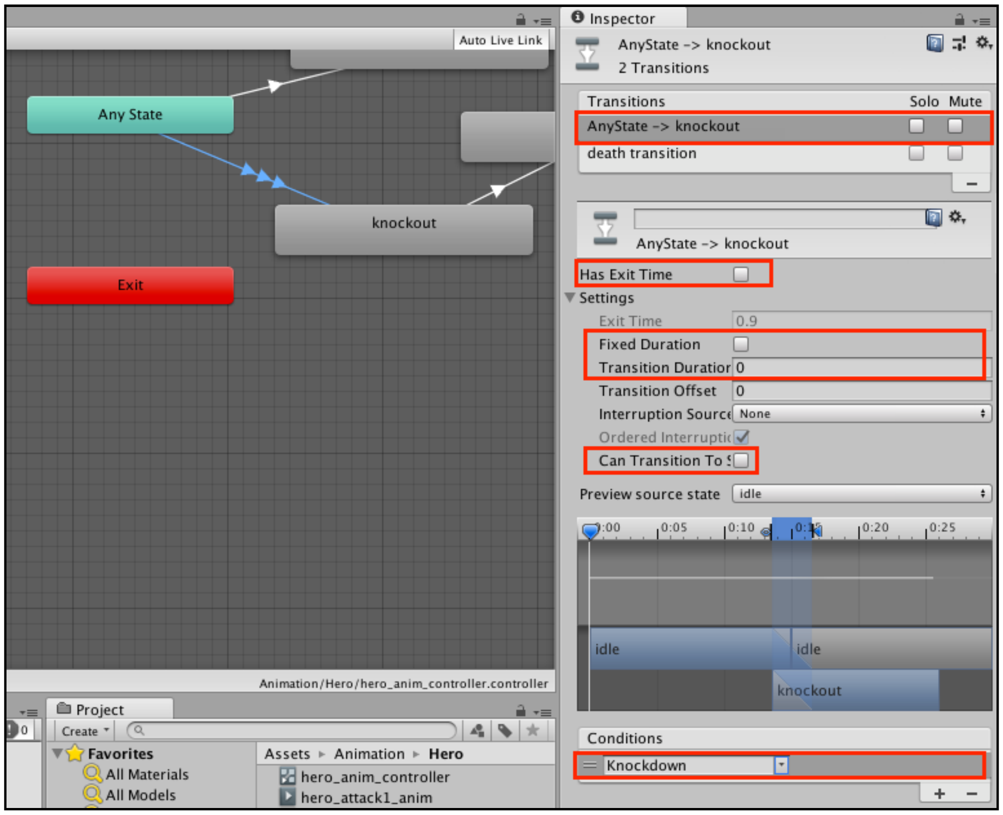
Conditions
IsAlive false

Add another transition from **Any State** to **knockout**. As this is the second transition between these states, the transition arrow will change to a multipoint arrow.



Select the transition with the name **AnyState -> knockout**.

Keep **Has Exit Time** unchecked. Uncheck both **Fixed Duration** and **Can Transition To Self** and set **Transition Duration** to `0`. Add a condition with the **Knockdown** trigger.



AnyState -> knockout
2 Transitions

Transitions	Solo	Mute
AnyState -> knockout	☐	☐
death transition	☐	☐

AnyState -> knockout

Has Exit Time ☐
Settings
Exit Time 0.9
Fixed Duration ☐
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓
Can Transition To Self ☐
Preview source state idle

Conditions
Knockdown

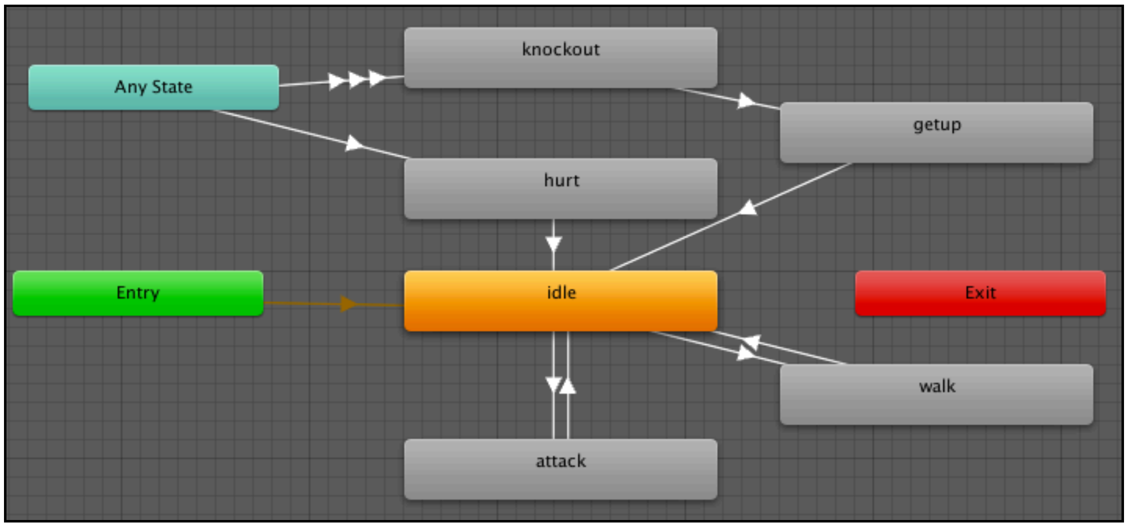
Great work! You're finished with the states and transitions for the hero knockdown effect.

Repeat for the droids

Now just do the same thing for the robot's Animator Controller. Well, mostly the same. Use the **robot_anim_controller** as the controller, and the **robot_getup_anim** as the animation clip.

Use the same naming conventions for the robot controller and follow the same structure.

Your robot state machine should look like this:



State positions may vary on your screen — just make sure that you've set up the transitions correctly.

Coding for KOs

Now you just need to tell the game when to use those awesome animations.

Open the **Actor.cs** script and add the `knockdownRoutine` and `isKnockedOut` variables:

```
protected Coroutine knockdownRoutine;
public bool isKnockedOut;
```

You'll use `knockdownRoutine`, a coroutine, to animate the actor's knockdown, and `isKnockedOut` is a Boolean that tells you whether the actor is knocked down or not.

Insert the following condition at the start of the `Die` method:

```
if (knockdownRoutine != null) {
  StopCoroutine(knockdownRoutine);
}
```

Here you stop any knockdown coroutines that are running when the actor dies.

Add the following method to the class:

```
public void DidGetUp() {
  isKnockedOut = false;
}
```

You'll call `DidGetUp` when the actor's get-up animation finishes. It resets the `isKnockedOut` Boolean to false.

Replace the contents of the `CanBeHit` method with the following:

```
public bool CanBeHit() {
  return isAlive && !isKnockedOut;
}
```

`CanBeHit` checks if the hero is alive and not knocked down before allowing damage. An actor cannot be damaged when it is on the floor, or worse, dead.

Next, add this to the Actor class:

```
protected virtual IEnumerator KnockdownRoutine() {
  isKnockedOut = true;
  baseAnim.SetTrigger("Knockdown");
  yield return new WaitForSeconds (1.0f);
  baseAnim.SetTrigger ("GetUp");
  knockdownRoutine = null;
}
```

`KnockdownRoutine` handles the animation for the knockdown.

1. Sets `isKnockedOut` to `true`, then triggers the "Knockdown" event parameter in the `baseAnim` Animator.
2. Pauses for 1 second then triggers the "Getup" animation and clears the `isKnockedOut` flag.

Your next move might trigger some errors, because you are adding more parameters to the `TakeDamage` method — ignore them. You'll fix them by the time you're done here.

Change the declaration of `TakeDamage` from this:

```
public virtual void TakeDamage(float value, Vector3 hitVector) {
```

To this:

```
public virtual void TakeDamage(float value, Vector3 hitVector, bool
knockdown = false) {
```

This will add a third parameter, `knockdown`, to the method. You're giving it a default value of `false`, so you needn't explicitly state that when calling this method.

With `TakeDamage` changed, you can use either `TakeDamage(10, Vector.one)` or `TakeDamage(10, Vector3.one, false)` and yield the same result.

However, you will need to explicitly enter `true` as the third parameter when you want to set `knockdown` to `true`.

Save the Actor script and open **Enemy.cs**.

Find the below `TakeDamage` method signature:

```
public override void TakeDamage(float value, Vector3 hitVector) {
```

And replace it with:

```
public override void TakeDamage(float value, Vector3 hitVector, bool
knockdown = false) {
```

That should convince the compiler to stop complaining.

Find this at the bottom of the `TakeDamage` method:

```
base.TakeDamage(value, hitVector);
```

Change it to:

```
base.TakeDamage(value, hitVector, knockdown);
```

Here you enable the enemy to use the new `knockdown` parameter of the `TakeDamage` method.

Save Enemy.cs and open **Actor.cs** again. In the `EvaluateAttackData` method, replace this line:

```
TakeDamage(data.attackDamage, hitVector);
```

With this line:

```
TakeDamage(data.attackDamage, hitVector, data.knockdown);
```

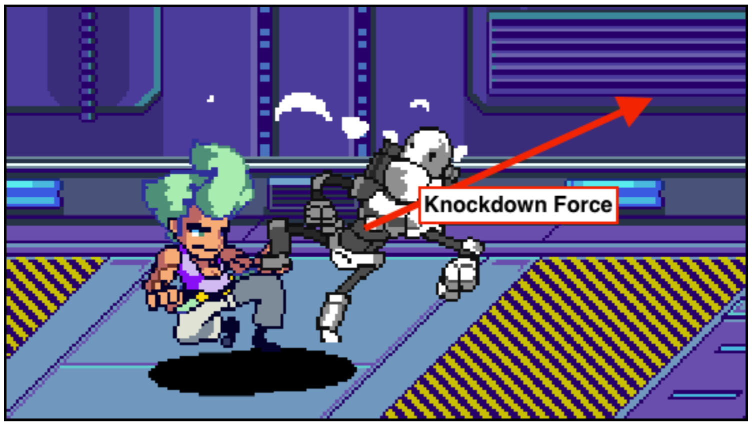
This enables all actor instances to take the third parameter of the `TakeDamage` method, assuming they aren't overriding `EvaluateAttackData`.

Insert the following condition in the `TakeDamage` method, just after the closing `}` of the `if` statement but before the `else`.

```
else if (knockdown) {
  if (knockdownRoutine == null) {
    Vector3 pushbackVector = (hitVector + Vector3.up*0.75f).normalized;
    body.AddForce (pushbackVector* 250 );
    knockdownRoutine = StartCoroutine(KnockdownRoutine());
  }
}
```

You're just sliding this little conditional between the `if` check of death and the hurt animation.

If the attack is not lethal and is a knockdown, this `else if` condition is used. When `knockdown` is `true`, and no knockdown animation is currently playing, it exerts backward and upward force and plays the actor's `KnockdownRoutine` coroutine.



Your next task will make the hero instantly fall to the ground when he takes a hit while airborne.

Save Actor.cs and open **Hero.cs**, and add the following method override:

```
public override void TakeDamage(float value, Vector3 hitVector, bool
knockdown = false) {
  if (!isGrounded) {
    knockdown = true;
  }
  base.TakeDamage(value, hitVector, knockdown);
}
```

Here you override the Actor's `TakeDamage` method. First you check if the hero is not grounded. If `isGrounded` is false, you set the `knockdown` parameter to true.

When the get-up animation completes, you'll need to trigger the `DidGetUp` method in the Actor class. To do so, you need to add a way for your Animators to trigger this method. So, you'll make a new component for that.

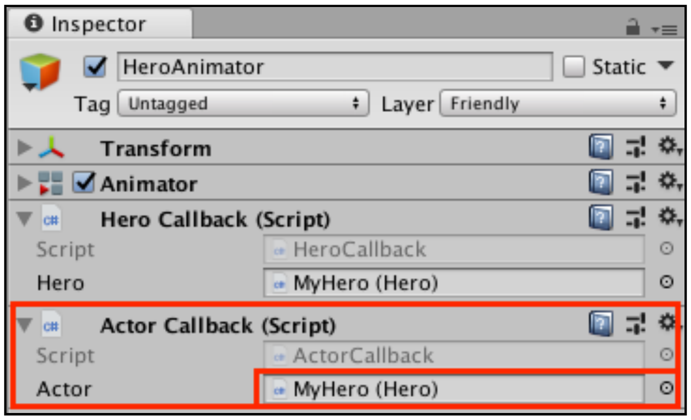
Save Hero.cs and create a new **C# script** in **Assets / Scripts** named **ActorCallback**.

Replace its contents with the following:

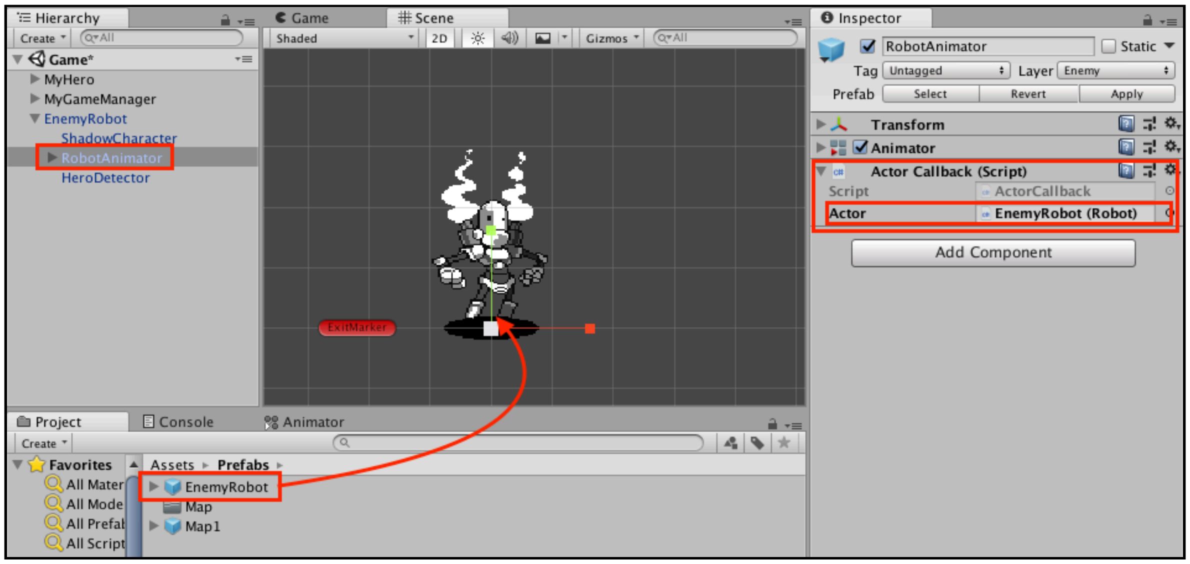
```
using UnityEngine;

public class ActorCallback : MonoBehaviour {
  public Actor actor;

  public void DidGetUp() {
    actor.DidGetUp();
  }
}
```

Here you have a reference to a target `Actor`. Then in `DidGetUp`, you simply forward the call to that actor's `DidGetUp` method. As you can see, this script's sole purpose is to trigger the `DidGetUp` method of the actor assigned to it. It acts as a standalone component that adds this single function to any GameObject. In this case, you will add it to your Animator GameObjects.

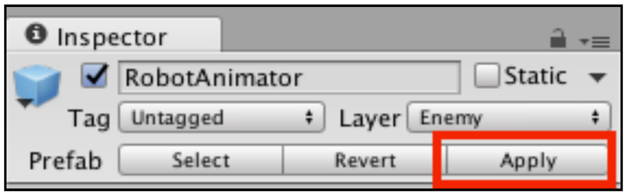
Save this script and return to Unity. Select **HeroAnimator** in the Hierarchy and add an **ActorCallback**. Set **MyHero** as its **Actor** variable.



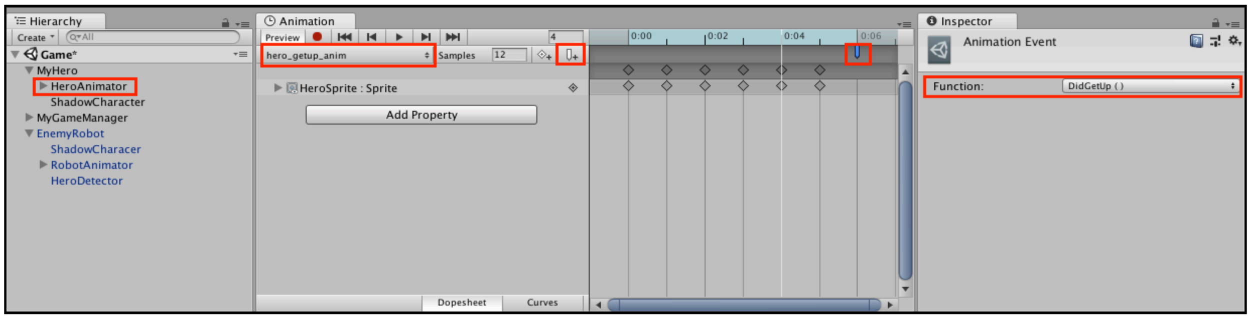
Drag an **EnemyRobot** prefab into the scene from the **Prefabs** folder. Select its **RobotAnimator** and add an **ActorCallback** component. Set its **Actor** to **EnemyRobot**.



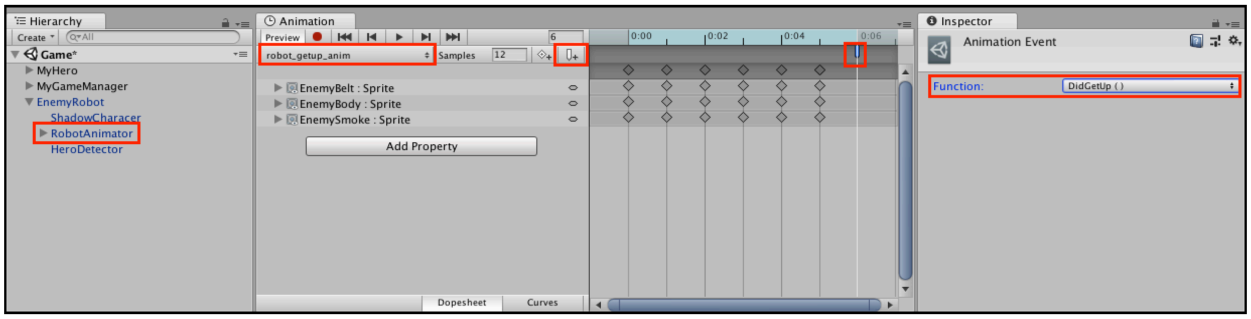
Click **Apply** near the top-right corner of the Inspector to overwrite the existing prefab with the instance you just modified.



Open the **Animation window** then select **HeroAnimator** in the Hierarchy. Select the **hero_getup_anim** clip. At the last frame, add an **Animation Event** and set **DidGetUp()** as its **Function**.

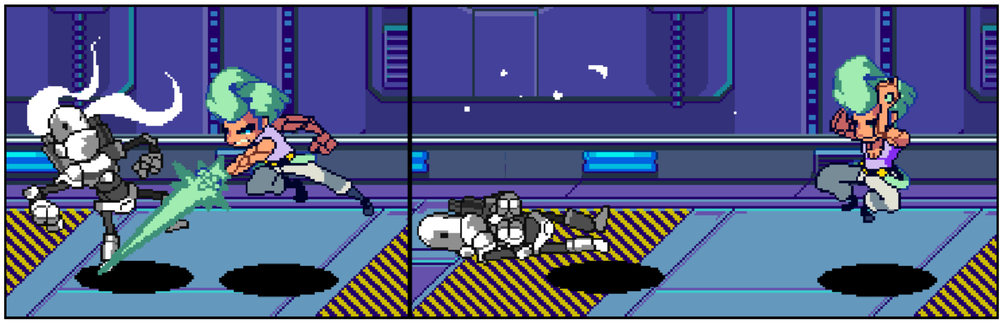


Next, select the **RobotAnimator** in the Hierarchy, and in the Animation window, select the **robot_getup_anim** clip. Also, add an animation event at the last frame with the **DidGetUp** method.

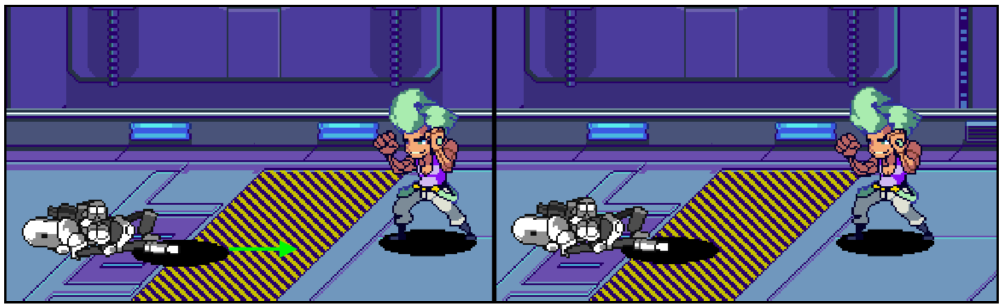


Delete the **EnemyRobot** in the Hierarchy. No worries, your changes committed to the prefab!

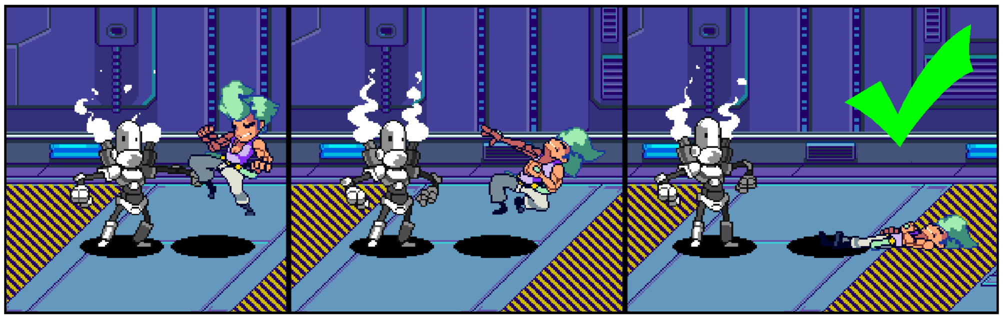
Save the scene and project then test the game. Try the jump attack again. Ha! Poor droid goes flying backward now.

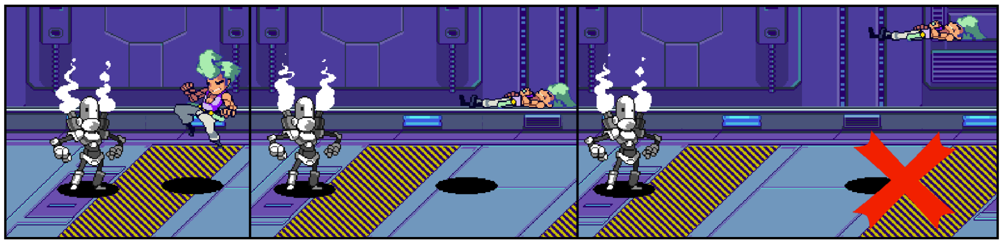


Defiantly it sneaks towards you in spite of being knocked down. Creepy…



Next, try jumping when a robot is about to punch you. When you get hit while in the air, you'll be either knocked down (which is good!) or fly out in the air. Yikes!





Next, try moving while the hero is down. Whoah! That's not right. Your logic is more or less functional but bug-stricken, as you might expect when you change all the things.

Exterminate a few bugs

Consider this your debugging intermission. Ahead of you are more features to add, and you'll get to them after you debug the last set of features. It is a good practice to debug fully in between feature sets; it keeps your code clean and helps prevent superbugs down the road.

Open **Hero.cs** and add the following variable declaration to the top of the script:

```
bool isHurtAnim;
```

This Boolean will track whether the current animation is the hurt clip or not.

Next, in the `Update` method, just after the `isJumpingAnim` assignment statement, add the following line:

```
isHurtAnim =
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("hurt");
```

This sets the `isHurtAnim` variable to `true` when the hurt animation plays.

Find this:

```
isAttackingAnim =
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1");
```

Replace it with:

```
isAttackingAnim =
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1")  ||
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_attack");
```

Two pipes (`||`) are how you add an `or` condition to the statement, so now a jump_attack animation evaluates as a type of attack. Remember this because you'll do it again soon.

While still in the `Update` method, find this:

```
if (jump &&
```

And replace it with this:

```
if (jump && !isKnockedOut &&
```

This prevents the hero from jumping when he is knocked down.

Next, in the `Update` method, find this:

```
if (attack && Time.time >= lastAttackTime + attackLimit) {
    lastAttackTime = Time.time;
    Attack();
  }
```

And upgrade it to this:

```
  if (attack && Time.time >= lastAttackTime + attackLimit && !isKnockedOut)
  {
    lastAttackTime = Time.time;
    Attack();
  }
```

Here you prevent the hero from attacking when he's down.

Next, in the `FixedUpdate` method, find this after the line `Vector3 moveVector = currentDir * speed;`

```
  if (isGrounded && !isAttackingAnim) {
```

And change it to this:

```
  if (isGrounded && !isAttackingAnim && !isJumpLandAnim && !isKnockedOut &&
  !isHurtAnim) {
```

Here you disable the hero's movement during a jump, and when hurt or knocked down.

While still in `FixedUpdate` method, find this line:

```
  if (moveVector != Vector3.zero) {
```

Replace with the following:

```
  if (moveVector != Vector3.zero && isGrounded && !isKnockedOut && !
  isAttackingAnim) {
```

You're preventing the hero from flipping back and forth while airborne or during a hit.

Next, add this override method:

```
  public override bool CanWalk () {
    return (isGrounded && !isAttackingAnim && !isJumpLandAnim && !
  isKnockedOut && !isHurtAnim);
  }
```

You're overriding `CanWalk` from the `Actor` class — you might recall it determines when the `Walker` class makes an actor walk. With this method, the hero will not walk while knocked down or hurt.

Another scenario presents with a bug: when punched mid-air, the hero remains immune to gravity. You need to reset the Rigidbody's `useGravity` variable to `true` in this case.

First, add the following `using namespace` statement to Hero.cs, if not already added:

```
using System.Collections;
```

You will use the `IEnumerator` interface for coroutines from the `System.Collections` namespace.

Now add an override for the `KnockedDownRoutine` method:

```
protected override IEnumerator KnockdownRoutine() {
  body.useGravity = true;
  return base.KnockdownRoutine();
}
```

With that, you prevent the hero from getting stuck in the air by enabling gravity for the hero's Rigidbody before triggering the knockdown.

Save the Hero script and open **Enemy.cs**. Robots have a little zombie bug to zap.

In the `CanWalk` method, replace the `return` statement with the following:

```
return !baseAnim.GetCurrentAnimatorStateInfo(0).IsName("hurt") &&
  !baseAnim.GetCurrentAnimatorStateInfo(0).IsName("getup");
```

You're adding the "getup" animation to the robot's walk state. The Walker will prevent walking when the robot is getting up.

Save Enemy.cs and open **Robot.cs**. You're here to disable the AI when the enemy is knocked down.

Add the following override to the class:

```
protected override IEnumerator KnockdownRoutine() {
  isKnockedOut = true;
  baseAnim.SetTrigger ("Knockdown");
  ai.enabled = false;

  yield return new WaitForSeconds (2.0f);
  baseAnim.SetTrigger ("GetUp");
  ai.enabled = true;

  knockdownRoutine = null;
}
```

This overrides the `KnockdownRoutine` of the Actor class so that you can enable different behavior for the robot knockdown. First, you disable the AI and play the knockdown animation when the knockdown routine starts. Then you keep the robot down a little longer than the hero and trigger the get-up animation and AI.

Save **Robot.cs**, return to Unity and click **Play**. Try jumping just before a robot punches to trigger a knockdown. Note how you're unable to move while you're on the ground.

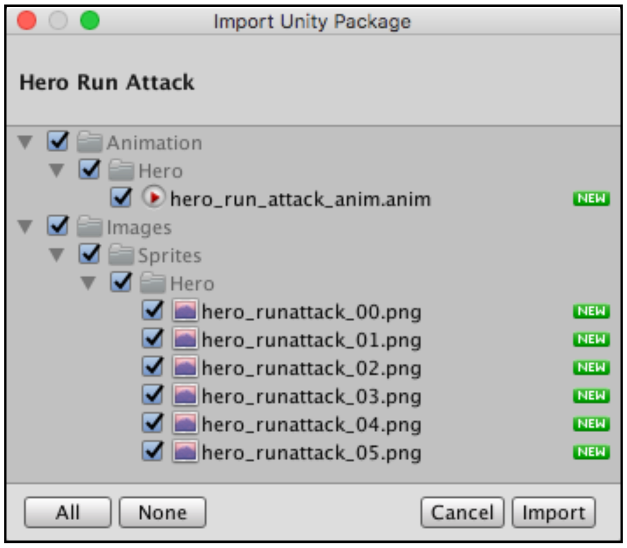
Congratulations Pompadoured Hero, your game has "learned" the jump attack! You have the knockdown working: The hero falls to the ground and seems to black out momentarily when he takes a vicious hit in the air.

Next up, another special attack for the protaganist: **Run Attack**!

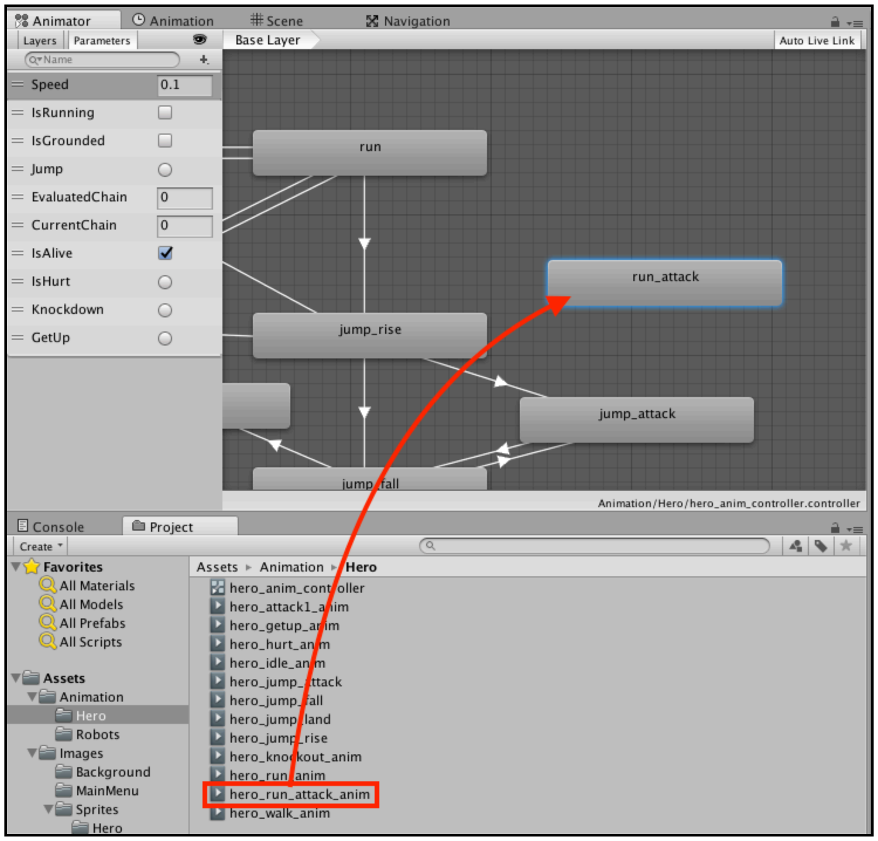
Run...attack!

One special attack is hardly enough to keep a player's interest, so you'll introduce a run attack. While running, the hero will lunge, kick, and knock down the enemy — channeling his inner kickboxer.

Import **Hero Run Attack.unitypackage** from the **Unity Packages** folder.



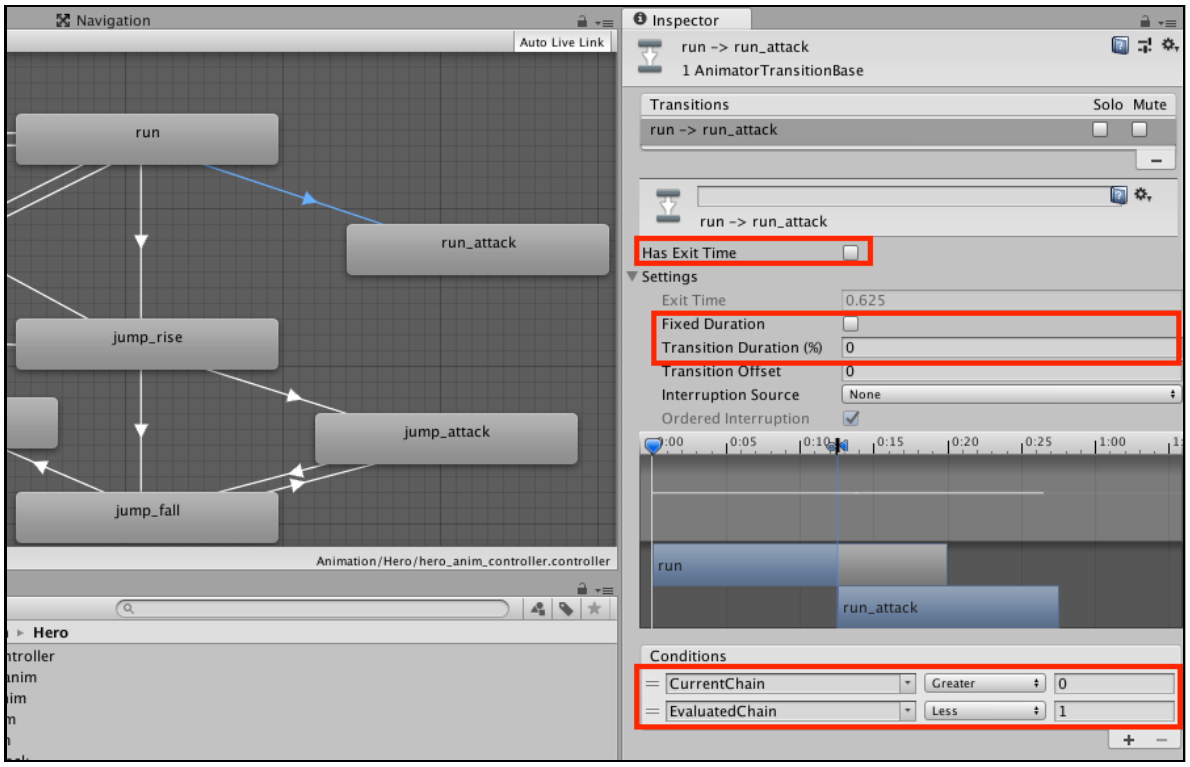
Add an animation state to the hero's animator: double-click **hero_anim_controller** to open the **Animator window**. Drag **hero_run_attack_anim** into the grid layout. Rename this new state to **run_attack**.



Add a transition from **run** to **run_attack**. Uncheck **Has Exit Time** and **Fixed Duration**. Set **Transition Duration** to `0`.

Add two conditions: **CurrentChain** being **Greater** than `0` and **EvaluatedChain Less** than `1`.

Your new run_attack state will occur when the player presses attack while the hero is running.



run -> run_attack
1 AnimatorTransitionBase

Transitions	Solo	Mute
run -> run_attack	☐	☐

run -> run_attack

Has Exit Time ☐

Settings

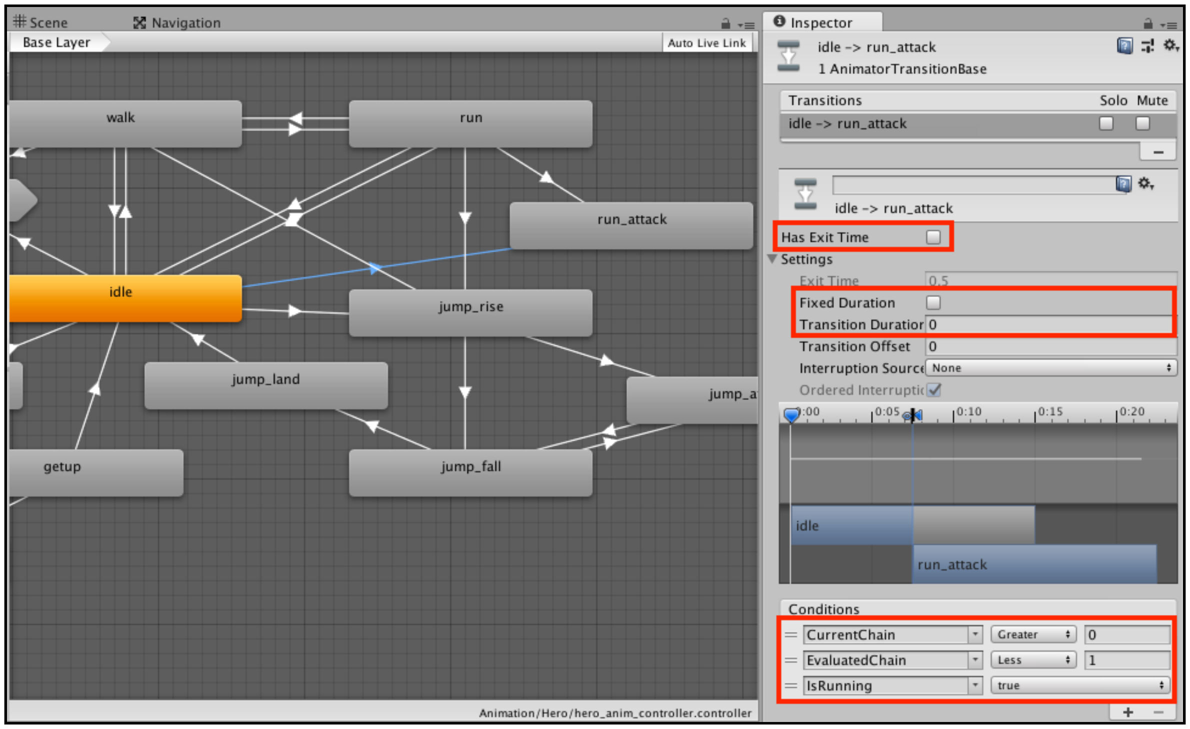
Exit Time	0.625
Fixed Duration	☐
Transition Duration (%)	0
Transition Offset	0
Interruption Source	None
Ordered Interruption	✓

Conditions

CurrentChain	Greater	0
EvaluatedChain	Less	1

Add a transition from **idle** to **run_attack**. Set parameters the same as the prior transition, but with the addition of a third condition of **IsRunning** set to **true**.

This transition will play when the player immediately presses attack when the run animation hasn't started.



idle -> run_attack
1 AnimatorTransitionBase

Transitions	Solo	Mute
idle -> run_attack	☐	☐

idle -> run_attack

Has Exit Time ☐

Settings

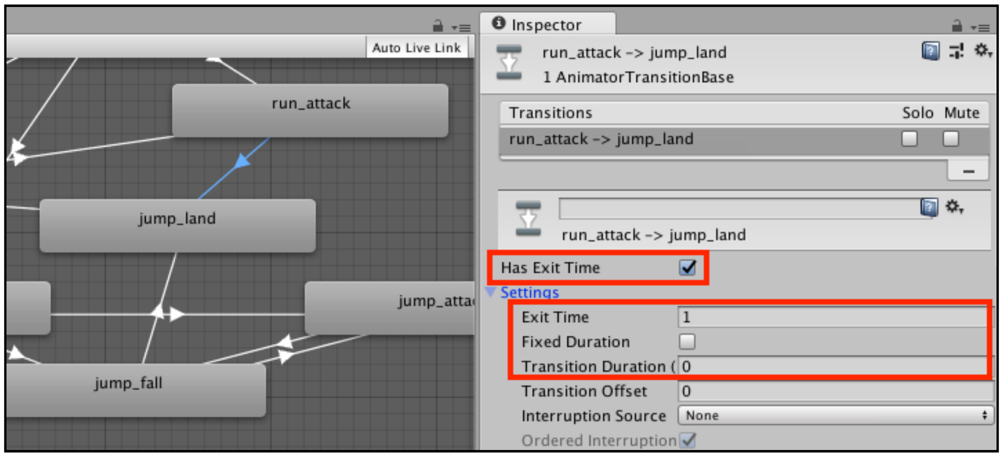
Exit Time	0.5
Fixed Duration	☐
Transition Duratior	0
Transition Offset	0
Interruption Sourc	None
Ordered Interrupti	✓

Conditions

CurrentChain	Greater	0
EvaluatedChain	Less	1
IsRunning	true	

Lastly, add a transition from **run_attack** to **jump_land**. Keep **Has Exit Time** checked and set **Exit Time** to 1. Uncheck **Fixed Duration** and set **Transition Duration** to `0`.

When the run_attack finishes the jump_land state will play.



run_attack -> jump_land
1 AnimatorTransitionBase

Transitions	Solo	Mute
run_attack -> jump_land	☐	☐

run_attack -> jump_land

Has Exit Time ✓
Settings
Exit Time 1
Fixed Duration ☐
Transition Duration (0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Save the scene and the project. Time to add the logic to implement the animations!

Open **Hero.cs** and add the following variables:

```
public AttackData runAttack;
public float runAttackForce = 1.8f;
```

`runAttack` stores the attack values for the hero's run attack. `runAttackForce` dictates how far the hero lunges forward when performing a run attack. You've made it a public variable so you can easily adjust it in the Inspector.

In the `Update` method, append the following to `isAttackingAnim`:

```
|| baseAnim.GetCurrentAnimatorStateInfo(0).IsName("run_attack")
```

Now the new run_attack state is considered an attack state.

In the `Attack` method, replace the `else` condition with:

```
else {
  if (isRunning) {
    //1
    body.AddForce((Vector3.up + (frontVector * 5)) * runAttackForce,
ForceMode.Impulse);
    //2
    currentAttackChain = 1;
    evaluatedAttackChain = 0;
    baseAnim.SetInteger("CurrentChain", currentAttackChain);
    baseAnim.SetInteger("EvaluatedChain", evaluatedAttackChain);
  } else {
    //3
    currentAttackChain = 1;
```

```
      evaluatedAttackChain = 0;
      baseAnim.SetInteger ("EvaluatedChain", evaluatedAttackChain);
      baseAnim.SetInteger ("CurrentChain", currentAttackChain);
    }
  }
```

Here you make it so that when the hero is running, he performs a run attack. Otherwise, he performs a regular punch.

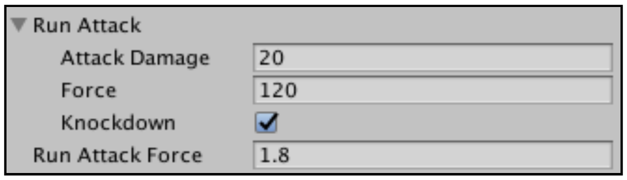
1. Lunges the hero forward thanks to a bit of upward and forward force, multiplied by `runAttackForce`, on his rigidbody. The second parameter, `ForceMode.Impulse`, applies the force instantly and compensates for the Rigidbody's mass.
2. Triggers the attack, which is like a regular punch, by assigning `CurrentChain` to `1` and `EvaluatedChain` to `0`.
3. Triggers the regular punch if `isRunning` is false.

Find the `HitActor` method, and add this `else if` condition before the last closing bracket:

```
  else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("run_attack")) {
    AnalyzeSpecialAttack (runAttack, actor, hitPoint, hitVector);
  }
```

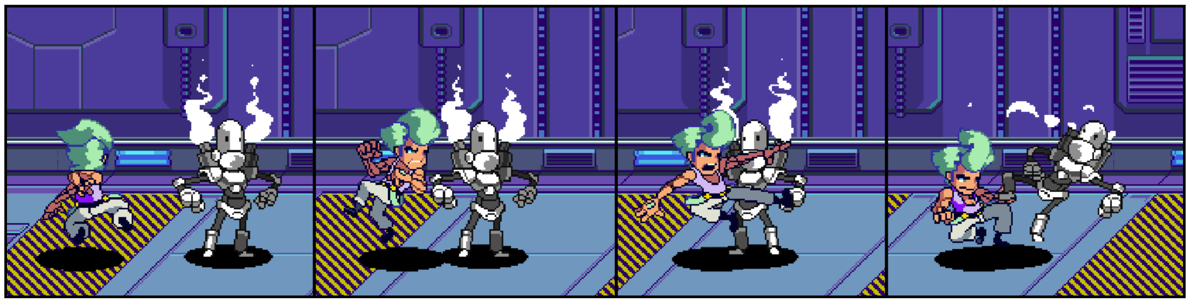
Here you process the `run_attack` and associated damage.

Save the script and return to Unity. Select **MyHero** and set the values of the **Run Attack** variable: **Attack Damage** to `10`, **Force** to `120`, **Knockdown** to **true** and **Run Attack Force** to `1.8` in the Hero component.



Run Attack	
Attack Damage	20
Force	120
Knockdown	✓
Run Attack Force	1.8

Run the game. While running, press attack to perform a spinning kick — let the bodies hit the floor!



Awesome! You've taught the hero two special attacks and given the player some variety to work with. Don't stop now!

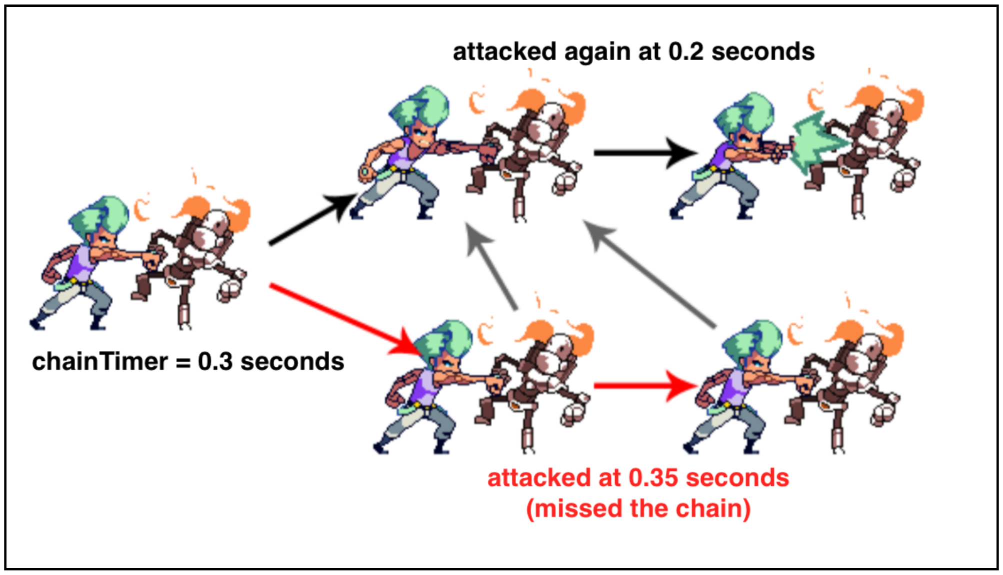
> **Note**: Actually, if you do need to step away to important stuff like stretch, answer some pesky email or get your blood pumping by doing 100 jumping jacks, you're in a perfect spot to do so.

1-2-3 combo

Up until now, you've been jabbing away at robots to defeat them. It takes like 10 punches to take down a droid. My thumb hurts just thinking about it!

To add some variety and strength to the hero's attacks, you're going to teach him to do a timed three-punch combo — a chain of punches!

The logic will work like this:



If the first attack connects with an enemy, there will be a 0.3-second time window in which to chain the attack. If the player triggers the second attack before 0.3 seconds elapses, then the hero will perform the next attack in the chain. If the second attack occurs after 0.3 seconds, however, then the hero will merely do the first attack in the chain again.

Similar logic applies to the third attack: if the player triggers it within the 0.3-second limit, the hero will execute the final, more powerful punch.

Open **Hero.cs** and add the following variables:

```
//1
public AttackData normalAttack2;
public AttackData normalAttack3;

//2
float chainComboTimer;
public float chainComboLimit = 0.3f;
const int maxCombo = 3;
```

1. The first two variables store the AttackData for the other two attacks, with `normalAttack2` being the AttackData for the second punch and `normalAttack3` being used for the third, more powerful punch.
2. These variables store values for the combo-chaining logic. `chainComboLimit` stores the timeframe where the player can trigger the next combo attack, `maxCombo` specifies the maximum number of possible attacks, and `chainComboTimer` stores the actual amount of time remaining in the combo chain.

In the `Update` method, replace `isAttackingAnim` with:

```
isAttackingAnim =
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1") ||
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack2") ||
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack3") ||
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("run_attack") ||
  baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_attack");
```

The `isAttackingAnim` flag now contains all attack states that the Hero can do, including "attack2" and "attack3".

Still in `Update`, after the block of code wrapped in `if (!isAttackingAnim)`, add the following:

```
//1
if (chainComboTimer > 0) {
  chainComboTimer -= Time.deltaTime;
  //2
  if (chainComboTimer < 0) {
    chainComboTimer = 0;
    currentAttackChain = 0;
    evaluatedAttackChain = 0;
    baseAnim.SetInteger ("CurrentChain", currentAttackChain);
    baseAnim.SetInteger ("EvaluatedChain", evaluatedAttackChain);
  }
}
```

The game will run this before the `Attack()` method in the `Update` loop.

1. Checks if the hero is performing a chainAttack by checking if `chainComboTimer` is greater than `0`. If true, it reduces chainComboTimer by deltaTime.
2. Checks if the `chainComboTimer` expired, then resets the `chainComboTimer`, as well as the variables `currentAttackChain` and `evaluatedAttackChain`, then it resets the animator.

Next, wrap all the lines of code inside the `Attack` method with this `if`:

```
if (currentAttackChain <= maxCombo) {
<Code goes here>
}
```

Here you curb the hero’s ability to make combo attacks. He can’t exceed `maxCombo`, which currently limits him to a three-punch combo.

Still in `Attack`, find these lines inside the `else` condition of the `if(isRunning)` block:

```
currentAttackChain = 1;
evaluatedAttackChain = 0;
```

Replace with:

```
if (currentAttackChain == 0 || chainComboTimer == 0 ) {
  currentAttackChain = 1;
  evaluatedAttackChain = 0;
}
```

Here you reset values for `currentAttackChain` and `evaluatedAttackChain` after the `chainComboTimer` resets.

Still in Hero.cs, add this method:

```
private void AnalyzeNormalAttack(AttackData attackData, int attackChain,
Actor actor, Vector3 hitPoint, Vector3 hitVector) {
  actor.EvaluateAttackData(attackData, hitVector, hitPoint);
  currentAttackChain = attackChain;
  chainComboTimer = chainComboLimit;
}
```

The main difference between the above method and `AnalyzeSpecialAttack` is that you handle chaining by setting `currentAttackChain` to `attackChain`. You then call `EvaluateAttackData` on the victim — I mean actor — that received the attack, and finally, you reset `chainComboTimer`.

Add this line to the end of the `AnalyzeSpecialAttack` method:

```
chainComboTimer = chainComboLimit;
```

You're now updating the chain timer when there is a special attack.

In the `HitActor` method, replace this:

```
if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1")){
  base.HitActor(actor, hitPoint, hitVector);
}
```

With this beauty:

```
if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1")) {
  AnalyzeNormalAttack (normalAttack, 2, actor, hitPoint, hitVector);
} else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack2")) {
  AnalyzeNormalAttack (normalAttack2, 3, actor, hitPoint, hitVector);
} else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack3")) {
  AnalyzeNormalAttack (normalAttack3, 1, actor, hitPoint, hitVector);
}
```

Which AttackData to use will depend on the state of animation. With this enhancement, `HitActor` now accepts the two (soon to be added) punch combo animations, attack2, and attack3 as potential attacks.

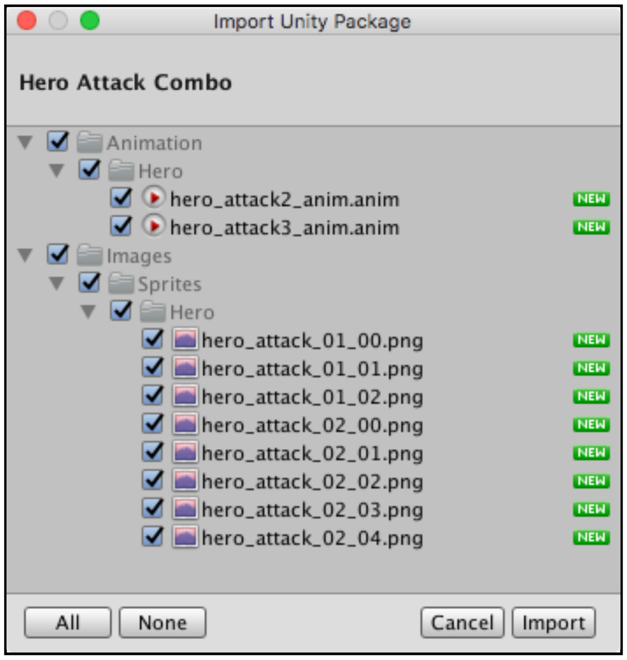
Replace the contents of `DidChain` with:

```
evaluatedAttackChain = chain;
baseAnim.SetInteger("EvaluatedChain", evaluatedAttackChain);
```

Here you assign `evaluatedAttackChain` as `chain` when an attack animation performs.

Save the script and return to Unity to finish off this feature by adding animations for the hero!

Import **Hero Attack Combo.unitypackage** from the "Unity Packages" folder. Find the two remaining clips for the hero combo inside.



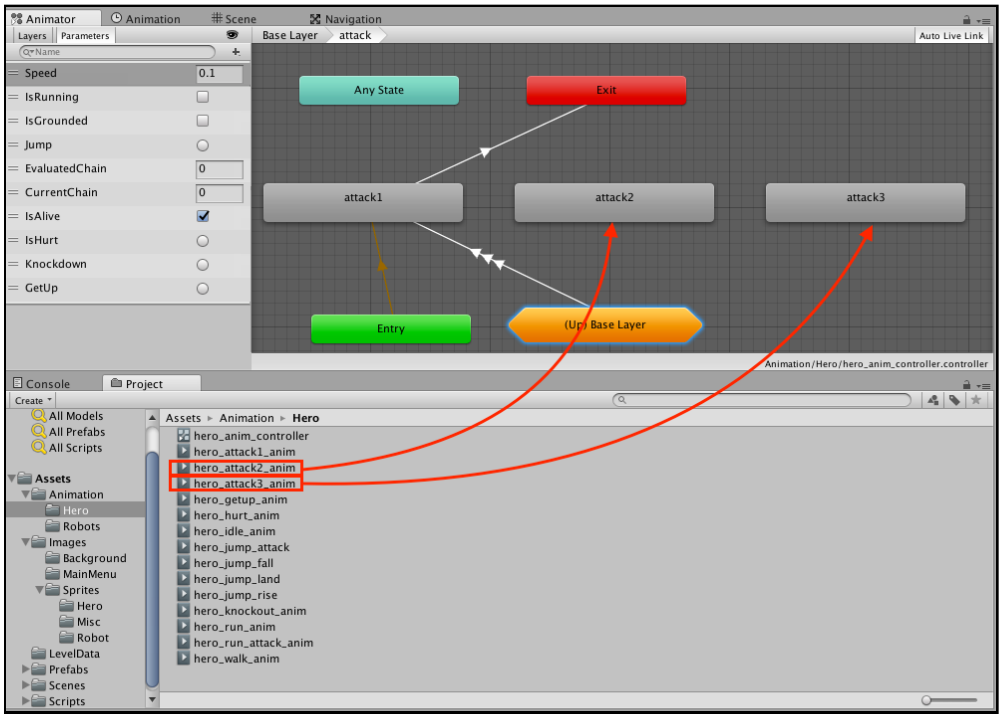
Import Unity Package

Hero Attack Combo

- Animation
  - Hero
    - hero_attack2_anim.anim NEW
    - hero_attack3_anim.anim NEW
- Images
  - Sprites
    - Hero
      - hero_attack_01_00.png NEW
      - hero_attack_01_01.png NEW
      - hero_attack_01_02.png NEW
      - hero_attack_02_00.png NEW
      - hero_attack_02_01.png NEW
      - hero_attack_02_02.png NEW
      - hero_attack_02_03.png NEW
      - hero_attack_02_04.png NEW

Open the Animator by double-clicking **hero_anim_controller** in the **Assets/ Animation/Hero** folder. Double-click the **attack** sub-state machine.

Drag **hero_attack2_anim** clip into the grid layout and name the new state **attack2**. Repeat for **here_attack3_anim** clip, but name it **attack3**.



Speed	0.1
IsRunning	☐
IsGrounded	☐
Jump	○
EvaluatedChain	0
CurrentChain	0
IsAlive	☑
IsHurt	○
Knockdown	○
GetUp	○

Animation/Hero/hero_anim_controller.controller

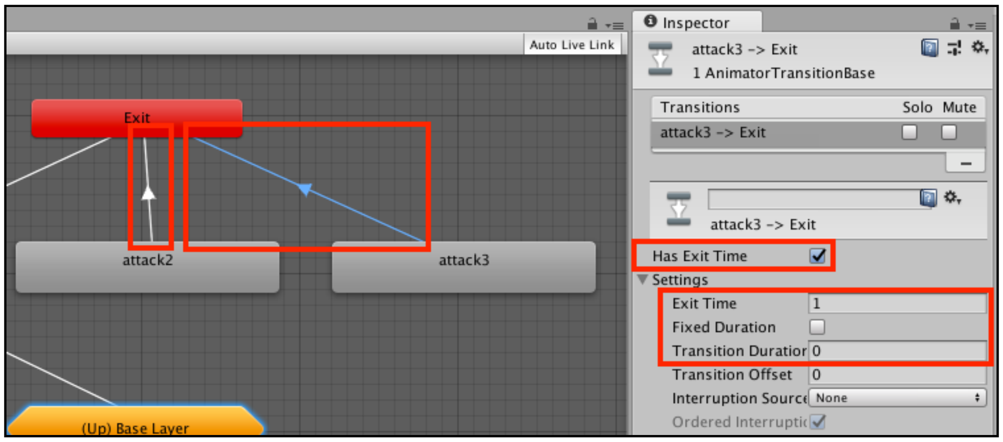
Assets ▸ Animation ▸ Hero

hero_anim_controller
hero_attack1_anim
hero_attack2_anim
hero_attack3_anim
hero_getup_anim
hero_hurt_anim
hero_idle_anim
hero_jump_attack
hero_jump_fall
hero_jump_land
hero_jump_rise
hero_knockout_anim
hero_run_anim
hero_run_attack_anim
hero_walk_anim

Add a transition from **attack2** to **Exit**. Keep **Has Exit Time** checked and set **Exit Time** to `1`. Uncheck **Fixed Duration** and set **Transition duration** to `0`.

Add a transition from **attack3** to the **Exit** state and use the same settings.

These transitions return the state machine to idle when the new attack states finish their animations.

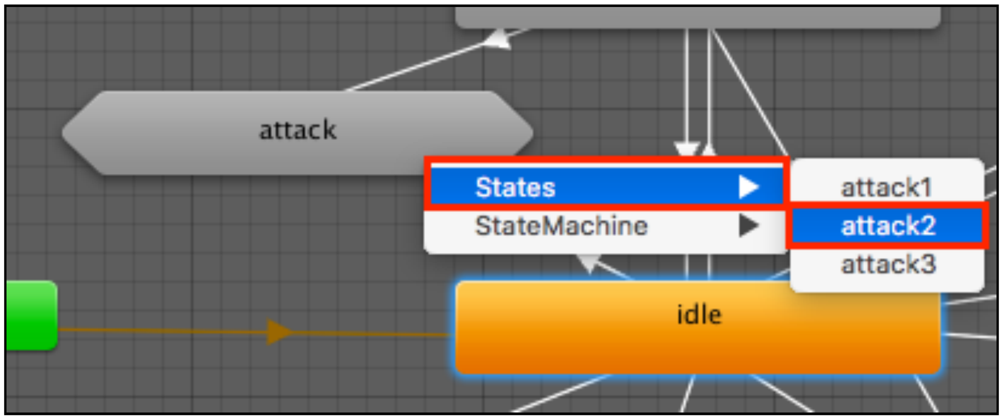


attack3 -> Exit
1 AnimatorTransitionBase

Transitions Solo Mute
attack3 -> Exit

attack3 -> Exit
Has Exit Time ✓
Settings
Exit Time 1
Fixed Duration
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

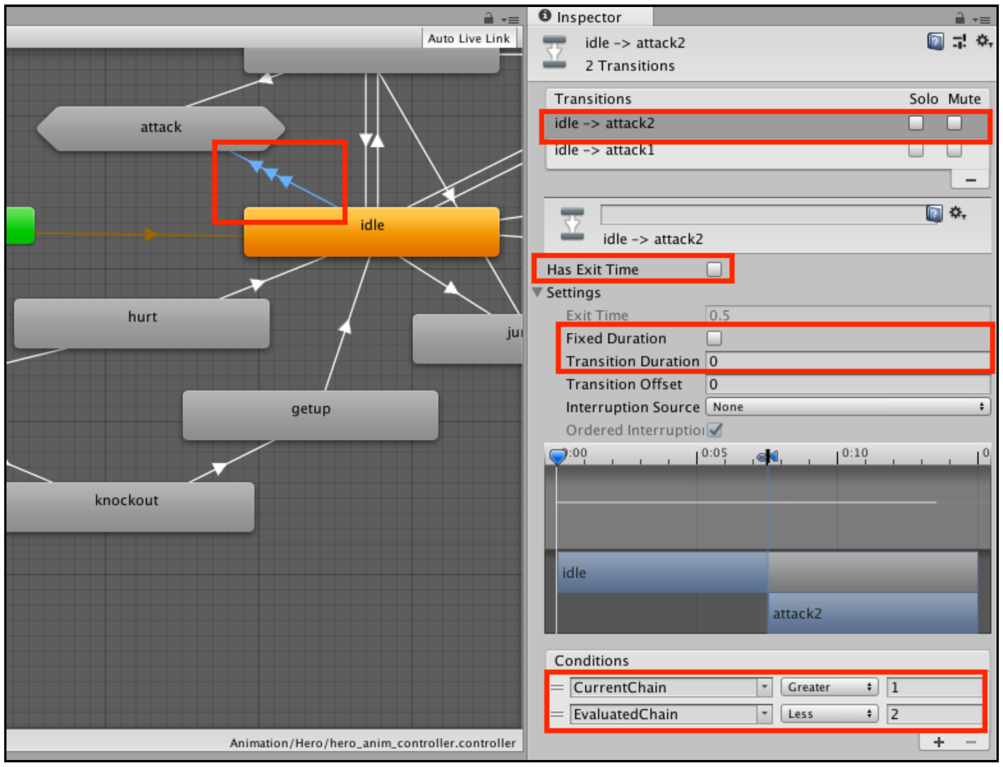
Return to the **Base Layer**, and add a transition from the **idle** to the **attack2** state.



Select the **transition arrow** from **idle** to **attack**. Select the **idle -> attack2** transition, and uncheck **Has Exit Time** and **Fixed Duration**. Set **Transition Duration** to `0`.

Finish by adding these two conditions: **CurrentChain** as **Greater** than `1` and **EvaluatedChain** as **Less** than `2`.

Voila — your transition to the second punch in the combo.



idle -> attack2
2 Transitions

Transitions	Solo	Mute
idle -> attack2	☐	☐
idle -> attack1	☐	☐

idle -> attack2

Has Exit Time ☐

Settings

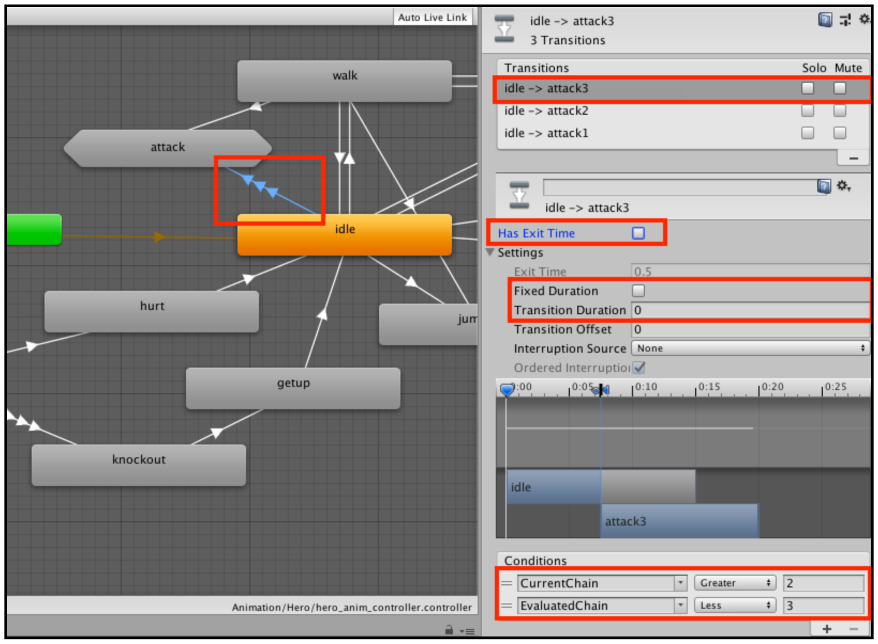
Exit Time	0.5
Fixed Duration	☐
Transition Duration	0
Transition Offset	0
Interruption Source	None
Ordered Interruption	✓

Conditions

CurrentChain	Greater	1
EvaluatedChain	Less	2

Create another transition from **idle** to the **attack** sub-state and select **States / attack3**. Select the transition and pick the **idle -> attack3** transition.

Set the same as you did for the **idle -> attack2** transition, but make **CurrentChain Greater** than 2 and **EvaluatedChain Less** than 3.



idle -> attack3
3 Transitions

Transitions	Solo	Mute
idle -> attack3	☐	☐
idle -> attack2	☐	☐
idle -> attack1	☐	☐

idle -> attack3

Has Exit Time ☐

Settings

Exit Time	0.5
Fixed Duration	☐
Transition Duration	0
Transition Offset	0
Interruption Source	None
Ordered Interruption	✓

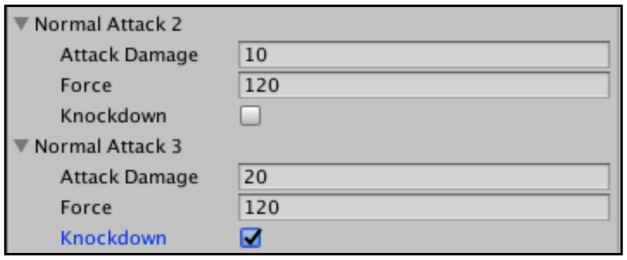
Conditions

CurrentChain	Greater	2
EvaluatedChain	Less	3

Huzzah! Animations wired. Over to the Inspector with you to stitch it all together.

In the Hero component of MyHero, set the **Normal Attack 2** variable with **Attack Damage** set to `10`, **Force** set to `30`, and **Knockdown** set to **false**.

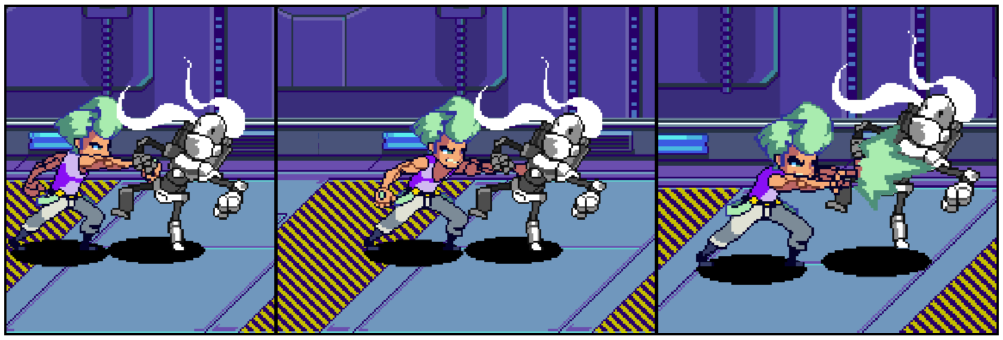
For **Normal Attack 3**, set **Attack Damage** to `20`, **Force** to `120`, and check **Knockdown** to make it true.



▼ Normal Attack 2	
Attack Damage	10
Force	120
Knockdown	☐
▼ Normal Attack 3	
Attack Damage	20
Force	120
Knockdown	✓

Save the scene and click **Play**.

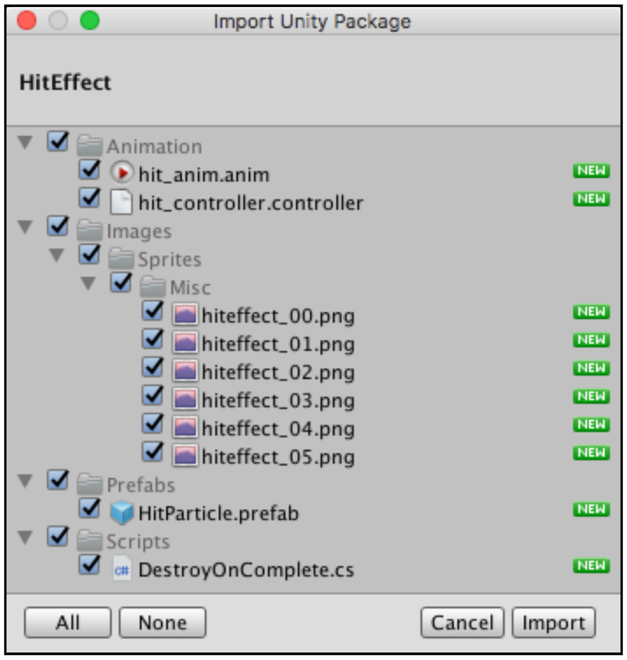
How's that three-punch combo working? Press the attack button repeatedly to trigger it. Each punch is more bruising than the last and the sequence ends with the enemy on the floor.



It's about to get sparky in here

You're almost done adding features in this chapter. When an actor successfully punches another actor, there should be a visual effect — a spark! Your next few steps will result in this classic beat-em-up effect, and you'll be pleased to see I've made it very easy to integrate.

Import **HitEffect.unitypackage** from the **Unity Packages** folder.

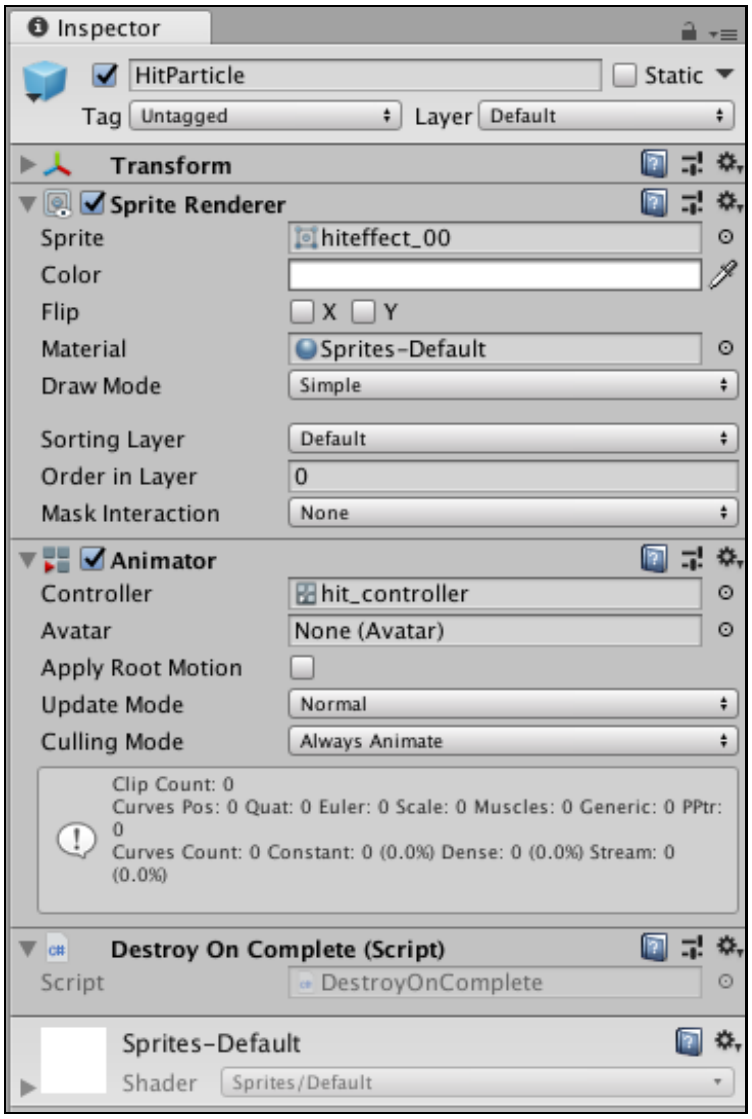

Import Unity Package

HitEffect

- Animation
  - hit_anim.anim NEW
  - hit_controller.controller NEW
- Images
  - Sprites
    - Misc
      - hiteffect_00.png NEW
      - hiteffect_01.png NEW
      - hiteffect_02.png NEW
      - hiteffect_03.png NEW
      - hiteffect_04.png NEW
      - hiteffect_05.png NEW
- Prefabs
  - HitParticle.prefab NEW
- Scripts
  - DestroyOnComplete.cs NEW

Here's what you're about to set up: When one actor hits another, or misses and connects with an object instead, the game will create the hit effect then destroy it when the animation finishes.

Find the prefab named **HitParticle**.



HitParticle ☐ Static
Tag Untagged Layer Default

Transform

Sprite Renderer
Sprite: hiteffect_00
Color:
Flip: ☐ X ☐ Y
Material: Sprites-Default
Draw Mode: Simple
Sorting Layer: Default
Order in Layer: 0
Mask Interaction: None

Animator
Controller: hit_controller
Avatar: None (Avatar)
Apply Root Motion: ☐
Update Mode: Normal
Culling Mode: Always Animate

Clip Count: 0
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 0 Generic: 0 PPtr: 0
Curves Count: 0 Constant: 0 (0.0%) Dense: 0 (0.0%) Stream: 0 (0.0%)

Destroy On Complete (Script)
Script: DestroyOnComplete

Sprites-Default
Shader: Sprites/Default

For convenience, it contains a SpriteRenderer to display the hit effect, an animator to handle the animation, and a `DestroyOnComplete` script:

```
public class DestroyOnComplete : MonoBehaviour {

  public void DidComplete() {
    Destroy(gameObject);
  }
}
```

When something calls `DidComplete()`, the gameObject will destroy itself. The method is called as an Animation Event in the particle's animation clip.

All you need to do is instantiate this at the right time. Open **Actor.cs** and add this variable:

```
public GameObject hitSparkPrefab;
```

Now the `hitSparkPrefab` variable contains the prefab for the hit effect.

Next, add the following method:

```
protected void ShowHitEffects(float value, Vector3 position) {
  GameObject sparkObj = Instantiate(hitSparkPrefab);
  sparkObj.transform.position = position;
}
```

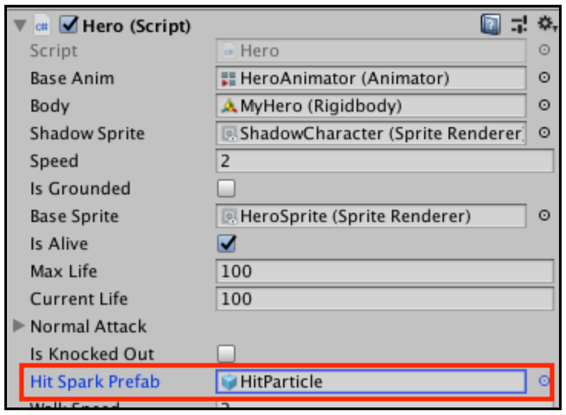
`ShowHitEffects` takes the amount of damage and the position of the collision then creates an instance of the `HitParticle` prefab at the precise position of the hit. You'll make use of the `value` parameter in the next chapter.

Find `EvaluateAttackData` and insert the following at the end of the method:

```
ShowHitEffects(data.attackDamage, hitPoint);
```

Here you call `ShowHitEffects` to create the hit effect.

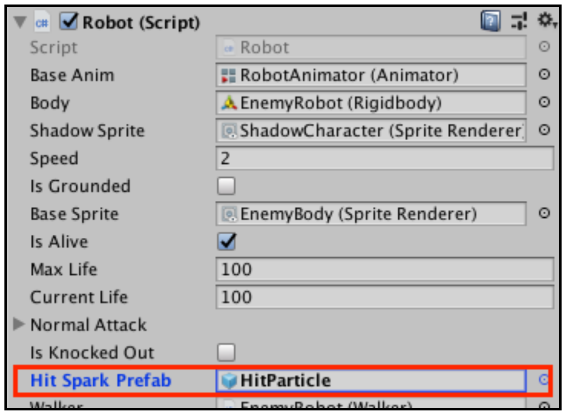
Save the script and return to Unity to set the prefab references for the `HitParticle`. Select **MyHero** in the Hierarchy and assign **HitParticle** as the **Hit Spark Prefab** variable.



Hero (Script)

Script	Hero
Base Anim	HeroAnimator (Animator)
Body	MyHero (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Renderer
Speed	2
Is Grounded	☐
Base Sprite	HeroSprite (Sprite Renderer)
Is Alive	✓
Max Life	100
Current Life	100
Normal Attack	
Is Knocked Out	☐
Hit Spark Prefab	HitParticle

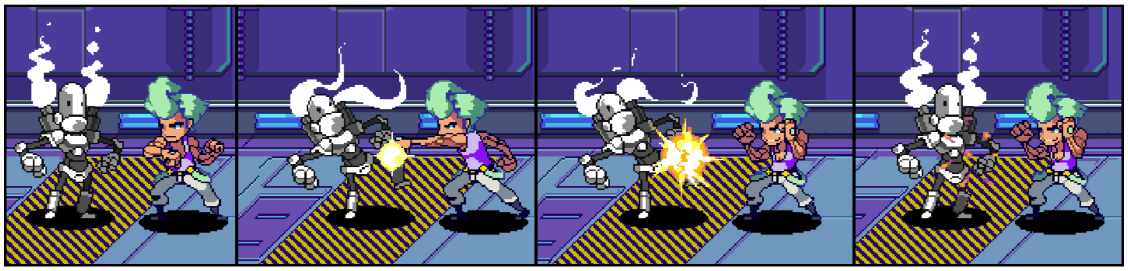
Then, select the **EnemyRobot** prefab in the **Assets** / **Prefab** folder. Also, set **Hit Spark Prefab** as the **HitParticle** prefab.



Robot (Script)

Script	Robot
Base Anim	RobotAnimator (Animator)
Body	EnemyRobot (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Renderer
Speed	2
Is Grounded	☐
Base Sprite	EnemyBody (Sprite Renderer)
Is Alive	✓
Max Life	100
Current Life	100
Normal Attack	
Is Knocked Out	☐
Hit Spark Prefab	HitParticle
Walker	EnemyRobot (Walker)

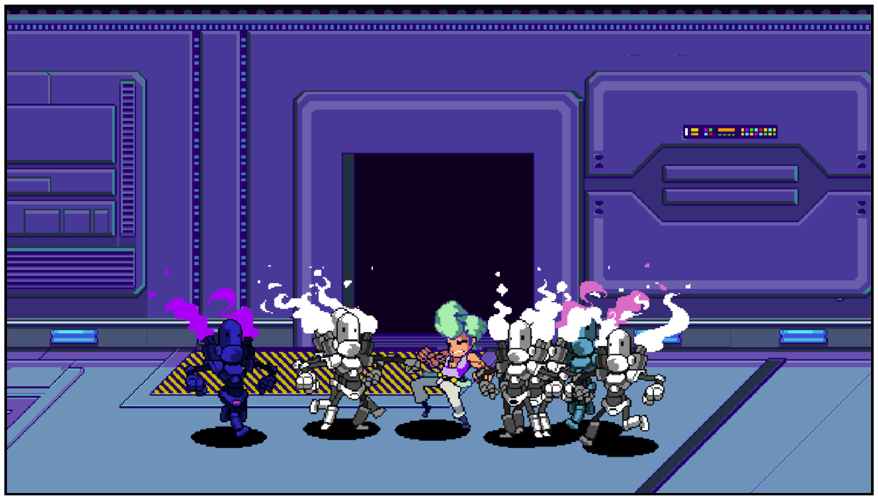
Save the scene and play. Many spark. Such wow! Players will love the instant gratification of a spark when they destroy some poor droid.



I get knocked down!

You've hung in there for a lot of drudge work in this chapter — you had to know it was coming! Worth it though, right? You're probably seeing the light at the end of the tunnel with this game, and you're about to add the last feature for this chapter.

Conditions exist that allow the hero to get mauled to death by a mob of sentient scrap metal. It's an embarrassing situation for him, honestly — the stuff of nightmares!



Unfortunately, if the hero takes continuous hits, he can't escape because the hurt animation requires him to be frozen with attacks disabled. Your game also allows him to take hits during this animation, but there is no reason to change that logic.

To increase the hero's chances of surviving multiple attackers, you will make him a little weaker and a tad faster to hit the ground. Quite the contradiction, huh?

Remember that when he is down, our pompadoured protaganist is immune from damage. If it takes fewer hits to trigger a knockdown, then the hero can escape the mob sooner.

Open **Hero.cs** and add the following variables:

```
public float hurtTolerance;
public float hurtLimit = 20;
public float recoveryRate = 5;
```

These variables store how much of a beating the hero can take before getting knocked down.

- `hurtTolerance` stores the amount of damage the hero can take before collapsing. You'll set it dynamically shortly.
- `hurtLimit` stores the maximum value for `hurtTolerance`.
- `recoveryRate` is the amount by which `hurtTolerance` will be increased per second until it reaches the `hurtLimit` value.

Find `Update` and insert the following at the end of the method:

```
if (hurtTolerance < hurtLimit) {
  hurtTolerance += Time.deltaTime * recoveryRate;
  hurtTolerance = Mathf.Clamp(hurtTolerance, 0, hurtLimit);
}
```

This statement calculates the `hurtTolerance` value per second. If `hurtTolerance` is less than the maximum `hurtLimit`, it increases by the time difference since the last update, expressed as `Time.deltaTime`, multiplied by the `recoveryRate`. This value is then clamped between `0` and the `hurtLimit`.

In `TakeDamage`, replace the following lines:

```
if (!isGrounded) {
  knockdown = true;
}
```

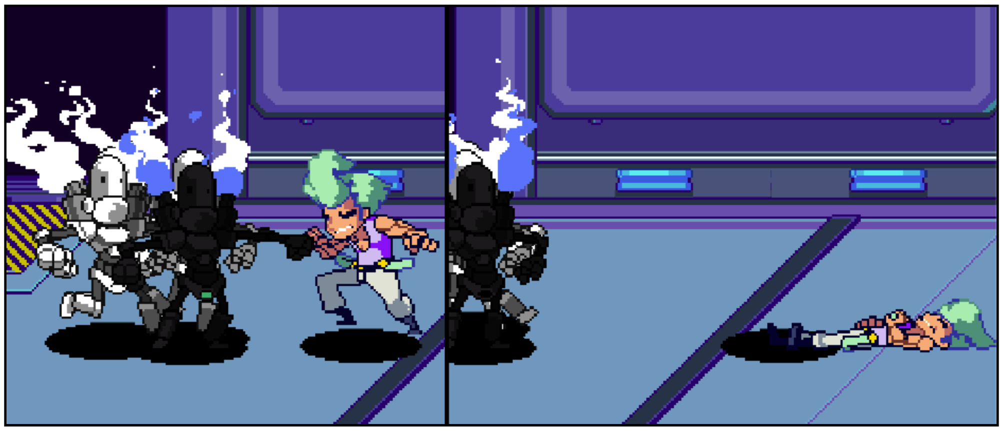
With these lines:

```
hurtTolerance -= value;
if (hurtTolerance <= 0 || !isGrounded) {
  hurtTolerance = hurtLimit;
  knockdown = true;
}
```

Here you subtract from `hurtTolerance` the damage taken by the hero. If `hurtTolerance` is less than `0`, or the hero is airborne, it triggers a knockdown and resets `hurtTolerance`.

Save the script and play the game! Since the default values of the variables are set, you don’t need to modify the Hero component.

Play the game, charge a group of robots and let them take you down. Since you cannot take damage while slithering around, you can take a moment to collect yourself and revise your strategy.



Where to go from here?

You should be feeling spectacular about now. Your hours of toil on PompaDroid are starting to pay off! You started with a rough game and now have something to be proud of. It only gets better from here. And ponder for a moment all that you’ve learned about Unity. At this point, you’ve explored most of the engine and might even find yourself predicting the next steps before you read them.

In this chapter, you’ve:

- Created a brand new AttackData class.
- Enabled knockdowns for all, but also made sure they can get up again, because you’re a generous maker like that.

- Cleaned up your code and fixed major bugs.
- Added three combination attacks to make the hero badass.
- Made sparks fly
- Created a way for the hero to escape an angry mob of metal.

What else does it need? Definitely a soundtrack — you'll get to that later. It also needs a better interface and to be playable on a mobile device. Turn the page to start working on the heads-up display and mobile UI!

Chapter 9: Heads-up Display and Mobile UI

Pompadroid feels like a proper game now. Unless you try to play it on a mobile device. I bet you'd like to see your health and how close those droids are to death.

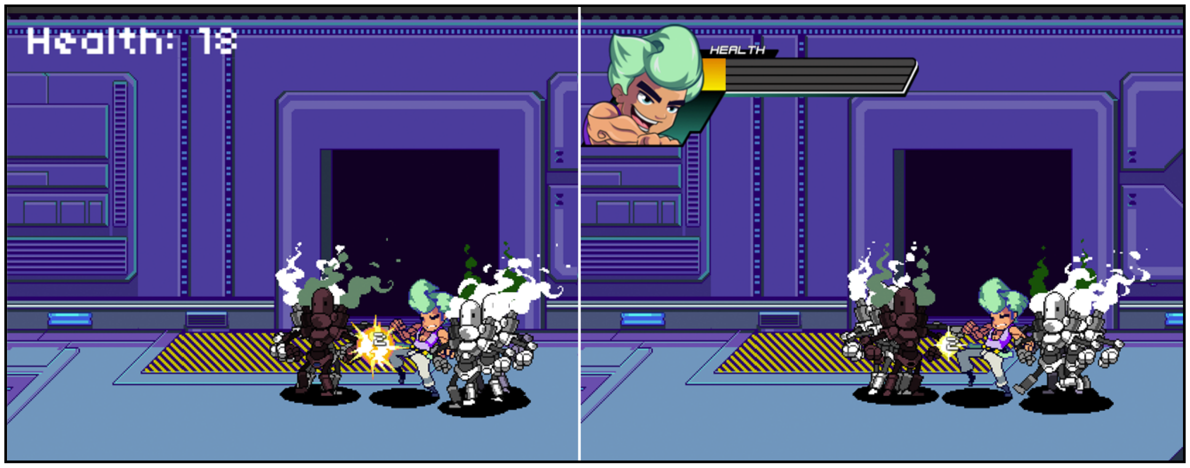
In this chapter, you'll build a heads-up display (HUD) so players know how badly they've hurt an enemy, everybody's health levels, and when the battle is over.

You'll get very familiar with Unity's UI system as you add the following features to the game:

- Health bar for the hero and foes
- GO sign to indicate when the hero should proceed
- Damage display
- On-screen attack, jump and movement buttons for mobile devices

Creating the UI

The UI displays game information such as the hero's health. Two popular designs are simple number values and progress bars.



In the case of PompaDroid, health will show as a simple number on the left and a progress bar on the right. Both ultimately show the same data, but I've found that players tend to prefer a progress bar to an on-screen number. You should know how to implement both.

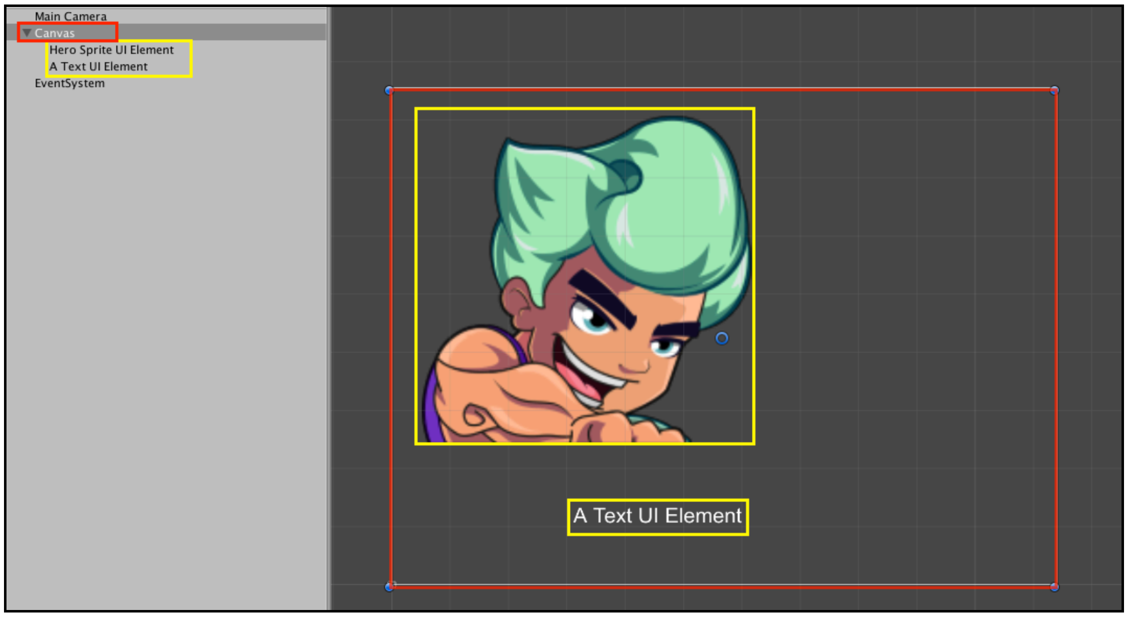
Information is vital in a game as it guides the player's decisions. For example, players tend to be more cautious when health is low.

Unity's user interface 101

Unity's old UI system caused many a developer to have a furrowed brow. You either had to use third-party solutions or resign yourself to hours of coding. When Unity 4.6 came along, developing UI became much easier.

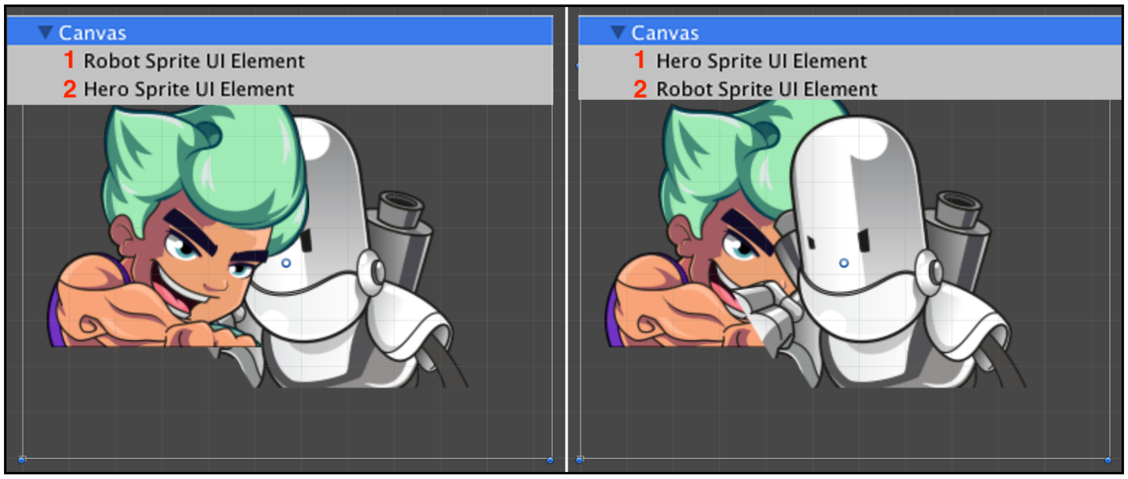
The base of the Unity's UI system is the **Canvas**. A canvas is a GameObject that has a canvas component attached. All Unity UI elements must be attached to a canvas object in order to render.

Unity depicts a canvas as rectangle-shaped space in the scene. In the image below, it's the red rectangle and children UI elements are the yellow rectangles.



Though a canvas is just another component, there are special rules for rendering its content.

First, the order in which it draws elements depends on their order in the Hierarchy. When two sprites overlap on the same canvas, Unity draws the first child of a UI element first.

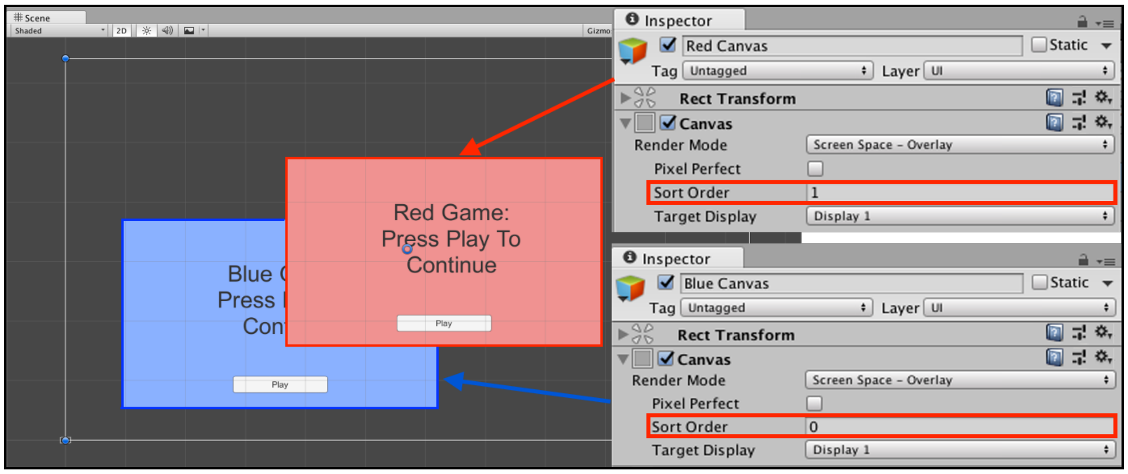


A canvas shows its elements via various **Render Modes**:

- **Screen Space: Overlay** renders the canvas on top of the scene. This mode resizes the canvas when the size of the screen changes.
- **Screen Space: Camera** draws the canvas in front of the camera while taking in its camera settings. It’s useful for interfaces that should appear in front of the camera. This mode resizes the canvas when the camera’s frustum size changes.

- **World Space**: behaves like any other object in a scene, so it can do things like hide behind objects and walls. This is useful when displaying UI elements that are part of the game, such as speech bubbles and damage displays.

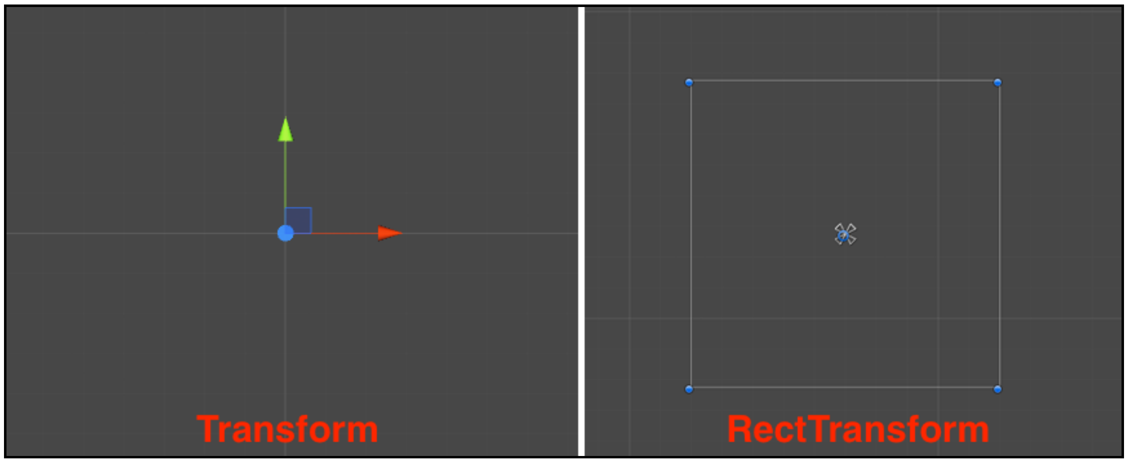
A canvas, and all of its children, are done in a single draw, so a canvas renders atop other canvases based on their sort order or — if applicable — their distances from the camera.



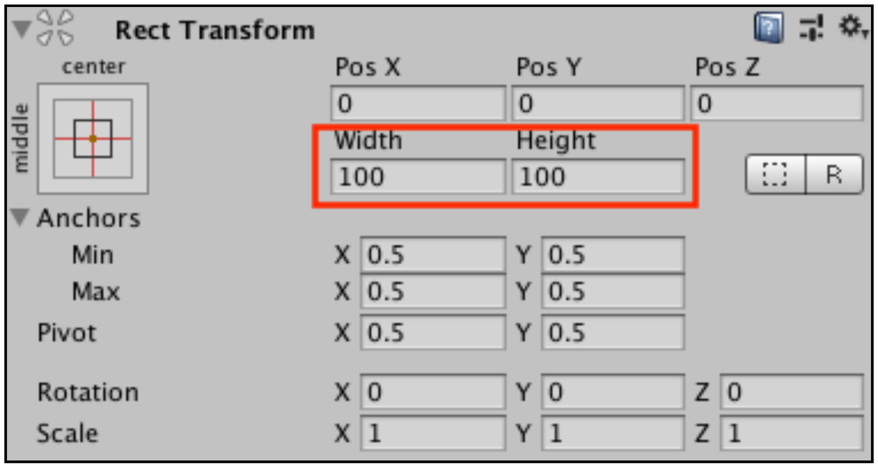
Red Canvas
Tag Untagged Layer UI
Render Mode: Screen Space - Overlay
Pixel Perfect
Sort Order: 1
Target Display: Display 1

Blue Canvas
Tag Untagged Layer UI
Render Mode: Screen Space - Overlay
Pixel Perfect
Sort Order: 0
Target Display: Display 1

Unity UI also has a special Transform called a **RectTransform**. Unlike a regular Transform, which tracks a point in space, a RectTransform tracks a rectangular space in a 3D world.



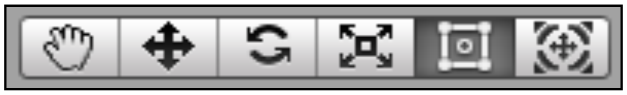
Similar to regular Transforms, they calculate based on position, rotation and scale. They also contain width and height values.


Rect Transform

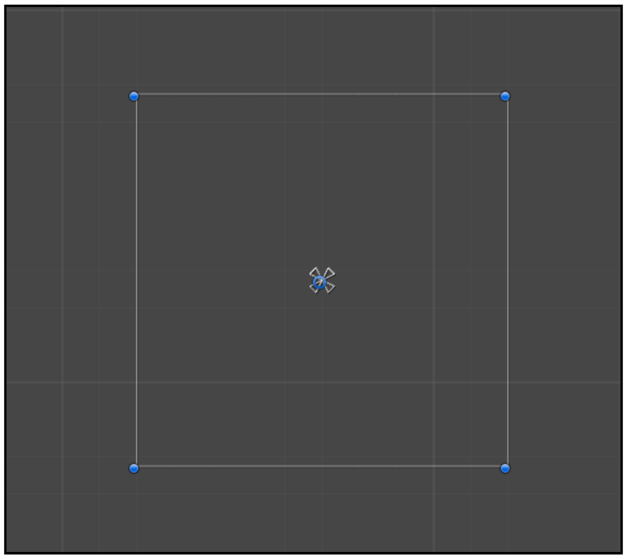
center, middle

	Pos X	Pos Y	Pos Z
	0	0	0
	Width	Height	
	100	100	
Anchors			
Min	X 0.5	Y 0.5	
Max	X 0.5	Y 0.5	
Pivot	X 0.5	Y 0.5	
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

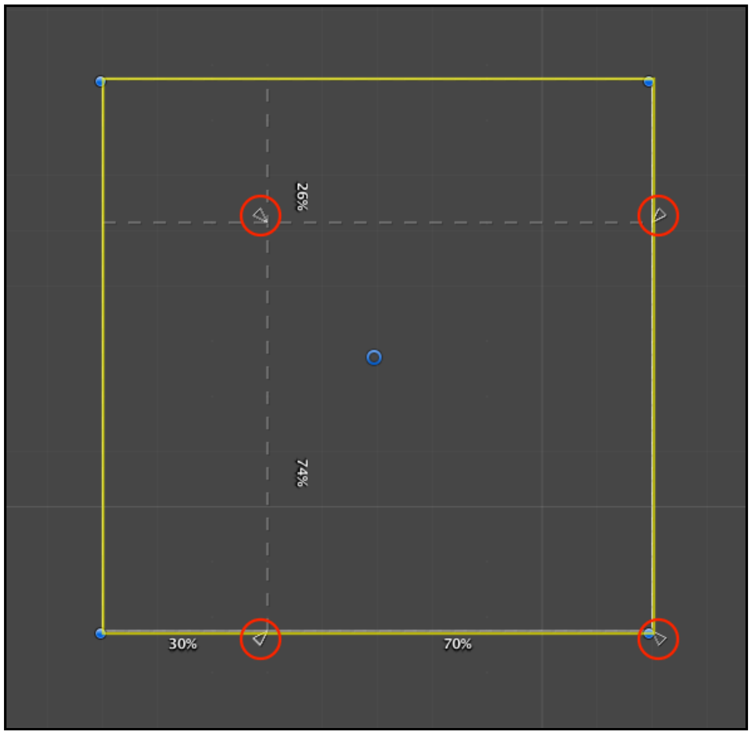
The RectTransform is visible in the scene when you use the Rect tool located in the upper-left part of the toolbar.



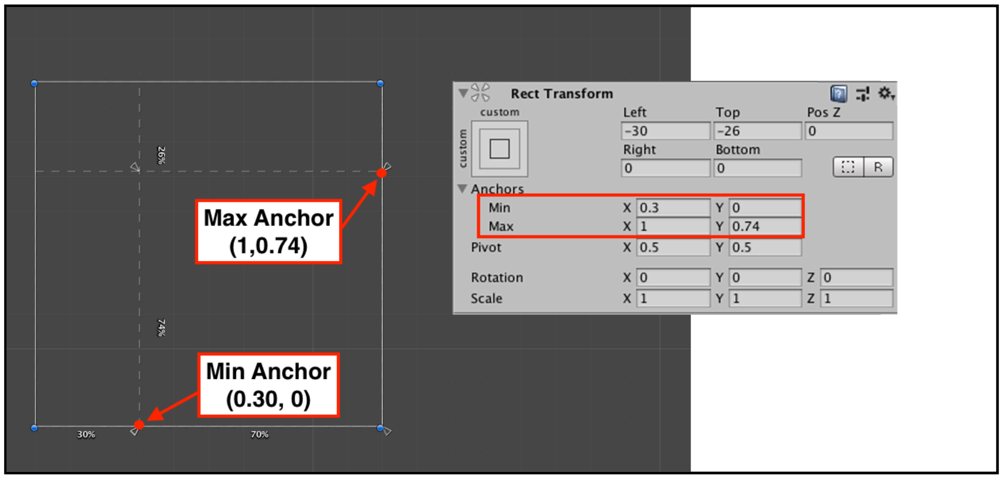
A rectangle dotted by blue circles depicts a RectTransform. You position the rectangle in your game world by employing the concept of **Anchoring**.



Anchors — indicated by small triangles — are positioned by percentages on the parent rectangle. Below, you can see anchors (circled in red) set as percentages of their parent's rectangle (the yellow square).

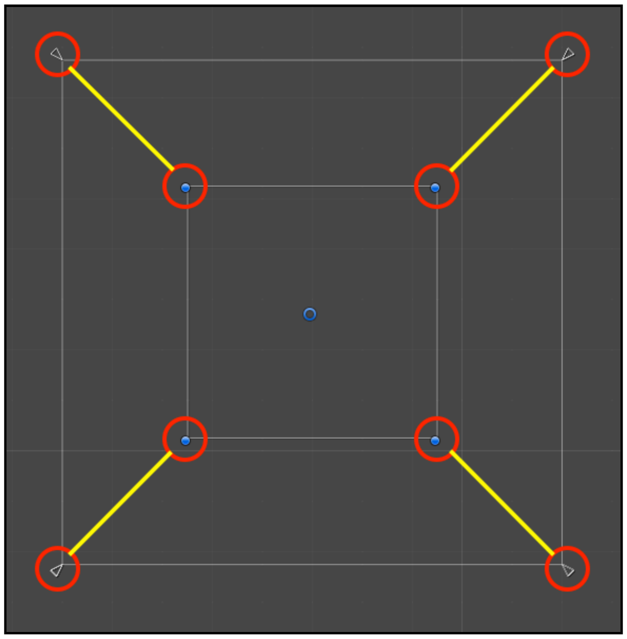


The minimum and maximum anchors are the bottom-left and the top-right percentages, respectively. The example below shows minimum `(0.3, 0)` and maximum `(1, 0.74)`.

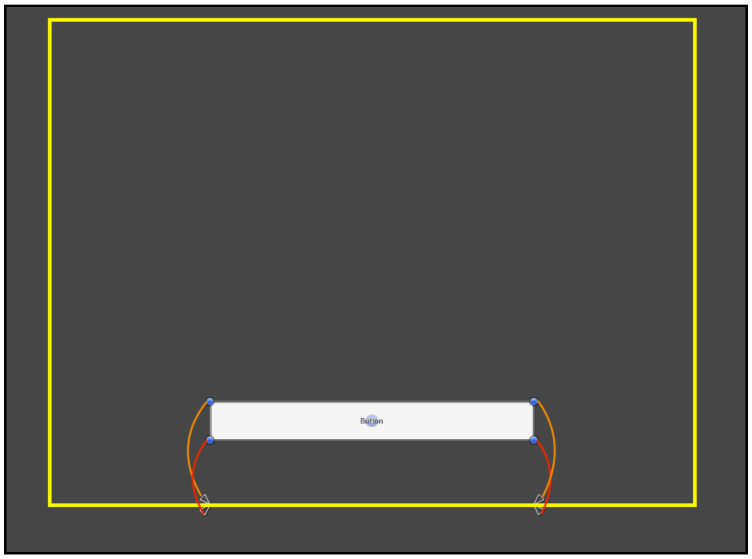


	X	Y	Z
Left / Top / Pos Z	-30	-26	0
Right / Bottom	0	0	
Anchors Min	0.3	0	
Anchors Max	1	0.74	
Pivot	0.5	0.5	
Rotation	0	0	0
Scale	1	1	1

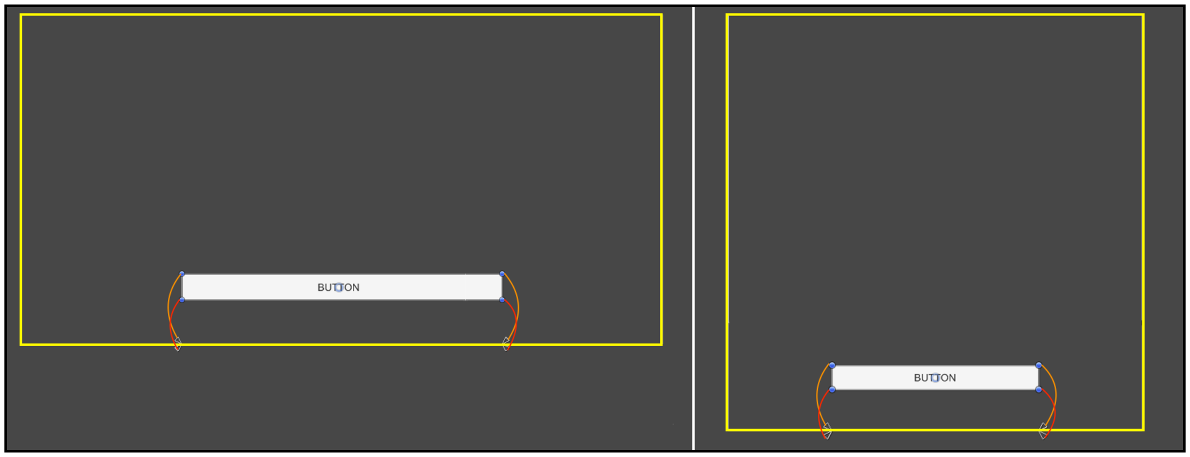
Anchors pair with corresponding corners of the rectangles. The anchor and its corresponding corner enforces a fixed distance between them, depicted by the yellow lines in the following image.



Anchors permit UI to dynamically resize the rectangle based on the size of the parent rectangle.

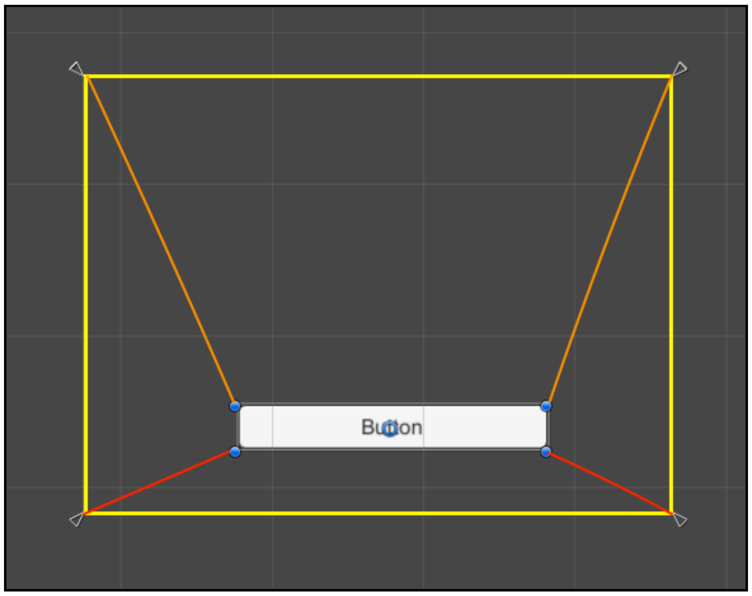


In the image above, the anchors are positioned at the bottom edge of their parent. If you made the parent wider (left) or taller (right) it would look like the following image.

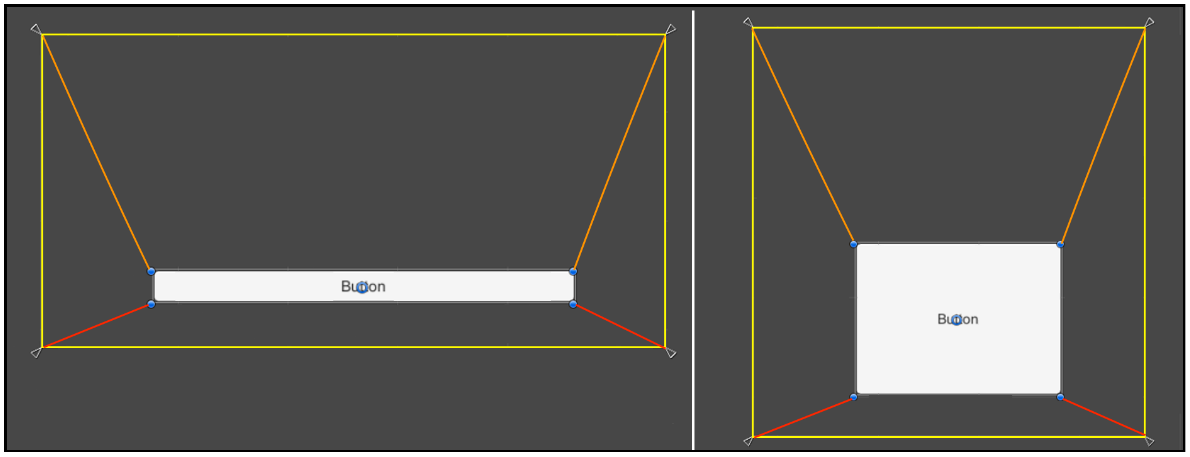


The button is fixed to the bottom of the parent rectangle because the anchors are along the bottom edge of the parent rectangle. Widening the parent rectangle increases the button's width. However, making the rectangle taller did not affect its size.

On the other hand, if you set up the anchors like this:



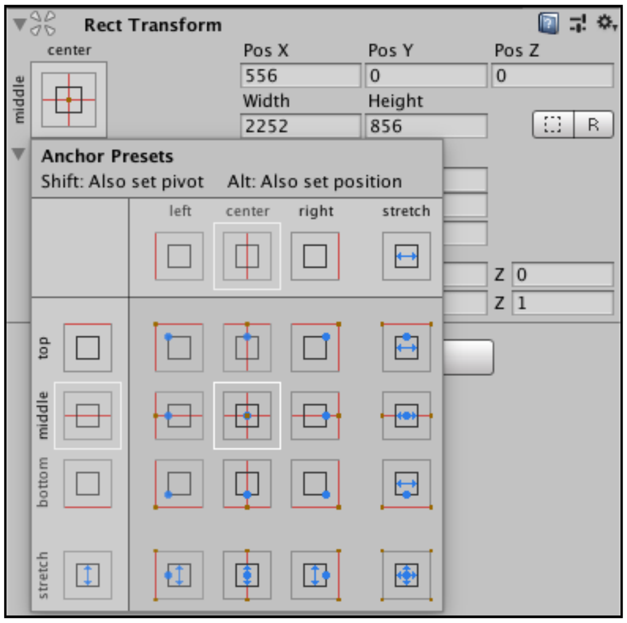
The anchors would be positioned at the corners of their parent rectangle. Resizing the parent would yield the following results:



Widening works the same as before, but this time, when you make the parent taller the button also becomes taller.

To simplify your work, Unity provides Anchor Presets, or common values for RectTransform's anchors.

Click the icon named **Anchor Presets** on the left to access these:



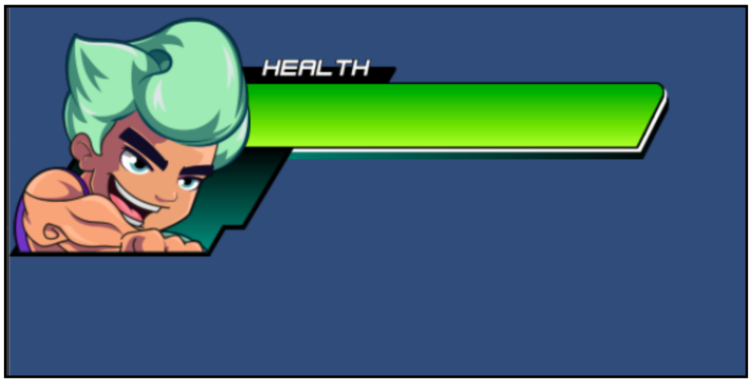
Experiment with these presets until you get a good grasp of how they work and what they do for your game.

Unity's UI uses an **Image** component instead of a **SpriteRenderer** to render sprites. Be careful, it's easy to accidentally add a SpriteRenderer to a canvas and not get the behavior you were expecting!

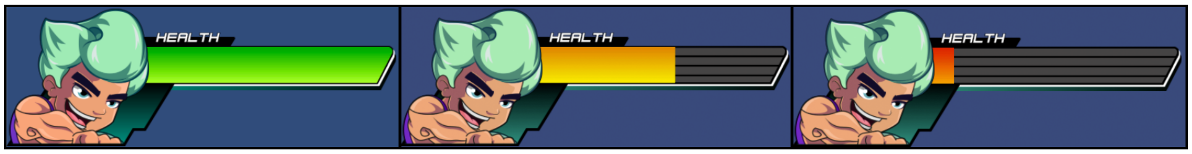
Now that you have a general idea of how the UI works, it's time to create one for Pompadroid!

Implementing health bars

You'll create a health bar with a portrait of the protagonist to show enemies' health.

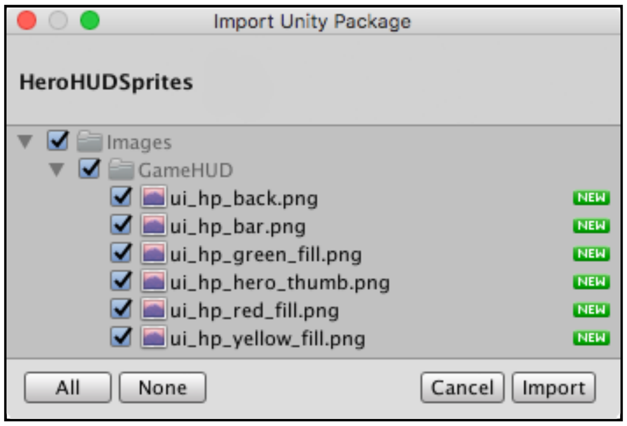


The health bar's behavior is simple. When health falls below a certain threshold, the bar changes color. It'll be green when full, yellow when about half full, and red when the character is a few hits away from certain death.



Health bar for the hero

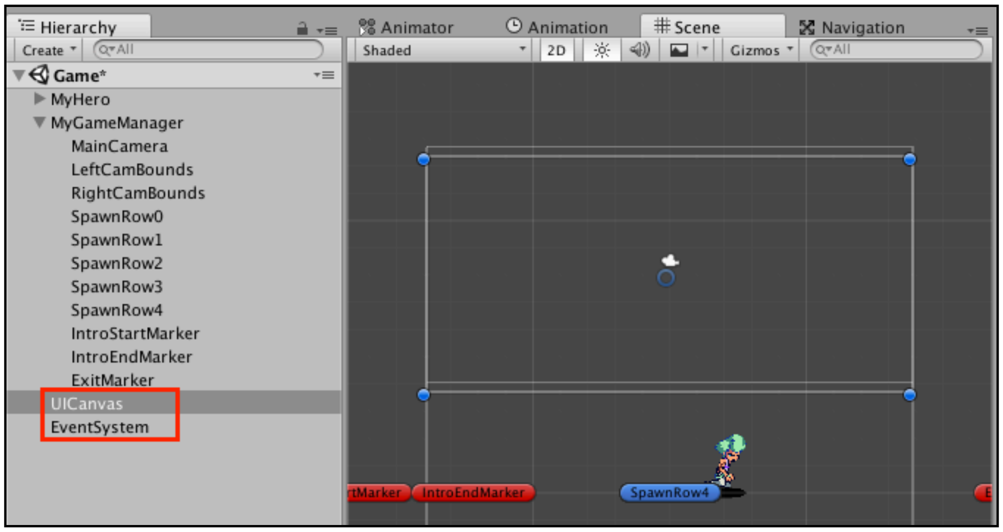
Import the **HeroHUDSprites.unitypackage** located in the **Unity Packages** folder, which contains the sprites for the health bar.


Import Unity Package

HeroHUDSprites

- [✓] Images
  - [✓] GameHUD
    - [✓] ui_hp_back.png NEW
    - [✓] ui_hp_bar.png NEW
    - [✓] ui_hp_green_fill.png NEW
    - [✓] ui_hp_hero_thumb.png NEW
    - [✓] ui_hp_red_fill.png NEW
    - [✓] ui_hp_yellow_fill.png NEW

Open the Game scene and create a **UI \ Canvas** in the Hierarchy. Rename it **UICanvas**. You'll attach all life bar UI components to this canvas root. It will also create an **EventSystem** GameObject to handle all events for the UI, such as button presses.

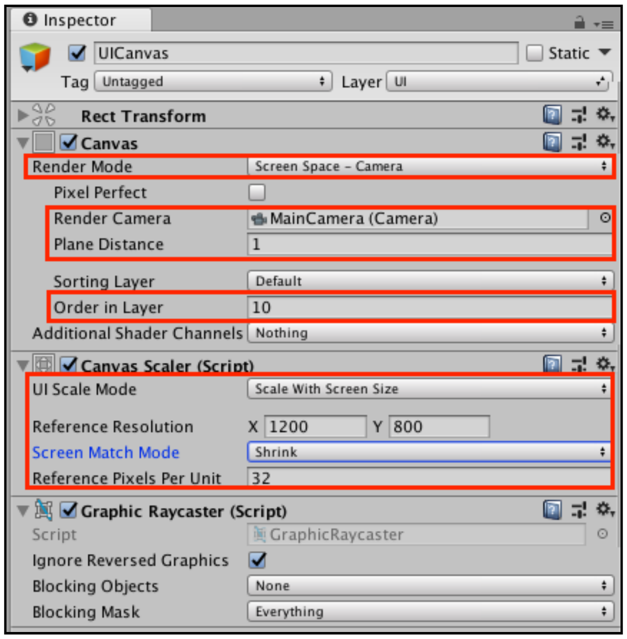


```
Game*
  MyHero
  MyGameManager
    MainCamera
    LeftCamBounds
    RightCamBounds
    SpawnRow0
    SpawnRow1
    SpawnRow2
    SpawnRow3
    SpawnRow4
    IntroStartMarker
    IntroEndMarker
    ExitMarker
  UICanvas
  EventSystem
```

Select **UICanvas**, set **Render Mode** to **Screen Space - Camera** and set **Camera** to **MyGameManager's** child, **MainCamera**.

Set **Plane Distance** to `1` and **Order In Layer** to `10`. Now this canvas will render in front of MainCamera.

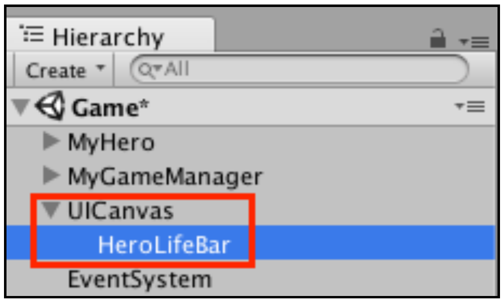
In the Canvas Scaler, set **UI Scale Mode** to **Scale With Screen Size**, **Reference Resolution** to `(X:1200, Y:800)`, **Screen Match Mode** to **Shrink**, and **Reference Pixels Per Unit** to `32`.



```
UICanvas
Tag  Untagged          Layer  UI
Rect Transform
Canvas
  Render Mode                  Screen Space - Camera
  Pixel Perfect                ☐
  Render Camera                MainCamera (Camera)
  Plane Distance               1
  Sorting Layer                Default
  Order in Layer               10
  Additional Shader Channels   Nothing
Canvas Scaler (Script)
  UI Scale Mode                Scale With Screen Size
  Reference Resolution         X 1200   Y 800
  Screen Match Mode            Shrink
  Reference Pixels Per Unit    32
Graphic Raycaster (Script)
  Script                       GraphicRaycaster
  Ignore Reversed Graphics     ✓
  Blocking Objects             None
  Blocking Mask                Everything
```

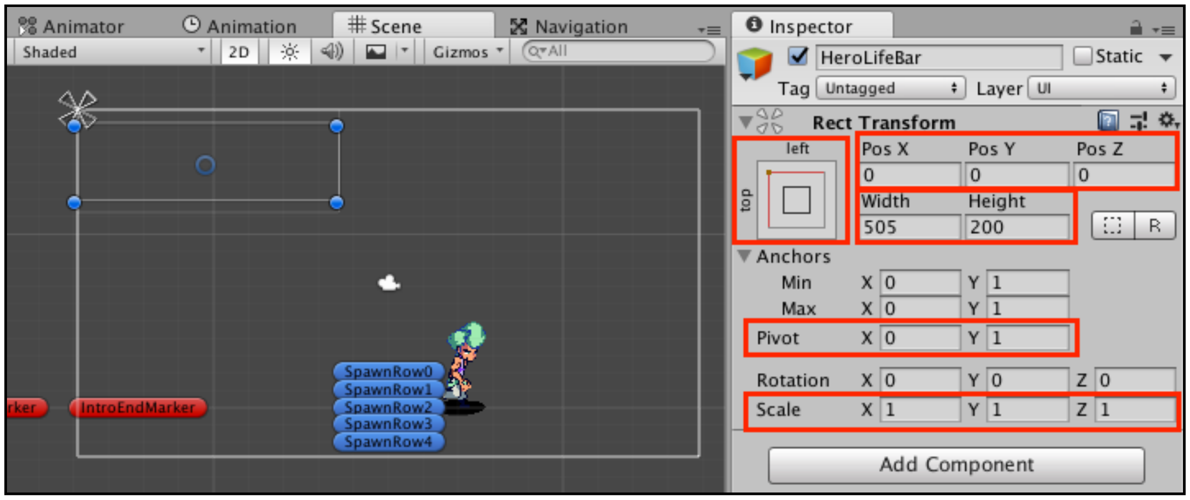
These settings will scale the UI to a size of `1200` units wide by `800` units tall. By default, it'll shrink to fit the device. It'll also use 32 pixels per unit (ppu) — a sprite that renders at 32 ppu means that one pixel equals one unit.

Create a child of **UICanvas** by right-clicking it and selecting **Create Empty**. Rename this child **HeroLifeBar**.



You've just created a GameObject with a RectTransform instead of a plain Transform that will serve as the parent of the hero's life bar.

Set the **Anchor Preset** to **Top Left**, set **Pivot** to `(X:0, Y:1)`, **Pos X**, **Pos Y** and **Pos Z** to `0`, **Width** to `505` and **Height** to `200` on the **RectTransform** on the **HeroLifeBar**.



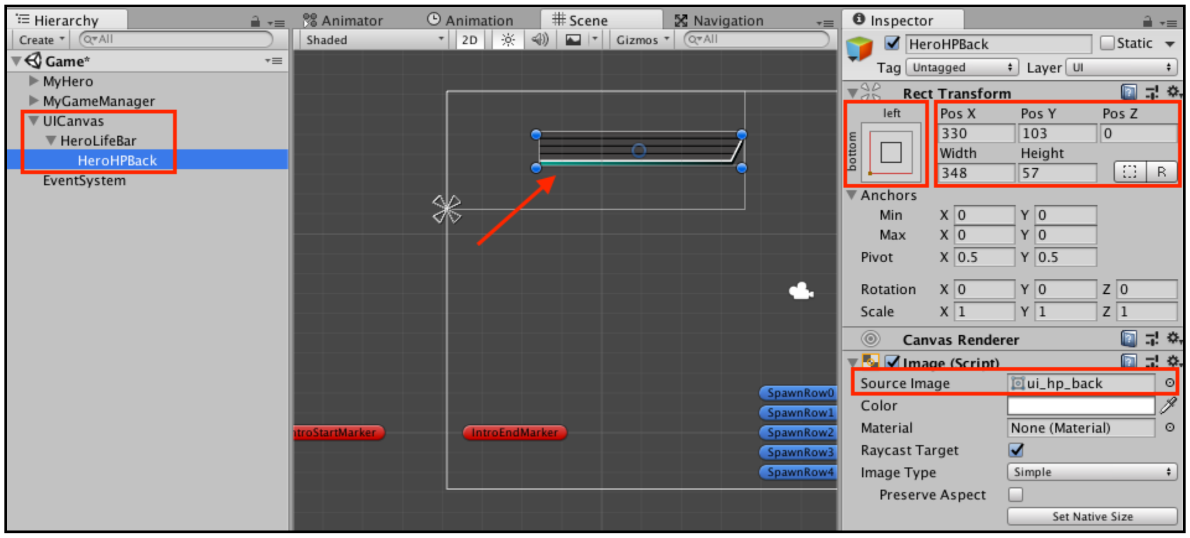
HeroLifeBar
Tag: Untagged
Layer: UI
Pos X: 0, Pos Y: 0, Pos Z: 0
Width: 505, Height: 200
Anchors Min: X 0, Y 1
Anchors Max: X 0, Y 1
Pivot: X 0, Y 1
Rotation: X 0, Y 0, Z 0
Scale: X 1, Y 1, Z 1

You've just pinned the RectTransform to the top-left side of the UICanvas. It should show as a white box with blue corners when you select it with the Rect Tool.

Right-click **HeroLifeBar**, and select **UI\Image** to create an **Image child** of **HeroLifeBar** that will serve as the background of the life bar.

Rename it **HeroHPBack**, and then set the **ui_hp_back** sprite as its **Source Image**.

Position the sprite inside its parent HeroLifeBar's rectangle by setting the **Anchor Preset** to **Bottom Left**, **Pos X**, **Pos Y** and **Pos Z** to `(X:330, Y:103, Z:0)`, the **Width** to `348`, and finally, **Height** to `57`.

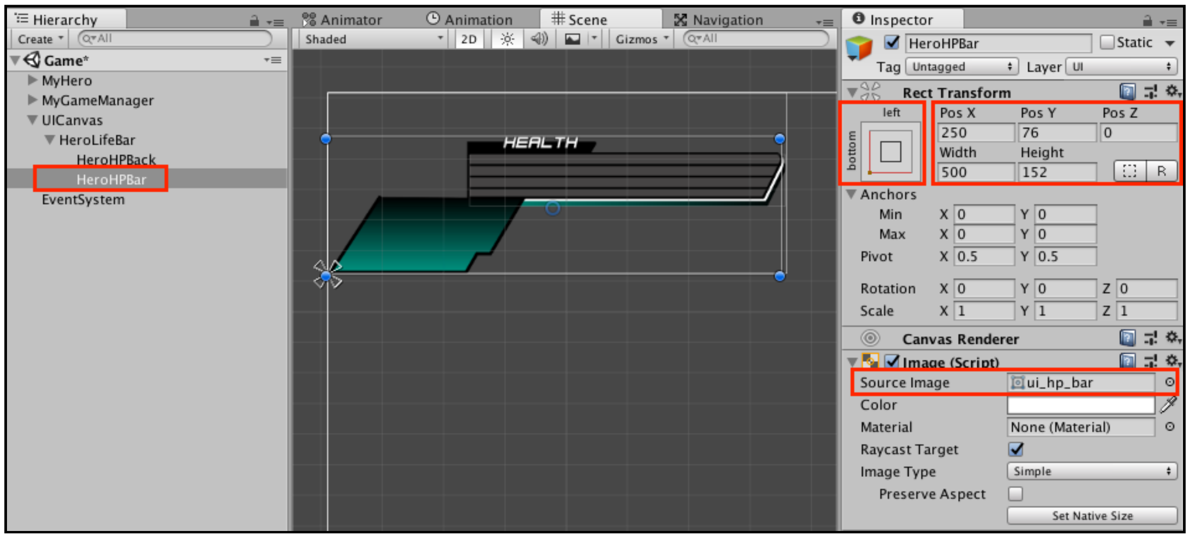


HeroHPBack Static
Tag Untagged Layer UI
Rect Transform
left / bottom
Pos X 330 Pos Y 103 Pos Z 0
Width 348 Height 57
Anchors
Min X 0 Y 0
Max X 0 Y 0
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Canvas Renderer
Image (Script)
Source Image ui_hp_back
Color
Material None (Material)
Raycast Target ✓
Image Type Simple
Preserve Aspect

NOTE: When creating a new **Image**, remember that if you don't assign a sprite to it, you'll see it render as a white image on the screen.

Add another **Image** child to **HeroLifeBar** and rename it **HeroHPBar**. This will serve as the background for the hero's thumbnail. Make sure this is the second child of HeroLifeBar because this must render on top of HeroHPBack.

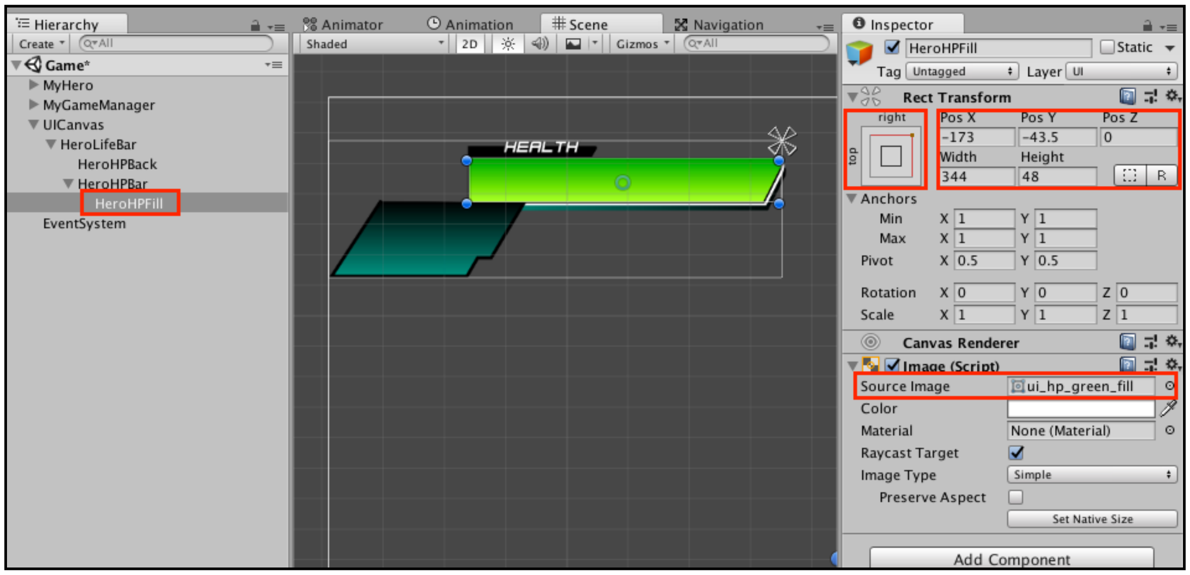
Set its **Source Image** as **ui_hp_bar**, the **Anchor Preset** to **Bottom Left**, and **Pos X**, **Pos Y** and **Pos Z** to `(X:250, Y:76, Z:0)`. Set the **Width** to `500` and **Height** to `152`.



HeroHPBar Static
Tag Untagged Layer UI
Rect Transform
left / bottom
Pos X 250 Pos Y 76 Pos Z 0
Width 500 Height 152
Anchors
Min X 0 Y 0
Max X 0 Y 0
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Canvas Renderer
Image (Script)
Source Image ui_hp_bar
Color
Material None (Material)
Raycast Target ✓
Image Type Simple
Preserve Aspect

Looking good. The UI is coming along nicely!

Add an **Image** child to **HeroHPBar**, which will become the hero's life bar. Rename it **HeroHPFill**. Set its **Source Image** as **ui_hp_green_fill**. In its RectTransform, set **Anchor Preset** To **Top Right**, **Pos X**, **Pos Y** and **Pos Z** to `(X:-173, Y:-43.5, Z:0)`, **Width** to `344` and **Height** to `48`.



Inspector: HeroHPFill, Tag Untagged, Layer UI

Rect Transform: right / top; Pos X -173, Pos Y -43.5, Pos Z 0; Width 344, Height 48

Anchors: Min X 1, Y 1; Max X 1, Y 1; Pivot X 0.5, Y 0.5

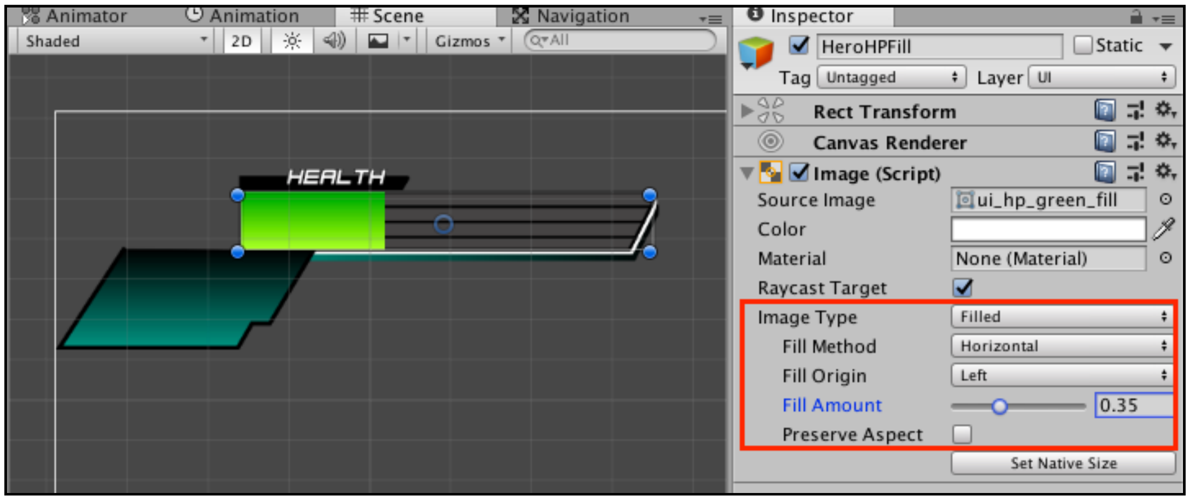
Rotation X 0, Y 0, Z 0; Scale X 1, Y 1, Z 1

Image (Script): Source Image ui_hp_green_fill; Color; Material None (Material); Raycast Target ✓; Image Type Simple; Preserve Aspect

In the **Image** component of HeroHPFill, set the **Image Type** to **Filled** and **Fill Method** to **Horizontal**.

Try changing the **Fill Amount** variable to see the hero's life bar behave like a progress bar — a value of `0.0` appears empty and `1.0` appears full.

NOTE: Remember to revert to its initial value of `1.0`.

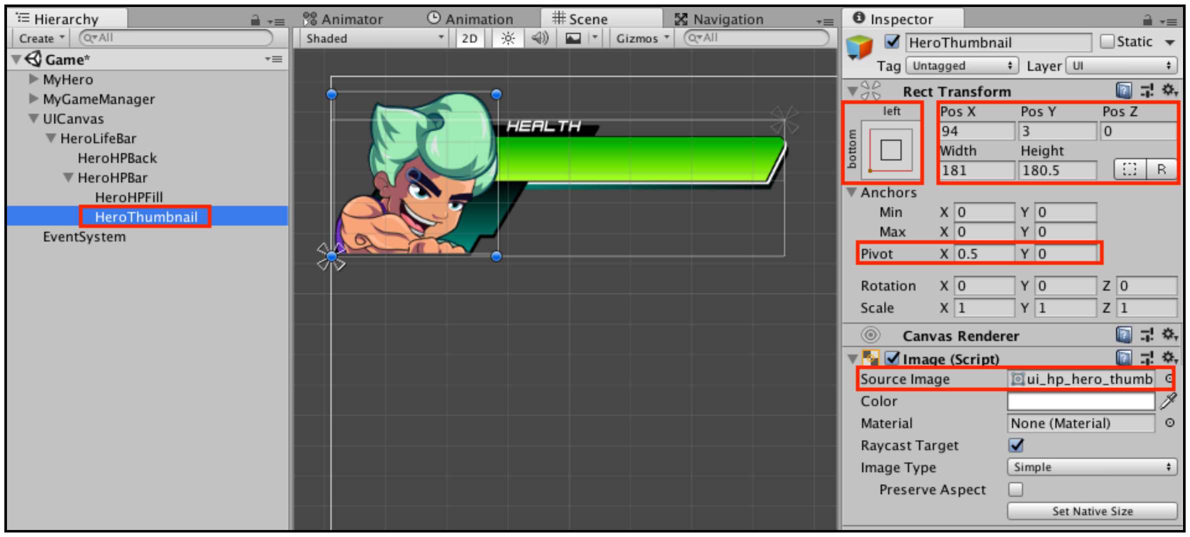


Inspector: HeroHPFill, Tag Untagged, Layer UI

Image (Script): Source Image ui_hp_green_fill; Color; Material None (Material); Raycast Target ✓; Image Type Filled; Fill Method Horizontal; Fill Origin Left; Fill Amount 0.35; Preserve Aspect

Now to make the thumbnail for the Hero. Create another **Image** child of **HeroHPBar**. Make sure it is the **second child**, and rename it **HeroThumbnail**.

Set its **Source Image** to **ui_hp_hero_thumb**, the **Anchor Preset** to **Bottom Left**, **Pivot** to **(0.5, 0)**, **Pos X**, **Pos Y** and **Pos Z** to `(X:94,Y:3, Z:0)`, and finally, **Width** to `181` and **Height** to `180.5`.



Game*
MyHero
MyGameManager
UICanvas
HeroLifeBar
HeroHPBack
HeroHPBar
HeroHPFill
HeroThumbnail
EventSystem

HeroThumbnail Static
Tag Untagged Layer UI

left / bottom	Pos X	Pos Y	Pos Z
	94	3	0
	Width	Height	
	181	180.5	

Anchors
Min X 0 Y 0
Max X 0 Y 0
Pivot X 0.5 Y 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Source Image ui_hp_hero_thumb
Color
Material None (Material)
Raycast Target ✓
Image Type Simple
Preserve Aspect

That’s it for your hero’s UI. Guess what you’ll do next — that’s right, it’s time for some scripting to add control for the life bar.

Create a new C# script in the **Scripts** folder named **LifeBar**. Open the script and replace its contents with the following:

```
//1
using UnityEngine;
using UnityEngine.UI;

public class LifeBar : MonoBehaviour {
  //2
  public Image fillImage;
  public Image thumbnailImage;

  //3
  public Sprite[] fillSprites;
}
```

1. `LifeBar` class will use the namespace `UnityEngine.UI` because the UI components and classes are inside that namespace.

2. `fillImage` is the progress bar this script will be attached to. `thumbnailImage` is a reference to the hero’s thumbnail.

3. This stores an array of sprites to color the progress bar. These will range from red (low health) to green (full health).

Add the following methods below the variable declarations:

```
//1
void Start() {
  SetProgress(1.0f);
}

//2
private Sprite SpriteForProgress(float progress) {
  if (progress >= 0.5f) {
    return fillSprites[0];
  }
  if (progress >= 0.25f) {
    return fillSprites[1];
  }
  return fillSprites[2];
}

//3
public void SetThumbnail(Sprite image, Color color) {
  thumbnailImage.sprite = image;
  thumbnailImage.color = color;
}

//4
public void SetProgress(float progress) {
  fillImage.fillAmount = progress;
  fillImage.sprite = SpriteForProgress(progress);
}

//5
public void EnableLifeBar(bool enabled) {
  foreach (Transform tr in transform) {
    tr.gameObject.SetActive(enabled);
  }
}
```

These methods establish the behavior of the LifeBar class.

1. `Start` sets the LifeBar to the full value of `1.0`.
2. `SpriteForProgress`, a helper method, returns a sprite for a given `progress` parameter. When progress is `0.5` or higher, it returns the first sprite. When it's `0.25` or higher you get the second sprite. Otherwise, it returns the third and last sprite. Note that the `fillSprites` array will be set in the Inspector when in the game.
3. `SetThumbnail` replaces the sprite parameter of `thumbnailImage`. It also introduces a `color` parameter that will tint the `thumbnailImage`.

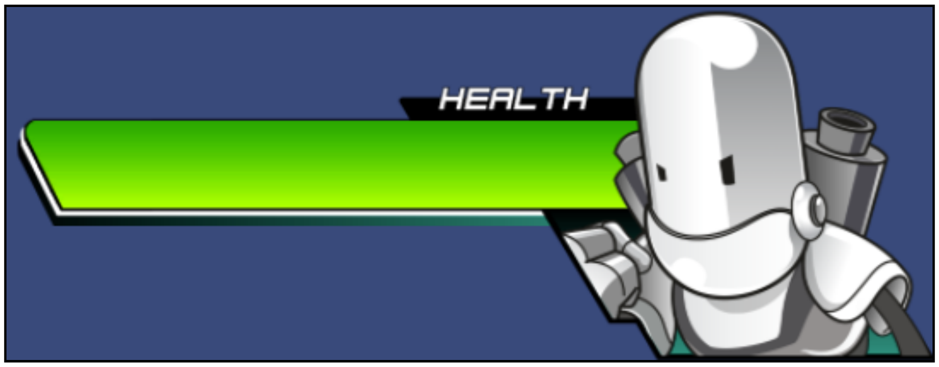
4. `SetProgress` updates the `fillImage` progress bar fill and gets the fill sprite via the helper method `SpriteForProgress`.

5. `EnableLifeBar` accepts the `enabled` parameter to toggle the life bar and its children on or off.

Save the script and return to the editor. You've just finished the pompadoured hero's life bar!

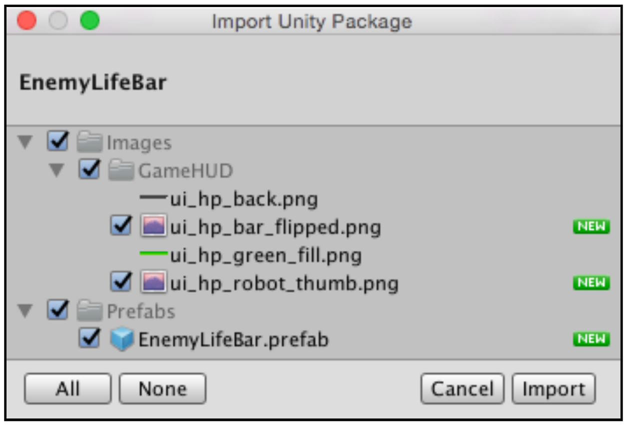
Health bar for the enemy

The enemy health will display on the right side of the screen, opposite of the hero's indicator. It'll display when an opponent takes a hit.



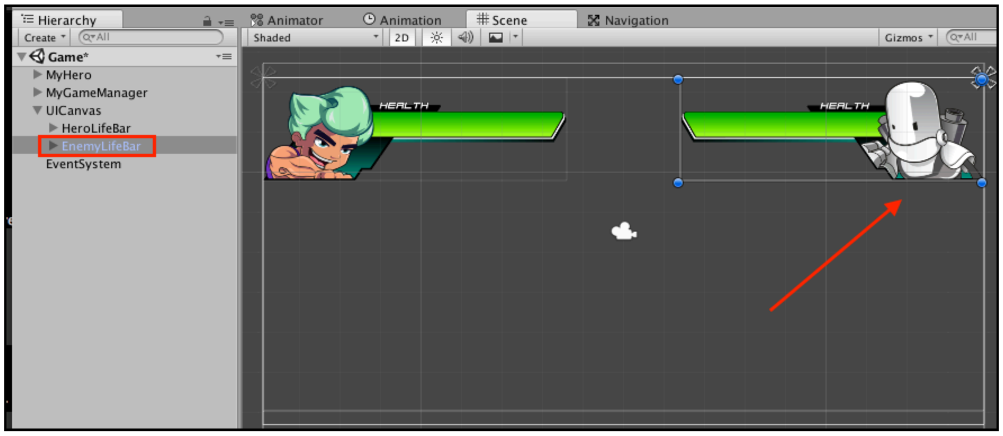
For convenience, the enemy's health indicator is mostly complete. You'll just add the LifeBar component.

Import **EnemyLifeBar.unitypackage** from the Unity Packages folder, which contains sprites and a prefab for the enemy's life bar.

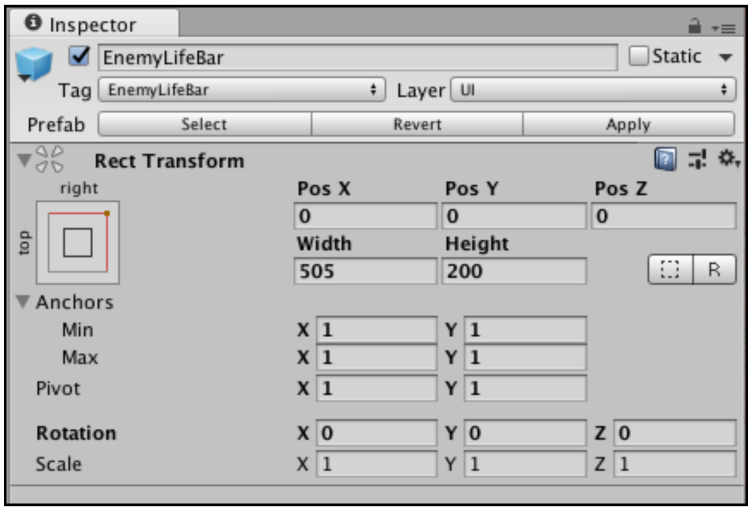


Drag the **EnemyLifeBar** prefab from the **Prefabs** folder onto **UICanvas** in the Hierarchy, and then set it as the second child of UICanvas.

The enemy's life bar will appear on the right side of the camera's view.



If the **EnemyLifeBar** is out of place, set **RectTransform** to **Anchor Preset Top Right**, its **Pivot** to `(X:1,Y:1)`, **Pos X,Y,Z** to `(X:0, Y:0, Z:0)`, **Width** to `505`, **Height** to `200`, **Rotation** to `(X:0, Y:0, Z:0)` and **Scale** to `(X:1, Y:1, Z:1)`.



EnemyLifeBar
Tag EnemyLifeBar   Layer UI

Rect Transform

	Pos X	Pos Y	Pos Z
	0	0	0
	Width	Height	
	505	200	

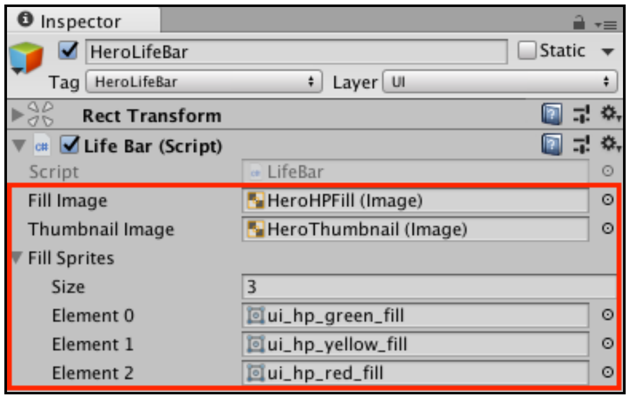
Anchors

Min	X 1	Y 1	
Max	X 1	Y 1	
Pivot	X 1	Y 1	
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Both life bars now display on the screen. Now you need to add the **LifeBar** component for control over their behavior.

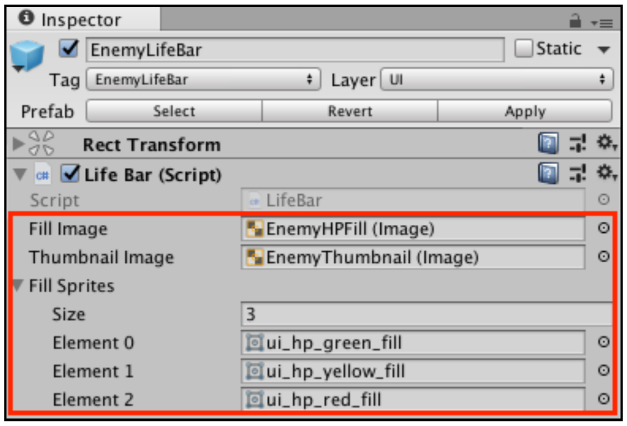
Add the **LifeBar** component to **HeroLifeBar**, drag **HeroHPFill** to the **Fill Image** slot and **HeroThumbnail** to **Thumbnail Image**.

Drag **ui_hp_green_fill**, **ui_hp_yellow_fill** and **ui_hp_red_fill** — in that exact order — from the **Images\GameHUD** folder to **FillSprites**.



HeroLifeBar
Tag HeroLifeBar Layer UI
Life Bar (Script)
Script LifeBar
Fill Image HeroHPFill (Image)
Thumbnail Image HeroThumbnail (Image)
Fill Sprites
Size 3
Element 0 ui_hp_green_fill
Element 1 ui_hp_yellow_fill
Element 2 ui_hp_red_fill

Select **EnemyLifeBar**, add a **LifeBar** component to it, and then fully expand it and its children. Drag **EnemyHPFill** to **Fill Image**, then drag **EnemyThumbnail** to **Thumbnail Image**, and then drag the following to **FillSprites** in the order listed: **ui_hp_green_fill**, **ui_hp_yellow_fill** and **ui_hp_red_fill**.

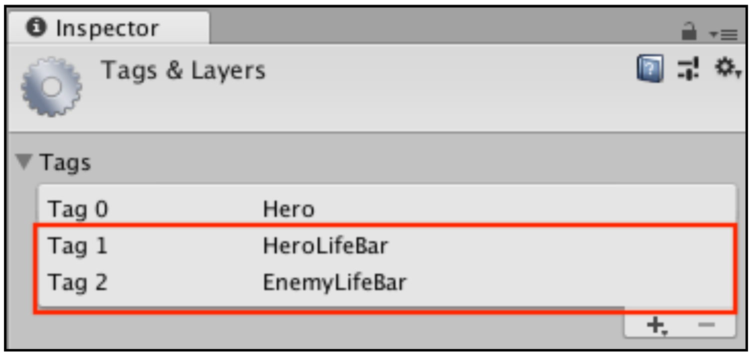


EnemyLifeBar
Tag EnemyLifeBar Layer UI
Prefab
Life Bar (Script)
Script LifeBar
Fill Image EnemyHPFill (Image)
Thumbnail Image EnemyThumbnail (Image)
Fill Sprites
Size 3
Element 0 ui_hp_green_fill
Element 1 ui_hp_yellow_fill
Element 2 ui_hp_red_fill

Note: Tags make it easy to access these LifeBar components from a script.

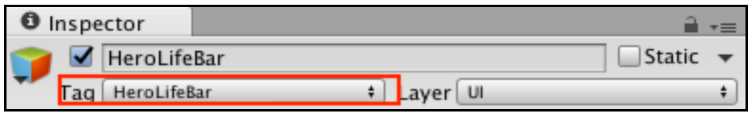
Open the **Tags List** by selecting **Edit \ Project Settings \ Tags & Layers** in the **Top Menu** and expanding **Tags**.

Add **HeroLifeBar** and **EnemyLifeBar** as new tags.

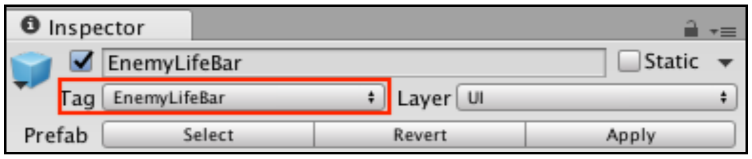


Tags
Tag 0 Hero
Tag 1 HeroLifeBar
Tag 2 EnemyLifeBar

Select **HeroLifeBar** in the Hierarchy then set its **tag** to **HeroLifeBar**. Select **EnemyLifeBar** and set its **tag** to **EnemyLifeBar**.

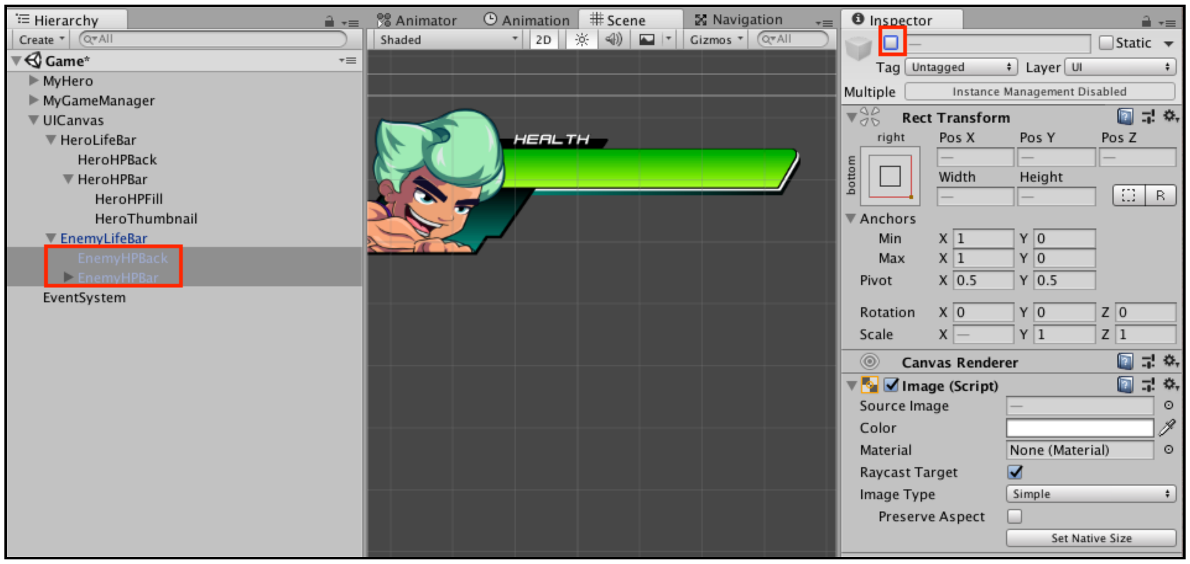


HeroLifeBar
Tag HeroLifeBar Layer UI



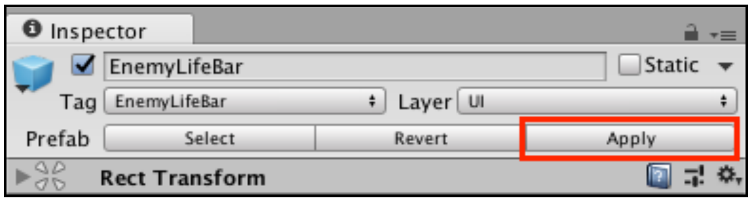
EnemyLifeBar
Tag EnemyLifeBar Layer UI

Disable the children of EnemyLifeBar by selecting **EnemyHPBack** and **EnemyHPBar** in the Hierarchy and clearing the checkbox near the top-left of the Inspector. By default, these GameObjects will be disabled. They'll be enabled when the enemy life bar needs to be displayed.



Tag Untagged Layer UI
Multiple Instance Management Disabled
right Pos X — Pos Y — Pos Z —
bottom Width — Height —
Anchors
Min X 1 Y 0
Max X 1 Y 0
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X — Y 1 Z 1
Source Image —
Color
Material None (Material)
Raycast Target ✓
Image Type Simple
Preserve Aspect ☐

Select **EnemyLifeBar** from the Hierarchy and click **Apply** in the **Inspector** to save its prefab.



EnemyLifeBar
Tag EnemyLifeBar Layer UI

The scene is now set!

Now you need to allow the scripts to use the life bars. Open the **Actor** script and add the following variables right above the `Start()` method:

```
public LifeBar lifeBar;
public Sprite actorThumbnail;
```

`lifebar` saves a reference to a `LifeBar` component, while `actorThumbnail` makes a reference to the thumbnail of this actor.

When the actor takes damage, the value of the `lifeBar` needs to update, so insert the following script at the end of `TakeDamage`:

```
lifeBar.EnableLifeBar(true); // 1
lifeBar.SetProgress(currentLife / maxLife); // 2
Color color = baseSprite.color; // 3
if (currentLife < 0) { // 4
  color.a = 0.75f;
}
lifeBar.SetThumbnail(actorThumbnail, color); // 5
```

This block updates the actor's `lifebar` when it takes damage. Specifically, it:

1. Shows the lifebar.
2. Sets the lifebar amount to the percentage value of the actor's life.
3. Gets the color of this actor's sprite.
4. Makes the actor semi-transparent when its health falls below 0.
5. Places the specific actor's thumbnail next to the lifebar and tints it with the actor's color.

Save the script.

Open the **Hero** script — you need a reference in the script to the hero's lifebar. Add the following override right above the `Update()` method:

```
protected override void Start() {
  base.Start();
  lifeBar =
GameObject.FindGameObjectWithTag("HeroLifeBar").GetComponent<LifeBar>();
  lifeBar.SetProgress(currentLife / maxLife);
}
```

This method overrides the Actor's `Start` method.

1. Calls the base class `Start()` method.
2. Searches the GameObjects for the one tagged with `HeroLifeBar` and gets the attached `LifeBar` component.

3. Updates the lifebar's progress.

Save the **Hero** script and open the **Enemy** script to implement references for the dreadful droid lifebar. Add the following override method:

```
protected override void Start() {
  base.Start();
  lifeBar =
GameObject.FindGameObjectWithTag("EnemyLifeBar").GetComponent<LifeBar>();
  lifeBar.SetProgress(currentLife / maxLife);
}
```

This method behaves similarly to the one you set up for the hero, but it searches for the `EnemyLifeBar` tag instead of the `HeroLifeBar` tag.

Save this script and open the **GameManager** script. Add the following variable at the top of the script.

```
public LifeBar enemyLifeBar;
```

This variable creates a reference to the enemy's life bar.

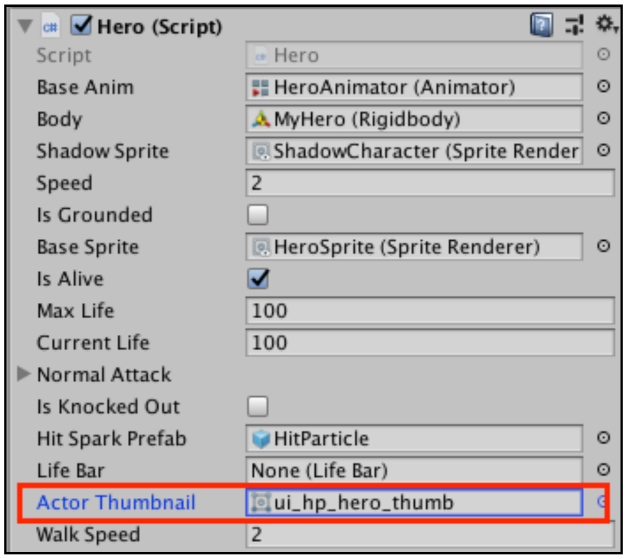
Next, in the `CompleteCurrentEvent` method, insert the following line before the `if (!hasRemainingEvents)` condition:

```
enemyLifeBar.EnableLifeBar (false);
```

This hides the enemy's life bar when the player completes a battle event.

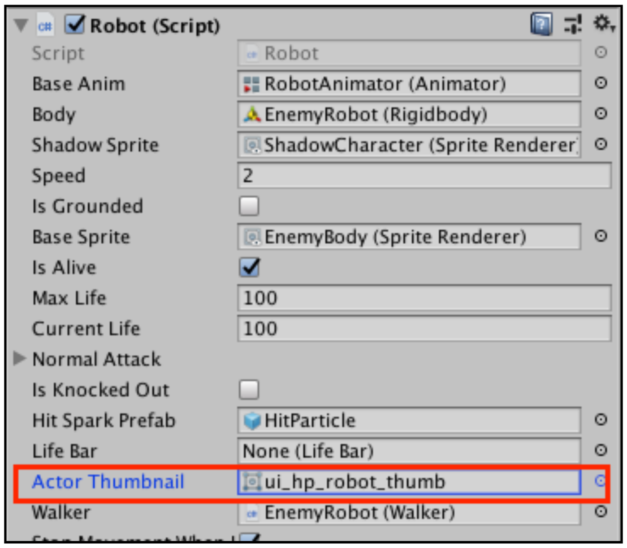
Save this script and return to the editor. It's time to assign sprites.

Select **MyHero** in the Hierarchy and set the **Actor Thumbnail** variable of the hero to **ui_hp_hero_thumb**.



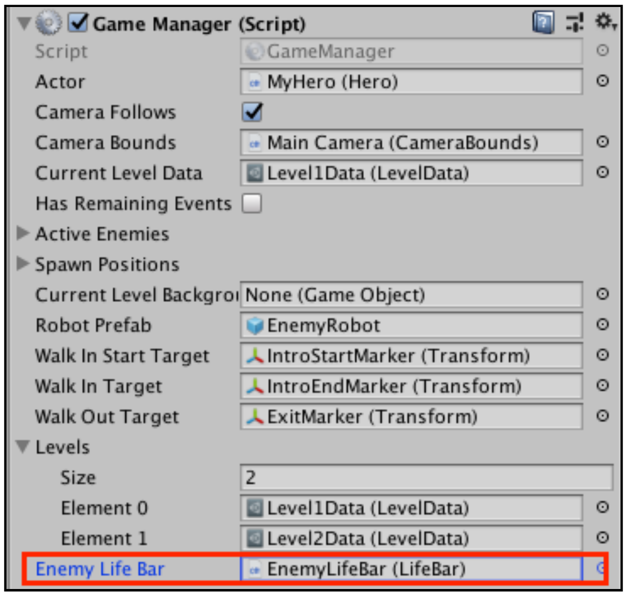
Hero (Script)	
Script	Hero
Base Anim	HeroAnimator (Animator)
Body	MyHero (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Render
Speed	2
Is Grounded	☐
Base Sprite	HeroSprite (Sprite Renderer)
Is Alive	✓
Max Life	100
Current Life	100
Normal Attack	
Is Knocked Out	☐
Hit Spark Prefab	HitParticle
Life Bar	None (Life Bar)
Actor Thumbnail	ui_hp_hero_thumb
Walk Speed	2

Select the **EnemyRobot** prefab in the **Prefabs** folder, and then set the robot's **Actor Thumbnail** variable to **ui_hp_robot_thumb**.


Robot (Script)

Script	Robot
Base Anim	RobotAnimator (Animator)
Body	EnemyRobot (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Renderer
Speed	2
Is Grounded	☐
Base Sprite	EnemyBody (Sprite Renderer)
Is Alive	✓
Max Life	100
Current Life	100
Normal Attack	
Is Knocked Out	☐
Hit Spark Prefab	HitParticle
Life Bar	None (Life Bar)
Actor Thumbnail	ui_hp_robot_thumb
Walker	EnemyRobot (Walker)

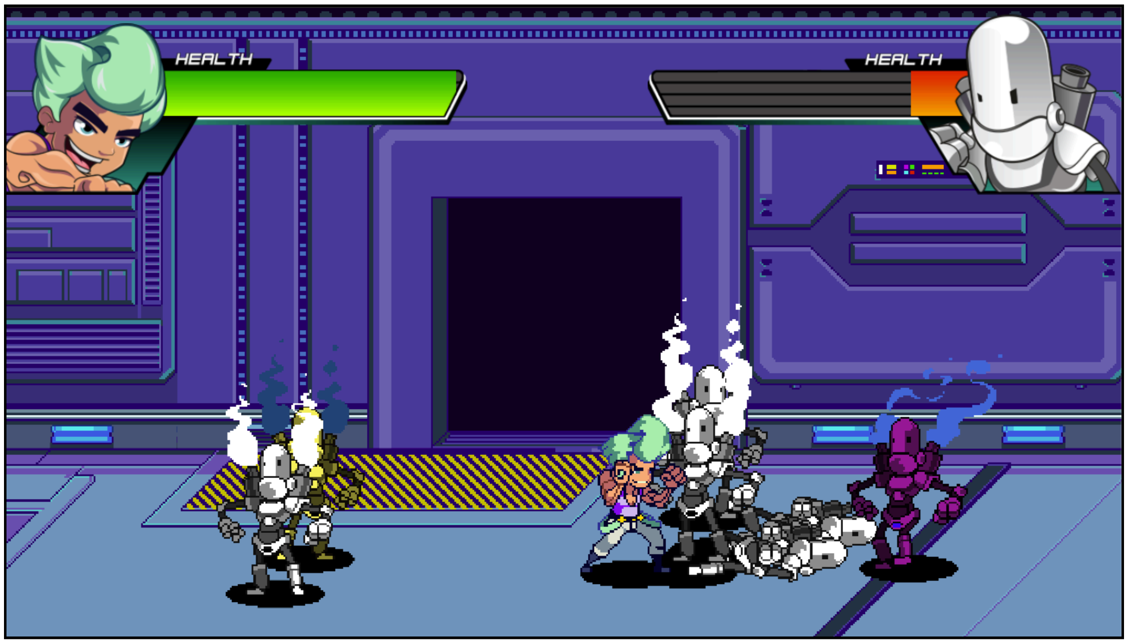
Select **MyGameManager** in the Hierarchy and drag **UICanvas**' child, **EnemyLifeBar** to the **Enemy Life Bar** slot.


Game Manager (Script)

Script	GameManager
Actor	MyHero (Hero)
Camera Follows	✓
Camera Bounds	Main Camera (CameraBounds)
Current Level Data	Level1Data (LevelData)
Has Remaining Events	☐
Active Enemies	
Spawn Positions	
Current Level Backgroi	None (Game Object)
Robot Prefab	EnemyRobot
Walk In Start Target	IntroStartMarker (Transform)
Walk In Target	IntroEndMarker (Transform)
Walk Out Target	ExitMarker (Transform)
Levels	
Size	2
Element 0	Level1Data (LevelData)
Element 1	Level2Data (LevelData)
Enemy Life Bar	EnemyLifeBar (LifeBar)

Click **Play** and commence with droid bashing.

The hero's HUD will appear at the top of the screen and display his current health. Watch how the EnemyLifeBar shows itself when you punch a droid. Awesome!

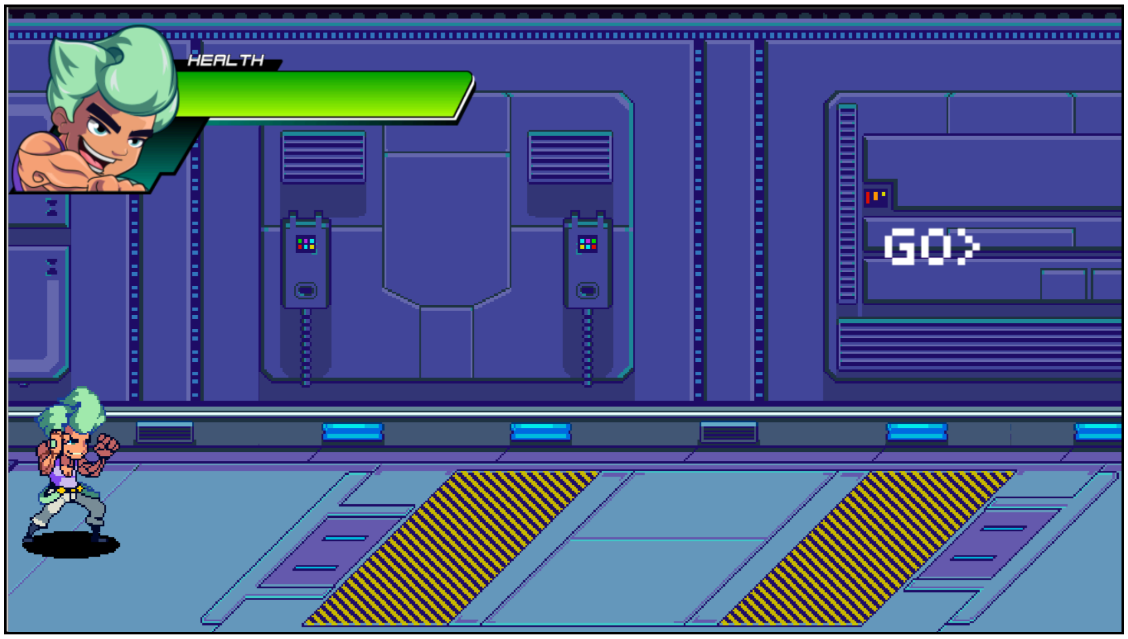


Keep bashing until you finish a battle event. Hmmm…something is missing. What could it be?

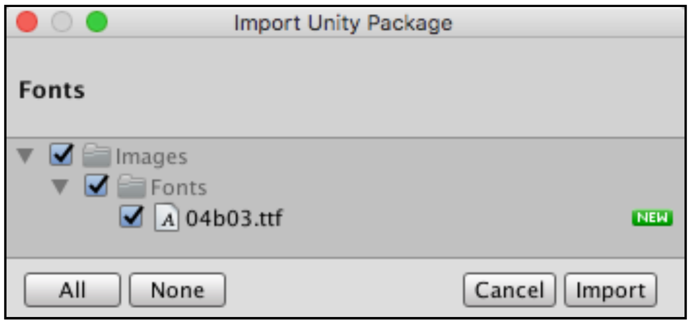
Give me a sign!

Beat-em-up games typically tell you to go forward with an unmissable "go" graphic that shows up after you've killed all the things. Arrows are often involved.

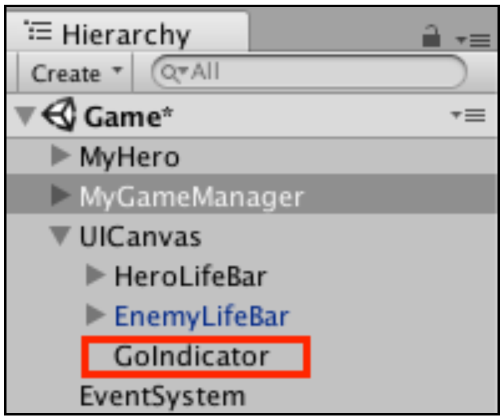
In the case of this game, the go sign will be a simple string of characters dressed in a font. A left angle bracket will serve as the arrow.



Import **Fonts.unitypackage** from the Unity Packages folder, which contains the `04b03` font.

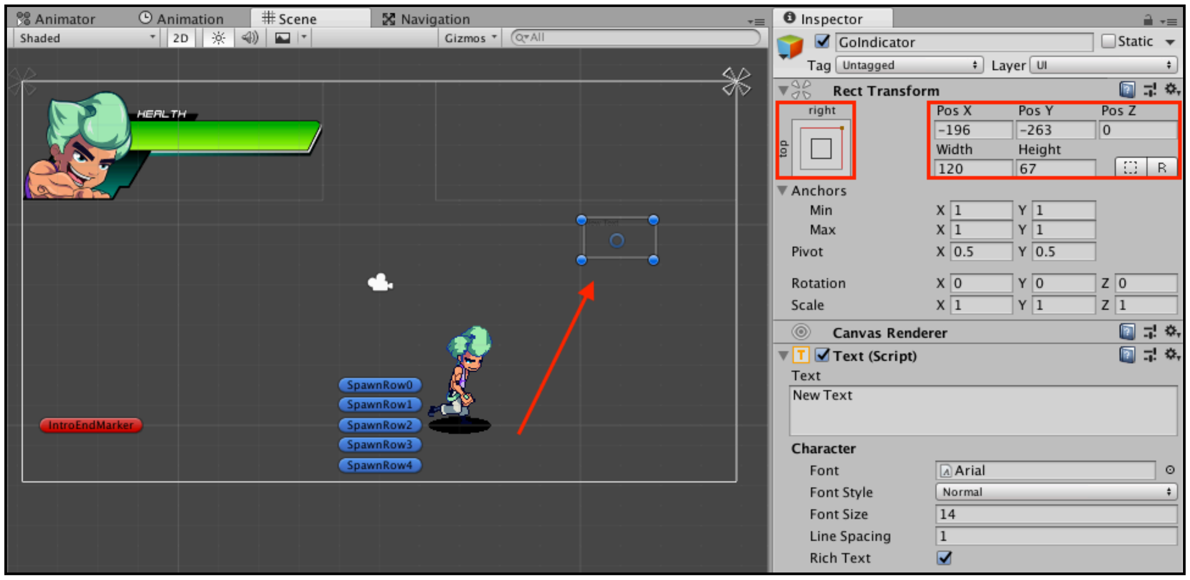


Select **UICanvas** in the Hierarchy, create a new **UI\Text** and rename it **GoIndicator** — you've just created a new GameObject with an attached text component.



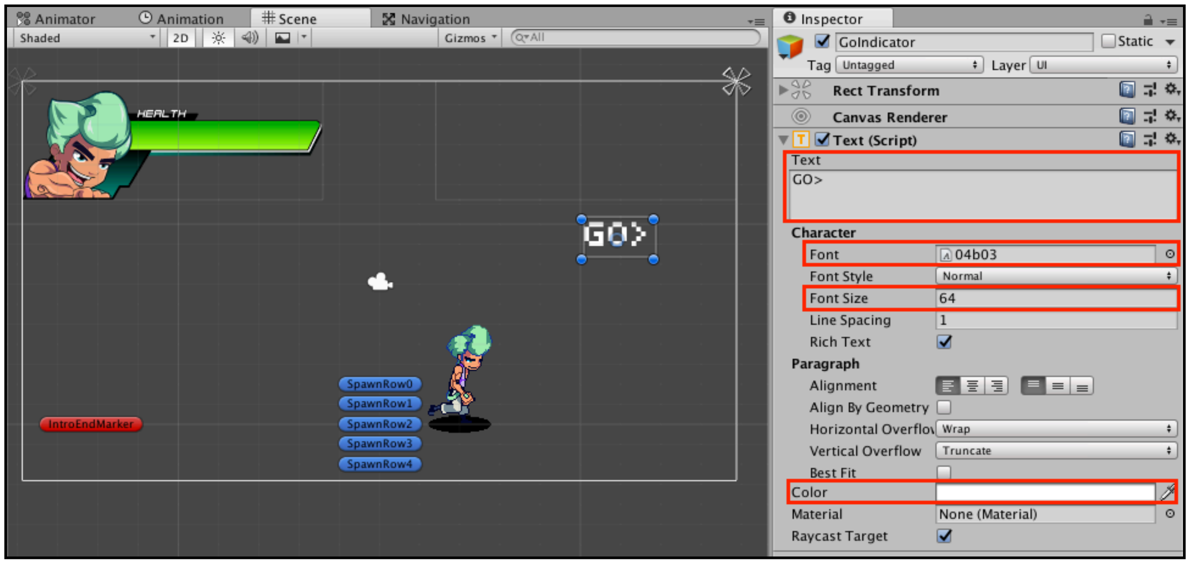
Set its **Anchor Preset** to **Top Right** and change **Pos X,Y,Z** to `(X:-196, Y:-263, Y:0)`.

Set **Width** to `120` and **Height** to `67` to position the text near the top-right corner of the main camera's frame.



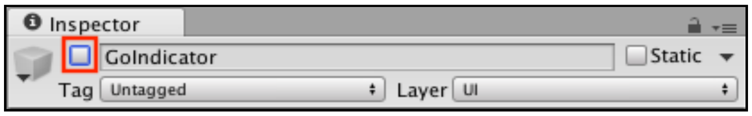
Inspector
GoIndicator Static
Tag Untagged Layer UI
Rect Transform
right
top
Pos X -196 Pos Y -263 Pos Z 0
Width 120 Height 67
Anchors
Min X 1 Y 1
Max X 1 Y 1
Pivot X 0.5 Y 0.5
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Canvas Renderer
Text (Script)
Text
New Text
Character
Font Arial
Font Style Normal
Font Size 14
Line Spacing 1
Rich Text ✓

In the text component, set **Text** to **GO>**, set **Font** to `04b03`, change **Font Size** to `64`, and then set the **Color** to pure white.



Inspector
GoIndicator Static
Tag Untagged Layer UI
Rect Transform
Canvas Renderer
Text (Script)
Text
GO>
Character
Font 04b03
Font Style Normal
Font Size 64
Line Spacing 1
Rich Text ✓
Paragraph
Alignment
Align By Geometry
Horizontal Overflow Wrap
Vertical Overflow Truncate
Best Fit
Color
Material None (Material)
Raycast Target ✓

Disable this GameObject.



Inspector
GoIndicator Static
Tag Untagged Layer UI

When a battle event ends, the go sign should flicker. You'll set up this behavior in a script.

Open **GameManager.cs** and add the following variable below `enemyLifeBar`:

```
public GameObject goIndicator;
```

This adds a reference to the go indicator.

Add the following methods below `AnimateNextLevel` method to animate the go indicator:

```
//1
private void ShowGoIndicator() {
  StartCoroutine(FlickerGoIndicator(4));
}

//2
private IEnumerator FlickerGoIndicator(int count = 4) {
  while (count > 0) {
    goIndicator.SetActive(true);
    yield return new WaitForSeconds(0.2f);
    goIndicator.SetActive(false);
    yield return new WaitForSeconds(0.2f);
    count--;
  }
}
```

Here's how the animation works:

1. `ShowGoIndicator` triggers the flicker by starting the `FlickerGoIndicator` coroutine.
2. `FlickerGoIndicator` animates the flicker according to the parameter `count` times — it toggles the active state for `GoIndicator` between on and off every 0.2 seconds.

Now you'll add method calls to trigger this animation. Insert the following line at the end of `DidFinishIntro`:

```
ShowGoIndicator();
```

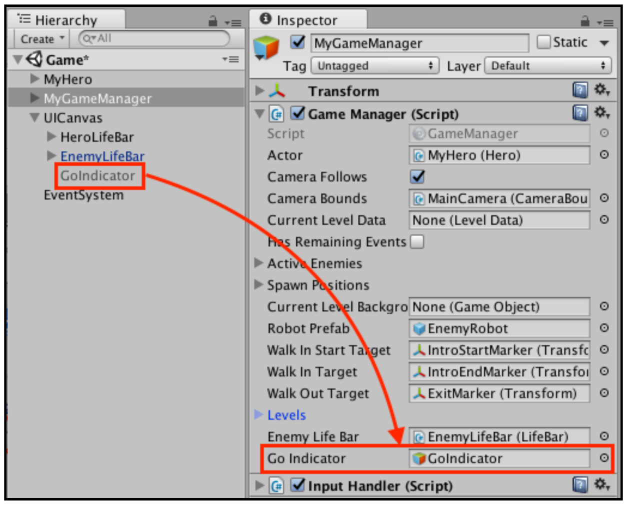
This method creates a "go time" queue, in the form of a flicker on the indicator at the end of the hero's walk-in animation.

Add the below `else` condition to the bottom of the `CompleteCurrentEvent()` method.

```
else {
  ShowGoIndicator();
}
```

This triggers the indicator when the player finishes a battle event.

Save the script and open the editor. Select **MyGameManager** in the Hierarchy. Set the **Go Indicator** variable in the GameManager script to **UICanvas\GoIndicator**.

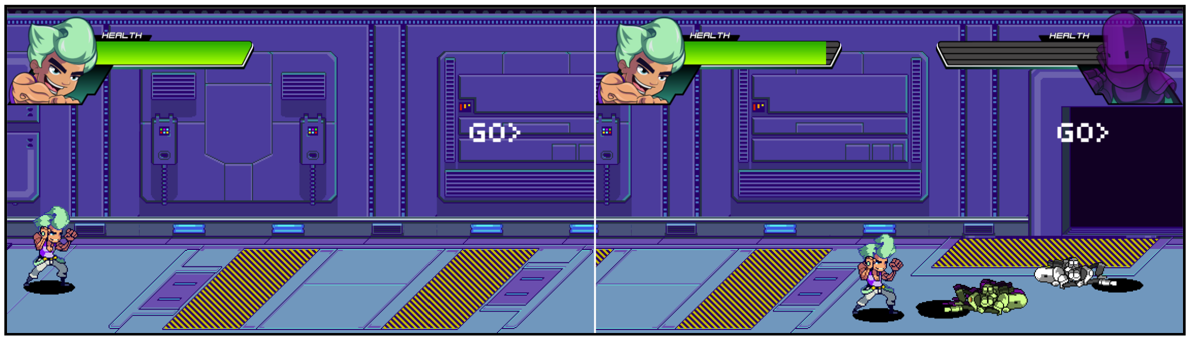


MyGameManager
Tag Untagged Layer Default

Game Manager (Script)

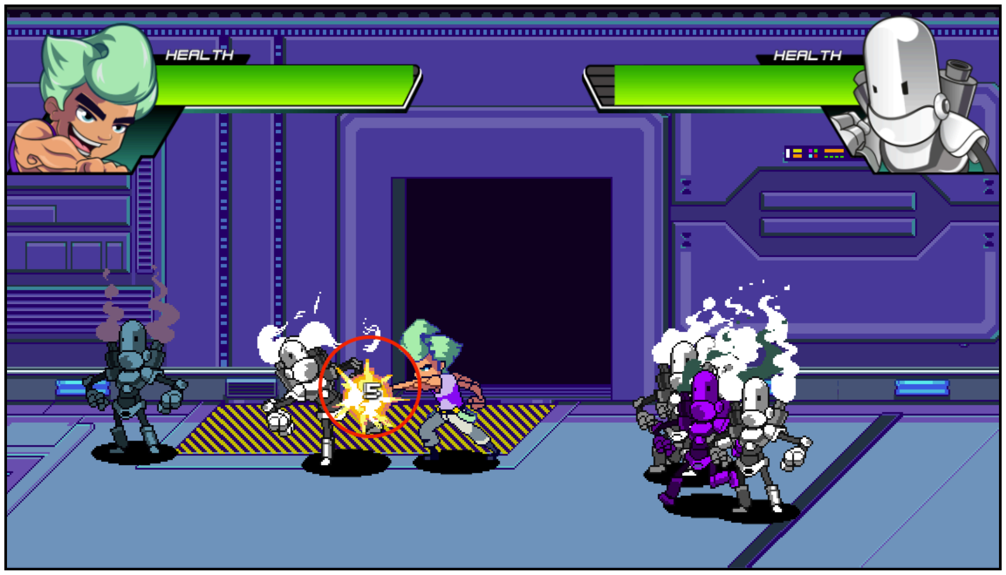
Script	GameManager
Actor	MyHero (Hero)
Camera Follows	✓
Camera Bounds	MainCamera (CameraBou
Current Level Data	None (Level Data)
Has Remaining Events	
Active Enemies	
Spawn Positions	
Current Level Backgro	None (Game Object)
Robot Prefab	EnemyRobot
Walk In Start Target	IntroStartMarker (Transfc
Walk In Target	IntroEndMarker (Transfoı
Walk Out Target	ExitMarker (Transform)
Levels	
Enemy Life Bar	EnemyLifeBar (LifeBar)
Go Indicator	GoIndicator

Save the scene and click **Play**! Watch for the flickering "GO>" sign when the game starts and after you've defeated every enemy in a battle event.



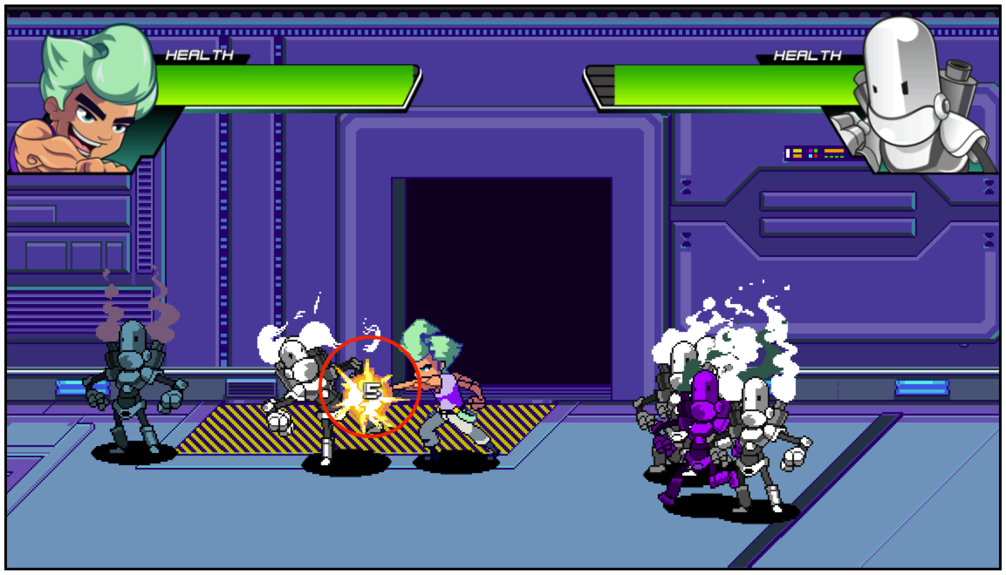
Quantify that damage for me!

A good strike to a metal hunk of junk is always more satisfying when you know just how badly you've hurt it. That's precisely what this UI feature is all about!



Import **HitValueText.unitypackage** from the **Unity Packages** folder, which contains a prefab you’ll use for the hit value.

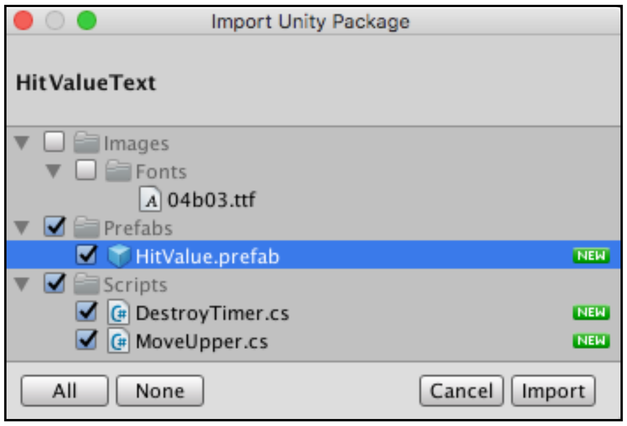
Next, you will use your newly found UI skills to add visual cues such as the one shown below.



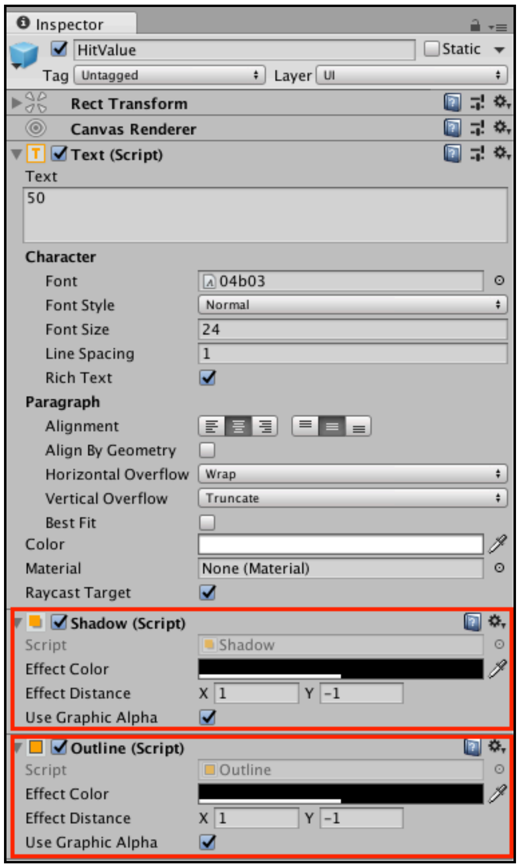
In the previous chapter, you added a hit effect animation for every attack that connects. However, with all the beat-downs going on, and all the hero’s awesome moves, it’s hard to tell which attack is the most effective. As the developer, you have knowledge of all the numbers in the game, including how much damage each attack inflicts. But the average player has no way of knowing, for example, that the second punch is stronger than the first punch.

One obvious solution is to display the damage numbers for every attack as it occurs, as long as the attack connects with an enemy. That's what you'll do next.

Import **HitValueText.unitypackage** from the **Unity Packages** folder, which contains a prefab you'll use for the hit value.



The **HitValue** prefab uses two text components to render the damage value: a shadow component and an outline component.

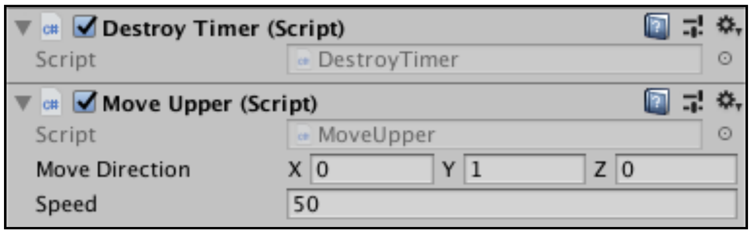


Text
50
Character
Font 04b03
Font Style Normal
Font Size 24
Line Spacing 1
Rich Text ✓
Paragraph
Alignment
Align By Geometry
Horizontal Overflow Wrap
Vertical Overflow Truncate
Best Fit
Color
Material None (Material)
Raycast Target ✓

Shadow (Script)
Script Shadow
Effect Color
Effect Distance X 1 Y -1
Use Graphic Alpha ✓

Outline (Script)
Script Outline
Effect Color
Effect Distance X 1 Y -1
Use Graphic Alpha ✓

This prefab also uses two new components: A **DestroyTimer** component, which destroys the GameObject to which it is attached after a set time, and a **MoveUpper** component that moves any Transform in any direction as determined by its **Move Direction** parameter.



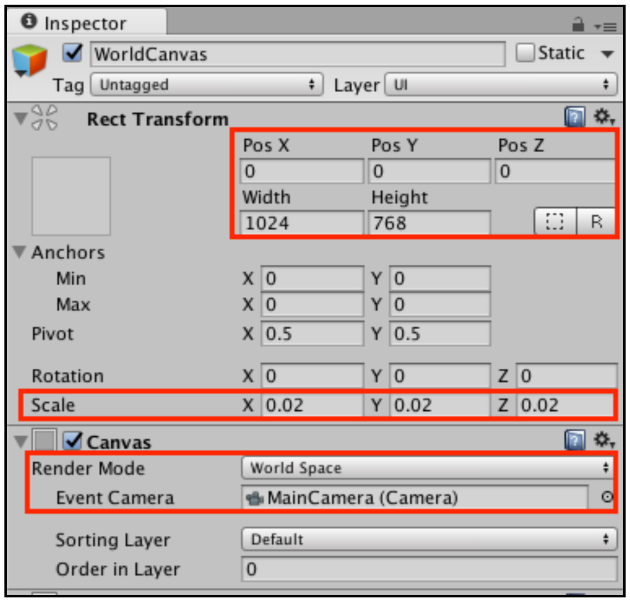
Destroy Timer (Script)
Script: DestroyTimer

Move Upper (Script)
Script: MoveUpper
Move Direction: X 0, Y 1, Z 0
Speed: 50

Create a new **Canvas** in the Hierarchy. Name it **WorldCanvas** — this will be the parent to all the damage values.

Set its **Render Mode** to **World Space**. Set its **Event Camera** to **MainCamera**, which is a child of **MyGameManager**, and adjust **Position** to `(X: 0, Y: 0, Z: 0)`.

Set **Scale** to `(X:0.02, Y:0.02, Z:0.02)`, and then adjust its **Width** to `1024` and **Height** to `768`.

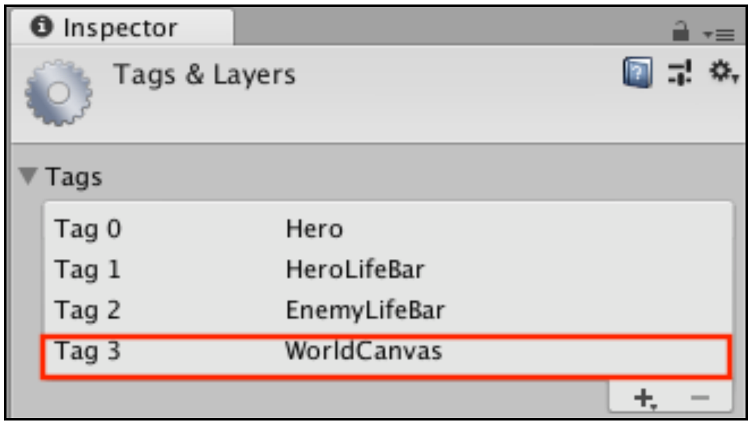


Inspector
WorldCanvas — Static
Tag: Untagged — Layer: UI

Rect Transform
Pos X: 0, Pos Y: 0, Pos Z: 0
Width: 1024, Height: 768
Anchors
Min: X 0, Y 0
Max: X 0, Y 0
Pivot: X 0.5, Y 0.5
Rotation: X 0, Y 0, Z 0
Scale: X 0.02, Y 0.02, Z 0.02

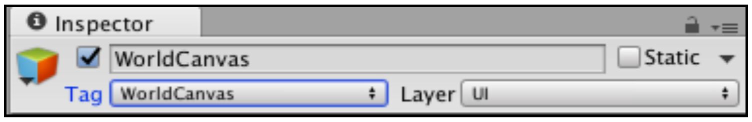
Canvas
Render Mode: World Space
Event Camera: MainCamera (Camera)
Sorting Layer: Default
Order in Layer: 0

Navigate to **Edit \ Project Settings \ Tags & Layers** and open the **Tags & Layers** settings from the top menu. Add a tag named **WorldCanvas**.



Tag 0	Hero
Tag 1	HeroLifeBar
Tag 2	EnemyLifeBar
Tag 3	WorldCanvas

Select **WorldCanvas** in the Hierarchy and set its tag to **WorldCanvas**.



WorldCanvas
Tag WorldCanvas
Layer UI

Open the Actor script. Add this `using` line to the top:

```
using UnityEngine.UI;
```

This `using` line includes a reference to the UI namespace in the script so that it can access the `UnityEngine.UI.Text` class.

Add the following variable below `actorThumbnail`:

```
public GameObject hitValuePrefab;
```

`hitValuePrefab` references the prefab you just imported. When an actor takes a hit, this object will be instantiated.

Insert the code below at the end of the `ShowHitEffects` method:

```
//1
GameObject obj = Instantiate(hitValuePrefab);
obj.GetComponent<Text>().text = value.ToString();
obj.GetComponent<DestroyTimer>().EnableTimer(1.0f);

//2
GameObject canvas = GameObject.FindGameObjectWithTag("WorldCanvas");
obj.transform.SetParent(canvas.transform);
obj.transform.localRotation = Quaternion.identity;
obj.transform.localScale = Vector3.one;
obj.transform.position = position;
```

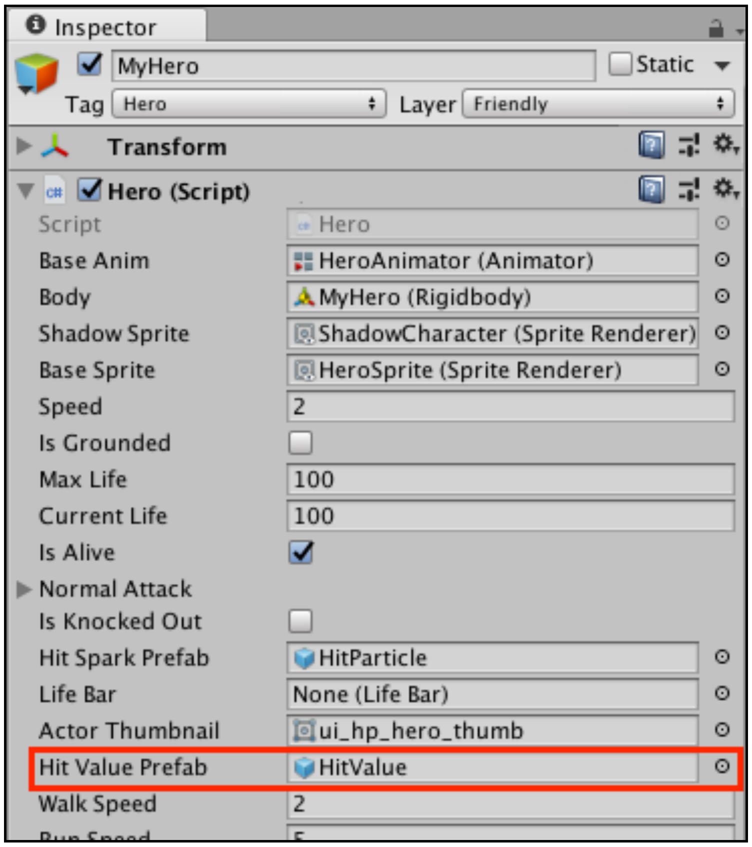
This code block instantiates the `hitValuePrefab` damage value GameObject.

1. Creates a new instance of the `hitValuePrefab` and sets its text to the amount of damage taken. After one second, it triggers the `DestroyTimer` script.

2. Finds `WorldCanvas` by looking for the object tagged with the `WorldCanvas` tag. Then it becomes the child of the damage value GameObject. Lastly, the damage value is positioned in the place of the hit.

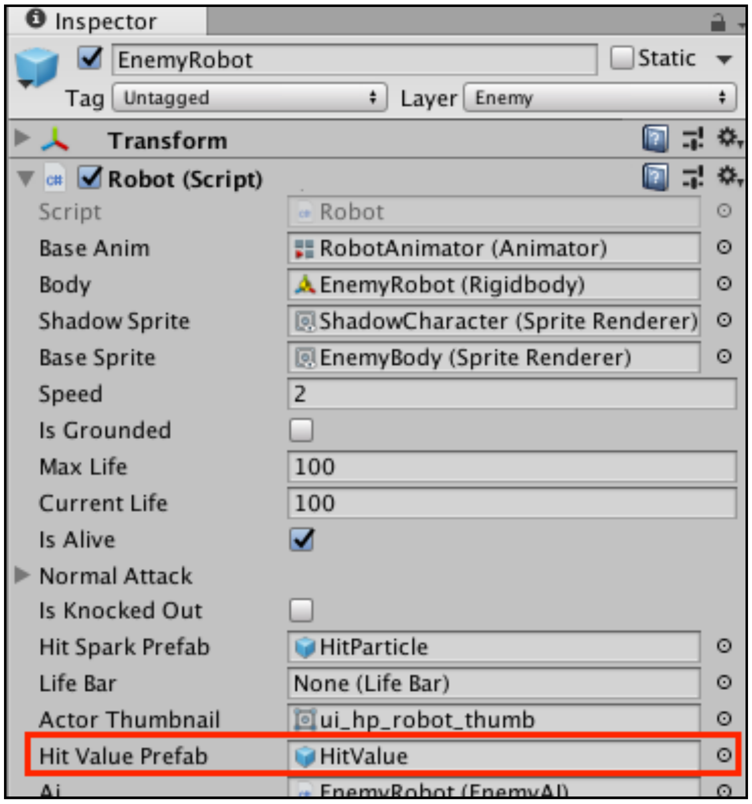
Now you're ready to add the hit value prefab for the hero and robot. Save the script and return to the editor.

Find the **Hero** component of **MyHero**. Find the **HitValue** prefab in the **Prefabs** folder and drag it into the **Hit Value Prefab** slot.



Script	Hero
Base Anim	HeroAnimator (Animator)
Body	MyHero (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Renderer)
Base Sprite	HeroSprite (Sprite Renderer)
Speed	2
Is Grounded	☐
Max Life	100
Current Life	100
Is Alive	✓
Normal Attack	
Is Knocked Out	☐
Hit Spark Prefab	HitParticle
Life Bar	None (Life Bar)
Actor Thumbnail	ui_hp_hero_thumb
Hit Value Prefab	HitValue
Walk Speed	2

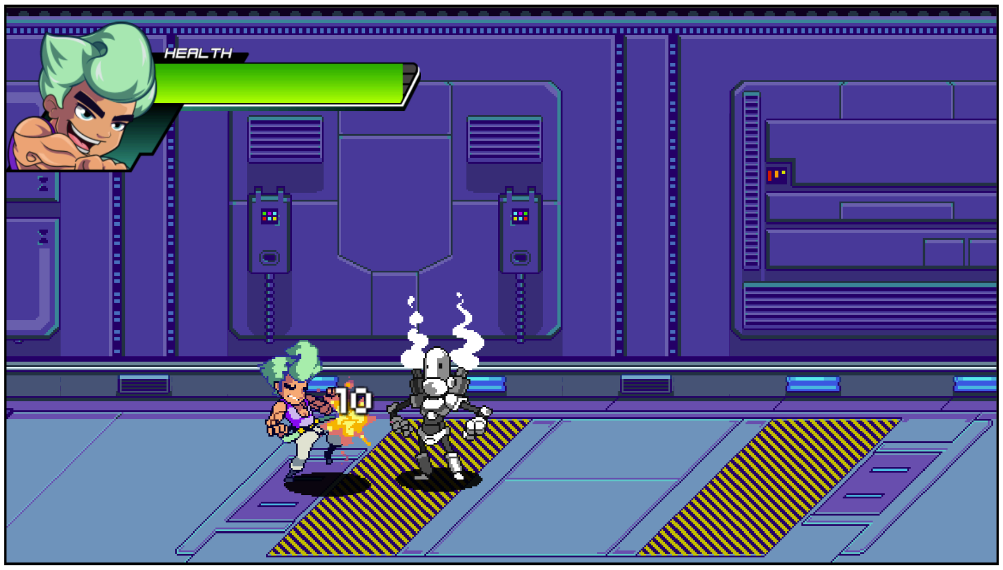
Select the **EnemyRobot** prefab in **Assets / Prefabs** and drag **HitValue** to its **Hit Value Prefab** slot.



Script	Robot
Base Anim	RobotAnimator (Animator)
Body	EnemyRobot (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Renderer)
Base Sprite	EnemyBody (Sprite Renderer)
Speed	2
Is Grounded	☐
Max Life	100
Current Life	100
Is Alive	✓
Normal Attack	
Is Knocked Out	☐
Hit Spark Prefab	HitParticle
Life Bar	None (Life Bar)
Actor Thumbnail	ui_hp_robot_thumb
Hit Value Prefab	HitValue
Ai	EnemyRobot (EnemyAI)

Save the scene and the project. Run the game.

Every time you destroy a droid, you'll get the satisfaction of knowing exactly how much damage you did to that miserable hunk of junk.



Congrats for finishing up the health bars! If you've been glued to your seat since you cracked open this chapter, now is a great time to treat yourself to a break. You just finished the HUD, and next up is tackling the missing Mobile UI.

Bashing droids on-the-go with mobile UI

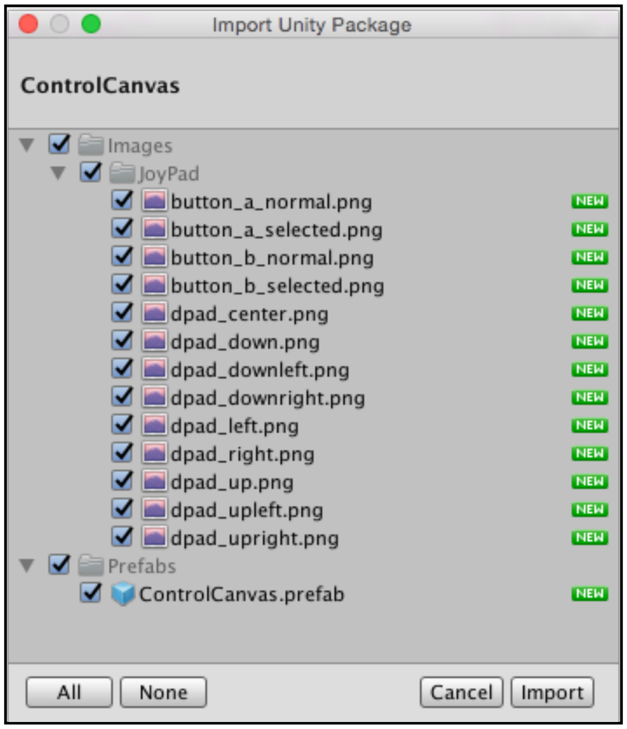
Things are coming along in Pompadroid. So far, you've built two life bars, a go indicator, and a damage value indicator.

At this point, it's time to turn your attention to the UI for mobile devices. Your keyboard works, but there is nothing for players who want to thrash droids on the go.

Pompadroid's on-screen controls will feature a directional pad (d-pad) for movement and two buttons: one for attacking and one for jumping. You'll position these controls at the bottom of the screen.

Implementing the d-pad

Import **ControlCanvas.unitypackage** from the **Unity Packages** folder. This package contains the sprites you'll need to implement the controls. Note the prefab named **ControlCanvas** — its hierarchy is already complete.



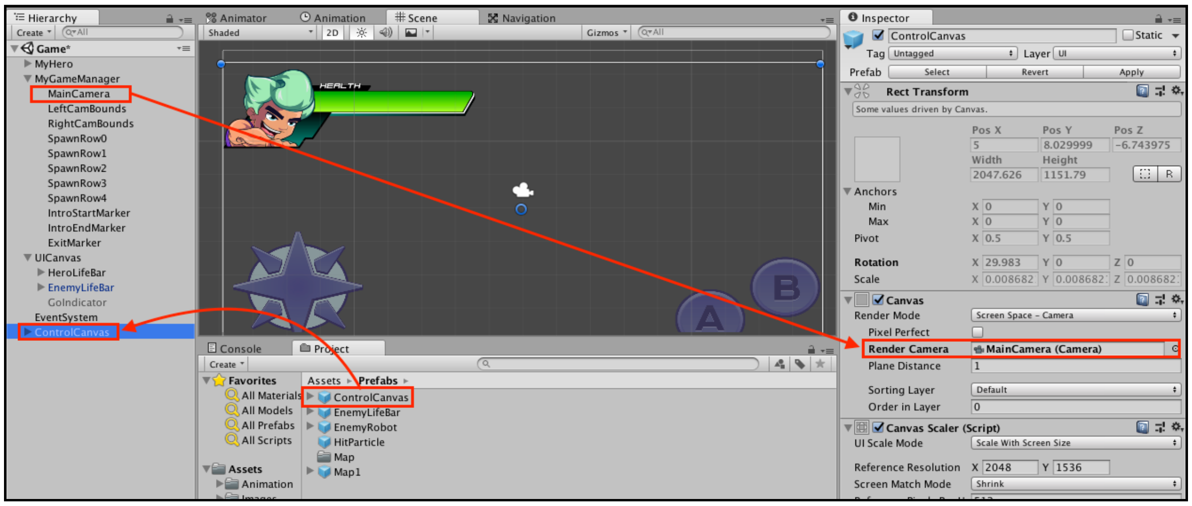
Import Unity Package

ControlCanvas

```
▼ ☑ Images
  ▼ ☑ JoyPad
      ☑ button_a_normal.png       NEW
      ☑ button_a_selected.png     NEW
      ☑ button_b_normal.png       NEW
      ☑ button_b_selected.png     NEW
      ☑ dpad_center.png           NEW
      ☑ dpad_down.png             NEW
      ☑ dpad_downleft.png         NEW
      ☑ dpad_downright.png        NEW
      ☑ dpad_left.png             NEW
      ☑ dpad_right.png            NEW
      ☑ dpad_up.png               NEW
      ☑ dpad_upleft.png           NEW
      ☑ dpad_upright.png          NEW
▼ ☑ Prefabs
    ☑ ControlCanvas.prefab        NEW
```

All | None | Cancel | Import

Drag the **ControlCanvas** prefab into the scene. Set the **Render Camera** in the Canvas component to **MainCamera**.



```
Hierarchy
▼ Game*
  ▶ MyHero
  ▼ MyGameManager
      MainCamera
      LeftCamBounds
      RightCamBounds
      SpawnRow0
      SpawnRow1
      SpawnRow2
      SpawnRow3
      SpawnRow4
      IntroStartMarker
      IntroEndMarker
      ExitMarker
  ▼ UICanvas
    ▶ HeroLifeBar
    ▶ EnemyLifeBar
      GoIndicator
    EventSystem
  ▶ ControlCanvas
```

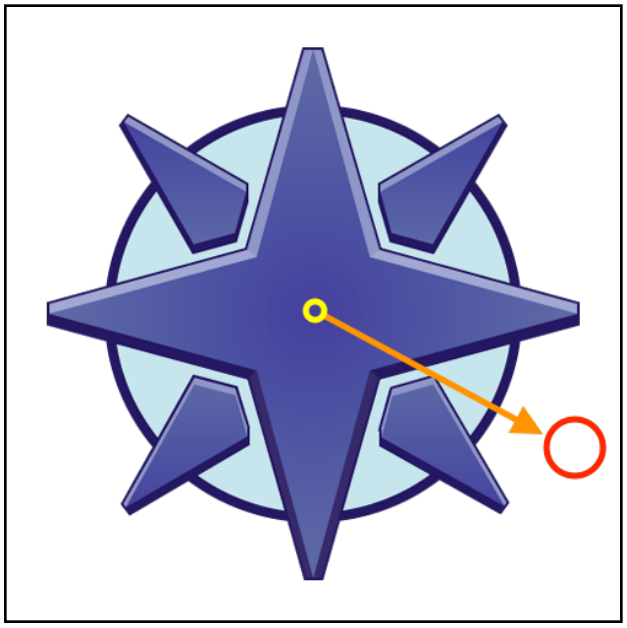
```
Project
Assets ▸ Prefabs
  ▶ ControlCanvas
  ▶ EnemyLifeBar
  ▶ EnemyRobot
    HitParticle
    Map
  ▶ Map1
```

```
Inspector
☑ ControlCanvas                              ☐ Static
Tag  Untagged          Layer  UI
Prefab   Select   Revert   Apply
▼ Rect Transform
Some values driven by Canvas.
            Pos X        Pos Y        Pos Z
            5            8.029999     -6.743975
            Width        Height
            2047.626     1151.79
▼ Anchors
    Min     X 0          Y 0
    Max     X 0          Y 0
Pivot       X 0.5        Y 0.5
Rotation    X 29.983     Y 0          Z 0
Scale       X 0.008682   Y 0.008682   Z 0.008682
▼ ☑ Canvas
Render Mode         Screen Space - Camera
  Pixel Perfect     ☐
  Render Camera     MainCamera (Camera)
  Plane Distance    1
  Sorting Layer     Default
  Order in Layer    0
▼ ☑ Canvas Scaler (Script)
UI Scale Mode              Scale With Screen Size
Reference Resolution       X 2048    Y 1536
Screen Match Mode          Shrink
```

To make the UI work, you'll need a class that handles joystick movement.

The diagram below shows how the d-pad calculates direction. First, it detects a touch on the d-pad sprite, shown as the red circle. Then it takes the touch position and the center of the d-pad sprite to calculate the direction vector.

The direction vector then becomes the input vector, which allows the hero to move around the map. The d-pad sprite also changes depending on which sprite is currently selected.



Note that the touch action doesn’t end until the player releases their finger from the d-pad. This allows the player to change direction by dragging over the d-pad sprite.

Scripting for touch handling

Create a new **C# script** called **ActionDPad**. Open the file and replace its contents with the following lines:

```
using System;
using UnityEngine;
using UnityEngine.Events;
using UnityEngine.EventSystems;
using UnityEngine.UI;

public class ActionDPad : UIBehaviour, IBeginDragHandler,
IEndDragHandler, IDragHandler, IPointerDownHandler, IPointerUpHandler {

  //1
  public enum ActionPadDirection {
    Up = 1,
    UpRight = 2,
    Right = 3,
    DownRight,
    Down,
    DownLeft,
    Left,
    UpLeft,
    None = 999
  };

  //2
  [SerializeField]
  float radius = 1;

  [HideInInspector]
  bool isHeld;

  //3
  [SerializeField]
  Sprite[] directionalSprites;

  //4
  [Serializable]
  public class JoystickMoveEvent : UnityEvent<ActionPadDirection> { }
  public JoystickMoveEvent OnValueChange;

}
```

The initial declaration of this class includes the following variables:

1. `ActionPadDirection` holds all possible directions.
2. `radius` determines the maximum radius for touches to register. `isHeld` sends up a flag when a touch happens.
3. This array of sprites holds the d-pad sprites for the button presses.
4. This `UnityEvent` class declaration processes d-pad movement for the UI's EventSystem.

Add the following method below the variables:

```
  private ActionPadDirection UpdateTouchSprite(Vector2 direction) {
    //1
    float angle = Mathf.Atan2(direction.x, direction.y) * Mathf.Rad2Deg;

    //2
    if (angle < 0) {
      angle += 360;
    }

    //3
    ActionPadDirection currentPadDirection = ActionPadDirection.None;
    if (angle <= 22.5f || angle > 337.5f) {
      currentPadDirection = ActionPadDirection.Up;
    } else if (angle > 22.5 && angle <= 67.5) {
      currentPadDirection = ActionPadDirection.UpRight;
    } else if (angle > 67.5 && angle <= 112.5) {
      currentPadDirection = ActionPadDirection.Right;
    } else if (angle > 112.5 && angle <= 157.5) {
      currentPadDirection = ActionPadDirection.DownRight;
    } else if (angle > 157.5 && angle <= 202.5) {
      currentPadDirection = ActionPadDirection.Down;
    } else if (angle > 202.5 && angle <= 247.5) {
      currentPadDirection = ActionPadDirection.DownLeft;
    } else if (angle > 247.5 && angle <= 292.5) {
      currentPadDirection = ActionPadDirection.Left;
    } else if (angle > 292.5 && angle <= 337.5) {
      currentPadDirection = ActionPadDirection.UpLeft;
    }

    //4
    int index = 0;
    if (currentPadDirection != ActionPadDirection.None) {
      index = (int)currentPadDirection;
    }
    GetComponent<Image>().sprite = directionalSprites[index];
    return currentPadDirection;
  }
```

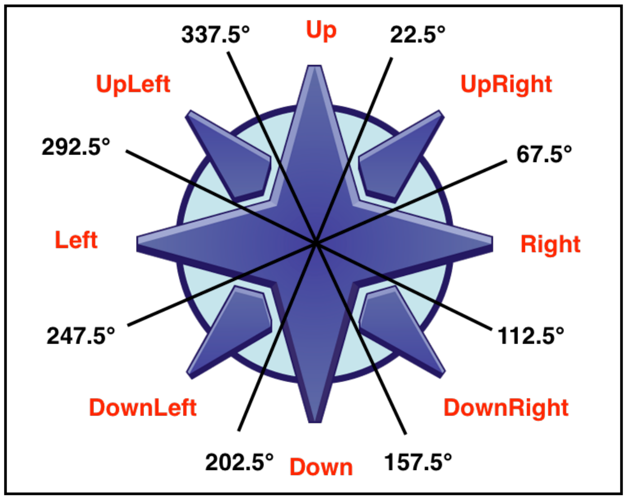
This method updates the d-pad’s look and returns the ActionPadDirection of the `direction` parameter.

1. Calculates the `angle` of the direction vector and converts it to degrees using the built-in `Mathf.Atan` method.

2. Normalizes the `angle` to a value between `0` and `360` by adding 360 when the angle is less than zero.

3. Converts `angle` to an ActionPadDirection after checking if the angle is between a certain range of values. See the diagram below for the values.

4. Updates the **Image** using the `directionalSprites` array and returns `currentPadDirection`.



Next up is adding three methods to the class:

```
public void OnBeginDrag(PointerEventData eventData) {
    if (!IsActive()) {
      return;
    }

    //1
    RectTransform thisRect = transform as RectTransform;
    Vector2 touchDir;
    bool didConvert =
  RectTransformUtility.ScreenPointToLocalPointInRectangle(thisRect,
  eventData.position, eventData.enterEventCamera, out touchDir);

    //2
    if (touchDir.sqrMagnitude > radius * radius) {
      touchDir.Normalize();
      isHeld = true;
      ActionPadDirection currentDirection = UpdateTouchSprite(touchDir);
      OnValueChange.Invoke(currentDirection);
    }
  }

  //3
  public void OnEndDrag(PointerEventData eventData) {
    OnValueChange.Invoke(ActionPadDirection.None);
    GetComponent<Image>().sprite = directionalSprites[0];
  }

  //4
  public void OnDrag(PointerEventData eventData) {
    if (isHeld) {

      RectTransform thisRect = transform as RectTransform;
      Vector2 touchDir;
      RectTransformUtility.ScreenPointToLocalPointInRectangle(thisRect,
  eventData.position, eventData.enterEventCamera, out touchDir);
      touchDir.Normalize();

      //2
      ActionPadDirection currentDirection = UpdateTouchSprite(touchDir);
      OnValueChange.Invoke(currentDirection);
    }
  }
```

These methods allow `ActionDPad` to receive the `OnDrag` events.

1. `OnBeginDrag` converts the touch position to a local `Vector2` using the `RectTransformUtility.ScreenPointToLocalPointInRectangle` method.
2. `If` the magnitude of the calculated vector is less than the radius variable, then the vector is reset so that it's between a value of `0` and `1`. `isHeld` is set true, then this statement updates the d-pad sprite and invokes the `OnValueChange` event.
3. `OnEndDrag` handles updates when the drag action is complete. It calls the `OnValueChange` event with a value of `ActionPadDirection.None`. Then it updates the sprite to the first in the array — the "no-direction-pressed" sprite.

4. `OnDrag` handles drag actions that go across the screen. It checks if the `isHeld` variable is set to true, and if it is, processes the touch like in the `OnBeginDrag` method.

Add the following `IPointerDownHandler` and `IPointerUpHandler` interface methods below the `OnDrag()` method:

```
//1
public void OnPointerDown(PointerEventData eventData) {
  RectTransform thisRect = transform as RectTransform;
  Vector2 touchDir;
  RectTransformUtility.ScreenPointToLocalPointInRectangle(thisRect,
eventData.position, eventData.enterEventCamera, out touchDir);
  touchDir.Normalize();

  //2
  ActionPadDirection currentDirection = UpdateTouchSprite(touchDir);
  OnValueChange.Invoke(currentDirection);
}

//3
public void OnPointerUp(PointerEventData eventData) {
  OnValueChange.Invoke(ActionPadDirection.None);
  GetComponent<Image>().sprite = directionalSprites[0];
}
```

These two interface methods handle what happens when the player taps — not drags — on the d-pad.

1. `OnPointerDown` converts the touch position to the local position on the **ActionDPad** RectTransform.

2. It then updates the d-pad sprite using `UpdateTouchSprite`, and then invokes the `OnValueChange` event method with the calculated `ActionPadDirection`.

3. `OnPointerUp` simply does the same thing as `OnEndDrag`. It returns the d-pad to its neutral state.

With that, you've finished the ActionDPad script. Save it.

Open **InputHandler.cs**. You need to modify the script so it's possible to toggle between on-screen buttons and keyboard input.

Start by adding this `using` directive at the top of the file:

```
using UnityEngine.EventSystems;
```

This will allow the class to use Event classes.

Add the following variables below `maxJumpDuration`:

```
bool didAttack;
public bool useUI = true;
```

`didAttack` will help prevent missing attack chains for the hero, and `useUI` will ultimately determine whether to use the keyboard or UI buttons for input.

Replace the `Update` method with the following:

```
void Update() {
  if (useUI) {
    if (didAttack) {
      didAttack = attack = false;
    } else if (attack) {
      didAttack = true;
    }
  } else {
    horizontal = Input.GetAxisRaw("Horizontal");
    vertical = Input.GetAxisRaw("Vertical");
    attack = Input.GetButtonDown("Attack");


    if(!jump && !isJumping && Input.GetButton("Jump")) {
      jump = true;
      lastJumpTime = Time.time;
      isJumping = true;
    } else if(!Input.GetButton("Jump")) {
      jump = false;
      isJumping = false;
    }
  }


  if(jump && Time.time > lastJumpTime + maxJumpDuration) {
    jump = false;
  }
}
```

You added an `if` statement that checks `useUI's` value. When `useUI` is true, the `ActionDPad` handles movement and on-screen buttons. When `useUI` is false, the script directs the game to use the keyboard.

Add the following button methods to the script:

```
//1
public void DidPressAttack(BaseEventData data) {
  attack = true;
  didAttack = false;
}

//2
public void DidPressJump(BaseEventData data) {
  if (!jump) {
    jump = true;
    lastJumpTime = Time.time;
  }
}

//3
public void DidReleaseJump(BaseEventData data) {
  jump = false;
}
```

As you'd expect, the player triggers the above actions with UI buttons.

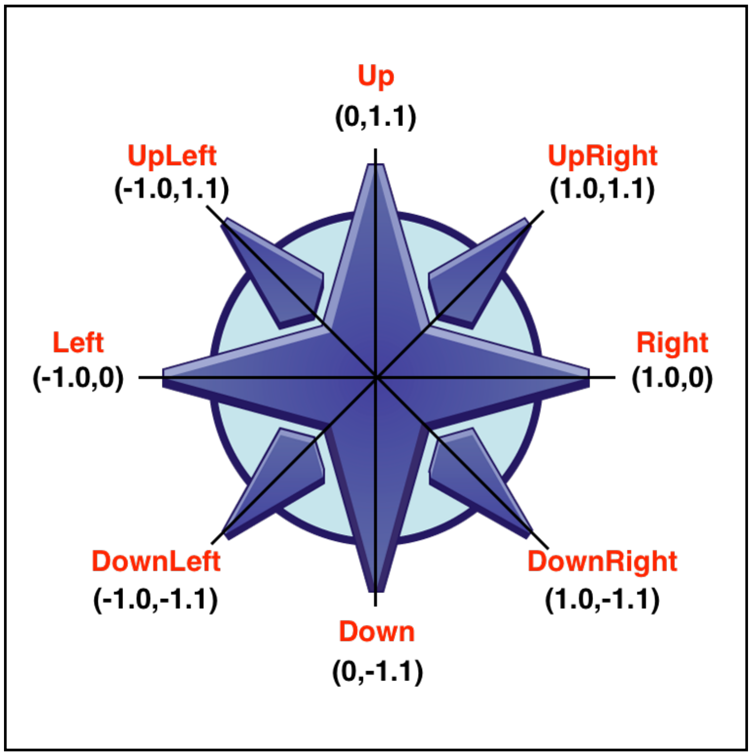
1. `DidPressAttack` sets `attack` to true and `didAttack` to `false`. It is called when the player presses the on-screen attack button.
2. `DidPressJump` triggers the jump when the `InputHandler` isn't in a jump state. It is called when the player presses the jump button.
3. `DidReleaseJump` sets the `jump` variable of InputHandler to `false`. It is called when the user releases the jump button.

Now add these methods:

```
  //1
  public Vector2 VectorForPadDirection(ActionDPad.ActionPadDirection
  padDirection) {
    float maxX = 1.0f;
    float maxY = 1.1f;
    switch (padDirection) {
      case ActionDPad.ActionPadDirection.None:
        return Vector2.zero;
      case ActionDPad.ActionPadDirection.Up:
        return new Vector2(0, maxY);
      case ActionDPad.ActionPadDirection.UpRight:
        return new Vector2(maxX, maxY);
      case ActionDPad.ActionPadDirection.Right:
        return new Vector2(maxX, 0);
      case ActionDPad.ActionPadDirection.DownRight:
        return new Vector2(maxX, -maxY);
      case ActionDPad.ActionPadDirection.Down:
        return new Vector2(0, -maxY);
      case ActionDPad.ActionPadDirection.DownLeft:
        return new Vector2(-maxX, -maxY);
      case ActionDPad.ActionPadDirection.Left:
        return new Vector2(-maxX, 0);
      case ActionDPad.ActionPadDirection.UpLeft:
        return new Vector2(-maxX, maxY);
      default:
        return Vector2.zero;
    }
  }

  //2
  public void OnActionPadChangeDirection(ActionDPad.ActionPadDirection
  direction) {
    Vector2 directionVector = VectorForPadDirection(direction);
    horizontal = directionVector.x;
    vertical = directionVector.y;
  }
```

1. `VectorForPadDirection` is a helper method that converts the ActionPadDirection parameters into a Vector2. The diagram below shows the `ActionPadDirection` and corresponding `directionVector` values.

2. `OnActionPadChangeDirection` gets the corresponding `directionVector` values and sets `horizontal` to the x-value of `directionVector` and `vertical` to the y-value of `directionVector`.



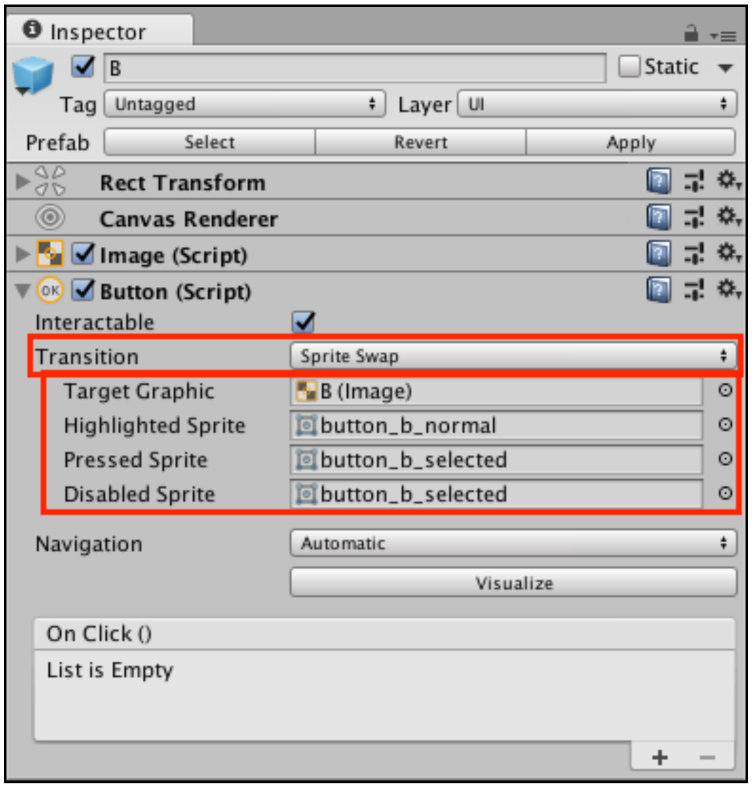
InputHandler is now complete. **Save** the script and return to Unity.

Wiring up the d-pad

Expand **ControlCanvas** and its child **Buttons**. Select **B** and add a **Button** component it.

Set **Transition** to **Sprite Swap**, **button_b_normal** as the value of the **Highlighted Sprite** field, and **button_b_selected** as the values for both **Pressed Sprite** and **Disabled Sprite**.

Now your sprites will change as the player presses the d-pad.

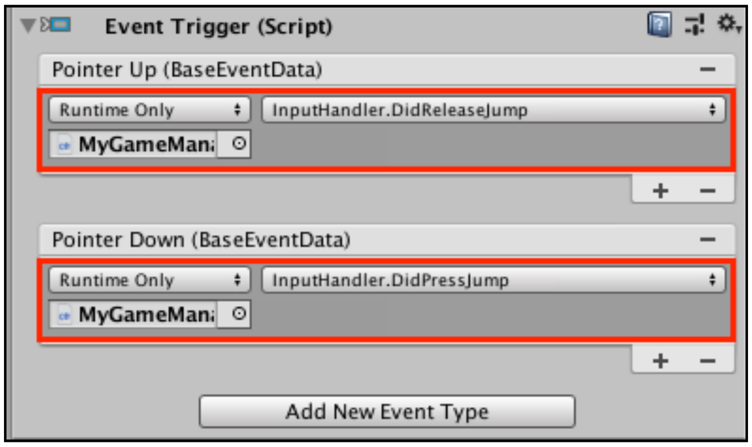


Inspector
B Static
Tag Untagged Layer UI
Prefab Select Revert Apply
Rect Transform
Canvas Renderer
Image (Script)
Button (Script)
Interactable ✓
Transition Sprite Swap
Target Graphic B (Image)
Highlighted Sprite button_b_normal
Pressed Sprite button_b_selected
Disabled Sprite button_b_selected
Navigation Automatic
Visualize
On Click ()
List is Empty

Now to make that B button into the jump control.

Add a new **Event Trigger** component to **B** and add two events: **Pointer Up** and **Pointer Down**. Add a new entry to both then set **MyGameManager** as the value of both **Object** fields.

Select the **InputHandler.DidReleaseJump** method for **Pointer Up** and choose **InputHandler.DidPressJump** for the **Pointer Down** event.

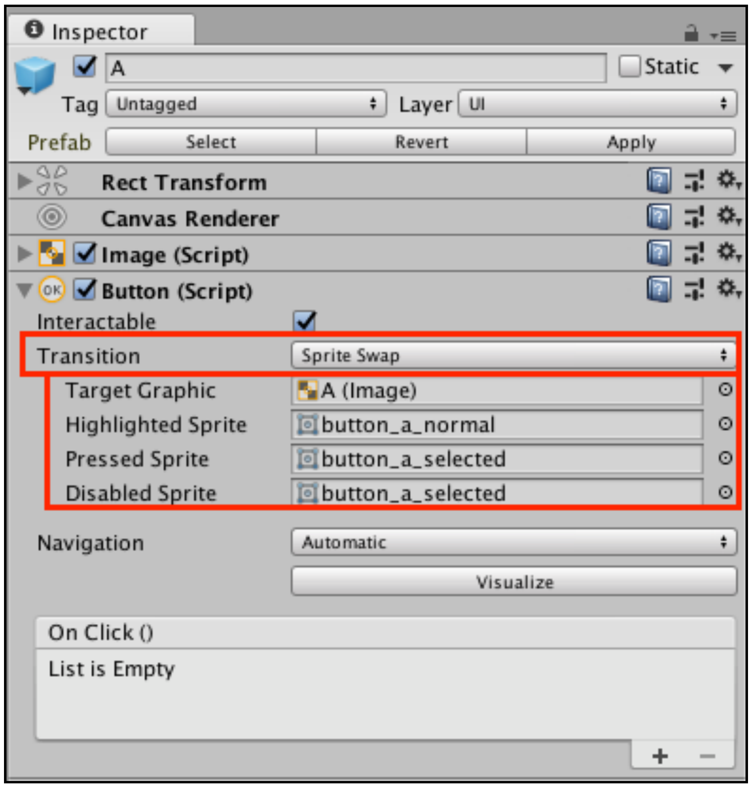


Event Trigger (Script)
Pointer Up (BaseEventData)
Runtime Only InputHandler.DidReleaseJump
MyGameMan:
Pointer Down (BaseEventData)
Runtime Only InputHandler.DidPressJump
MyGameMan:
Add New Event Type

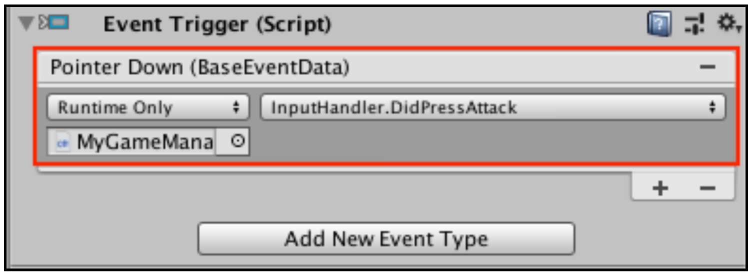
The A button is important — it's the attack button.

Expand **ControlCanvas** and its child **Buttons**. Select **A** and add a **Button** component it.

Set **Transition** to **Sprite Swap**, set **button_a_normal** as the value of **Highlighted Sprite** and **button_a_selected** as the value of both **Pressed Sprite** and **Disabled Sprite**.


Inspector
A
Static
Tag Untagged
Layer UI
Prefab Select Revert Apply
Rect Transform
Canvas Renderer
Image (Script)
Button (Script)
Interactable ✓
Transition: Sprite Swap
Target Graphic: A (Image)
Highlighted Sprite: button_a_normal
Pressed Sprite: button_a_selected
Disabled Sprite: button_a_selected
Navigation: Automatic
Visualize
On Click ()
List is Empty

Add an **Event Trigger** component and add a **Pointer Down** event. Add a new entry, set **MyGameManager** as the event's **Object**, and then apply **InputHandler.DidPressAttack** to the Function field.

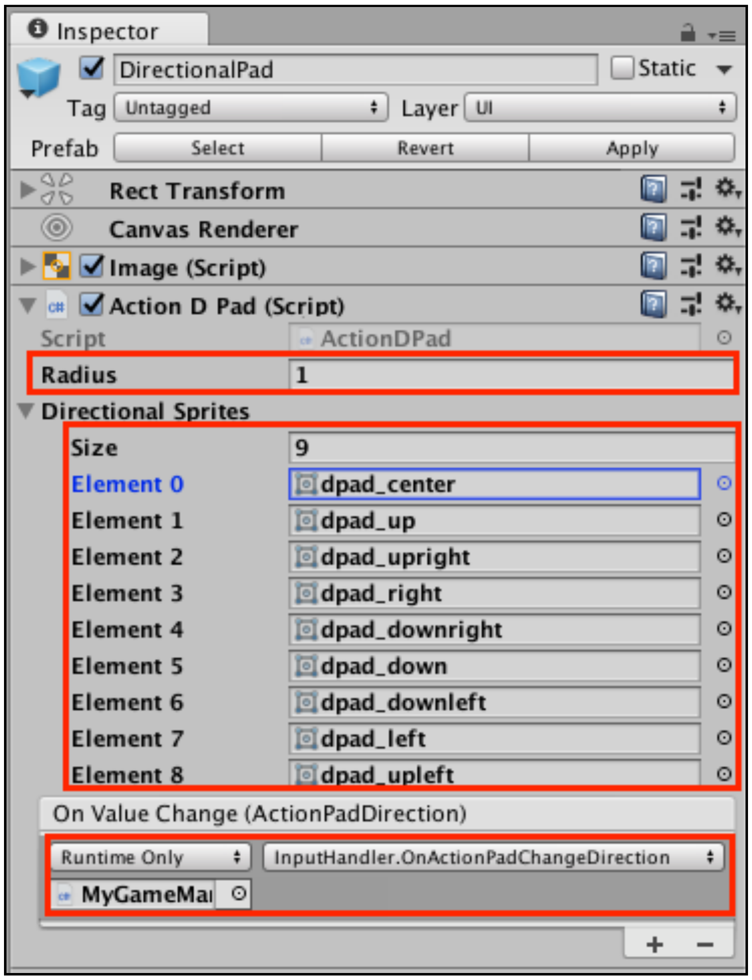

Event Trigger (Script)
Pointer Down (BaseEventData)
Runtime Only
InputHandler.DidPressAttack
MyGameMana
Add New Event Type

That takes care of the attack and jump buttons. Up next is setting references for the directional buttons.

Expand ControlCanvas and select **DirectionPad** in the Hierarchy.

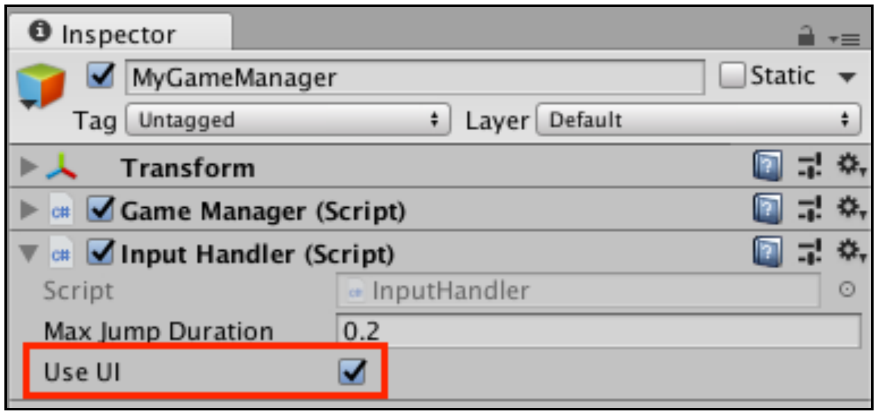
Add an **ActionDPad** component to it and set **Radius** to 5. Add a new event to **OnValueChange** and set **MyGameManager** as the **Object** and set **InputHandler.OnActionPadChangeDirection** as its function.

Set the **Size** of **Directional Sprites** to 9 and drag the sprites from the **Images** \ **JoyPad** folder to the list, setting them in the following order from Element 0 to Element 8: **dpad_center**, **dpad_up**, **dpad_upright**, **dpad_right**, **dpad_downright**, **dpad_down**, **dpad_downleft**, **dpad_left**, and **dpad_upleft**.



DirectionalPad
Tag: Untagged
Layer: UI
Script: ActionDPad
Radius: 1
Directional Sprites
Size: 9
Element 0: dpad_center
Element 1: dpad_up
Element 2: dpad_upright
Element 3: dpad_right
Element 4: dpad_downright
Element 5: dpad_down
Element 6: dpad_downleft
Element 7: dpad_left
Element 8: dpad_upleft
Runtime Only
InputHandler.OnActionPadChangeDirection
MyGameMai

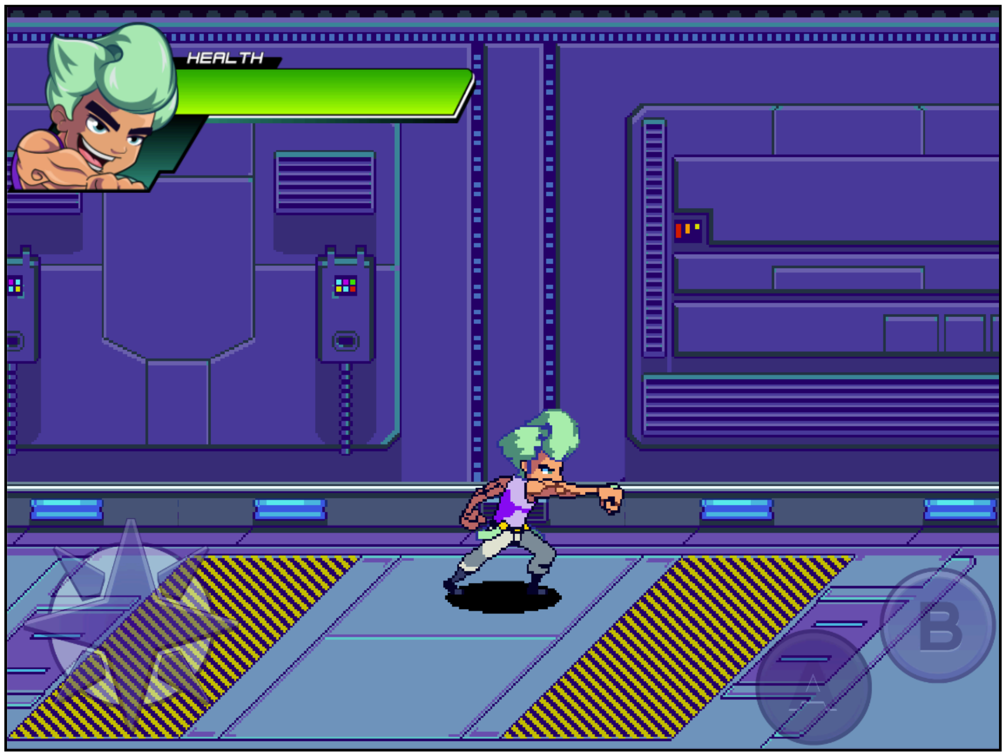
Select ControlCanvas and click **Apply** to save the prefab. Save the scene and run the game. Select **MyGameManager** in the Hierarchy and turn **Use UI** on and off to toggle between control modes.



MyGameManager
Tag: Untagged
Layer: Default
Script: InputHandler
Max Jump Duration: 0.2
Use UI: ✓

By keeping the Boolean variable **Use UI** enabled, you're allowing the **InputHandler** to use the on-screen UI as its input source. Otherwise, it would use the keyboard for input.

Make sure to uncheck this variable if you want to test the keyboard.



Where to go from here?

And there you have it! The UI is pretty much complete — great job setting all that up. You can see the hero and enemies' health and hit damage values, and you've got a nifty visual cue that the battle is done and you can go forward.

For today's on-the-go player who needs a daily dose of robot-bashing pleasure, you implemented on-screen controls.

In this chapter, you:

- Learned all about Unity's UI and its functionality.
- Created a health bar for the hero and his foes.
- Displayed a go sign to prompt the player to continue after they emerged victorious from a battle event.
- Satisfied the player's need for data with an on-screen damage value display.
- Added on-screen controls for the attack, jump and movement buttons.

Congratulations on a job well done! Pompadroid is looking pretty sharp. It's premature to declare "ship it", but it's close for mobile device deployment.

Up next, you'll add power-ups, garbage cans, and big bad bosses to the game!

Chapter 10: Big Bad Boss and Powerups

PompaDroid is looking pretty good so far. In fact, it is nearly complete!

Think about PompaDroid for a moment. Players need to be entertained. Challenged. Pushed to their limits. If they walk through the whole game — or level — without an unfortunate, untimely death, they'll tire of your game and move onto the next.

You, as the game designer, need to give players challenges to overcome. Double bonus points if you can induce a little ire when they lose! :]

In this chapter, you'll learn how to:

- Add a new, super-powerful antagonist, in this case, the mighty boss.
- Implement powerups, in this case, the POW300, a set of gloves that empower the hero to defeat the boss.
- Place powerups around the map.

By now, you know your way around the game and Unity, and you won't be doing anything "new", so this chapter should go pretty quickly!

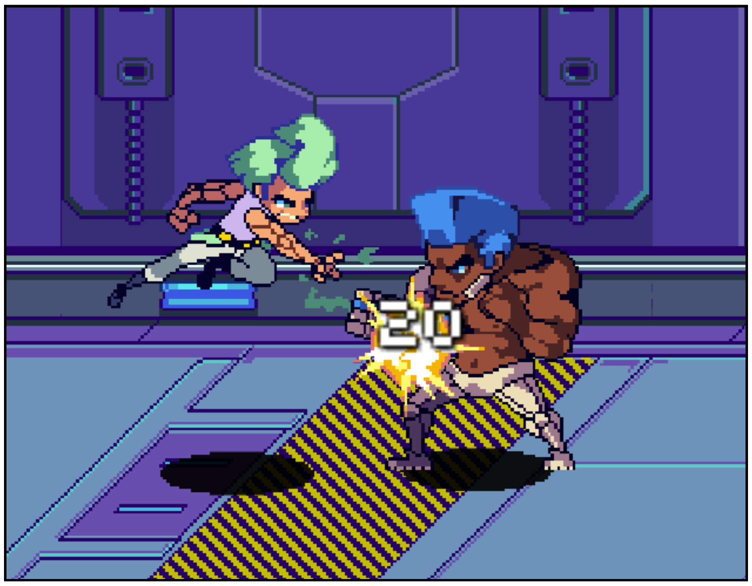
The big bad boss

Meet the new boss. He's larger than your standard-issue droids, as bosses often are (at least in their own minds), and he'll have a wicked mechanical arm that delivers maximum damage to our pompadoured protagonist. Ouch!

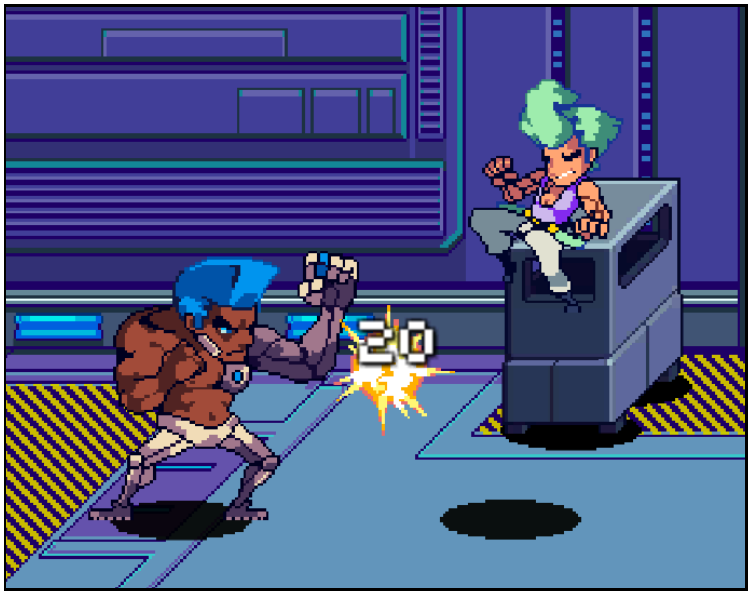
He's also got a pretty sweet hairdo. Makes you wonder if he's related to the hero, right?

The boss will behave similarly to the robots: he'll walk, attack, chase, and relentlessly pursue the hero. Yet he will be different.

To make things hard on the player, he won't feel pain or fall down. He'll shrug off all attacks he receives like a…boss!

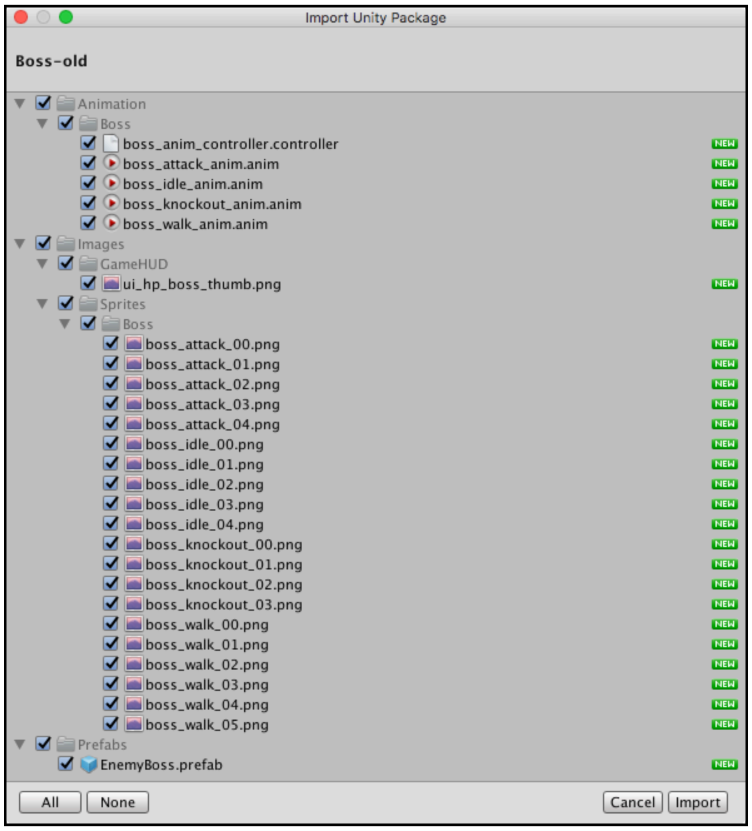


His attacks will impart substantially more damage. Each blow from his mechanical arm will knock the hero down. That mechanical arm is a force to be reckoned with!



Build the boss

Import **Boss.unitypackage** from the **UnityPackages** folder. Note that it contains all the assets you need, including a semi-complete EnemyBoss prefab.



```
Import Unity Package

Boss-old

▼ ✓ Animation
  ▼ ✓ Boss
      ✓ boss_anim_controller.controller        NEW
      ✓ boss_attack_anim.anim                  NEW
      ✓ boss_idle_anim.anim                    NEW
      ✓ boss_knockout_anim.anim                NEW
      ✓ boss_walk_anim.anim                    NEW
▼ ✓ Images
  ▼ ✓ GameHUD
      ✓ ui_hp_boss_thumb.png                   NEW
  ▼ ✓ Sprites
    ▼ ✓ Boss
        ✓ boss_attack_00.png                   NEW
        ✓ boss_attack_01.png                   NEW
        ✓ boss_attack_02.png                   NEW
        ✓ boss_attack_03.png                   NEW
        ✓ boss_attack_04.png                   NEW
        ✓ boss_idle_00.png                     NEW
        ✓ boss_idle_01.png                     NEW
        ✓ boss_idle_02.png                     NEW
        ✓ boss_idle_03.png                     NEW
        ✓ boss_idle_04.png                     NEW
        ✓ boss_knockout_00.png                 NEW
        ✓ boss_knockout_01.png                 NEW
        ✓ boss_knockout_02.png                 NEW
        ✓ boss_knockout_03.png                 NEW
        ✓ boss_walk_00.png                     NEW
        ✓ boss_walk_01.png                     NEW
        ✓ boss_walk_02.png                     NEW
        ✓ boss_walk_03.png                     NEW
        ✓ boss_walk_04.png                     NEW
        ✓ boss_walk_05.png                     NEW
▼ ✓ Prefabs
    ✓ EnemyBoss.prefab                         NEW
```

As it is for the hero and droids, the Actor class will handle damage for the boss. Modifying it is the first order of business.

Open **Actor.cs** and add the following variable below the class declaration:

```
protected bool canFlinch = true;
```

This Boolean determines whether the corresponding Actor flinches when it takes damage. It defaults to `true` for all instances of the Actor class.

Find the `TakeDamage` method and replace the following lines:

```
} else {
  baseAnim.SetTrigger("IsHurt");
}
```

With these lines:

```
} else if (canFlinch) {
  baseAnim.SetTrigger("IsHurt");
}
```

You’re adding a condition so that the hurt animation only plays when `canFlinch` is set to `true`.

The Actor script is complete for now. Save it, create a new **C# script** in the **Scripts** folder and name it **Boss.cs**.

Open it and replace its contents with the following:

```
using System.Collections;
using UnityEngine;
//1
public class Boss : Enemy {

  //2
  protected override void Start() {
    base.Start();
    canFlinch = false;
  }

  //3
  public override void TakeDamage(float value, Vector3 hitVector, bool
knockdown = false) {
    base.TakeDamage(value,hitVector,false);
  }
}
```

Here you’re setting up the `Boss` class.

1. `Boss`, just like `Robot`, is based on the `Enemy` class so it inherits the ability to use the `EnemyAI` script.

2. `Start` overrides the `Start` method from the base class by setting `canFlinch` to `false`.
3. `TakeDamage` overrides the base class while setting `knockdown` to `false`. It prevents the hero from knocking down the boss.

That should frustrate the player! **Save** the script and open **GameManager.cs**. Next up is creating an instance of the boss when the level demands it.

Add the following below `public GameObject robotPrefab`:

```
public GameObject bossPrefab;
```

This variable is a reference to the boss prefab.

Next, in the `SpawnEnemy` method of the same class, replace the following line:

```
GameObject enemyObj = Instantiate(robotPrefab);
```

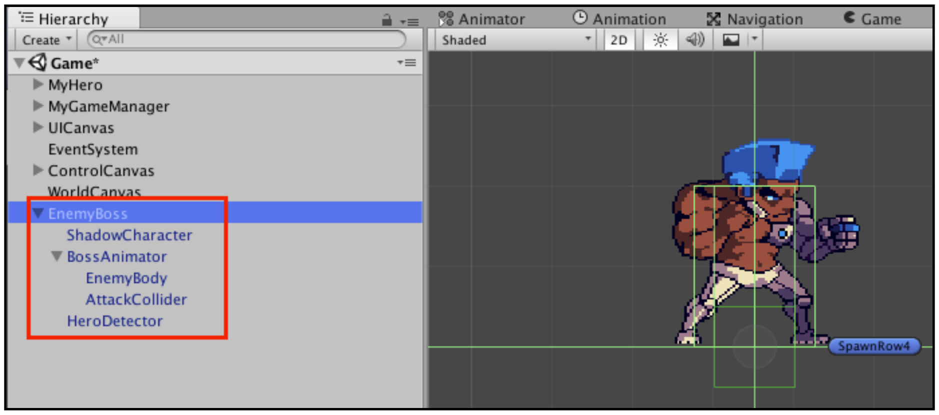
With this conditional:

```
GameObject enemyObj;
if (data.type == EnemyType.Boss) {
  enemyObj = Instantiate(bossPrefab);
} else {
  enemyObj = Instantiate(robotPrefab);
}
```

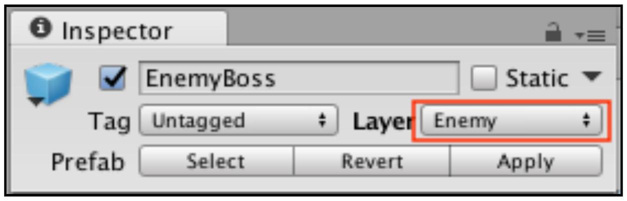
This `if` instantiates a boss when the `data` parameter requires an `EnemyType.Boss`. Otherwise, it generates a normal droid.

Save `GameManager.cs` and return to the editor to assemble the boss prefab.

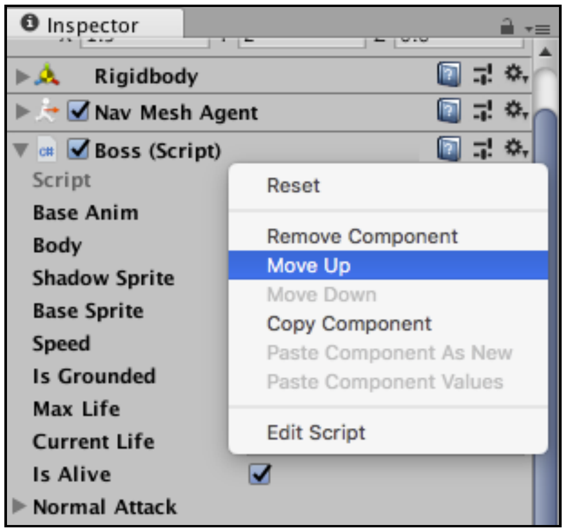
Open the Game scene and drag the **EnemyBoss** prefab from the **Assets** / **Prefabs** folder into the scene.



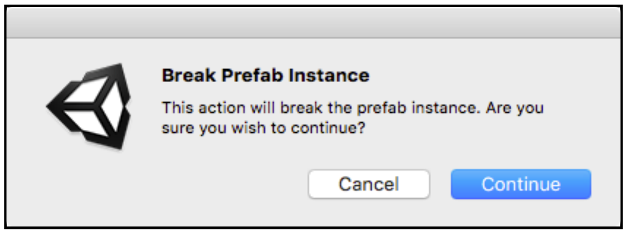
Change the layer of **EnemyBoss** to **Enemy**. Select **Yes, change children** when prompted.



Add a **Boss** component to **EnemyBoss**. Move it up in the Inspector by right-clicking and selecting **Move Up**. For convenience, make it the first element below the Transform of EnemyBoss.



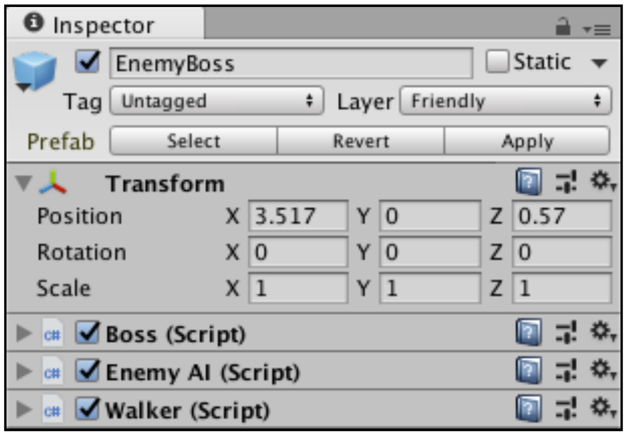
Rearranging components will trigger a prompt about breaking the prefab instance. Select **Continue** — you'll save the prefab later.


Break Prefab Instance

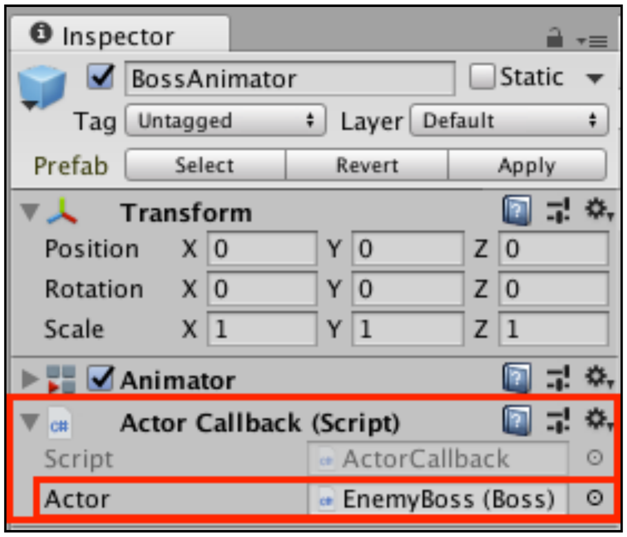
This action will break the prefab instance. Are you sure you wish to continue?

Note: In *most* cases, and PompaDroid is no exception, rearranging components has no bearing on how the code runs.

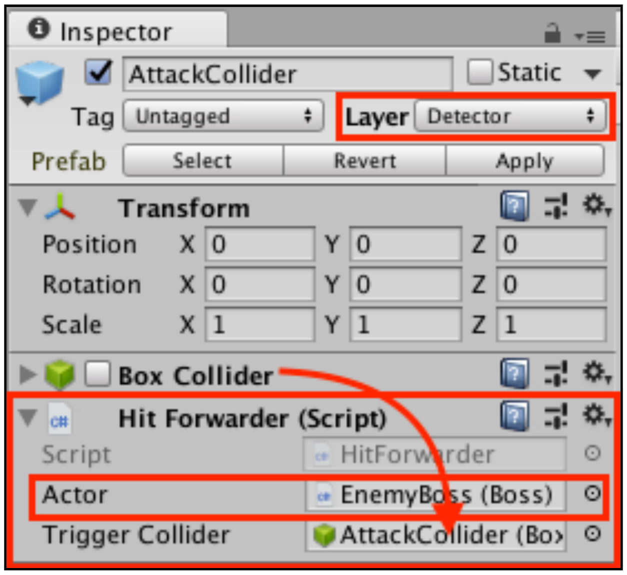
Add an **EnemyAI** and a **Walker** component to **EnemyBoss**. Move the components to the top of the GameObject.


Inspector
EnemyBoss Static
Tag Untagged Layer Friendly
Prefab Select Revert Apply
Transform
Position X 3.517 Y 0 Z 0.57
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Boss (Script)
Enemy AI (Script)
Walker (Script)

Select the **BossAnimator** child of EnemyBoss in the Hierarchy, and add an **ActorCallback** component to it. Point its **Actor** field to **EnemyBoss**.

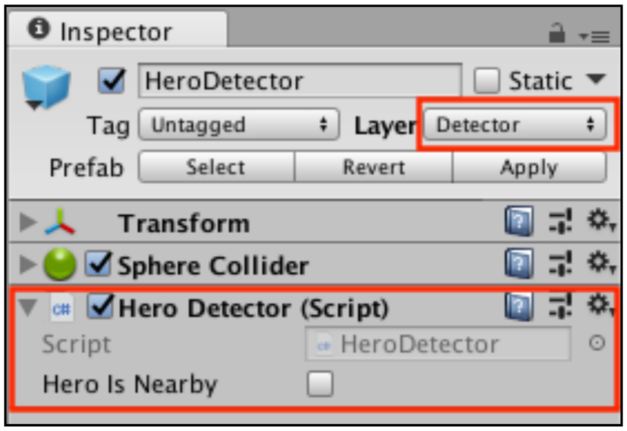

Inspector
BossAnimator Static
Tag Untagged Layer Default
Prefab Select Revert Apply
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Animator
Actor Callback (Script)
Script ActorCallback
Actor EnemyBoss (Boss)

Select the **AttackCollider** child of BossAnimator, and add a **HitForwarder** component. Point its **Actor** field to **EnemyBoss** and its **Trigger Collider** to its **Box Collider**. Set its layer to **Detector**.



AttackCollider
Static
Tag Untagged
Layer Detector
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Box Collider
Hit Forwarder (Script)
Script HitForwarder
Actor EnemyBoss (Boss)
Trigger Collider AttackCollider (Box

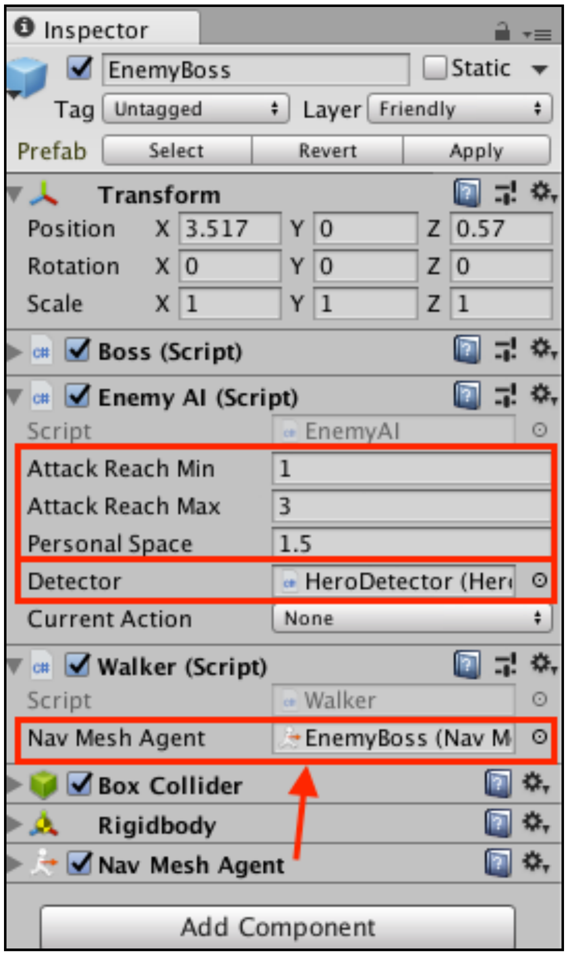
Select the **HeroDetector** child of EnemyBoss and add a **HeroDetector** component to it. Set its layer to **Detector** as well.



HeroDetector
Static
Tag Untagged
Layer Detector
Transform
Sphere Collider
Hero Detector (Script)
Script HeroDetector
Hero Is Nearby

Next, select **EnemyBoss** in the Hierarchy and set the **Walker** component's **Nav Mesh Agent** to the **Nav Mesh Agent** component on the same GameObject.

Point the **Hero Detector** field of **Enemy AI** to the **HeroDetector** child of EnemyBoss. Set **Attack Reach Min** to `1`, **Attack Reach Max** to `3`, and **Personal Space** to `1.5` in the **Enemy AI** component.



EnemyBoss Static
Tag Untagged Layer Friendly

Transform
Position X 3.517 Y 0 Z 0.57
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Boss (Script)

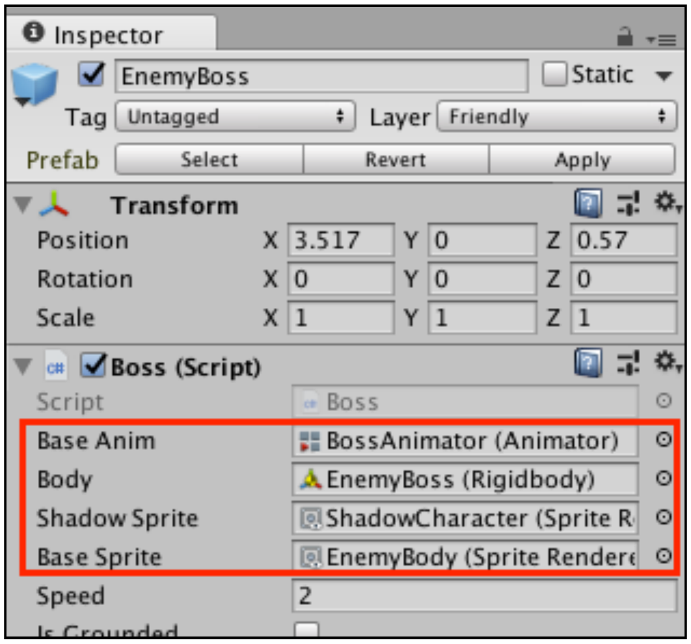
Enemy AI (Script)
Script EnemyAI
Attack Reach Min 1
Attack Reach Max 3
Personal Space 1.5
Detector HeroDetector (Her
Current Action None

Walker (Script)
Script Walker
Nav Mesh Agent EnemyBoss (Nav M

Box Collider
Rigidbody
Nav Mesh Agent

Set the **Base Anim** variable to the **BossAnimator** Animator.

Adjust **Body** to the **RigidBody** on the same GameObject. Set **Shadow Sprite** to **ShadowCharacter** and modify **BaseSprite** to **EnemyBody**.

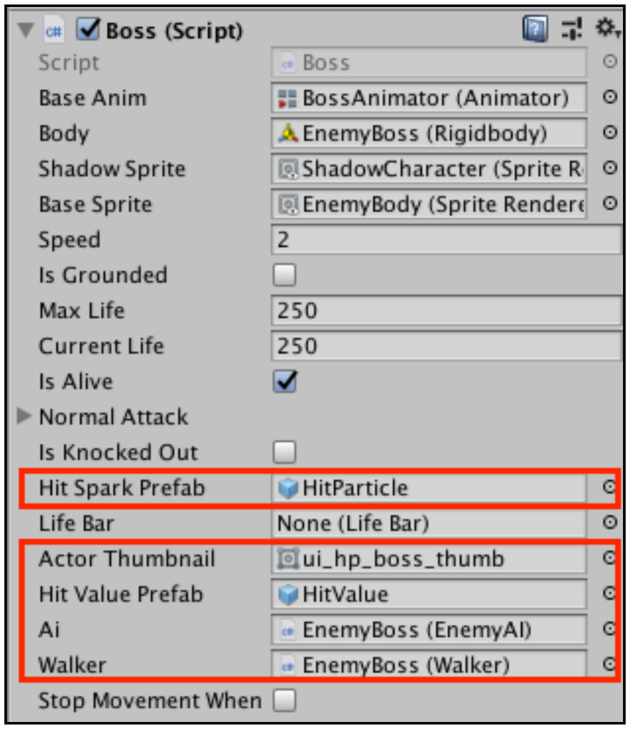


```
Inspector
EnemyBoss                    Static
Tag Untagged      Layer Friendly
Prefab  Select  Revert  Apply
Transform
Position   X 3.517  Y 0  Z 0.57
Rotation   X 0      Y 0  Z 0
Scale      X 1      Y 1  Z 1
Boss (Script)
Script          Boss
Base Anim       BossAnimator (Animator)
Body            EnemyBoss (Rigidbody)
Shadow Sprite   ShadowCharacter (Sprite R
Base Sprite     EnemyBody (Sprite Rendere
Speed           2
Is Grounded
```

Drag the **HitParticle** prefab located in **Assets / Prefabs** to the **Hit Spark Prefab** field.

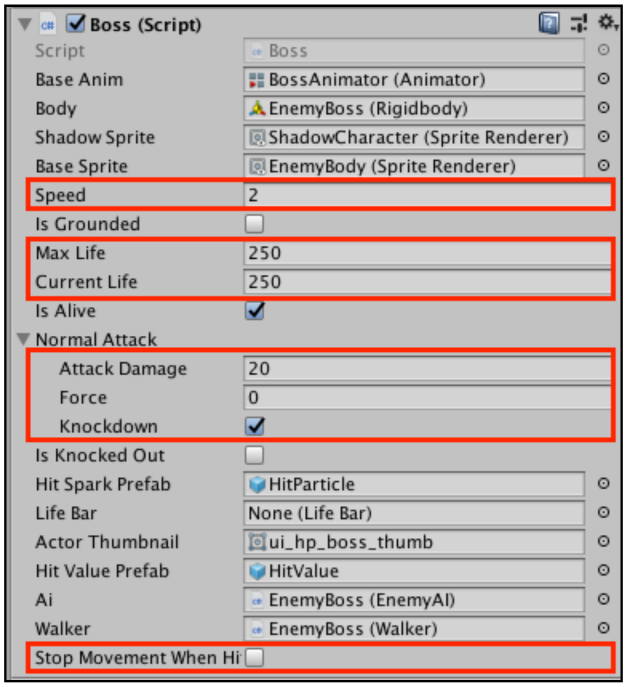
Set **Actor Thumbnail** to the **ui_hp_boss_thumb** sprite. Set **Hit Value Prefab** to **HitValue** located in **Assets / Prefabs**.

Point **AI** to the **EnemyAI** component in **EnemyBoss** and adjust **Walker** to the **Walker** component in the same GameObject.



```
Boss (Script)
Script              Boss
Base Anim           BossAnimator (Animator)
Body                EnemyBoss (Rigidbody)
Shadow Sprite       ShadowCharacter (Sprite R
Base Sprite         EnemyBody (Sprite Rendere
Speed               2
Is Grounded         ☐
Max Life            250
Current Life        250
Is Alive            ✓
Normal Attack
Is Knocked Out      ☐
Hit Spark Prefab    HitParticle
Life Bar            None (Life Bar)
Actor Thumbnail     ui_hp_boss_thumb
Hit Value Prefab    HitValue
Ai                  EnemyBoss (EnemyAI)
Walker              EnemyBoss (Walker)
Stop Movement When  ☐
```

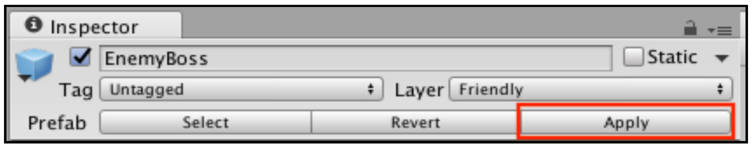
Set **Speed** to `2` and adjust both **MaxLife** and **CurrentLife** to `250`. Set **Normal Attack's Attack Damage** to `20`, check **Knockdown** and uncheck **Stop Movement When Hit**.



Boss (Script)	✓
Script	Boss
Base Anim	BossAnimator (Animator)
Body	EnemyBoss (Rigidbody)
Shadow Sprite	ShadowCharacter (Sprite Renderer)
Base Sprite	EnemyBody (Sprite Renderer)
Speed	2
Is Grounded	☐
Max Life	250
Current Life	250
Is Alive	✓
Normal Attack	
Attack Damage	20
Force	0
Knockdown	✓
Is Knocked Out	☐
Hit Spark Prefab	HitParticle
Life Bar	None (Life Bar)
Actor Thumbnail	ui_hp_boss_thumb
Hit Value Prefab	HitValue
Ai	EnemyBoss (EnemyAI)
Walker	EnemyBoss (Walker)
Stop Movement When Hi	☐

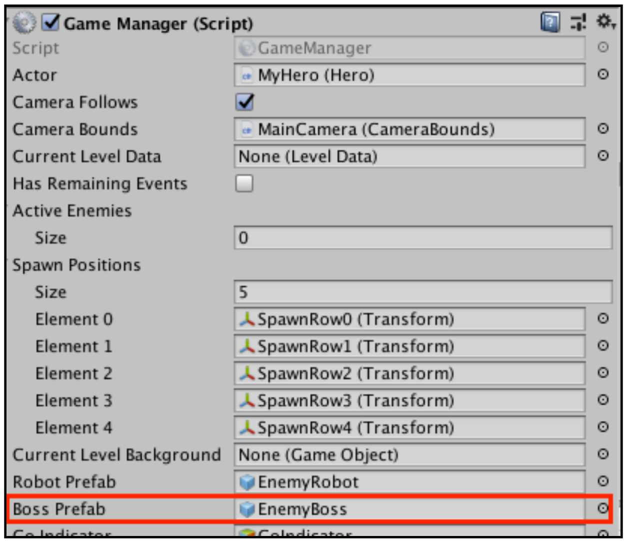
You just finished the boss. Like a boss, no less.

Select **Apply** at the top-right of the Inspector to save the prefab. Delete the **EnemyBoss** GameObject in the Hierarchy.



✓ EnemyBoss	☐ Static
Tag: Untagged	Layer: Friendly
Prefab	Select / Revert / Apply

Select **MyGameManager** in the Hierarchy and set the **Boss Prefab** variable as **EnemyBoss** in the **Prefabs** folder.

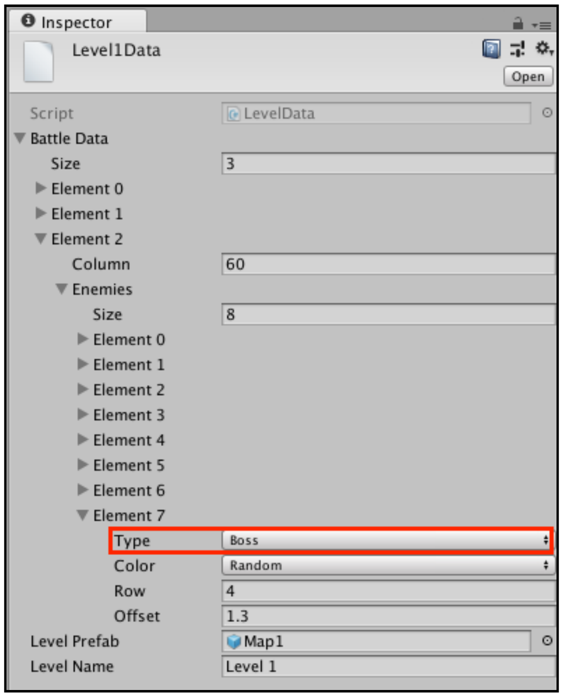


Game Manager (Script)

Script	GameManager
Actor	MyHero (Hero)
Camera Follows	✓
Camera Bounds	MainCamera (CameraBounds)
Current Level Data	None (Level Data)
Has Remaining Events	☐
Active Enemies	
Size	0
Spawn Positions	
Size	5
Element 0	SpawnRow0 (Transform)
Element 1	SpawnRow1 (Transform)
Element 2	SpawnRow2 (Transform)
Element 3	SpawnRow3 (Transform)
Element 4	SpawnRow4 (Transform)
Current Level Background	None (Game Object)
Robot Prefab	EnemyRobot
Boss Prefab	EnemyBoss

Select the **Level1Data** asset in the **LevelData** folder. In the Inspector, expand **Battle Data** and its **Enemies** list.

Duplicate the last item in **Enemies** by right-clicking then selecting **Duplicate Array Element**. Set its **Type** to **Boss** and leave the other settings as-is.



Inspector — Level1Data

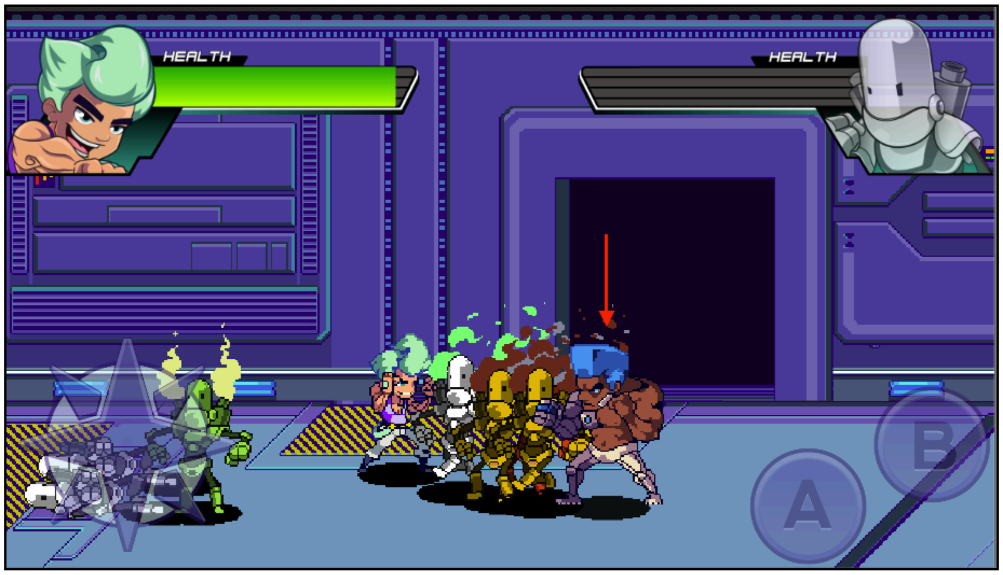
Script	LevelData
Battle Data	
Size	3
Element 0	
Element 1	
Element 2	
Column	60
Enemies	
Size	8
Element 0	
Element 1	
Element 2	
Element 3	
Element 4	
Element 5	
Element 6	
Element 7	
Type	Boss
Color	Random
Row	4
Offset	1.3
Level Prefab	Map1
Level Name	Level 1

Repeat those steps to add a boss to each level. Maybe two if you're feeling punchy.

Save the scene and the project.

And that, my dear reader, is how you spawn a boss the end of the first level.

Thrash some droids. Destroy them! When you reach the end, you'll meet the new boss.



Great job! The boss spawns, ready to fight! But wow. He's badass! Poor little pompadoured protagonist doesn't stand a chance.

Powerups: the boss' worst nightmare

But wait. This is *your* game. You're the original boss of this game. Look at your fingers for a moment. Look at them! Those very fingers control life and death in the world of PompaDroid. You have the power to give the hero something to work with.

Besides, you can't just implement a powerful boss and not give the player some way to defeat it. Players like to be challenged, but they deplore impossible challenges. They have to have at least some hope of prevailing in order for a boss battle to be "fun".

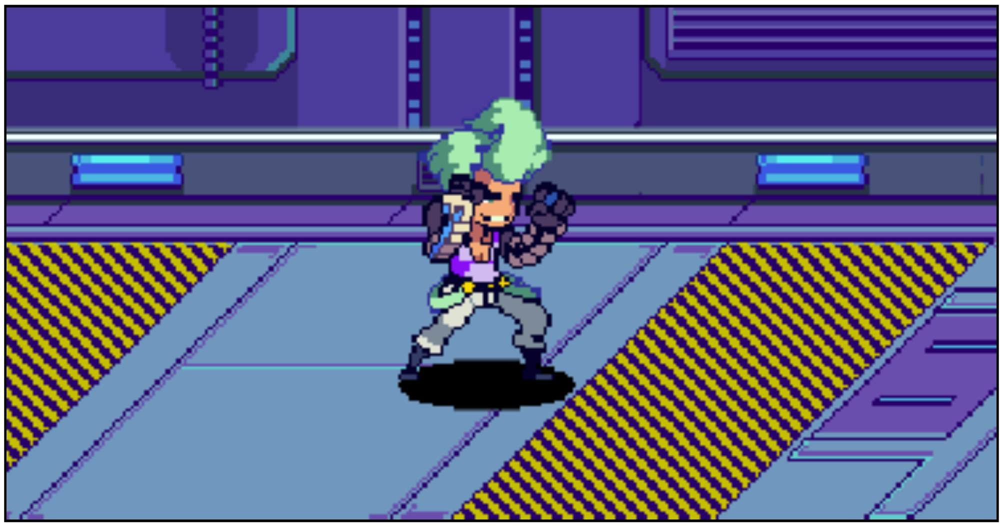
Players love stumbling across loot and powerups they can use to crush enemies in a more grandiose fashion. Powerups tend to have a few qualities: enhance the player's powers, limited duration, and some kind of trade off in abilities.

Gloves are a hero's best friend

I've set you up to implement the **Punch-a-Tron Operational Work Gloves 300**, or **POW300** for short!

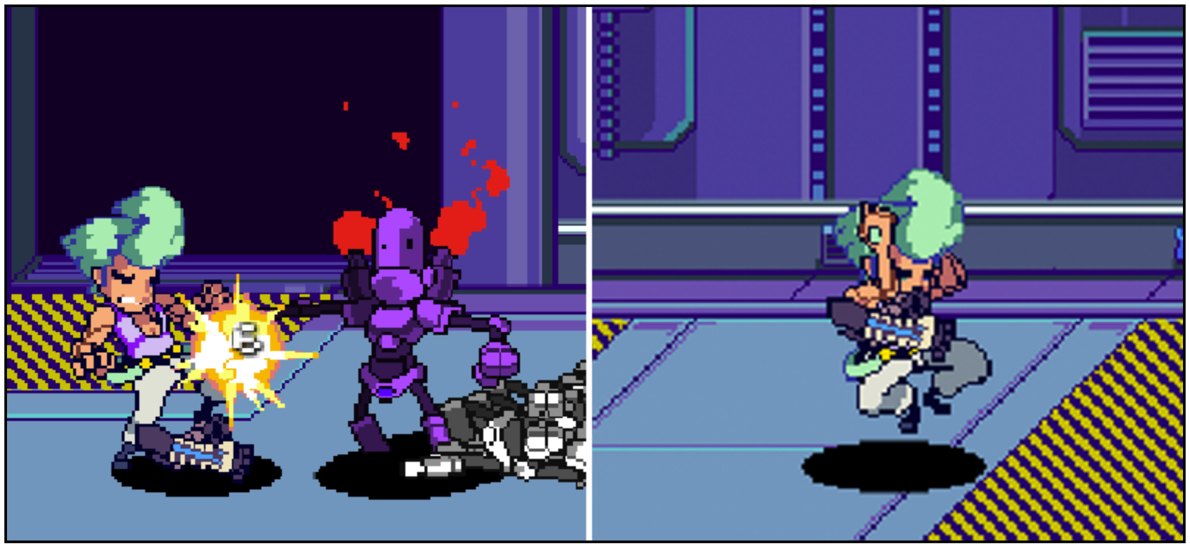
The POW300 grants super-human strength to the hero, but he can't run or jump while wearing them. They are too bulky!

Another downside? The gloves are made cheaply and simply fall apart after a few hits.



The last caveat is that they fall off when the hero takes damage.

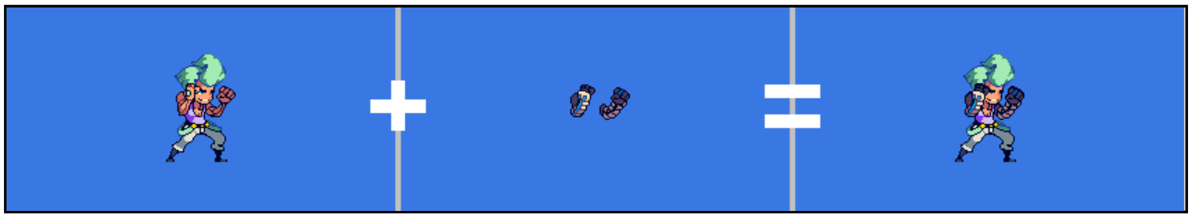
If the player wants to take them off, they can press the jump button.



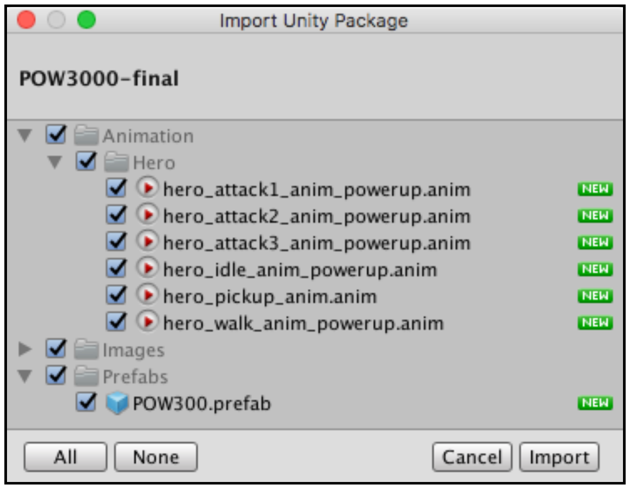
In spite its limitations, this powerup gives the hero a decisive advantage over enemies. You’ll stash POW300s in garbage cans to make them available to the player.

Setting up the gloves

When you equip them, the gloves sprite draw on top of the hero sprite. This is similar to how the robot sprite works: multiple layers of sprites create the illusion of a single, solid asset.



Import the **POW300.unitypackage** from the **UnityPackages** folder. In here, you'll find new and updated animation clips suffixed with "_powerup", and an incomplete POW300 prefab.



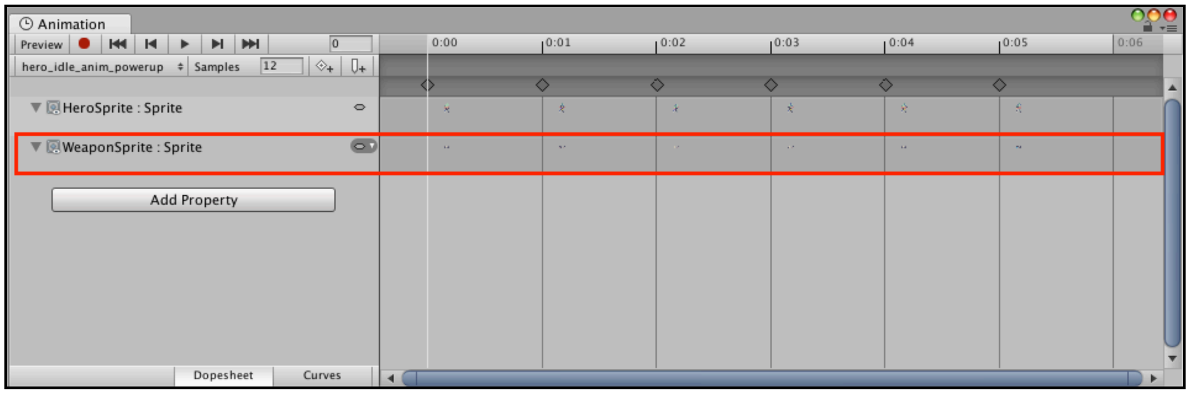
Import Unity Package

POW3000-final

- Animation
  - Hero
    - hero_attack1_anim_powerup.anim NEW
    - hero_attack2_anim_powerup.anim NEW
    - hero_attack3_anim_powerup.anim NEW
    - hero_idle_anim_powerup.anim NEW
    - hero_pickup_anim.anim NEW
    - hero_walk_anim_powerup.anim NEW
- Images
- Prefabs
  - POW300.prefab NEW

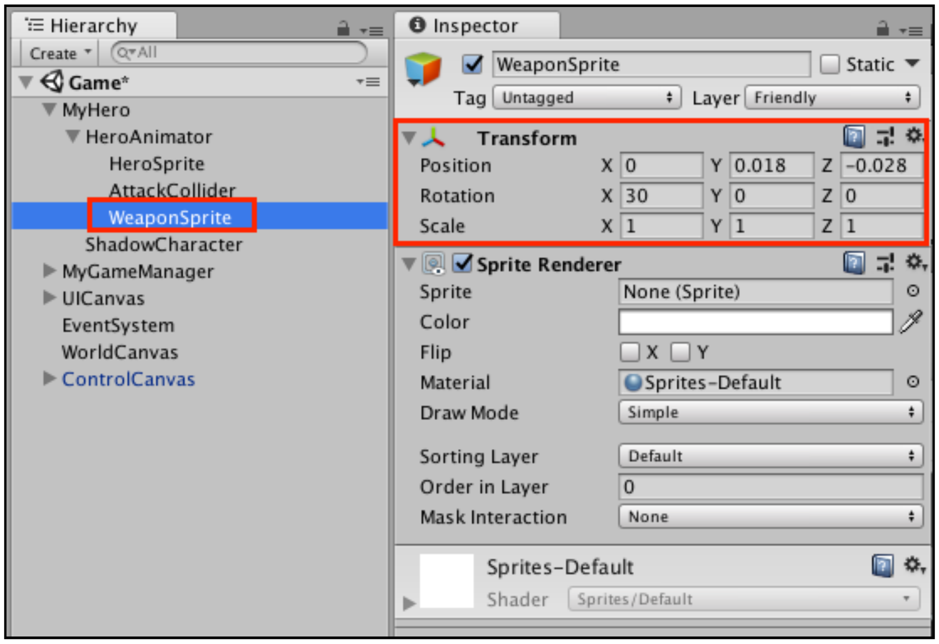
All | None | Cancel | Import

The animation clips reference a **WeaponSprite SpriteRenderer** parameter. This SpriteRenderer is responsible for rendering the POW300 assets on top of the hero's sprite.



You'll need a new SpriteRenderer before you can add the animator for the POW300 gloves.

In the Hierarchy, select **HeroAnimator** under MyHero, and add a new **2D Object \ Sprite** to it. Name it **WeaponSprite** then set **Transform** position and rotation to `(X:0, Y: 0.018, Z:-0.028)` and `(X:30, Y:0, Z:0)`, respectively.



WeaponSprite ✓ — Static ☐
Tag: Untagged — Layer: Friendly

Transform

	X	Y	Z
Position	0	0.018	-0.028
Rotation	30	0	0
Scale	1	1	1

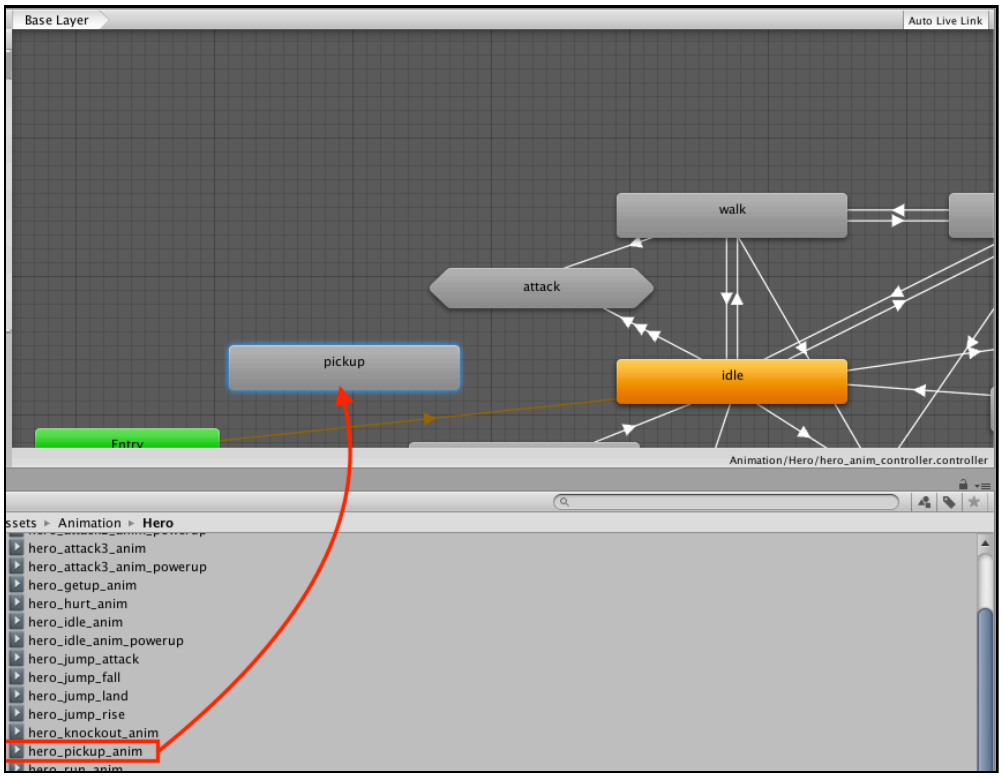
Sprite Renderer ✓

Sprite	None (Sprite)
Color	
Flip	☐ X ☐ Y
Material	Sprites-Default
Draw Mode	Simple
Sorting Layer	Default
Order in Layer	0
Mask Interaction	None

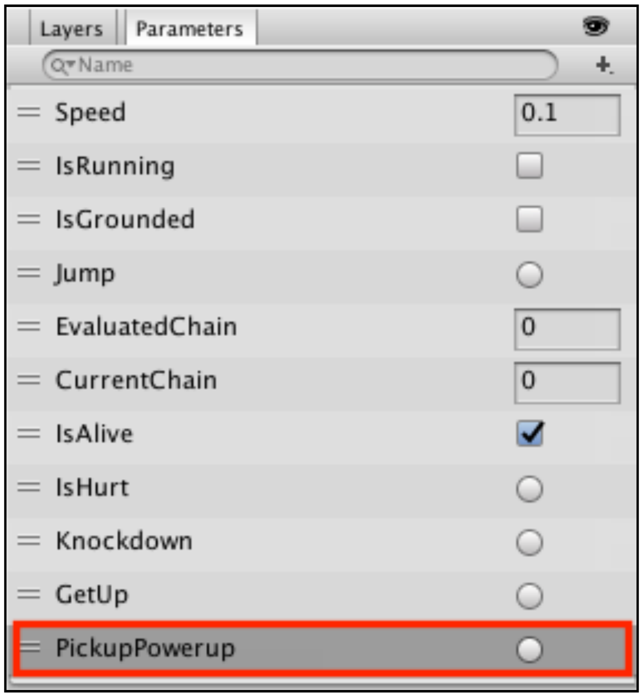
Sprites-Default
Shader: Sprites/Default

Select the **HeroAnimator** GameObject and open the **Animator Window**. You'll need to add a new pickup state in the statemachine.

Drag the **hero_pickup_anim** clip from **Animation \ Hero** into the Animator view and rename it **pickup**.

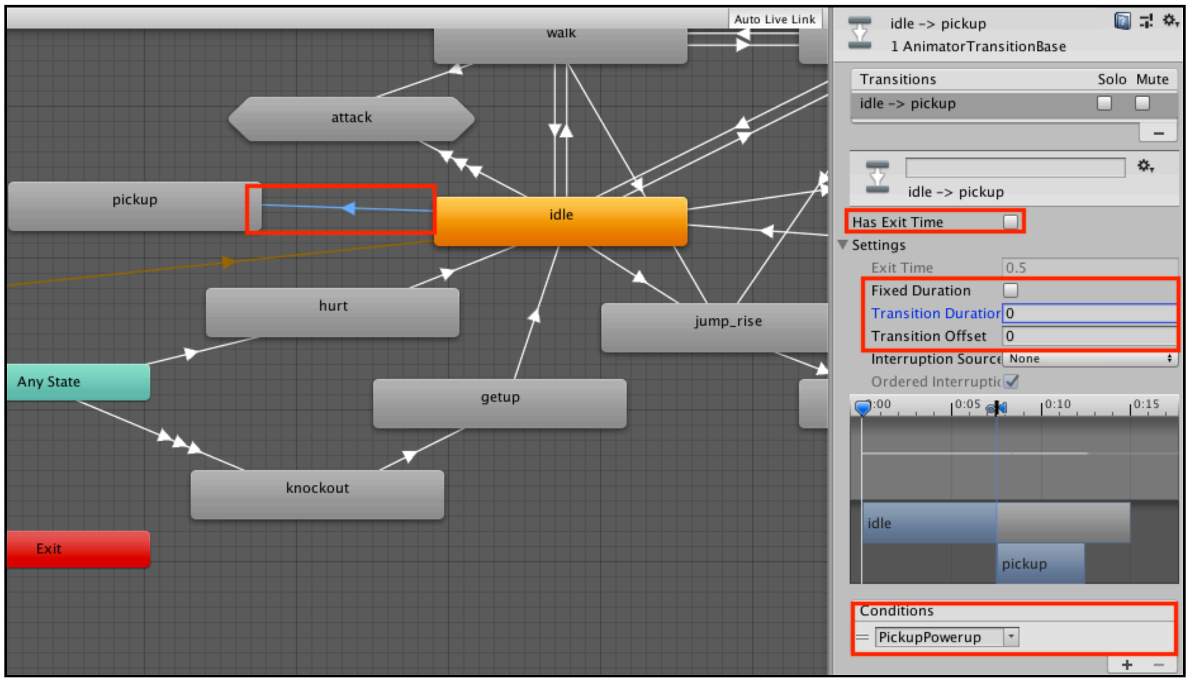


Add a new **Trigger Event** parameter named **PickupPowerup**.



Speed	0.1
IsRunning	☐
IsGrounded	☐
Jump	○
EvaluatedChain	0
CurrentChain	0
IsAlive	✓
IsHurt	○
Knockdown	○
GetUp	○
PickupPowerup	○

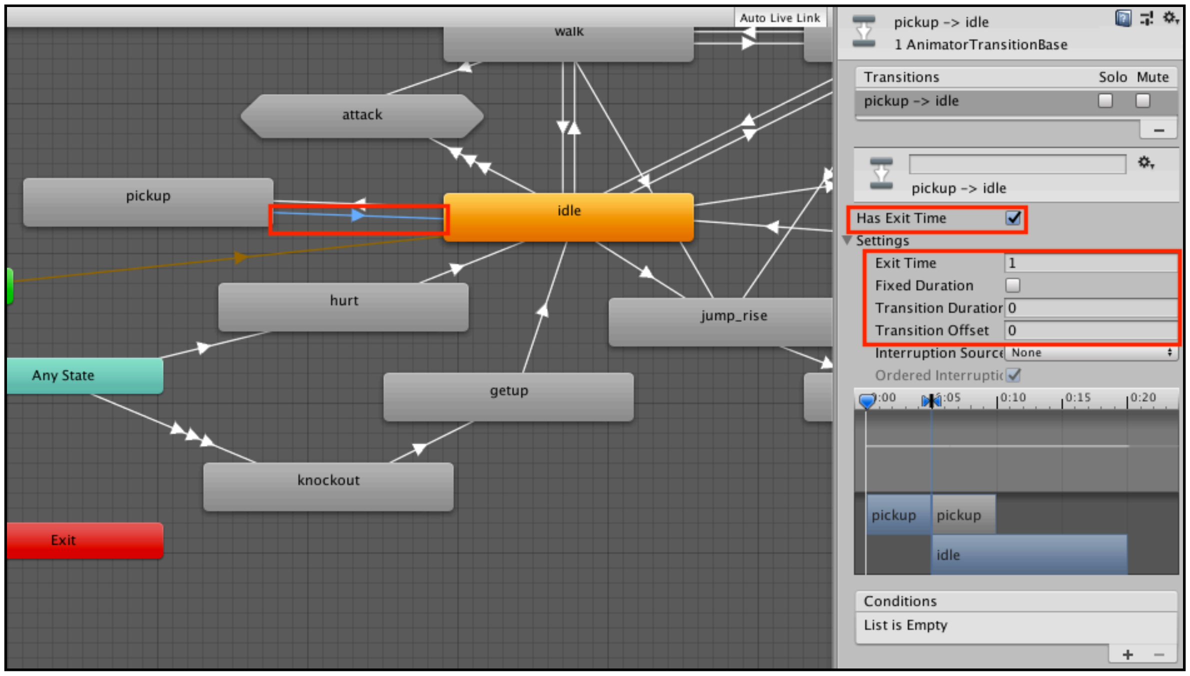
Add a transition from **idle** to **pickup**. Uncheck **Has Exit Time** and **Fixed Duration**. Set **Transition Duration** to `0`, and its condition as **PickupPowerup**.



idle -> pickup
1 AnimatorTransitionBase
Transitions
idle -> pickup
idle -> pickup
Has Exit Time ☐
Settings
Exit Time 0.5
Fixed Duration ☐
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓
Conditions
PickupPowerup

Add a transition from **pickup** to **idle**. Keep **Has Exit Time** checked. Set **Exit Time** to `1`. Uncheck **Fixed Duration** and set **Transition Duration** to `0`.

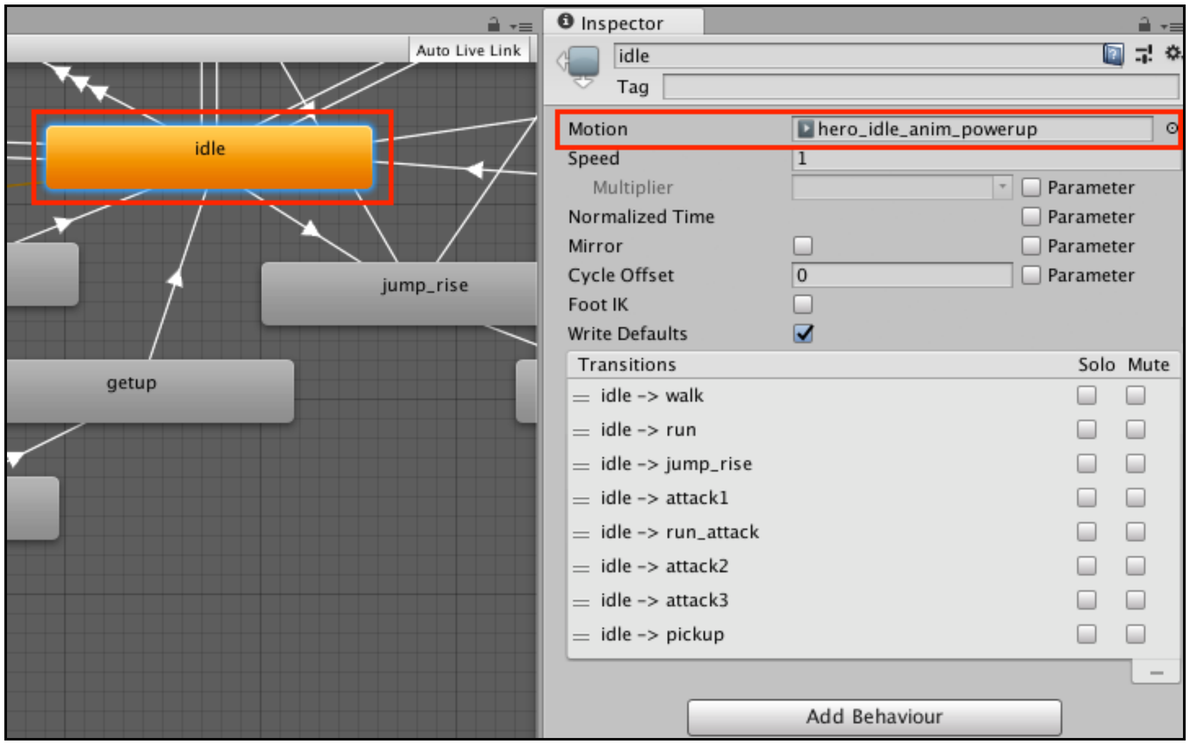
This will transition the state machine to an idle state when the pickup animation completes.



pickup -> idle
1 AnimatorTransitionBase

Has Exit Time ✓
Settings
Exit Time 1
Fixed Duration
Transition Duration 0
Transition Offset 0
Interruption Source None
Ordered Interruption ✓

Replace the animation clip of the idle animation state. Select the **idle** state, and in the Inspector, replace **Motion** with the **hero_idle_anim_powerup** animation clip.

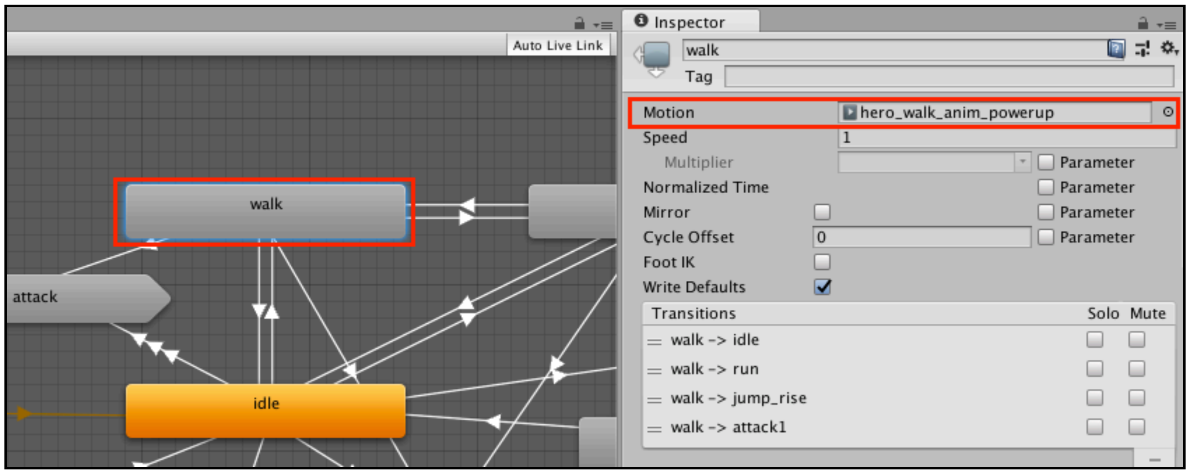


idle
Motion hero_idle_anim_powerup
Speed 1
Multiplier Parameter
Normalized Time Parameter
Mirror Parameter
Cycle Offset 0 Parameter
Foot IK
Write Defaults ✓

Transitions	Solo	Mute
idle -> walk		
idle -> run		
idle -> jump_rise		
idle -> attack1		
idle -> run_attack		
idle -> attack2		
idle -> attack3		
idle -> pickup		

Our hero needs to wear the gloves while walking.

Select the walk state, and replace **Motion** with **hero_walk_anim_powerup** animation clip.

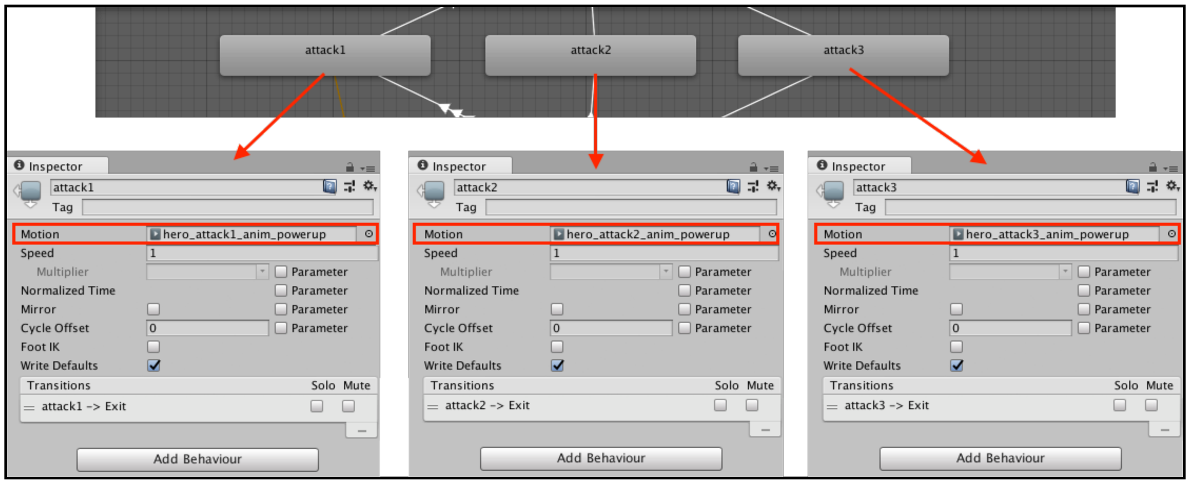


walk
Tag
Motion hero_walk_anim_powerup
Speed 1
Multiplier ☐ Parameter
Normalized Time ☐ Parameter
Mirror ☐ ☐ Parameter
Cycle Offset 0 ☐ Parameter
Foot IK ☐
Write Defaults ✓

Transitions	Solo	Mute
walk -> idle	☐	☐
walk -> run	☐	☐
walk -> jump_rise	☐	☐
walk -> attack1	☐	☐

And he definitely needs to wear them while throwing a punch.

Open the attack substate machine and set the **Motion** field of these states: **attack1**, **attack2** and **attack3** as the **hero_attack1_anim_powerup**, **hero_attack2_anim_powerup**, and **hero_attack3_anim_powerup** clips respectively.



attack1
Tag
Motion hero_attack1_anim_powerup
Speed 1
Multiplier ☐ Parameter
Normalized Time ☐ Parameter
Mirror ☐ ☐ Parameter
Cycle Offset 0 ☐ Parameter
Foot IK ☐
Write Defaults ✓

Transitions	Solo	Mute
attack1 -> Exit	☐	☐

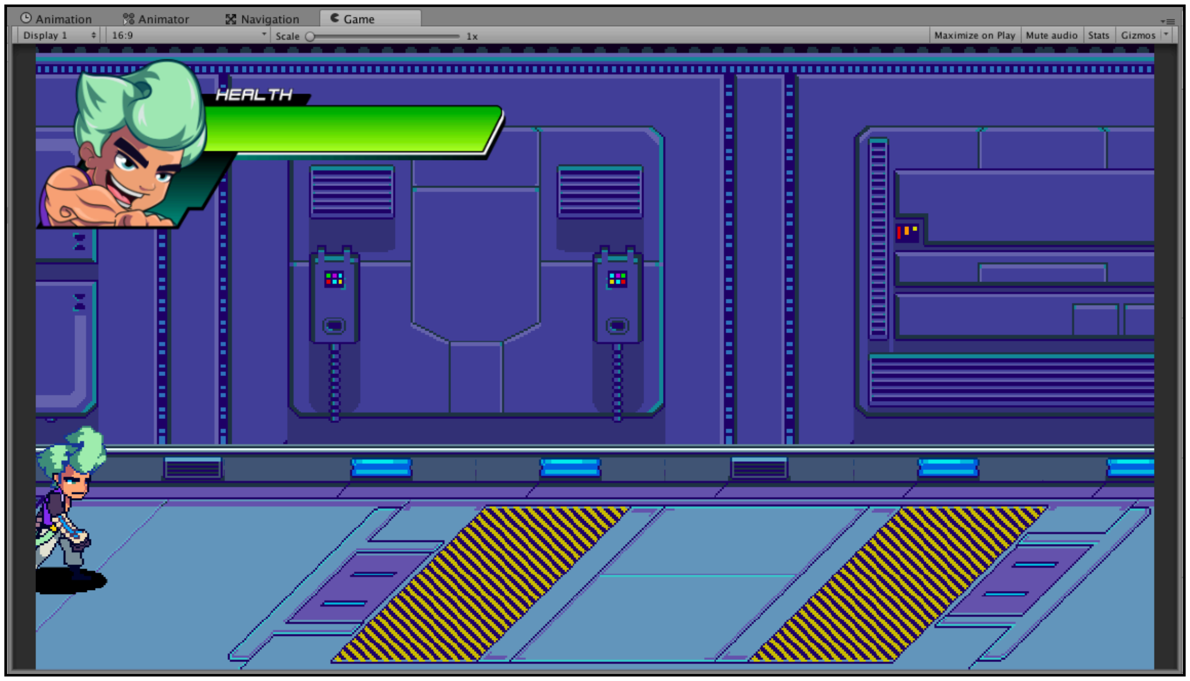
attack2
Tag
Motion hero_attack2_anim_powerup
Speed 1
Multiplier ☐ Parameter
Normalized Time ☐ Parameter
Mirror ☐ ☐ Parameter
Cycle Offset 0 ☐ Parameter
Foot IK ☐
Write Defaults ✓

Transitions	Solo	Mute
attack2 -> Exit	☐	☐

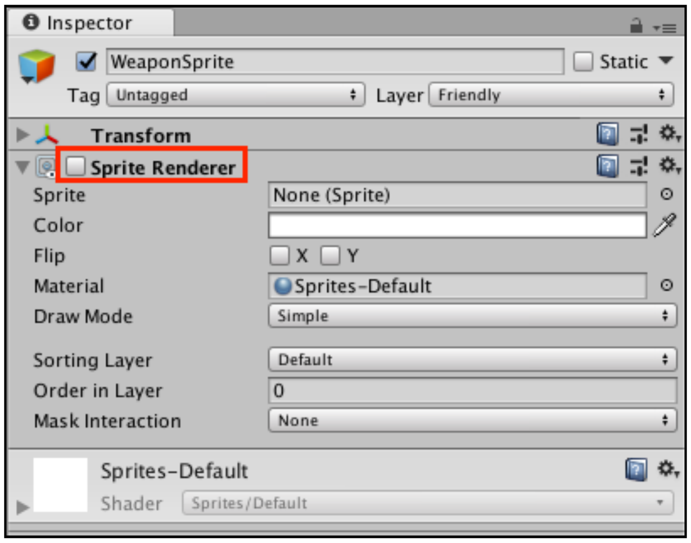
attack3
Tag
Motion hero_attack3_anim_powerup
Speed 1
Multiplier ☐ Parameter
Normalized Time ☐ Parameter
Mirror ☐ ☐ Parameter
Cycle Offset 0 ☐ Parameter
Foot IK ☐
Write Defaults ✓

Transitions	Solo	Mute
attack3 -> Exit	☐	☐

The hero’s animation is complete. So run the game. Does he always have the gloves on now? Looks like it. Your work is not done here.



Select the **WeaponSprite** child of HeroAnimator and disable the **SpriteRenderer**. Fixed!


Inspector
WeaponSprite
Static
Tag Untagged
Layer Friendly
Transform
Sprite Renderer
Sprite None (Sprite)
Color
Flip X Y
Material Sprites-Default
Draw Mode Simple
Sorting Layer Default
Order in Layer 0
Mask Interaction None
Sprites-Default
Shader Sprites/Default

But your work continues. You added new sprites and animations, so naturally, you’ll need to spend some time coding.

Create a new **C# script** named **Powerup.cs** in the **Scripts** folder. Open the script and replace its contents with this code:

```
using System.Collections;
using UnityEngine;
```

```
public class Powerup : MonoBehaviour {
  //1
  public GameObject rootObject;
  public GameObject shadowSprite;
  public Rigidbody body;
  public int uses = 20;
  public Hero user;
  public SpriteRenderer sprite;

  //2
  public AttackData attackData1;
  public AttackData attackData2;
  public AttackData attackData3;

  //3
  protected virtual void Update() {
    Vector3 spritePos = shadowSprite.transform.position;
    spritePos.y = 0;
    shadowSprite.transform.position = spritePos;
  }
}
```

The `Powerup` script will be used on all instances of the POW300 gloves.

1. Here are your references to the `Powerup` GameObject root, its shadow, its physical rigidBody and the current `user` who wields it. Also, the number of times the powerup can be used, with a default value of 20.

2. Here are your replacement `AttackData` methods. When the hero equips gloves, the game will use these `AttackData` values to calculate damage.

3. Here you have an overriding `Update` to the sprite's shadow position, similar to the Actor's shadow sprite.

Next, add the following methods:

```
//1
public void Use() {
  uses--;
  if (uses <= 0) {
    StartCoroutine(DestroyAnimation());
  }
}

//2
protected virtual void SetOpacity(float value) {
  Color color = sprite.color;
  color.a = value;
  sprite.color = color;
}

//3
private IEnumerator DestroyAnimation(int amount = 5) {
  int i = amount;
```

```
    while (i > 0) {
      SetOpacity(0.5f);
      yield return new WaitForSeconds(0.2f);
      SetOpacity(1.0f);
      yield return new WaitForSeconds(0.2f);
      i--;
    }
    Destroy(rootObject);
  }

  //4
  public bool CanEquip() {
    return uses > 0;
  }
```

1. `Use` subtracts one from the `uses` variable. When it reaches zero, it starts the `DestroyAnimation` coroutine.

2. `SetOpacity` is a helper method that sets the alpha value of the SpriteRenderer's `Color` variable to the value passed as its parameter.

3. `DestroyAnimation` is a coroutine that flickers the powerup object when it's almost spent by flickering the alpha value of the glove sprite numerous times before destroying its GameObject.

4. `CanEquip` determines whether the powerup can be used by the hero by checking if the powerup has remaining uses.

Save the script and open the `Hero.cs` script. Next, you'll let the hero pick up the powerup.

Add these variables just above `Start()`:

```
  //1
  bool isPickingUpAnim;
  bool weaponDropPressed = false;
  public bool hasWeapon;

  //2
  public bool canJump = true;

  //3
  public SpriteRenderer powerupSprite;
  public Powerup nearbyPowerup;
  public Powerup currentPowerup;
  public GameObject powerupRoot;
```

1. `weaponDropPressed` stores whether the player pressed jump, which forces the hero to drop any equipped weapons. `isPickingUpAnim` is true when the hero's animator is playing the pickup animation. Lastly, `hasWeapon` stores if the hero has equipped a weapon.

2. `canJump` flags whether the hero can jump or not.

3. These are references for the powerup’s behavior: First is a reference to the `powerupSprite` SpriteRenderer, then you have references to the `Powerup` class you created earlier — these track equipped `currentPowerup` variables and any (aptly named) `nearbyPowerup`, and lastly, a reference to the powerup’s root object.

Next, add the powerup equipping methods to the bottom of the class:

```
//1
public void PickupWeapon(Powerup powerup) {
  baseAnim.SetTrigger ("PickupPowerup");
}

public void DidPickupWeapon() {
  //2
  if (nearbyPowerup != null && nearbyPowerup.CanEquip()) {
    Powerup powerup = nearbyPowerup;
    hasWeapon = true;
    currentPowerup = powerup;
    nearbyPowerup = null;
    powerupRoot = currentPowerup.rootObject;
    powerup.user = this;
    //3
    currentPowerup.body.velocity = Vector3.zero;
    powerupRoot.SetActive (false);
    Walk ();

    //4
    powerupSprite.enabled = true;
    canRun = false;
    canJump = false;
  }
}
```

These methods allow the hero to use powerups.

1. `PickupWeapon` triggers the hero’s pick-up animation.

2. `DidPickupWeapon` picks up any powerups that are near the hero. It checks if a powerup is nearby. If `true`, it sets the references and values necessary for equipping the powerup.

3. Prevents the rigidbody of the powerup from moving and hides the powerup in the scene by setting `powerupRoot.SetActive(false)`. Constrains the hero’s motion to walking only by calling the `Walk()` method.

4. Enables the glove’s `powerupSprite` SpriteRenderer. Sets `canRun` and `canJump` flags to `false`, further preventing the hero from running or jumping.

Add these methods to allow the player to unequip the powerup:

```
public void DropWeapon() {
  //1
  powerupRoot.SetActive(true);
  powerupRoot.transform.position = transform.position + Vector3.up;
  currentPowerup.body.AddForce(Vector3.up * 100);

  //2
  powerupRoot = null;
  currentPowerup.user = null;
  currentPowerup = null;
  nearbyPowerup = null;

  //3
  powerupSprite.enabled = false;
  canRun = true;
  hasWeapon = false;
  canJump = true;
}
```

1. Enables the `powerupRoot` object again. Then `AddForce` causes the hero to toss the glove sprite upward.
2. Resets the member references to the powerup.
3. Disables the `powerupSprite` SpriteRenderer to hide the glove sprites on the hero. Restores the hero's `canRun` and `canJump` and sets `hasWeapon` to `false`.

You are getting close to finishing these gloves! But the hero needs to be able to tell when a powerup is nearby. Add these methods to the script:

```
//1
void OnTriggerEnter(Collider collider) {
  if (collider.gameObject.layer == LayerMask.NameToLayer("Powerup")) {
    Powerup powerup = collider.gameObject.GetComponent<Powerup>();
    if (powerup != null) {
      nearbyPowerup = powerup;
    }
  }
}

//2
void OnTriggerExit(Collider collider) {
  if (collider.gameObject.layer == LayerMask.NameToLayer("Powerup")) {
    Powerup powerup = collider.gameObject.GetComponent<Powerup>();
    if (powerup == nearbyPowerup) {
      nearbyPowerup = null;
    }
  }
}
```

These `OnTrigger` methods are called when a corresponding trigger collider overlaps the Hero's collider.

1. `OnTriggerEnter` checks if a colliding trigger collider is a member of the powerup layer. If it is, it fetches the Powerup component by using `GetComponent<Powerup>()`. If it finds a Powerup component, the code stores its reference as the `nearbyPowerup` variable.

2. The `OnTriggerExit` method does the opposite of the previous method: If it finds a powerup, and it is the same as the `nearbyPowerup` variable, it disregards it by setting `nearbyPowerup` to `null`.

Note: You'll add the Powerup layer to the game's layers later.

Next, you'll update the `HitActor` method to enable the hero to use the powerup's AttackData when he's equipped the gloves.

Change the first three `if` statements in `HitActor()` from this:

```
if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1")) {
  AnalyzeNormalAttack (normalAttack, 2, actor, hitPoint, hitVector);
} else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack2")) {
  AnalyzeNormalAttack (normalAttack2, 3, actor, hitPoint, hitVector);
} else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack3")) {
  AnalyzeNormalAttack (normalAttack3, 1, actor, hitPoint, hitVector);
}
```

To this:

```
if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1")) {
  //1
  AttackData attackData = hasWeapon ? currentPowerup.attackData1 :
normalAttack;
  //2
  AnalyzeNormalAttack (attackData, 2, actor, hitPoint, hitVector);
  //3
  if (hasWeapon) {
    currentPowerup.Use();
  }
} else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack2")) {
  AttackData attackData = hasWeapon ? currentPowerup.attackData2 :
normalAttack2;
  AnalyzeNormalAttack (attackData, 3, actor, hitPoint, hitVector);
  if (hasWeapon) {
    currentPowerup.Use();
  }

} else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack3")) {
  AttackData attackData = hasWeapon ? currentPowerup.attackData3 :
normalAttack3;
  AnalyzeNormalAttack (attackData, 1, actor, hitPoint, hitVector);
  if (hasWeapon) {
    currentPowerup.Use();
  }
  }
```

This block of code might look complicated at first, but its logic is quite simple. Take a look at the first `if` condition:

1. Which `AttackData` to use is determined by a ternary operator. If the `hasWeapon` flag is `true`, the `attackData1` of the powerup is used to calculate damage. Otherwise, the game sticks with the `normalAttack`.
2. This line handles the damage like before.
3. When a powerup weapon is equipped, the code subtracts its uses by calling `currentPowerup.Use()`.

The same logic applies to `attack2` and `attack3` states, but they use different AttackData parameters.

Find `TakeDamage()` and add the following lines above the `base.TakeDamage` line:

```
  if (hasWeapon) {
    DropWeapon();
  }
```

This forces the hero to drop his weapon when he takes a smack across the kisser.

The `Update` method needs a few changes too.

Add the following line in `Update()`, right below this line `isHurtAnim = baseAnim.GetCurrentAnimatorStateInfo(0).IsName("hurt")`.

```
  isPickingUpAnim =
    baseAnim.GetCurrentAnimatorStateInfo(0).IsName("pickup");
```

That updates the `isPickingUpAnim` Boolean when the pickup animation plays.

Find this `if` condition:

```
  if (attack && Time.time >= lastAttackTime + attackLimit && !isKnockedOut)
  {
```

And replace it with:

```
  if (attack && Time.time >= lastAttackTime + attackLimit && !isKnockedOut
  && !isPickingUpAnim) {
```

You're preventing the hero from attacking whilst picking up a powerup.

Find this jump `if` condition:

```
if (jump &&  !isKnockedOut &&
    !isJumpLandAnim && !isAttackingAnim &&
    (isGrounded || (isJumpingAnim && Time.time < lastJumpTime +
jumpDuration)) )
```

And replace with:

```
  if (canJump && jump &&  !isKnockedOut &&
    !isJumpLandAnim && !isAttackingAnim &&
    !isPickingUpAnim && !weaponDropPressed &&
    (isGrounded || (isJumpingAnim && Time.time < lastJumpTime +
jumpDuration)) )
```

These extra conditions prevent the hero from jumping when `canJump` is `false` and when he is picking up a powerup.

Insert the code below at the bottom of the `Update` method right above `if(attack && Time.time...`

```
if (attack && Time.time >= lastAttackTime + attackLimit && isGrounded
&& !isPickingUpAnim) {
  if (nearbyPowerup != null && nearbyPowerup.CanEquip ()) {
    lastAttackTime = Time.time;
    Stop ();
    PickupWeapon (nearbyPowerup);
  }
}
```

With this, you've made it so the hero will pick up a nearby weapon rather than attack when the player presses the attack button.

While still in the `Update` method, insert this above `if (canJump && jump...`

```
if (jump && hasWeapon) {
  weaponDropPressed = true;
  DropWeapon();
}

if (weaponDropPressed && !jump) {
  weaponDropPressed = false;
}
```

Here you let the hero drop the powerup when the player presses jump.

Great! Your Hero class is all set for now. **Save** it and open `HeroCallback.cs`.

Add the following method:

```
public void DidPickup() {
  hero.DidPickupWeapon ();
}
```

Here you call `hero.DidPickupWeapon()` when the pickup animation is done.

You're done in here for now. **Save** the script and open `Powerup.cs`.

Insert the following in the `Use` method above `StartCoroutine(DestroyAnimation());`

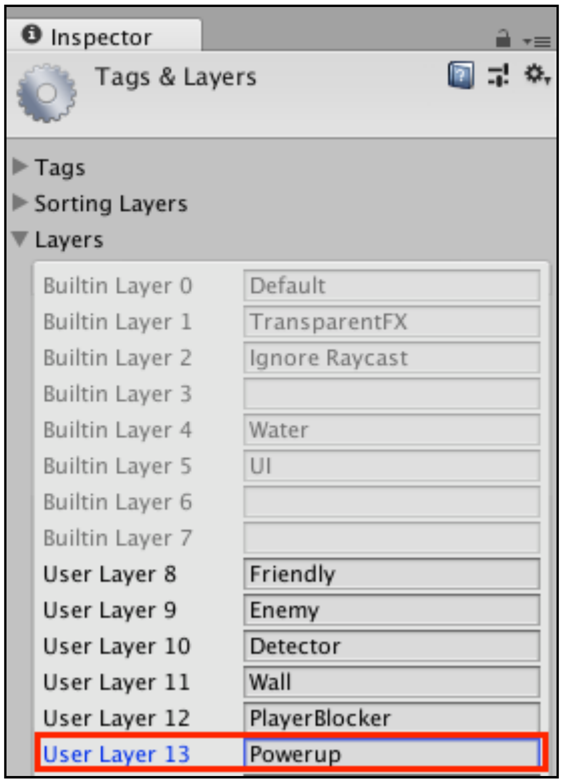
```
user.DropWeapon();
```

This tells the `user` to drop its current weapon when `uses` reaches `0`.

Okay! That's it for scripting. For now. **Save** it and head back into Unity.

Remember that change you needed to make to the layers of the **POW300** prefab?

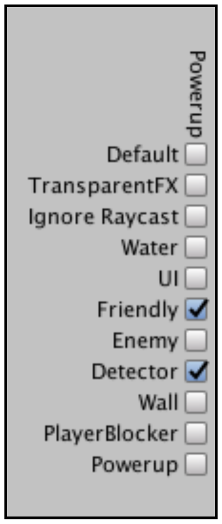
Open **Tags & Layers** in the Inspector by selecting **Edit \ Project Settings \ Tags & Layers** in the top menu. Add a new **layer** named **Powerup**.



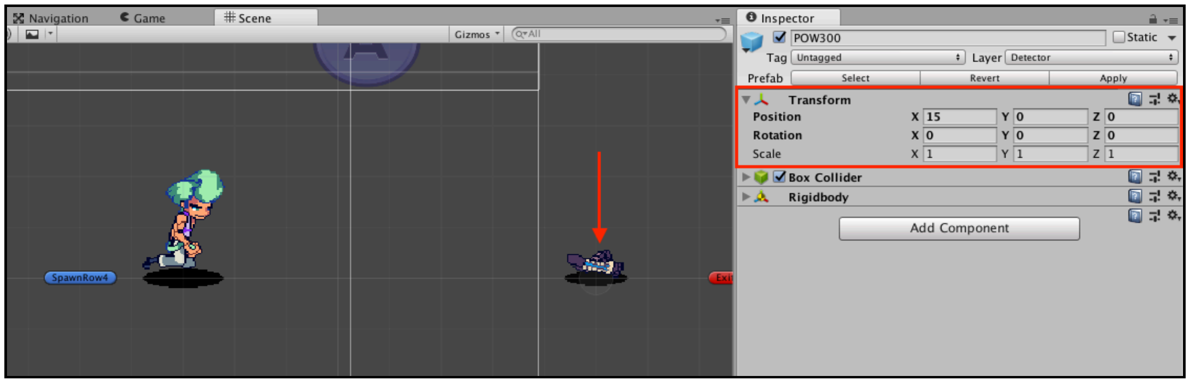
Builtin Layer 0	Default
Builtin Layer 1	TransparentFX
Builtin Layer 2	Ignore Raycast
Builtin Layer 3	
Builtin Layer 4	Water
Builtin Layer 5	UI
Builtin Layer 6	
Builtin Layer 7	
User Layer 8	Friendly
User Layer 9	Enemy
User Layer 10	Detector
User Layer 11	Wall
User Layer 12	PlayerBlocker
User Layer 13	Powerup

Now onto the physics collisions for your shiny new layer.

Select **Edit \ Project Settings \ Physics** in the Top Menu to open the Physics Manager in the Inspector. In the **Layer Collision Matrix**, uncheck **all** collisions for Powerup *except* for **Friendly** and **Detector**.

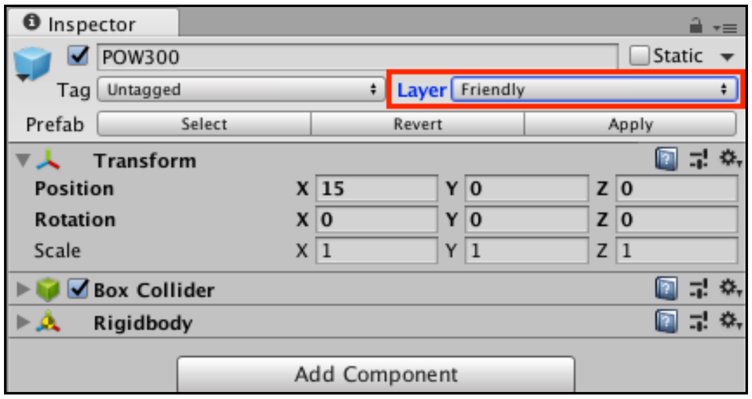

Powerup
Default ☐
TransparentFX ☐
Ignore Raycast ☐
Water ☐
UI ☐
Friendly ☑
Enemy ☐
Detector ☑
Wall ☐
PlayerBlocker ☐
Powerup ☐

Now drag the **POW300** prefab into the scene. Set its position to `(X:15, Y:0, Z:0)` to position the POW300 in front of the hero.



Inspector
POW300 Static
Tag Untagged Layer Detector
Prefab Select Revert Apply
Transform
Position X 15 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Box Collider
Rigidbody
Add Component

Set the **Layer** of the POW300 GameObject to **Friendly**. If prompted, choose **No, this object only**.


Inspector
POW300 Static
Tag Untagged Layer Friendly
Prefab Select Revert Apply
Transform
Position X 15 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Box Collider
Rigidbody
Add Component

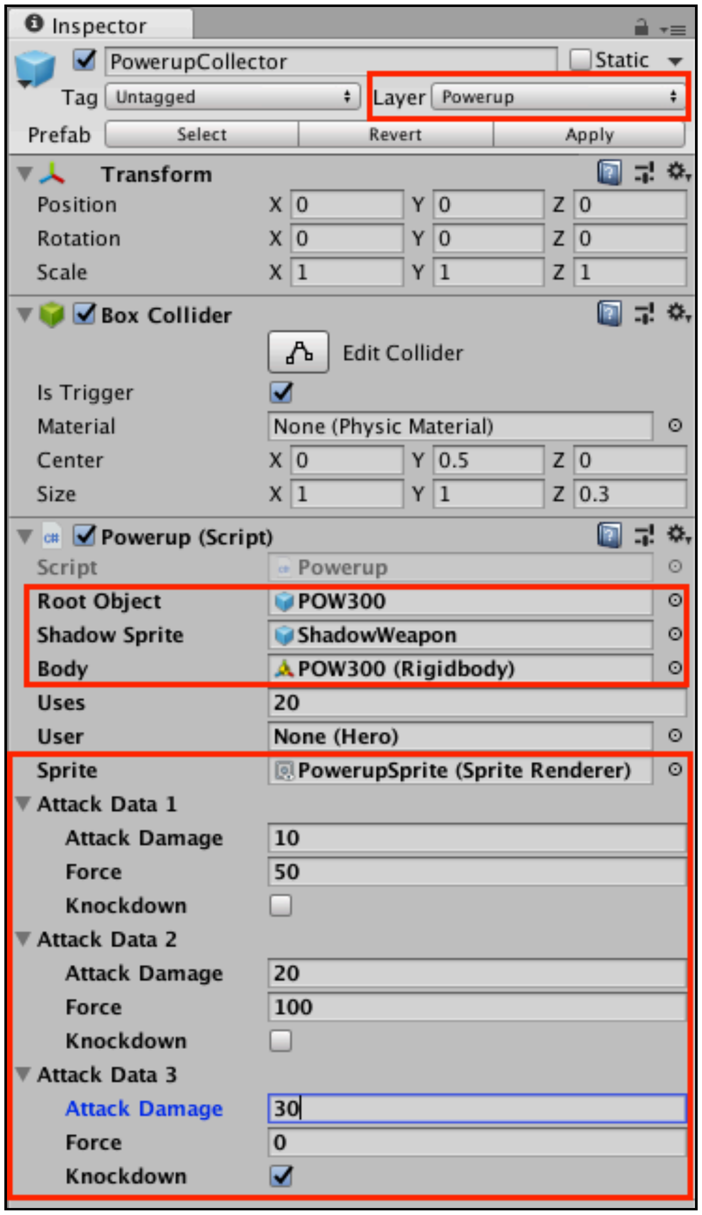
Next, select the **PowerupCollector** child of POW300. Set its layer to **Powerup**, and add a **Powerup** component to this GameObject.

Set **Root Object** to **POW300**, and **Shadow Sprite** to **ShadowWeapon**. Then set **Body** to **POW300**, and **Sprite** to **PowerupSprite**.

Set the `AttackData` values for the powerup like so:

- **Attack Data 1: Damage** `10`, **Force** `50` and **Knockdown** disabled.
- **Attack Data 2: Damage** `20`, **Force** `100` and **Knockdown** disabled.
- **Attack Data 3: Damage** `30`, **Force** `0` and **Knockdown** enabled.

With that, you’re done with the POW300 prefab! That took a bit of sweat, but you got to build on top of work you’ve already done. It’s a great place to be. Feels pretty good, right?



Inspector

PowerupCollector — Static

Tag: Untagged — Layer: Powerup

Prefab: Select | Revert | Apply

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Box Collider

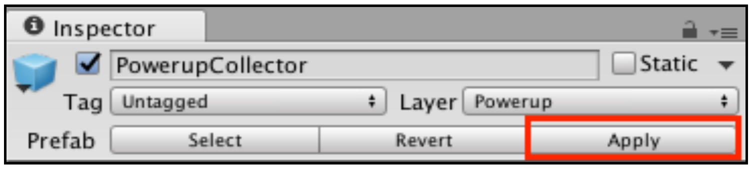
Edit Collider

Is Trigger	✓
Material	None (Physic Material)
Center	X 0, Y 0.5, Z 0
Size	X 1, Y 1, Z 0.3

Powerup (Script)

Script	Powerup
Root Object	POW300
Shadow Sprite	ShadowWeapon
Body	POW300 (Rigidbody)
Uses	20
User	None (Hero)
Sprite	PowerupSprite (Sprite Renderer)
Attack Data 1	
Attack Damage	10
Force	50
Knockdown	
Attack Data 2	
Attack Damage	20
Force	100
Knockdown	
Attack Data 3	
Attack Damage	30
Force	0
Knockdown	✓

Click **Apply** to save the POW300 prefab.



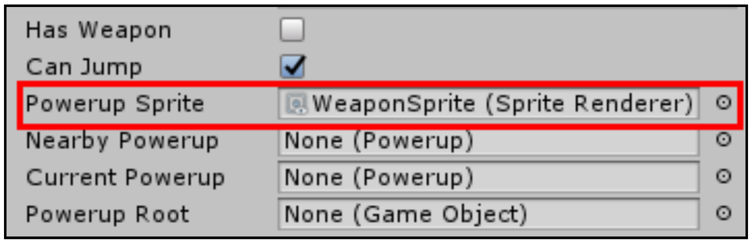
Inspector

PowerupCollector — Static

Tag: Untagged — Layer: Powerup

Prefab: Select | Revert | Apply

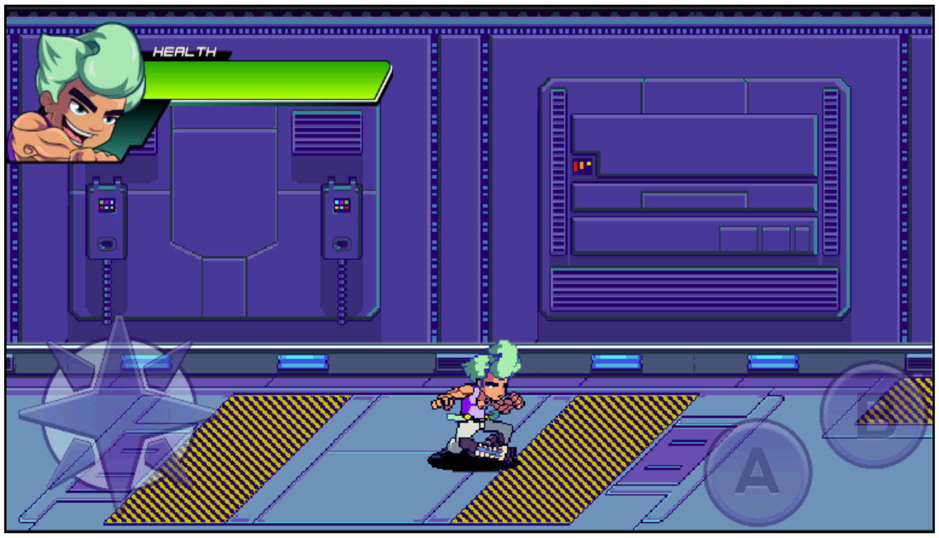
Next, select **MyHero** and adjust its Hero component. Set **Powerup Sprite** from the Hero component as the **WeaponSprite** child of HeroAnimator.



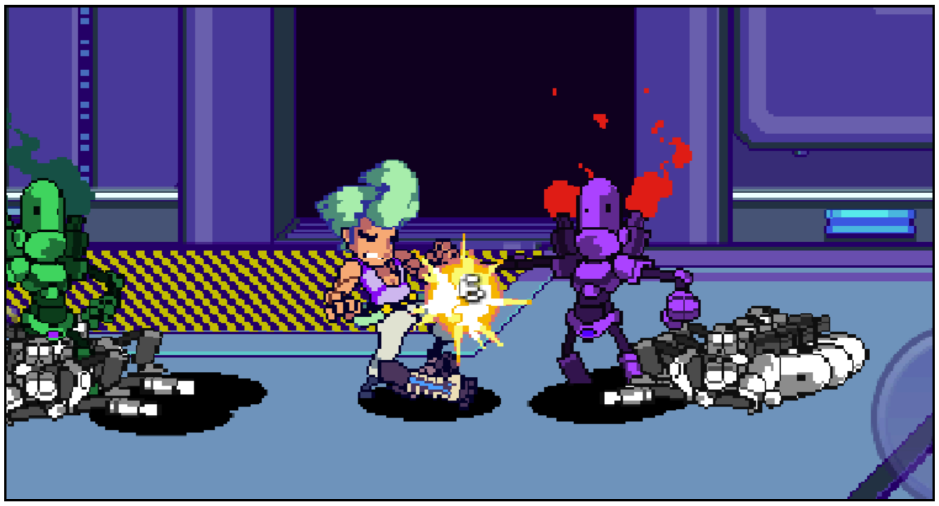
Has Weapon	☐
Can Jump	✓
Powerup Sprite	WeaponSprite (Sprite Renderer)
Nearby Powerup	None (Powerup)
Current Powerup	None (Powerup)
Powerup Root	None (Game Object)

Save the scene and the project. PLAY THE GAME. Play it!

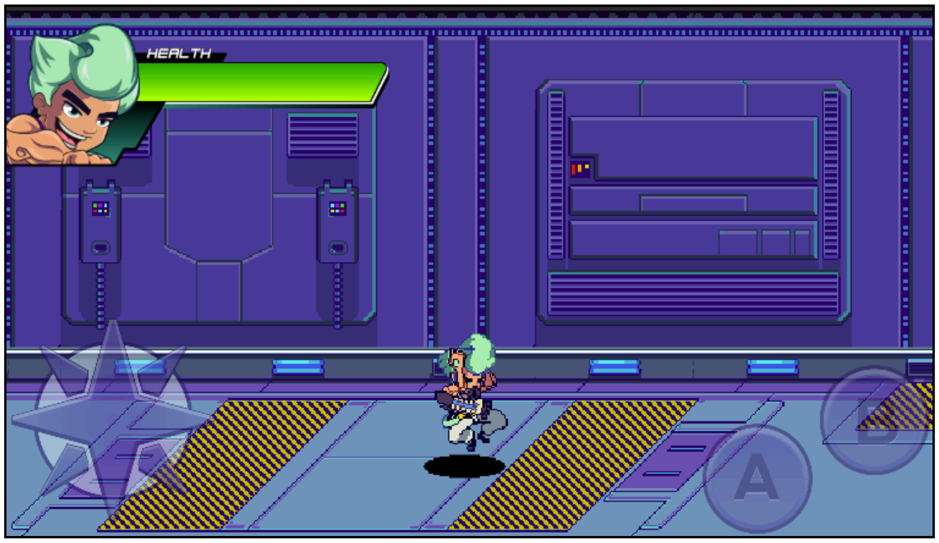
Walk right up to that POW300. Press that attack button. Watch as the hero kneels down and picks it up.



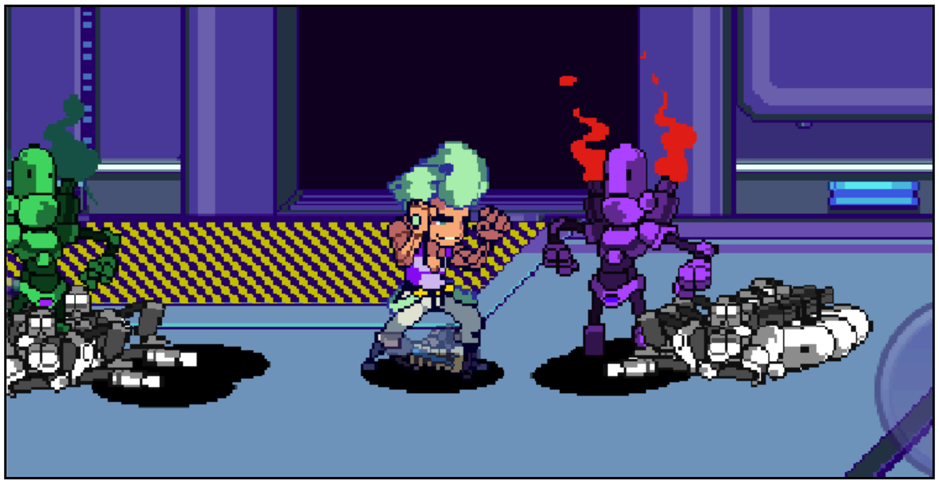
Go find some poor droid and lay it out. Hehehehe.



Try jumping or taking a hit. Per your design, the hero drops the gloves.



Okay, pick them back up and keep punching droids. Once all the uses are spent, the fancy gloves disappear.



Nice going. You took control of the game and gave the hero something to work with when that nasty boss shows up.

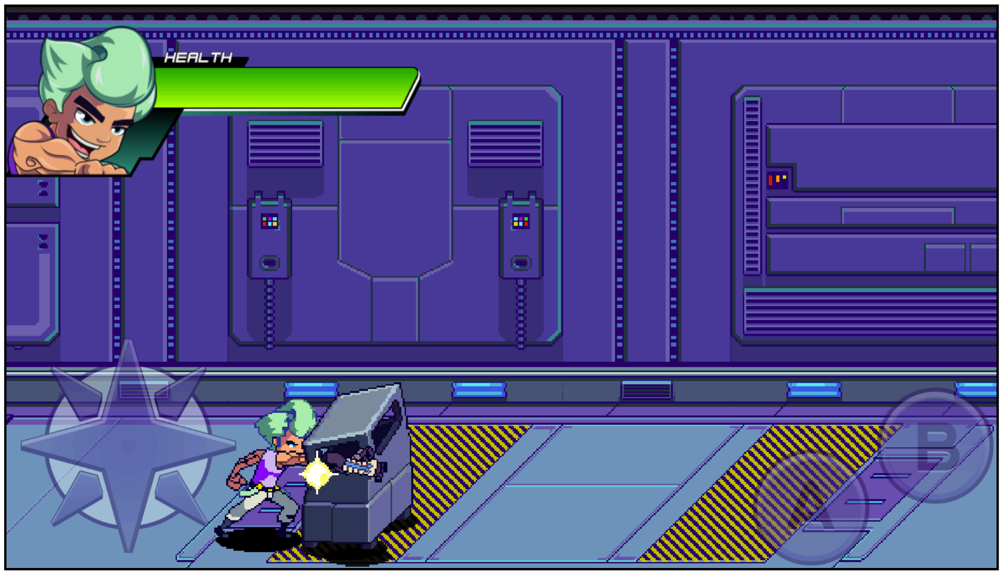
But who leaves gloves like that just laying out in the open? Seems like a lot of bosses would get punched if they were so readily available. :]

One robot’s trash is another hero’s treasure

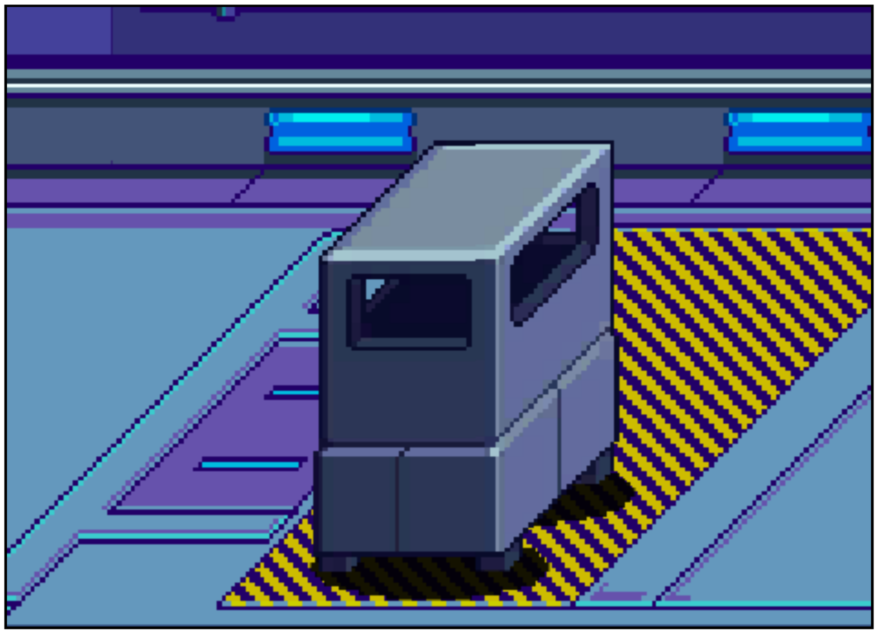
Trash cans will serve as the storage vessels for the gloves. They’re cheap, durable and discrete.

Discrete? Huh?

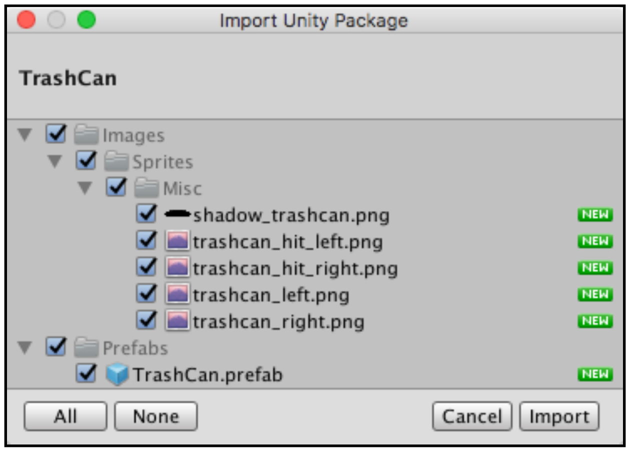
How often do you go digging through trash cans? Hopefully never. But admit it: You would dig through trash to get your hands on gloves like these!



In PompaDroid, trash cans have multiple purposes: blocking movement and serving as containers for powerups.



Import **TrashCan.unitypackage** from the **UnityPackages** folder. It contains a semi-complete prefab for trash cans and the sprites used to create them.

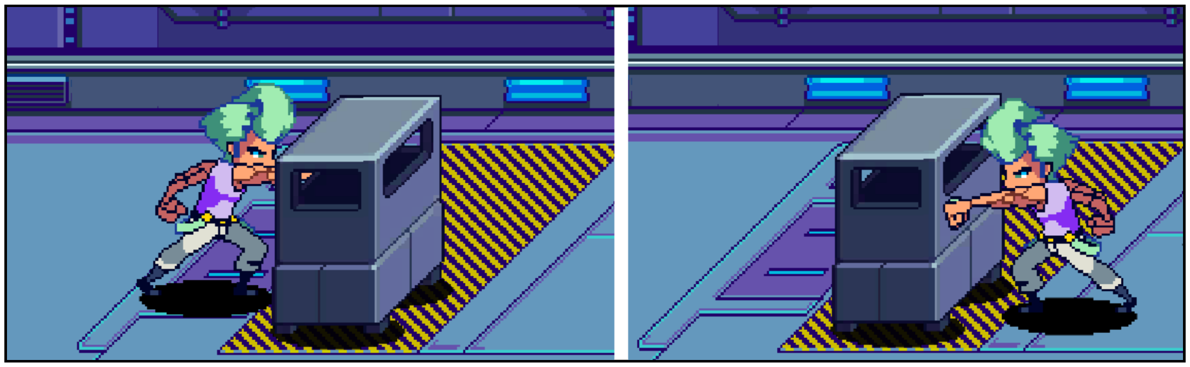

Import Unity Package

TrashCan

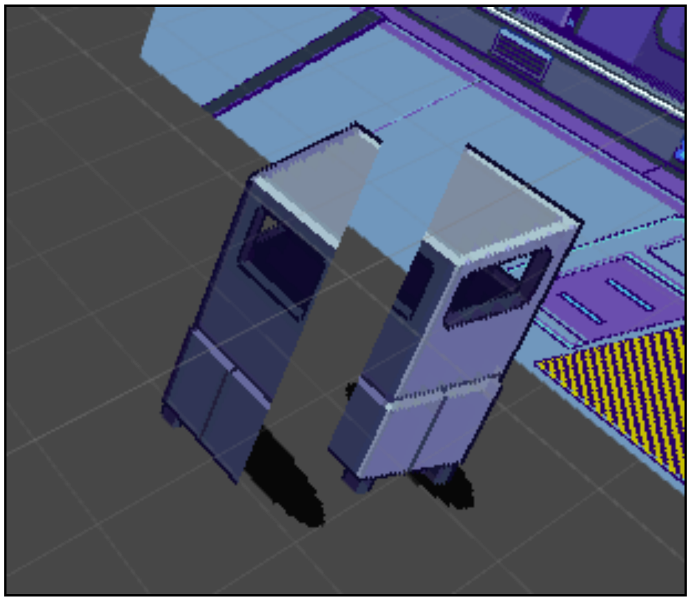
```
▼ ☑ Images
  ▼ ☑ Sprites
    ▼ ☑ Misc
        ☑ shadow_trashcan.png        NEW
        ☑ trashcan_hit_left.png      NEW
        ☑ trashcan_hit_right.png     NEW
        ☑ trashcan_left.png          NEW
        ☑ trashcan_right.png         NEW
▼ ☑ Prefabs
  ☑ TrashCan.prefab                  NEW
```

The trash can sprite is split into two sprites that serves as the left and right sides. You'll use them to accommodate layering needs when actors are near the garbage can.

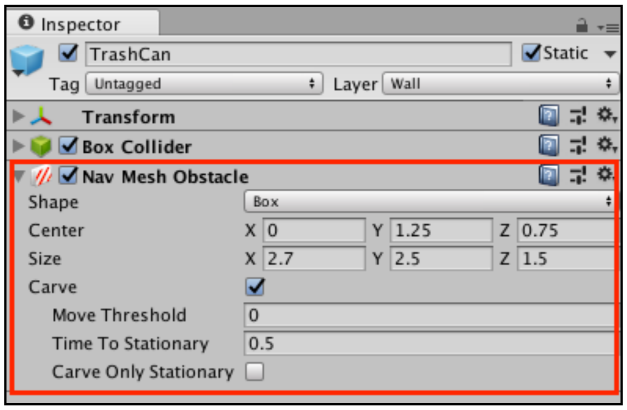
When another sprite, such as a hero, approaches the trash can from the left, the sprite should appear behind the trash can. When it approaches from the right, the sprite should appear in front of the trash can.



You'll achieve this effect by positioning the trash can sprites at different distances from the camera. They will, however, appear as a single sprite.



Your TrashCan prefab is configured accordingly. It also has a **NavMeshObstacle** component attached to it so the enemy's AI can pathfind around the trash cans.



Inspector

TrashCan ✓ Static

Tag: Untagged Layer: Wall

Transform

Box Collider

Nav Mesh Obstacle

Shape	Box		
Center	X 0	Y 1.25	Z 0.75
Size	X 2.7	Y 2.5	Z 1.5
Carve	✓		
Move Threshold	0		
Time To Stationary	0.5		
Carve Only Stationary	☐		

Implementing the trash cans

The prefab needs a component that spawns a POW300 when the trash can is hit.

Create a new **C# script** called **Container.cs** in the **Scripts** folder. Replace its contents with the following:

```
using UnityEngine;

public class Container : MonoBehaviour {
  //1
  bool isOpen;
  public GameObject prizePrefab;
  public Transform spawnPoint;

  //2
  public Sprite leftSpriteClose;
  public Sprite rightSpriteClose;
  public Sprite leftSpriteOpen;
  public Sprite rightSpriteOpen;
  public SpriteRenderer leftSpriteRenderer;
  public SpriteRenderer rightSpriteRenderer;

  //3
  public GameObject sparkPrefab;
}
```

Now the Container script has these variables:

1. A Boolean to flag if the trash can is opened or not, a GameObject reference to the prefab of its content, and a `spawnPoint` to serve as the position from which the loot will spawn.
2. References to the left, right, closed and open sprites, and their corresponding SpriteRenderers.
3. A reference to the hit spark prefab that shows when the container is hit.

Add these methods below the variable declarations:

```
//1
public bool CanBeOpened() {
  return isOpen != true;
}

//2
public void Hit(Vector3 hitPoint) {
  GameObject sparkObj = Instantiate(sparkPrefab);
  sparkObj.transform.position = hitPoint;
}

//3
public void Open(Vector3 hitPoint) {
  isOpen = true;
  SetSprites(leftSpriteOpen, rightSpriteOpen);
  GameObject obj = Instantiate(prizePrefab);
  obj.transform.position = spawnPoint.transform.position;

}

//4
private void SetSprites(Sprite leftSprite, Sprite rightSprite) {
  leftSpriteRenderer.sprite = leftSprite;
  rightSpriteRenderer.sprite = rightSprite;
}
```

Here you handle the container’s behavior.

1. `CanBeOpened` returns a Boolean to indicate whether the container can still be opened.
2. `Hit` spawns a sparking effect when the container is hit at its `hitPoint` parameter.
3. `Open` changes its sprite to the left and right versions of the open sprite. It then spawns an instance of the `prizePrefab` at the `spawnPoint` position.
4. `SetSprites` is a helper method that changes the sprites of the `leftSpriteRenderer` and `rightSpriteRenderer`.

Save the script and open `Hero.cs`. Remember how you made it so that actors couldn’t hit objects? That’s going to make it hard on the hero!

Add this override for the `DidHitObject` method from the Actor class:

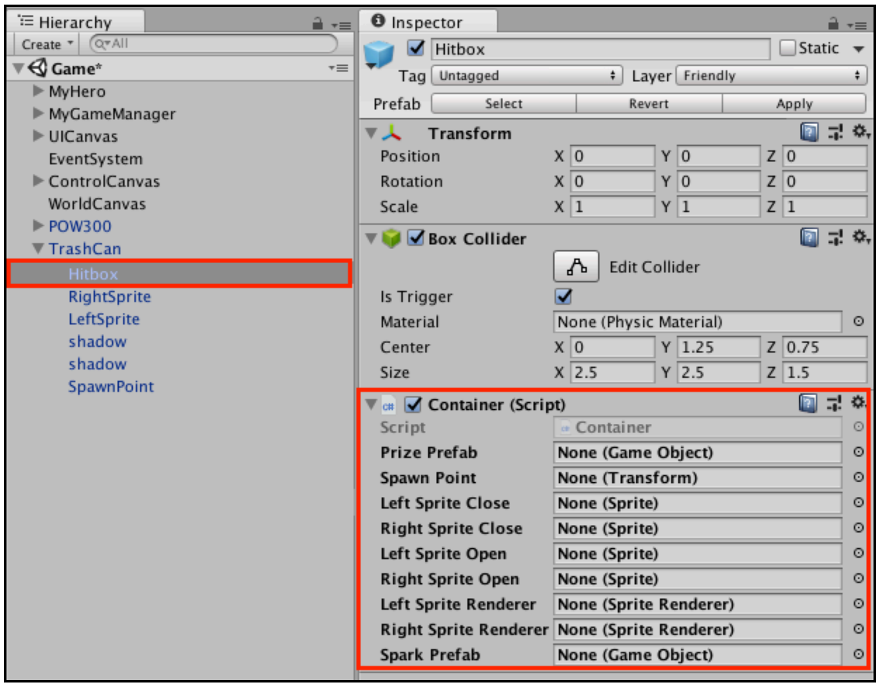
```
  public override void DidHitObject(Collider collider, Vector3 hitPoint,
  Vector3 hitVector) {
    Container containerObject = collider.GetComponent<Container>();
    //1
    if (containerObject != null) {
      containerObject.Hit(hitPoint);
      if (containerObject.CanBeOpened() && collider.tag != gameObject.tag)
  {
        containerObject.Open(hitPoint);
      }
    } else {
      //2
      base.DidHitObject(collider, hitPoint, hitVector);
    }
  }
```

Your fancy new override does the following:

1. If the object hit contains a `Container` script, then the attack will open the container.
2. Else, the base `Actor` class script processes the `DidHit` script by calling `base.DidHitObject(collider, hitPoint, hitVector);`

Save the script and return to the editor — time to assign some references to make the trash can prefab work.

Drag the **TrashCan** prefab into the Hierarchy to create an instance of it. Select the **HitBox** child of TrashCan and add a **Container** component to it.


Hierarchy
Create ▾ All
Game*
MyHero
MyGameManager
UICanvas
EventSystem
ControlCanvas
WorldCanvas
POW300
TrashCan
Hitbox
RightSprite
LeftSprite
shadow
shadow
SpawnPoint

Inspector
☑ Hitbox ☐ Static
Tag Untagged Layer Friendly
Prefab Select Revert Apply

Transform			
Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

☑ Box Collider
Edit Collider

Is Trigger	✓		
Material	None (Physic Material)		
Center	X 0	Y 1.25	Z 0.75
Size	X 2.5	Y 2.5	Z 1.5

☑ Container (Script)

Script	Container
Prize Prefab	None (Game Object)
Spawn Point	None (Transform)
Left Sprite Close	None (Sprite)
Right Sprite Close	None (Sprite)
Left Sprite Open	None (Sprite)
Right Sprite Open	None (Sprite)
Left Sprite Renderer	None (Sprite Renderer)
Right Sprite Renderer	None (Sprite Renderer)
Spark Prefab	None (Game Object)

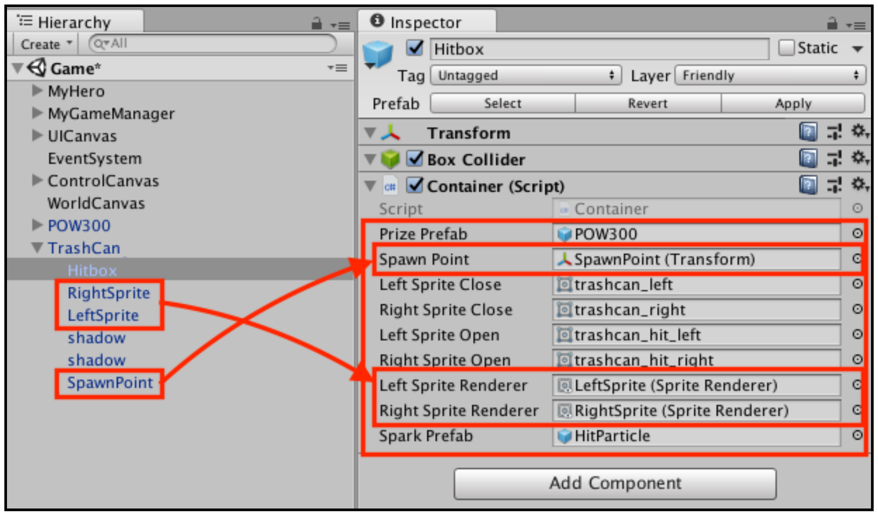
Set **Prize Prefab** to the prefab located at **Prefabs \ POW300** and drag its sibling **SpawnPoint** to the **Spawn Point** slot.

Set **Left Sprite Renderer** to its **LeftSprite** sibling.

Set **Right Sprite Renderer** to its **RightSprite** sibling and set **Spark Prefab** to the **Prefabs \ HitParticle** prefab.

Assign the sprites used by the trashcan:

- **Left Sprite Close** to the **trashcan_left** sprite
- **Right Sprite Close** to the **trashcan_right** sprite
- **Left Sprite Open** to the **trashcan_hit_left** sprite
- **Right Sprite Open** to the **trashcan_hit_right** sprite



Hierarchy

Game*
- MyHero
- MyGameManager
- UICanvas
- EventSystem
- ControlCanvas
- WorldCanvas
- POW300
- TrashCan
  - Hitbox
  - RightSprite
  - LeftSprite
  - shadow
  - shadow
  - SpawnPoint

Inspector

Hitbox — Static

Tag: Untagged — Layer: Friendly

Prefab: Select | Revert | Apply

Transform

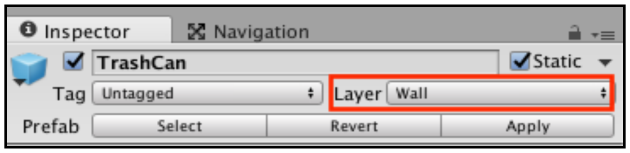
Box Collider

Container (Script)

Script	Container
Prize Prefab	POW300
Spawn Point	SpawnPoint (Transform)
Left Sprite Close	trashcan_left
Right Sprite Close	trashcan_right
Left Sprite Open	trashcan_hit_left
Right Sprite Open	trashcan_hit_right
Left Sprite Renderer	LeftSprite (Sprite Renderer)
Right Sprite Renderer	RightSprite (Sprite Renderer)
Spark Prefab	HitParticle

Add Component

On to setting the TrashCan's layer: Select **TrashCan** from the Hierarchy and set its **Layer** to **Wall**. When prompted, select **Yes, change Children**.



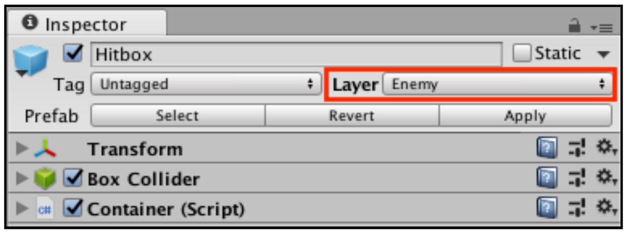
Inspector | Navigation

TrashCan — Static ✓

Tag: Untagged — Layer: Wall

Prefab: Select | Revert | Apply

Select the **HitBox** child of TrashCan. Set its **Layer** to **Enemy**.



Inspector

Hitbox — Static

Tag: Untagged — Layer: Enemy

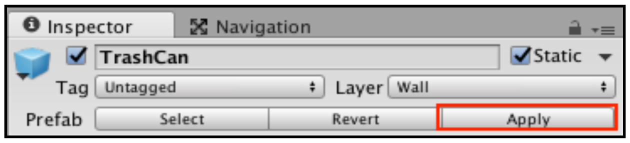
Prefab: Select | Revert | Apply

Transform

Box Collider

Container (Script)

Select **TrashCan** in the Hierarchy again and click **Apply** to save the prefab.



Inspector | Navigation

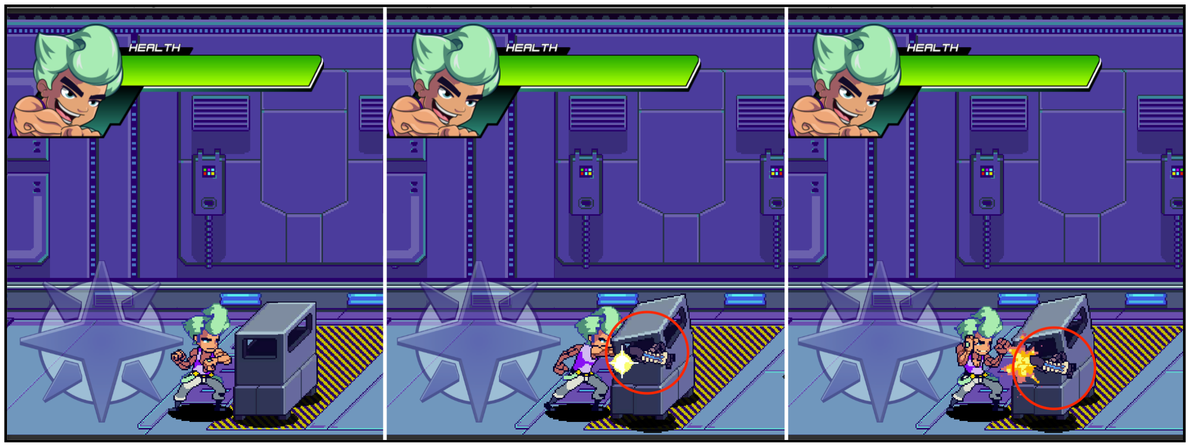
TrashCan — Static ✓

Tag: Untagged — Layer: Wall

Prefab: Select | Revert | Apply

Finally, set its position to `(X:12,Y:0, Z:-2)`. With that, you're placing the trash can in front of the player when the level starts.

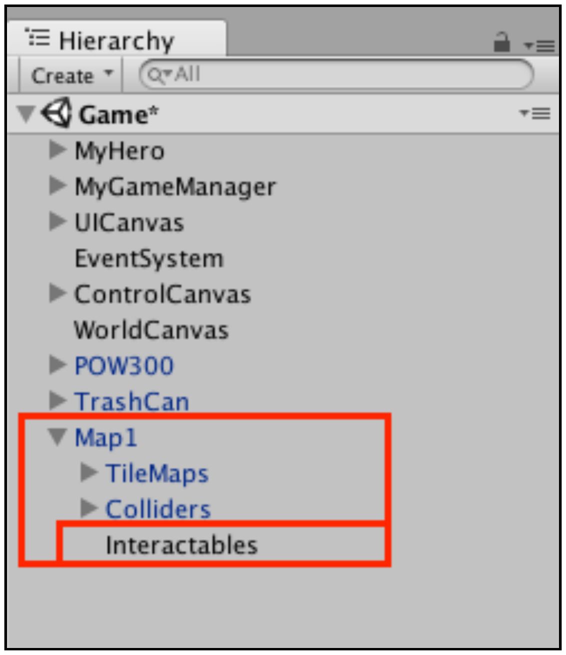
Play the game and find a trash can to punch.



It works as intended!

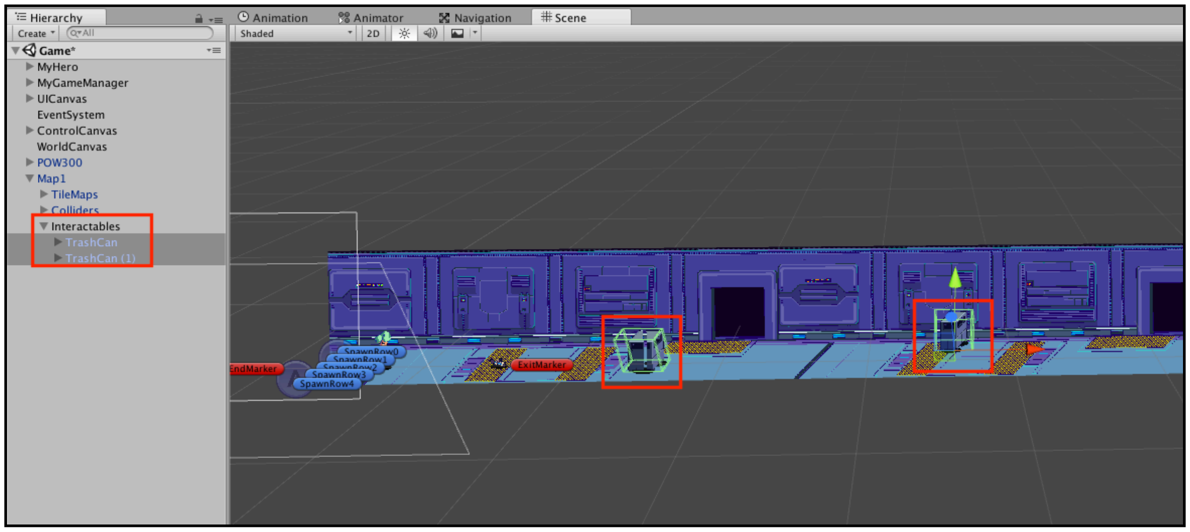
One trash can isn’t enough. Take a moment to place more trash cans!

Drag the **Prefabs \ Map1** prefab into the scene. Reset its **Transform.** Also add a new **Empty Child** to **Map1**, name it **Interactables** and reset its **Transform**.



Make **TrashCan** a child of **Interactables**. Set its position to `(X:25, Y:0, Z:-1)`.

Duplicate the **TrashCan** GameObject and position this second instance to `(X:46, Y:0, Z:1.27)`.

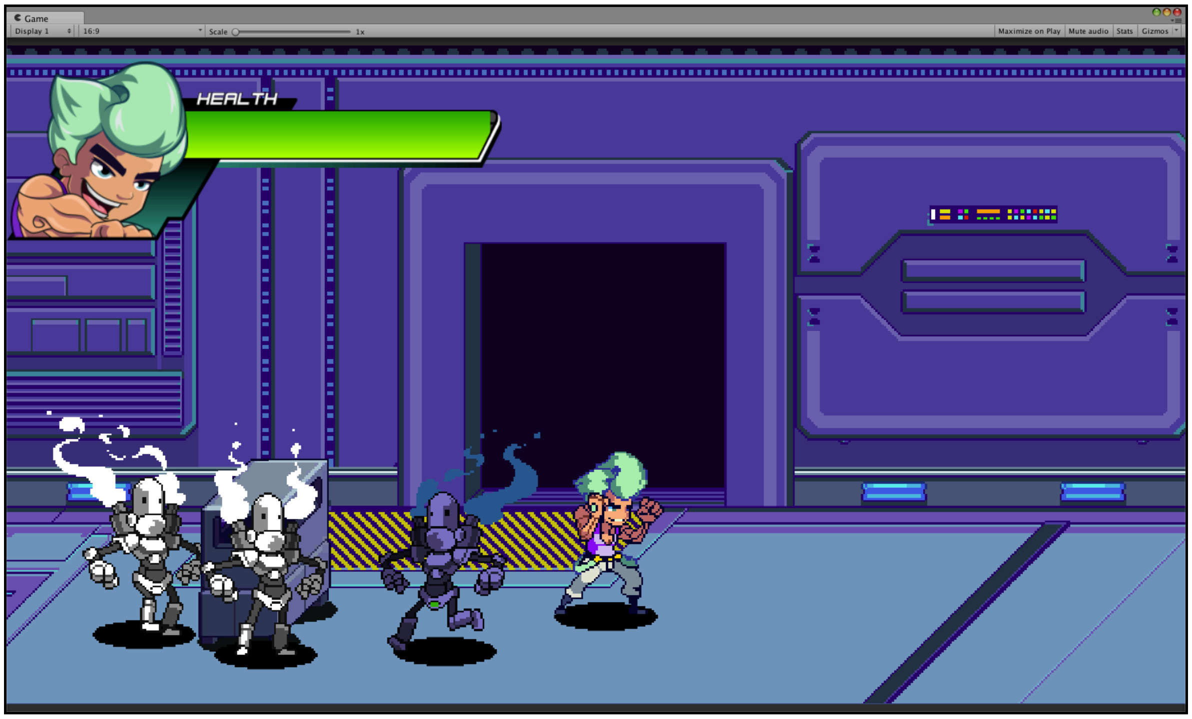


Select **Map1** in the Hierarchy and click **Apply**. Once saved, delete the Map1 instance.

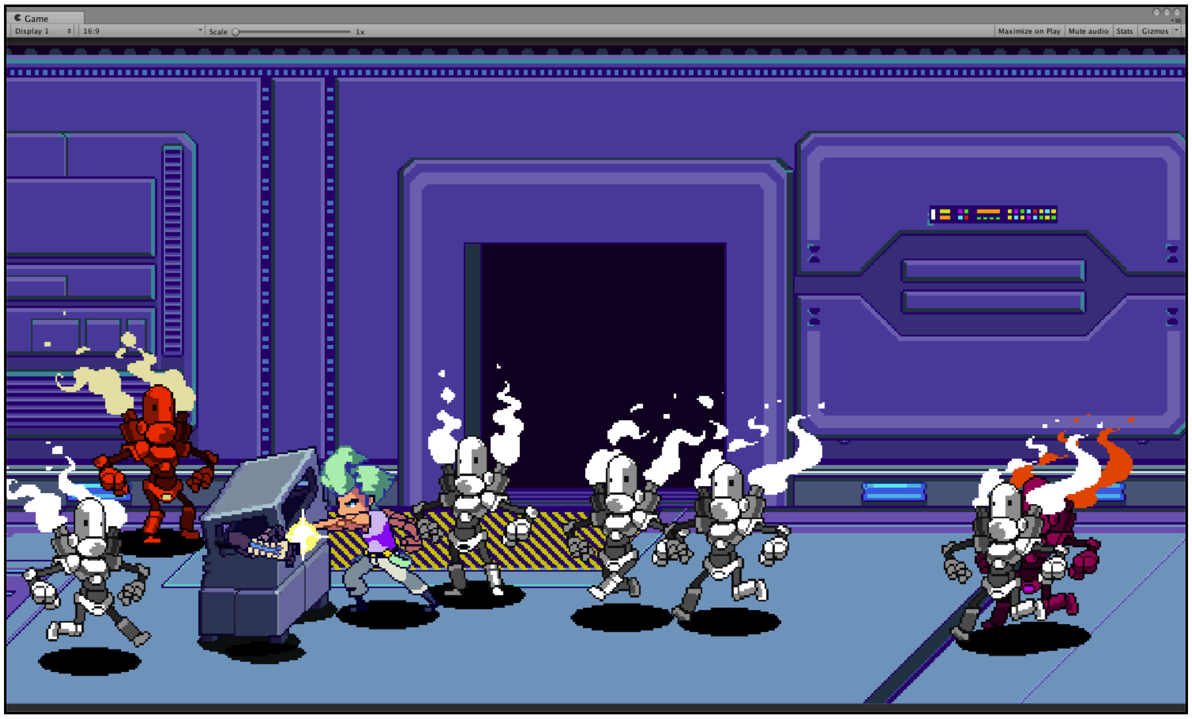
Delete all instances of **POW300** and **TrashCan** that are left in the scene as well.

Save the scene and project and press **Play**.

Note the conveniently placed trash can near the first battle event. The droids walk around the trash can, exactly as you designed!



Break the trash can to get a POW300 and smoke those dreadful droids!



Where to go from here?

Congratulations! You made PompaDroid more challenging with the addition of a boss and the hero's first powerup.

Perhaps the biggest thing you did in this chapter was build on top of your already functional game! Your work in this chapter was largely appending and overriding classes and methods, and it evolved the game a bit to make it more interesting. You added:

- A big, bad, boss
- A powerup to allow the hero to beat the boss: the POW300 (Punch-a-Tron Operational Work Gloves)
- Breakable containers for the powerup

You're almost ready to ship it! But don't even think about a release until you add audio.

PompaDroid needs some catchy background music, sound effects and a bit more polish. Any guesses about what you'll do in chapter 11?

Chapter 11: Audio and Final Touches

Take a moment to reflect on your progress. Some time ago, you started building this game with nothing but this book, your computer, and a jumble of assets. You may or may not have known much about programming, game design, or Unity. But obviously, you picked up this book because you wanted to learn how to build a game from the ground up. Consider that endeavor complete, because at this point, you have a functional game on your hands!

That said, if you were to ship the game as-is, you'd get a lot of, um, constructive criticism because of the odd little glitches and rough edges throughout. Players expect a high-quality game, even when they're not paying for it. One of the silliest things you can do is release a game before you've tested it and looked for easily solvable issues that could annoy your players — such as unwarranted restraint of movement, sound effect glitches, layering issues, funky colliders, etc. I think of this phase as "polishing" the game.

Since audio is pretty easy to implement, this chapter will include adding sound effects as well as polishing the game. In this (almost final) chapter, you'll:

- Add background music and sound effects.
- Add intro/outro text and a splash loading screen.
- Squash some annoying bugs to improve the player experience.
- Polish a few things here and there to improve quality of the final product.

By the end of this chapter, PompaDroid should be complete!

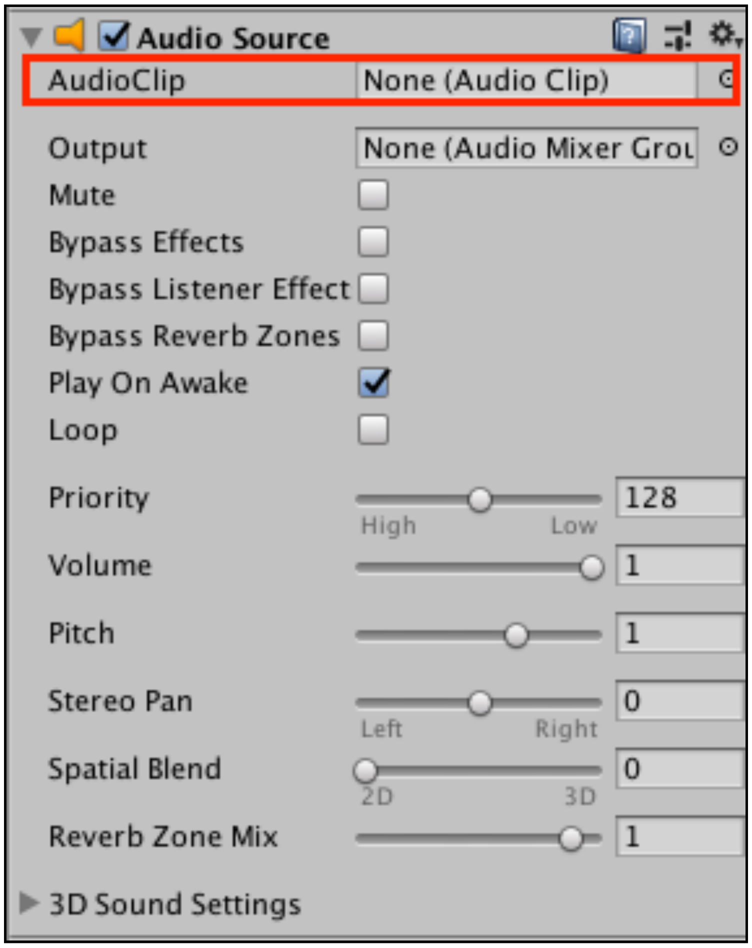
The importance of sound

PompaDroid is an action-packed game. As it stands right now, the game's fun and visually appealing in a retro kind of way.

Beauty is only skin deep, they say, and games are a multi-sensory experience. So any game worth playing will appeal to the ears at least as much as the eyes. Audio adds depth and immerses the player in the game-world you've created.

Overview: Unity's audio toolkit

Unity has an **AudioSource** component that handles and plays audio files. You just have to plug in logic, assets and references for each **audio clip** you want to play.


Audio Source

AudioClip: None (Audio Clip)

Output: None (Audio Mixer Grou

Mute: ☐

Bypass Effects: ☐

Bypass Listener Effect: ☐

Bypass Reverb Zones: ☐

Play On Awake: ✓

Loop: ☐

Priority: 128

Volume: 1

Pitch: 1

Stereo Pan: 0

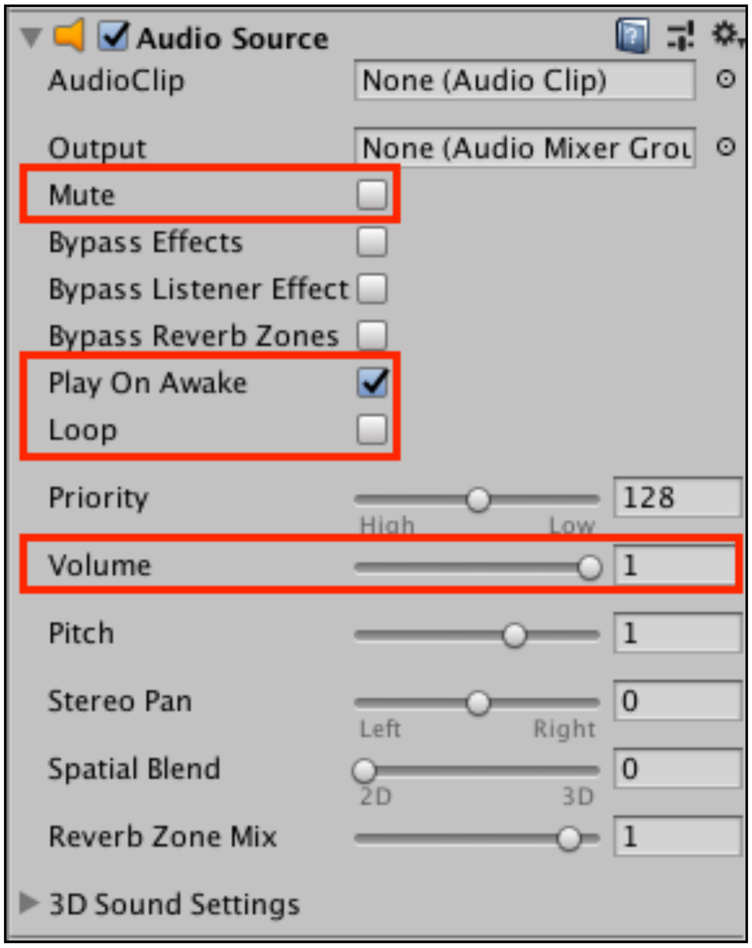
Spatial Blend: 0

Reverb Zone Mix: 1

3D Sound Settings

You have a number of variables to work with for each **AudioSource**, most notably:

- **Mute**: silences the AudioSource when checked.
- **Play On Awake**: plays the AudioClip upon creation of the component.
- **Loop**: repeats the playback of the AudioClip.
- **Volume**: adjusts how loudly the clip plays.



AudioClip: None (Audio Clip)
Output: None (Audio Mixer Grou
Mute: ☐
Bypass Effects: ☐
Bypass Listener Effect: ☐
Bypass Reverb Zones: ☐
Play On Awake: ✓
Loop: ☐
Priority: 128
Volume: 1
Pitch: 1
Stereo Pan: 0
Spatial Blend: 0
Reverb Zone Mix: 1

Note: Unity's documentation is the best place to learn about AudioSource and its variables. I'm about to brief you on it, but if you must peek right now, please go ahead. I'll wait for you right here. https://docs.unity3d.com/Manual/class-AudioSource.html.

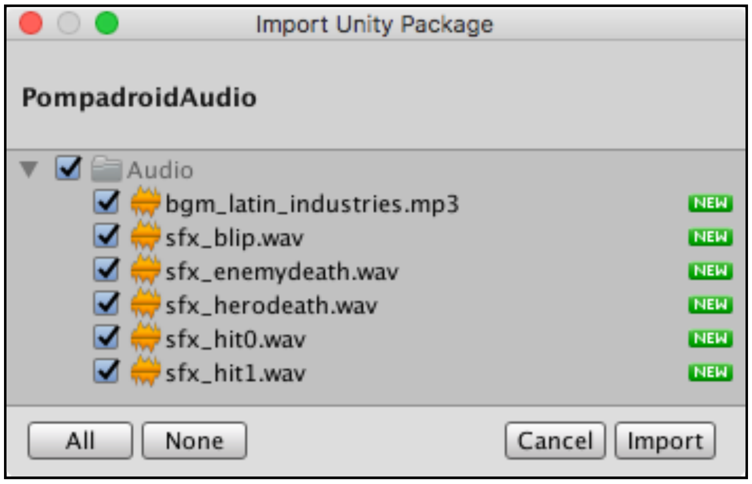
Every audio source needs a **listener**, and in the case of Unity, it is the **AudioListener**.

Similar to how a Camera component renders what it sees, an **AudioListener** plays what it hears. By default, Unity's cameras are created with an AudioListener component already attached. Makes sense, right?

That's all you need to know about Unity's audio system to get started! PompaDroid is about to level up and transform into a real game. Just wait until you hear the catchy tunes.

Adding audio

Import the **PompadroidAudio.unitypackage** from the **UnityPackages** folder. Inside, you'll find a single background music file in .mp3 format, and multiple sound effects in .wav format.



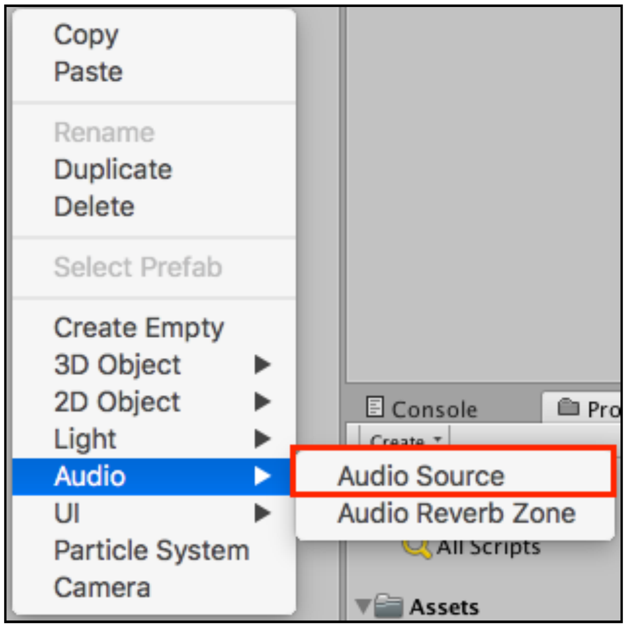
- ✓ bgm_latin_industries.mp3 NEW
- ✓ sfx_blip.wav NEW
- ✓ sfx_enemydeath.wav NEW
- ✓ sfx_herodeath.wav NEW
- ✓ sfx_hit0.wav NEW
- ✓ sfx_hit1.wav NEW

Note: Unity supports an array of audio file formats. For a full list, visit https://docs.unity3d.com/Manual/AudioFiles.html.

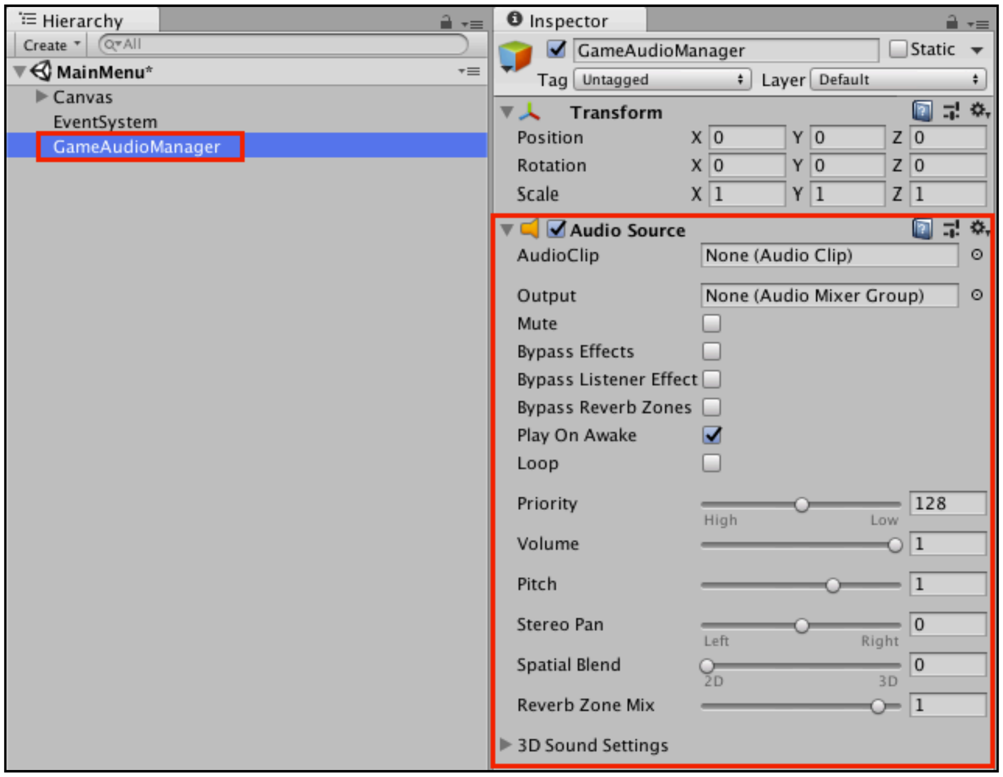
The audio file prefixed with **bgm_** will be your background music. It will loop during gameplay. Audio files prefixed with **sfx_** are sound effect files — short audio clips — that play when an action happens, for example, button clicks and player hits.

Play that funky music

Start by loading the **MainMenu** scene from the **Scenes** folder. Add a new **Audio/Audio Source** GameObject to the Hierarchy.



Rename it **GameAudioManager**. This AudioSource will play the background music of the game.

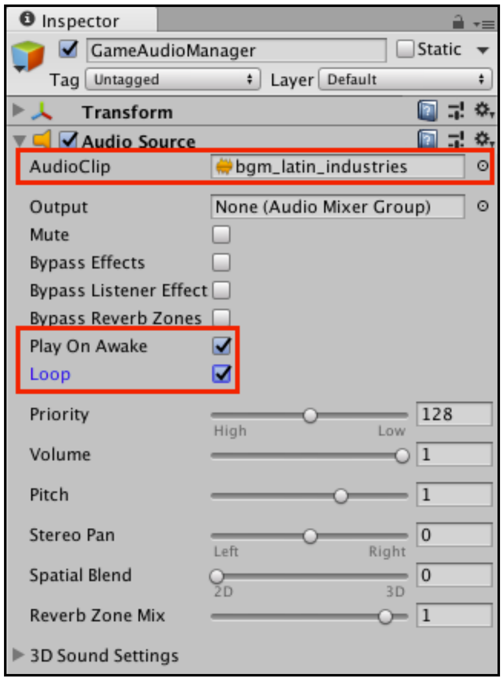


GameAudioManager
Tag Untagged Layer Default

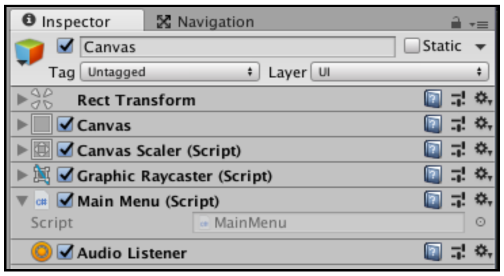
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Audio Source
AudioClip None (Audio Clip)
Output None (Audio Mixer Group)
Mute ☐
Bypass Effects ☐
Bypass Listener Effect ☐
Bypass Reverb Zones ☐
Play On Awake ✓
Loop ☐
Priority 128 (High – Low)
Volume 1
Pitch 1
Stereo Pan 0 (Left – Right)
Spatial Blend 0 (2D – 3D)
Reverb Zone Mix 1

Set **bgm_latin_industries** as its **AudioClip** and check both **Play On Awake** and **Loop**.


Inspector
GameAudioManager Static
Tag Untagged Layer Default
Transform
Audio Source
AudioClip bgm_latin_industries
Output None (Audio Mixer Group)
Mute ☐
Bypass Effects ☐
Bypass Listener Effect ☐
Bypass Reverb Zones ☐
Play On Awake ✓
Loop ✓
Priority 128 (High – Low)
Volume 1
Pitch 1
Stereo Pan 0 (Left – Right)
Spatial Blend 0 (2D – 3D)
Reverb Zone Mix 1
3D Sound Settings

Next, you'll need an AudioListener in the MainMenu scene since it doesn't have a camera. Select the **Canvas** and add an **AudioListener** component to it.


Inspector Navigation
Canvas Static
Tag Untagged Layer UI
Rect Transform
Canvas
Canvas Scaler (Script)
Graphic Raycaster (Script)
Main Menu (Script)
Script MainMenu
Audio Listener

Save the scene and click **Play**. Voila! Background music added. It's working but you might have noticed that when you touched the screen to start, the music stopped abruptly.

Theoretically, you could add an AudioSource to the Game scene, but that would result in the music starting over when you play the game. A better option is to retain the AudioSource in the MainMenu scene when the Game scene loads, so that the *only* time the music repeats from the start is when the hero dies and the game resets. For that to happen, you'll need custom logic to prevent the **GameAudioManager** from being destroyed when switching scenes.

Create a new C# script named **AudioManager.cs** in the Scripts folder. Replace its contents with:

```
using UnityEngine;
```

```
public class AudioManager : MonoBehaviour {
  //1
  public static AudioManager Instance;
  void Awake() {
    //2
    if (Instance == null) {
      Instance = this;
      DontDestroyOnLoad(gameObject);
    } else {
      //3
      if (Instance != this) {
        Destroy(gameObject);
      }
    }
  }
}
```

Your new AudioManager class acts as a **Singleton**: a programming design pattern that allows only one instance of a class to exist. Here is how it works:

1. First, create a `static` reference to AudioManager that points to the only existing instance of the AudioManager. As a static variable, you can access it from any class by calling `AudioManager.Instance`.

2. The `Awake` method checks if the static variable is assigned. If not, it sets the current AudioManager instance as the `Instance` variable and calls `DontDestroyOnLoad(gameObject)` to save the AudioManager from destruction when the game scene changes.

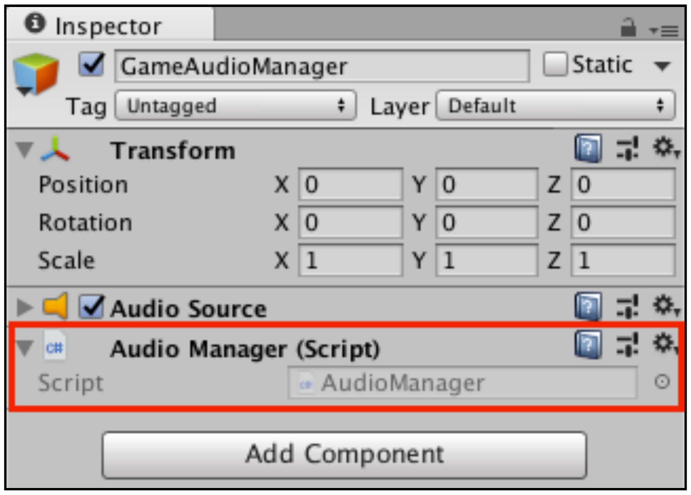
3. Alternatively, if `Instance` is not the same as the `AudioManager` instance, you destroy it because the game already has an instance of the AudioManager.

Save the script and open the **MainMenu** script. Append the `Start` method with the line below:

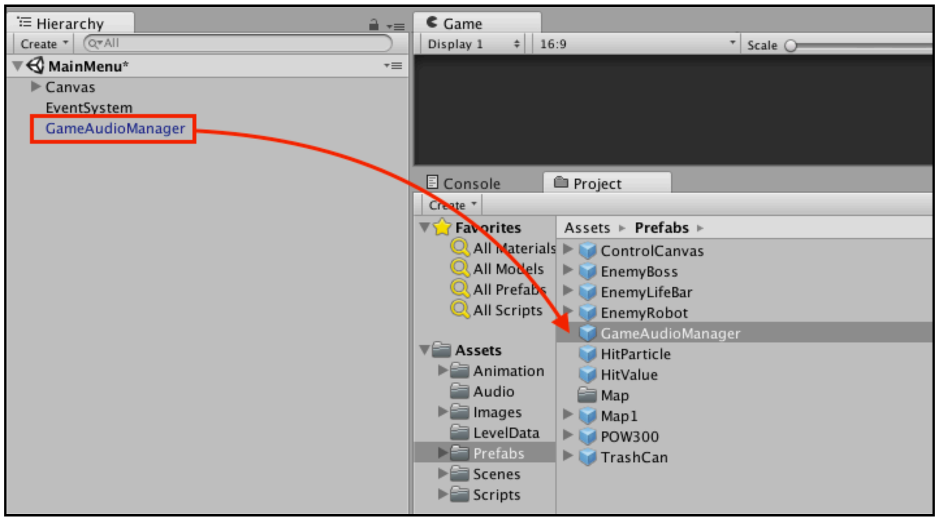
```
AudioManager.Instance.GetComponent<AudioSource>().Play();
```

With this, you reset the `AudioSource` component of `AudioManager` when the MainMenu scene loads.

Save the script and return to Unity. Add the **AudioManager** to the **GameAudioManager** GameObject.

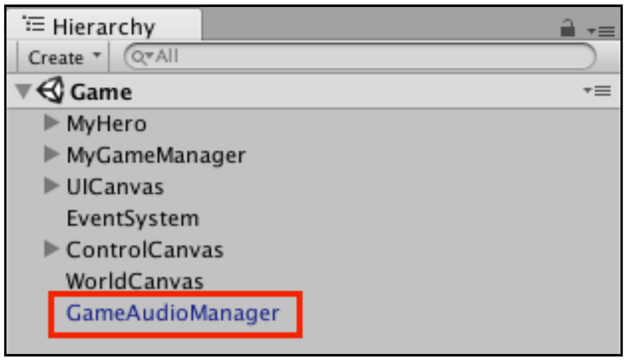


Drag **GameAudioManager** onto the **Prefabs** folder to create a prefab.



Save the scene and open the **Game** scene from the **Scenes** folder.

Drag the **GameAudioManager** prefab to the Hierarchy. **Save** the scene and navigate back to the MainMenu scene.



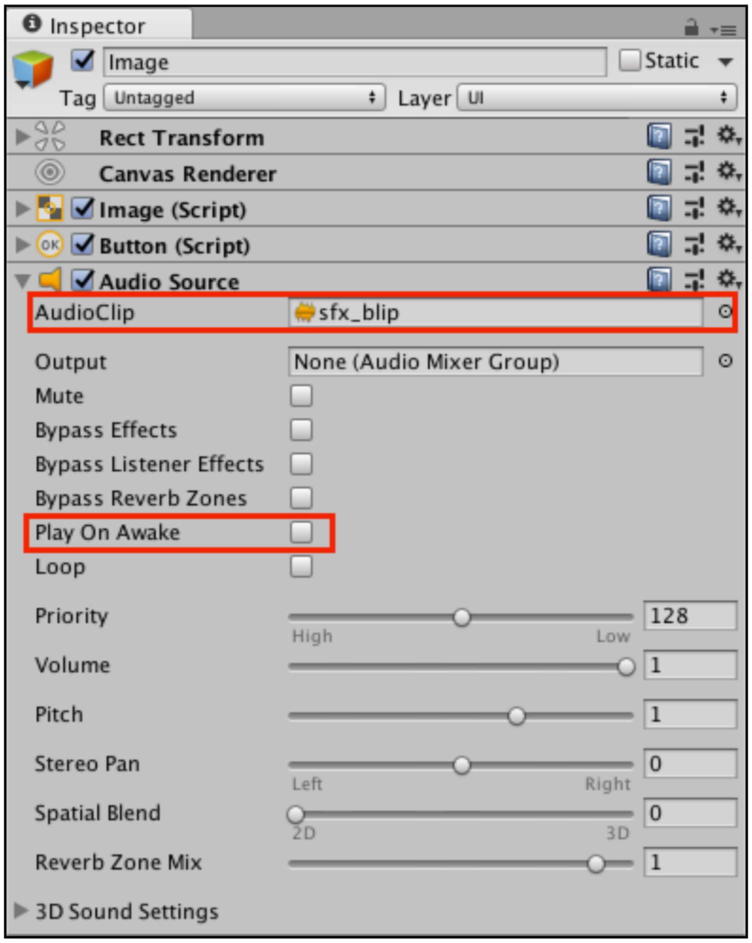
Run and play the game. Even when loading other scenes, the background music should keep playing. Boom. Audio track done!

Next, you'll add sound effects to make the game feel alive.

Adding sound effects

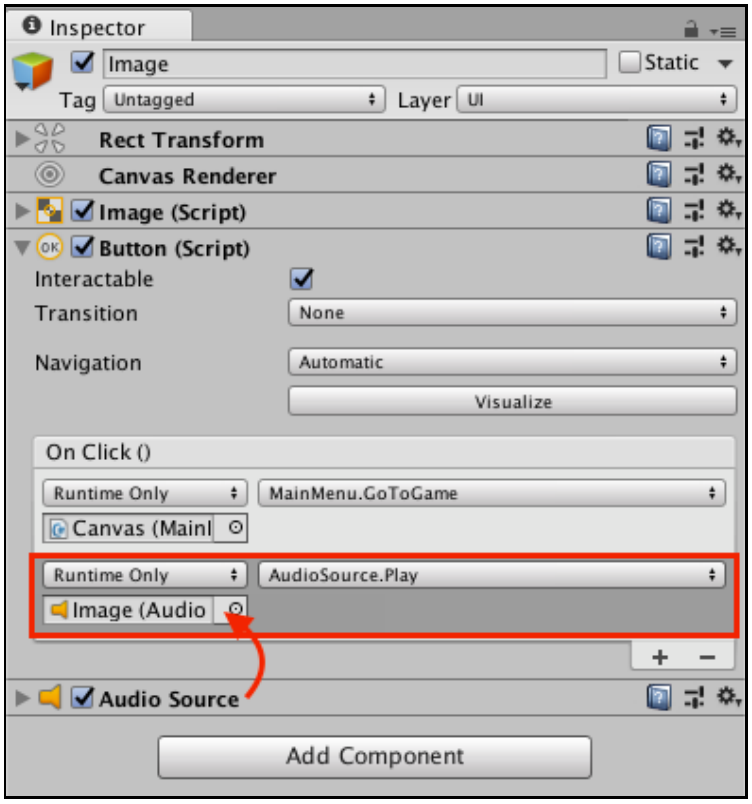
Sound effects add depth and improve the player's experience. Even subtle pops, clicks and dings add up to a more immersive, satisfying game. It pays off to be thoughtful when it comes to the audio environment! You'll start by adding an effect for the start button.

Expand **Canvas** in the Hierarchy and select its child **Image**. Add an **AudioSource** to it. Set **sfx_blip** as the **AudioClip** and uncheck **Play On Awake**.



Inspector
Image ✓ Static ☐
Tag Untagged Layer UI
Rect Transform
Canvas Renderer
Image (Script) ✓
Button (Script) ✓
Audio Source ✓
AudioClip sfx_blip
Output None (Audio Mixer Group)
Mute ☐
Bypass Effects ☐
Bypass Listener Effects ☐
Bypass Reverb Zones ☐
Play On Awake ☐
Loop ☐
Priority 128 (High – Low)
Volume 1
Pitch 1
Stereo Pan 0 (Left – Right)
Spatial Blend 0 (2D – 3D)
Reverb Zone Mix 1
3D Sound Settings

Still working with **Image**, find the **Button** component and add a new **On Click()** event to it. Drag the **AudioSource** to the **Object** field and select the **AudioSource\Play()** method. This should play a *blip* when the player touches the screen to start the game.



That's it for the MainMenu scene. **Save** the scene.

Next up is spicing up punches and deaths. But first, you need some logic. Open **Actor.cs** and add these variables below `hitValuePrefab`:

```
public AudioClip deathClip;
public AudioClip hitClip;

public AudioSource audioSource;
```

The first two variables reference the clips to play when the Actor punches or dies. The third variable will serve as a reference to the AudioSource responsible for playing these sound clips.

Add this method underneath `ShowHitEffects`:

```
public void PlaySFX(AudioClip clip) {
  audioSource.PlayOneShot (clip);
}
```

Your new helper method, `PlaySFX`, plays the `clip` parameter when called. It uses the `audioSource.PlayOneShot` method, which plays the sound clip once.

Append the end of the `Die` method with:

```
PlaySFX (deathClip);
```

Now `Die` will play the `deathClip` when the Actor dies.

Add this to the end of the `HitActor` method:

```
PlaySFX (hitClip);
```

Now `HitActor` plays the appropriate sound effect when one Actor hits another.

Save Actor.cs and open **Hero.cs**. You need to add the same logic to Hero since it overrides Actor's `HitActor` method.

In the `HitActor` method, add this line beneath `AnalyzeNormalAttack` inside the `if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1"))` condition and also below `AnalyzeNormalAttack` inside the `(baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack2"))` condition:

```
PlaySFX (hitClip);
```

For reference, the two insertions inside the two conditionals are shown in the snippet below:

```
if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack1")) {
  AttackData attackData = hasWeapon ? currentPowerup.attackData1
    : normalAttack;
  AnalyzeNormalAttack (attackData, 2, actor, hitPoint,
    hitVector);
  //1
  PlaySFX (hitClip);
  if (hasWeapon) {
    currentPowerup.Use();
  }
} else if
  (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack2")) {
  AttackData attackData = hasWeapon ? currentPowerup.attackData2
    : normalAttack2;
  AnalyzeNormalAttack (attackData, 3, actor, hitPoint,
    hitVector);
  //2
  PlaySFX (hitClip);
  if (hasWeapon) {
    currentPowerup.Use();
  }
}
```

These insertions will play the hit audio clip when the hero punches with the 1-2 combo and hits an enemy. You may be wondering why just the two, and not all of the hero's attacks. Well, this is because our pompadoured protagonist deserves a unique sound effect for his much stronger punches.

Still in `Hero.cs`, add the following variable beneath `powerupRoot`:

```
public AudioClip hit2Clip;
```

This variable will reference the special sound effect for strong attacks.

You need to play this sound effect in three places, once for each of the hero's three strong attacks.

In the `HitActor` method, add this line:

```
PlaySFX (hit2Clip);
```

to the following three locations:

1. Beneath `AnalyzeNormalAttack` inside the `else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack3"))` condition
2. Beneath `AnalyzeNormalAttack` inside the `else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_attack"))` condition
3. Beneath `AnalyzeNormalAttack` inside the `else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("run_attack"))` condition:

For reference, the three insertions are shown in the snippet below:

```
else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("attack3")) {
  AttackData attackData = hasWeapon ? currentPowerup.attackData3
    : normalAttack3;
  AnalyzeNormalAttack (attackData, 1, actor, hitPoint,
    hitVector);
  //1
  PlaySFX (hit2Clip);
  if (hasWeapon) {
    currentPowerup.Use();
  }
}
else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("jump_attack")) {
  AnalyzeSpecialAttack (jumpAttack, actor, hitPoint, hitVector);
  //2
  PlaySFX (hit2Clip);
}
else if (baseAnim.GetCurrentAnimatorStateInfo(0).IsName("run_attack")) {
  AnalyzeSpecialAttack (runAttack, actor, hitPoint, hitVector);
  //3
  PlaySFX (hit2Clip);
}
```

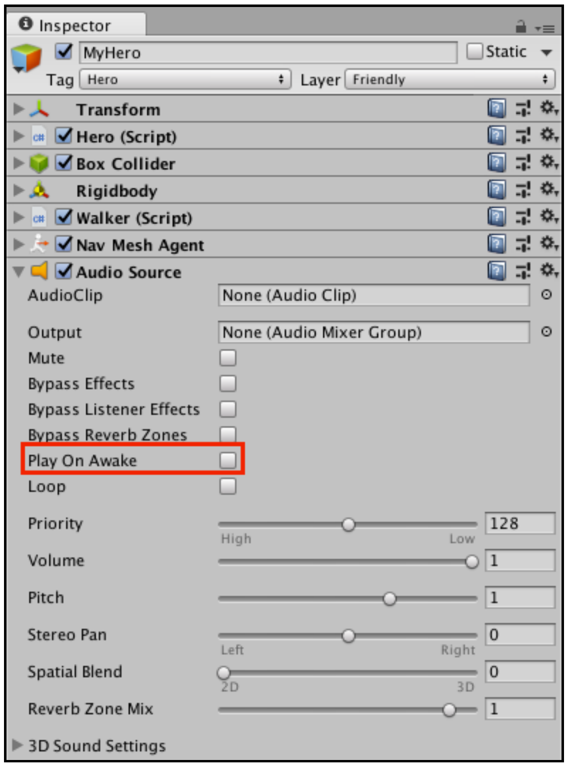
These insertions will play the stronger hit clip when the hero lands the third punch in a combo, the run attack or the jump attack.

Next, still in the `Hero` script, go to the `DidHitObject` method and add this line undearneath `containerObject.Hit(hitPoint);`:

```
PlaySFX(hitClip);
```

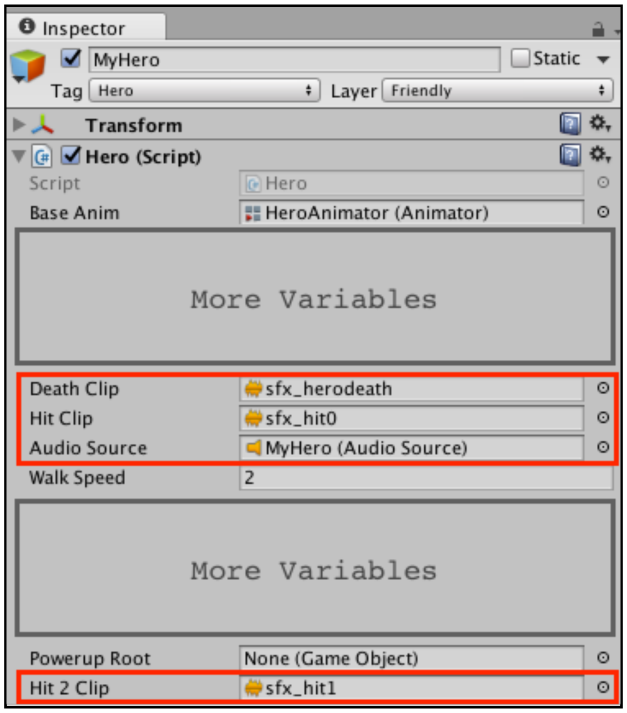
With that, the hit sound effect will play when the hero hits the trash can, or any other object you may add in the future.

Save the script and return to Unity to set up references to all that beautiful code. Open the **Game** scene, select **MyHero** in the Hierarchy and add an **AudioSource** to it. Uncheck **Play On Awake** to prevent the component from playing when it is first created.



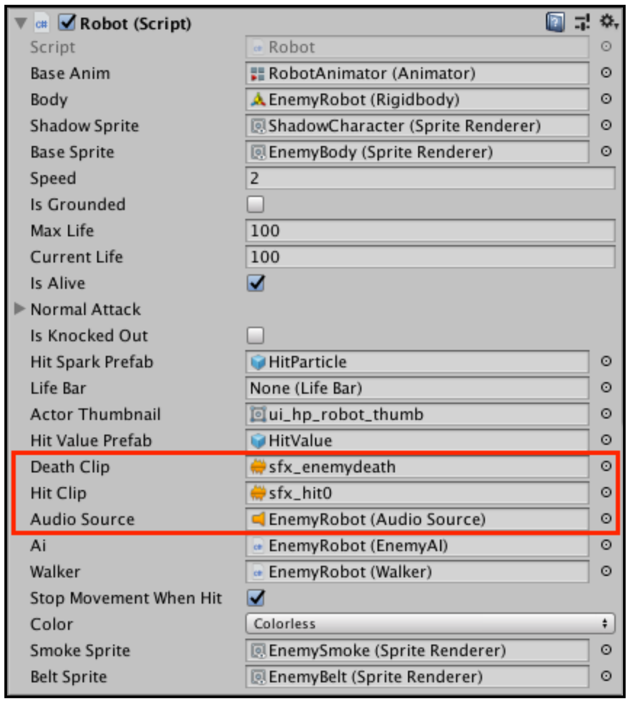
Next, you'll assign references for the Hero component.

Set **Death Clip** to **sfx_herodeath**, **Hit Clip** to **sfx_hit0**, and **Hit 2 Clip** to **sfx_hit1**. Drag the **Audio Source** you recently added to the **Audio Source** field. Be sure to fold in some of the components to make this easier.


Inspector
MyHero | Static
Tag: Hero | Layer: Friendly
Transform
Hero (Script)
Script: Hero
Base Anim: HeroAnimator (Animator)
More Variables
Death Clip: sfx_herodeath
Hit Clip: sfx_hit0
Audio Source: MyHero (Audio Source)
Walk Speed: 2
More Variables
Powerup Root: None (Game Object)
Hit 2 Clip: sfx_hit1

Your droids need some audio love too.

Select the **EnemyRobot** prefab in the **Prefabs** folder, add an **AudioSource** component to it and uncheck its **Play On Awake** field. In its **Robot** component, set **Death Clip** to **sfx_enemydeath**, **Hit Clip** to **sfx_hit0** and **Audio Source** to the one you just added.

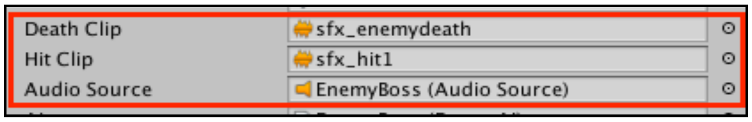

Robot (Script)
Script: Robot
Base Anim: RobotAnimator (Animator)
Body: EnemyRobot (Rigidbody)
Shadow Sprite: ShadowCharacter (Sprite Renderer)
Base Sprite: EnemyBody (Sprite Renderer)
Speed: 2
Is Grounded: ☐
Max Life: 100
Current Life: 100
Is Alive: ✓
Normal Attack
Is Knocked Out: ☐
Hit Spark Prefab: HitParticle
Life Bar: None (Life Bar)
Actor Thumbnail: ui_hp_robot_thumb
Hit Value Prefab: HitValue
Death Clip: sfx_enemydeath
Hit Clip: sfx_hit0
Audio Source: EnemyRobot (Audio Source)
Ai: EnemyRobot (EnemyAI)
Walker: EnemyRobot (Walker)
Stop Movement When Hit: ✓
Color: Colorless
Smoke Sprite: EnemySmoke (Sprite Renderer)
Belt Sprite: EnemyBelt (Sprite Renderer)

> **Note**: There is no need to save the prefab because you modified the prefab itself — not an instance in the scene.

The boss deserves unique audio, don't you think?

Select it, add an **AudioSource** component and uncheck **Play On Awake**. In its **Boss** component, set **Death Clip** to **sfx_enemydeath**, **Hit Clip** to **sfx_hit1** and **Audio Source** to the audio source you just added.

To emphasize the Boss' punches' impact, you give him the stronger sounding **sfx_hit1**.



Death Clip	sfx_enemydeath
Hit Clip	sfx_hit1
Audio Source	EnemyBoss (Audio Source)

Save the scene and project then click **Play**. Punch something. Now punch something else then let yourself take a punch. Sound effects make it so much more believable!

Great! Audio is now integrated into the game, serenading you with a catchy tune while you trash those droids!

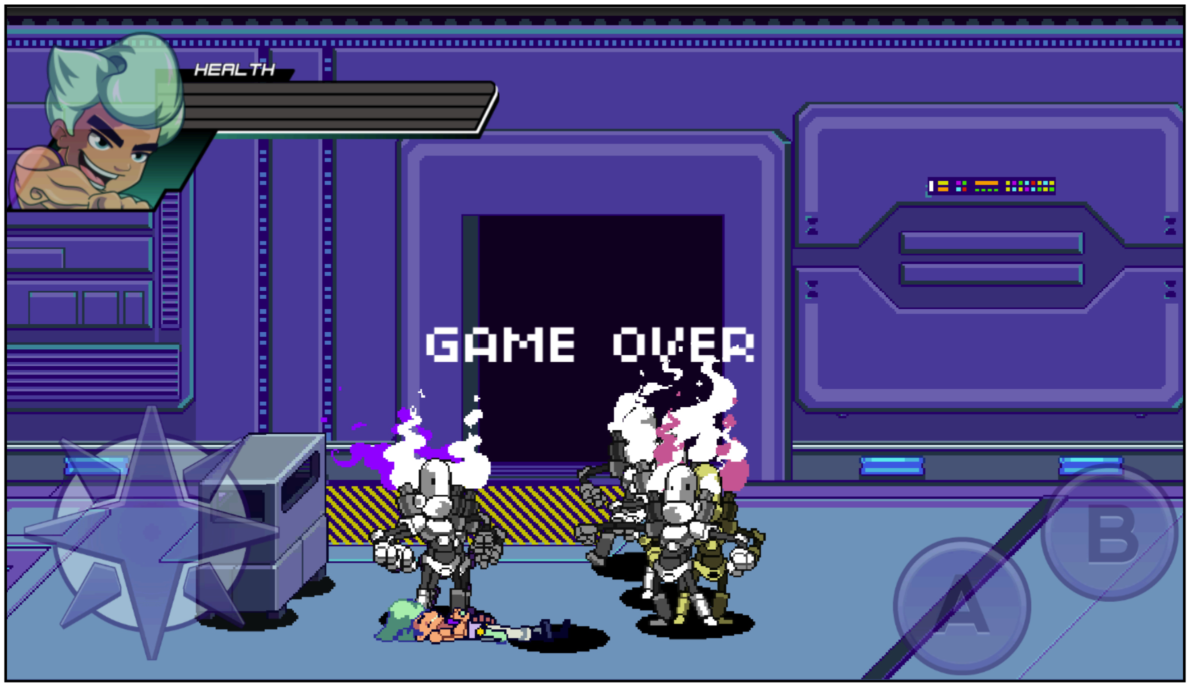
At this point, your game is nearly complete but not ready to ship. Before you can hang your hat on this game, you need to do some polishing and bug fixing. From this point forward in this chapter, you'll be cleaning things up and adding little touches that enhance the game.

Polish, admire, repeat

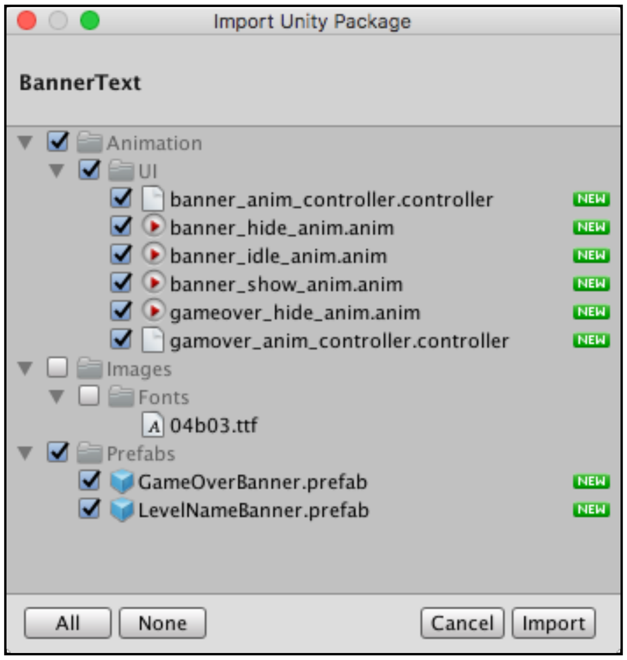
Players appreciate knowing what level they're playing, but PompaDroid doesn't give any clues to the player. Not even an obligatory **Game Over** message when the droids prevail. Don't leave the player hanging!

The first bit of polish is adding in prompts to tell the player which level they're on and when the level has ended.

You'll implement this by adding some level title text to the beginning and end of each level, and an ominous "GAME OVER" message when the hero dies.



Import **BannerText.unitypackage** to get the assets you'll need to create these text marquees.


Import Unity Package

BannerText

```
▼ ☑ Animation
  ▼ ☑ UI
      ☑ banner_anim_controller.controller     NEW
      ☑ banner_hide_anim.anim                 NEW
      ☑ banner_idle_anim.anim                 NEW
      ☑ banner_show_anim.anim                 NEW
      ☑ gameover_hide_anim.anim               NEW
      ☑ gamover_anim_controller.controller    NEW
▼ ☐ Images
  ▼ ☐ Fonts
        04b03.ttf
▼ ☑ Prefabs
    ☑ GameOverBanner.prefab                   NEW
    ☑ LevelNameBanner.prefab                  NEW
```

You'll see it contains two incomplete prefabs: a **GameOverBanner**, a prefab for the banner marquee when the hero dies, and a **LevelNameBanner** prefab, to be displayed at the beginning and end of each level.

To add these into the game, create a new **C# script** named **LoadMainMenu.cs** in the **Scripts** folder and open it. Replace its contents with the following code:

```
using UnityEngine;
using UnityEngine.SceneManagement;

public class LoadMainMenu : MonoBehaviour {
  public void GameOverDone() {
    SceneManager.LoadScene("MainMenu");
  }
}
```

The LoadMainMenu script will load the MainMenu scene when the `GameOverDone` method is called.

Save the script and open **GameManager.cs**. You’ll need to add the code that shows the banner texts when the hero dies or when a level has been started or completed.

Start by adding the following `namespace` import at the top of the file:

```
using UnityEngine.UI;
```

This allows the GameManager class to use `UI.Text` classes.

Next, add these variables underneath `bossPrefab`:

```
public GameObject levelNamePrefab;
public GameObject gameOverPrefab;

public RectTransform uiTransform;
```

Here you have references to the prefabs of the level name and the game over texts. It also creates a variable to hold a reference to the RectTransform that will serve as the parent of all UI elements in the GameManager.

Your next block goes below `FlickerGoIndicator`:

```
//1
private void ShowBanner(string bannerText, GameObject prefab) {
  GameObject obj = Instantiate(prefab);
  obj.GetComponent<Text>().text = bannerText;
  RectTransform rectTransform = obj.transform as RectTransform;
  rectTransform.SetParent(uiTransform);
  rectTransform.localScale = Vector3.one;
  rectTransform.anchoredPosition = Vector2.zero;
}

//2
public void GameOver() {
  ShowBanner("GAME OVER", gameOverPrefab);
}
```

```
//3
public void Victory() {
  ShowBanner("YOU WON", gameOverPrefab);
}

//4
public void ShowTextBanner(string levelName) {
  ShowBanner(levelName, levelNamePrefab);
}
```

Each of these methods serves a different purpose for the GameManager:

1. `ShowBanner` creates an instance of its prefab parameter and parents it to the class' `uiTransform` field. It also sets the title of the prefab's `Text` component to the value of the `bannerText parameter`.
2. `GameOver` is a helper method that instantiates the `gameOverPrefab` variable with the string "GAME OVER". You'll use this method when the player dies.
3. `Victory` is a helper method that instantiates the `gameOverPrefab` variable with the string "YOU WON". You'll use this when the player completes the game.
4. `ShowTextBanner` shows the `levelNamePrefab` with the value of the `levelName` parameter.

Find these in `DidFinishWalkout`:

```
Debug.Log("Game Completed!");
SceneManager.LoadScene("MainMenu");
```

And replace them with:

```
Victory();
```

With that, you put the victory banner on display when the player wins the last level.

Locate `AnimateNextLevel()` and replace its contents with this block:

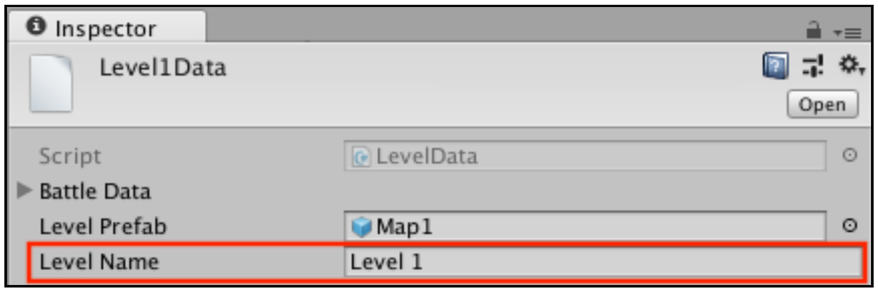
```
ShowTextBanner(currentLevelData.levelName + " COMPLETED");
yield return new WaitForSeconds(3.0f);
SceneManager.LoadScene("Game");
```

Here you show the text banner with a "LEVEL NAME COMPLETED" message when the last battle event of a level has been completed.

Locate `LoadLevelData` and append the end of the method with this line:

```
ShowTextBanner(currentLevelData.levelName);
```

That will show the current name of the level when the game starts. This name is defined as the **Level Name** parameter in the LevelData scriptable object.



Script: LevelData
Battle Data
Level Prefab: Map1
Level Name: Level 1

Save the script and open **Hero.cs** so that you can call the `GameOver` method when the hero dies.

Start by placing this variable below `hit2clip`:

```
public GameManager gameManager;
```

You've just added a reference to the GameManager in the Hero class.

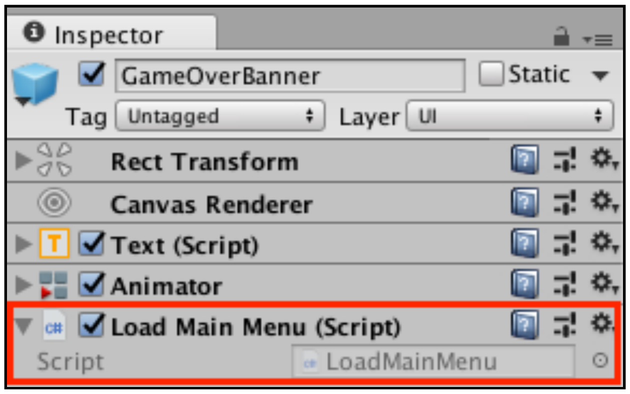
Add the following method to the bottom of the class:

```
protected override void Die() {
  base.Die();
  gameManager.GameOver();
}
```

This method calls `gameManager.GameOver` when the hero dies, displaying the game over text on screen.

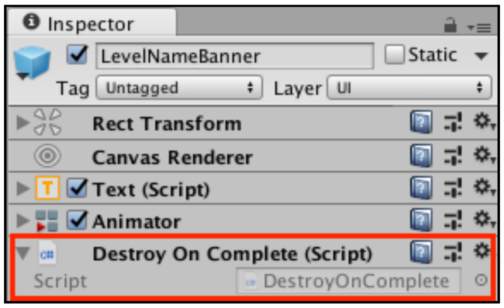
That should do it, **save** the scripts and return to Unity to play around with prefabs and references and get the text working.

Select the **GameOverBanner** prefab in the **Prefabs** folder and add a **LoadMainMenu** component to it. Since you've added the component to the prefab, all instances of the the prefab now contain the component.

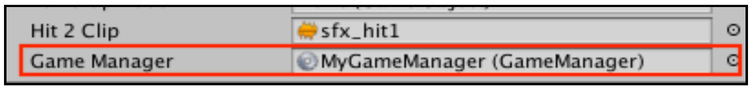


GameOverBanner
Static
Tag: Untagged
Layer: UI
Rect Transform
Canvas Renderer
Text (Script)
Animator
Load Main Menu (Script)
Script: LoadMainMenu

Now select **LevelNameBanner** in the **Prefabs** folder and add a **DestroyOnComplete** component to it.



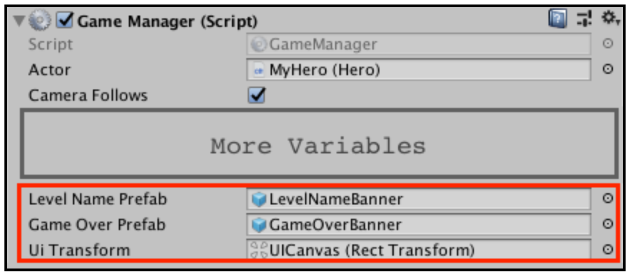
Select **MyHero** in the **Hierarchy**. Find the **Game Manager** field of the **Hero** component and drag **MyGameManager** onto the field.



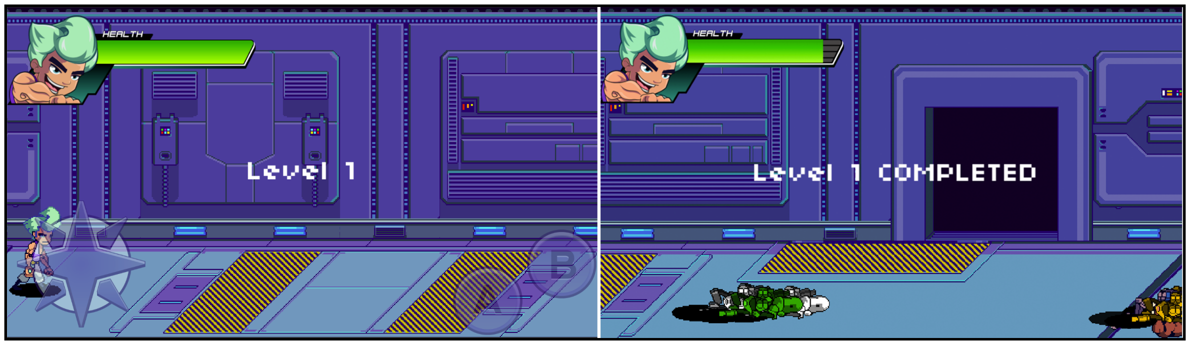
Select **MyGameManager** in the **Hierarchy**. Find the following items in the **Prefabs** folder and drag them to the corresponding fields in the **GameManager** component of **MyGameManager**.

- **LevelNameBanner** to **Level Name Prefab**
- **GameOverBanner** to **Game Over Prefab**

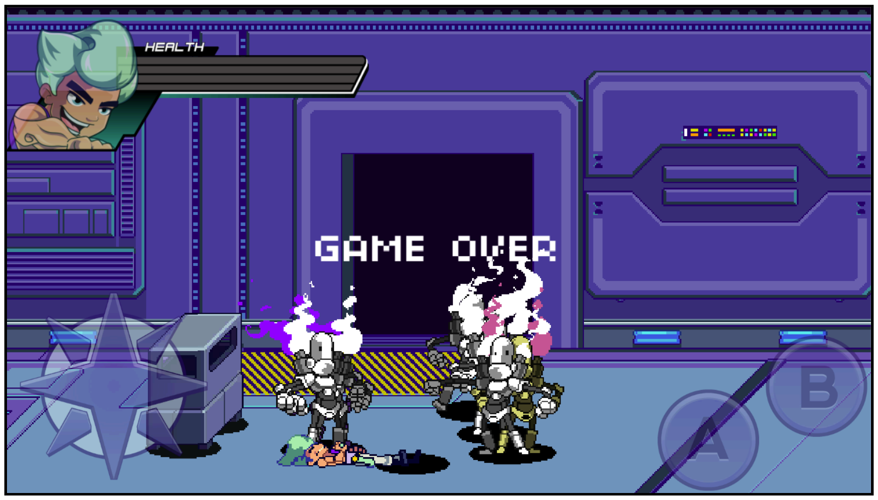
Finally, change the value of **UI Transform** to **UICanvas**.


Game Manager (Script)
Script GameManager
Actor MyHero (Hero)
Camera Follows ✓
More Variables
Level Name Prefab LevelNameBanner
Game Over Prefab GameOverBanner
Ui Transform UICanvas (Rect Transform)

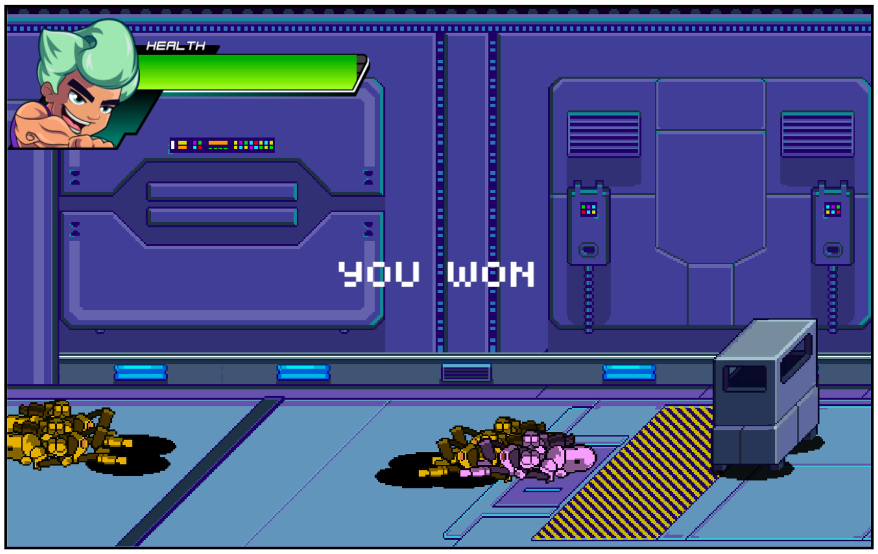
Save the scene and the project, and play through a level. Fancy! You can't miss the start and end of the levels anymore.



A bold GAME OVER message shows up when you lose. Once the animation finishes, the game kicks you back to the main menu.

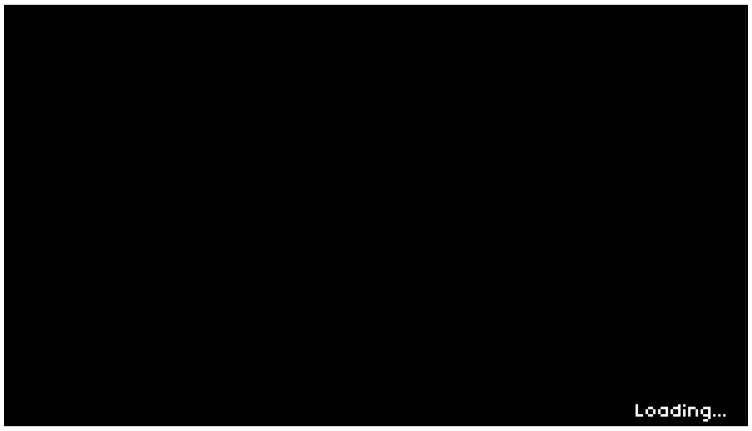


Make sure you play until you defeat the last boss on the last level to see YOU WON before you return to the main menu. Neat! PompaDroid is looking a bit sharper already.

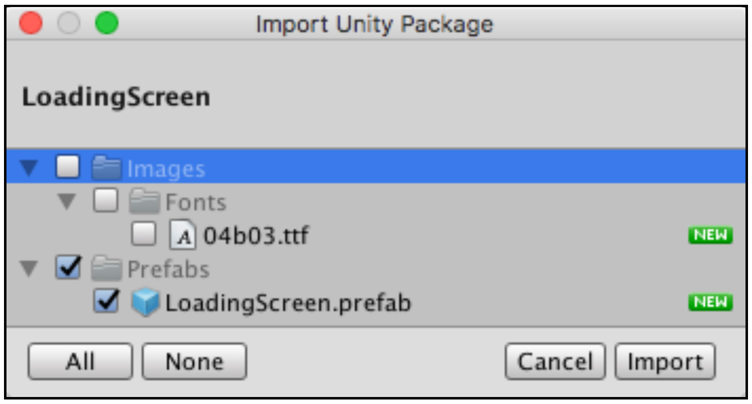


Turn your attention to the transition between the main menu and the game — it's a little jarring without some kind of loading indicator. You know how players hate to wait for no apparent reason. :]

Your polish will be simple but effective: a solid black overlay that displays "Loading..." in the bottom-right corner.



Import **LoadingScreen.unitypackage**, which contains the **LoadingScreen** prefab.



Of course, you'll need to add logic to get the loading screen to show up, so open **GameManager.cs** and add this variable declaration below `uiTransform`:

```
public GameObject loadingScreen;
```

This variable stores a reference to the loading screen.

Add this `Awake` method right below `Start()`:

```
void Awake() {
  loadingScreen.SetActive(true);
}
```

Here you show the `loadingScreen` upon creation of the GameManager.

Insert this at the end of the `LoadLevelData` method:

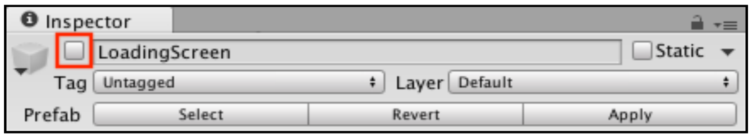
```
loadingScreen.SetActive(false);
```

Now the final task of `LoadLevelData` is to hide the loading screen.

Save the script and return to Unity.

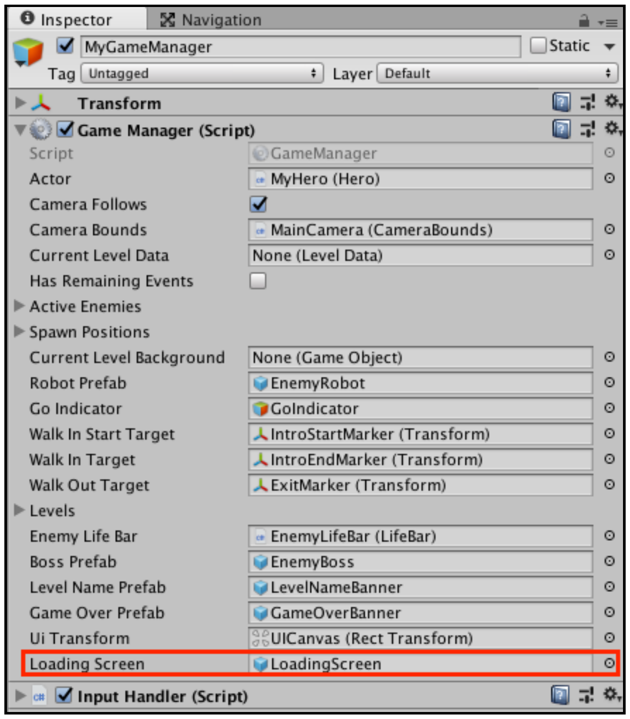
From the **Prefabs** folder, drag the **LoadingScreen** prefab into the Hierarchy to create an instance. Make sure to disable the GameObject of the **LoadingScreen** — it should be hidden by default.

Note: You may be wondering why you just disabled the loading screen when GameManager's `Awake()` method loads it. Allow me to explain: It's simply easier to work with the Game scene. Otherwise, you'd have to enable the loading screen every single time you test and build the game.



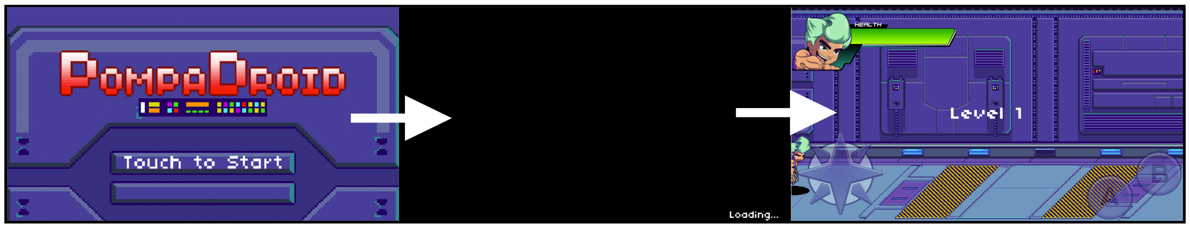
LoadingScreen
Static
Tag Untagged
Layer Default
Prefab

Drag the **LoadingScreen** you just added over to **MyGameManager** and into the **LoadingScreen** field.



MyGameManager
Static
Tag Untagged
Layer Default
Transform
Game Manager (Script)
Script GameManager
Actor MyHero (Hero)
Camera Follows ✓
Camera Bounds MainCamera (CameraBounds)
Current Level Data None (Level Data)
Has Remaining Events
Active Enemies
Spawn Positions
Current Level Background None (Game Object)
Robot Prefab EnemyRobot
Go Indicator GoIndicator
Walk In Start Target IntroStartMarker (Transform)
Walk In Target IntroEndMarker (Transform)
Walk Out Target ExitMarker (Transform)
Levels
Enemy Life Bar EnemyLifeBar (LifeBar)
Boss Prefab EnemyBoss
Level Name Prefab LevelNameBanner
Game Over Prefab GameOverBanner
Ui Transform UICanvas (Rect Transform)
Loading Screen LoadingScreen
Input Handler (Script)

Save the scene and the project. Open the **MainMenu** scene and run the game. Check that the "Loading..." text displays at the bottom right until the game loads.



Bling! PompaDroid is a little shinier than it was before. So far, you've added intro and exit texts, as well as a loading text. Don't forget about the audio! You're making excellent progress.

Squashing bugs

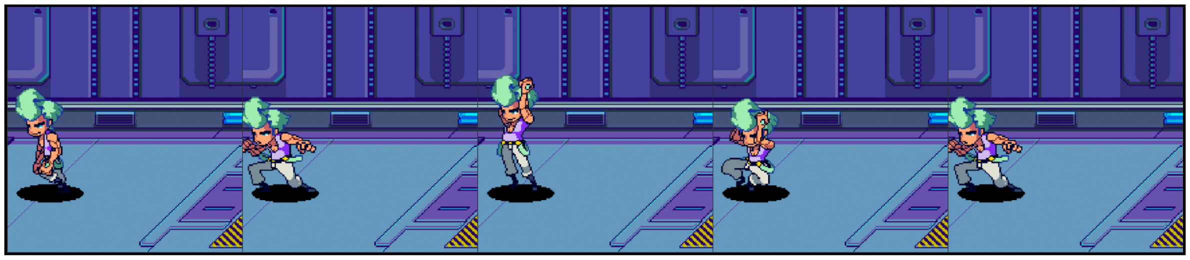
No doubt that you've noticed a few bugs as you've tested and played. Bugs are pretty much unavoidable when you're programming, so don't sweat it. Unexpected things can — and will — happen anytime you modify code. Remember that bugs can always be replicated and traced to specific blocks of code.

Note: Debugging can be a bit frustrating, but it's also kind of like a challenge. A treasure hunt. A noble quest for code domination, even. Over the next several steps, you'll fix some known bugs. In doing so, you might stumble across or even create new bugs.

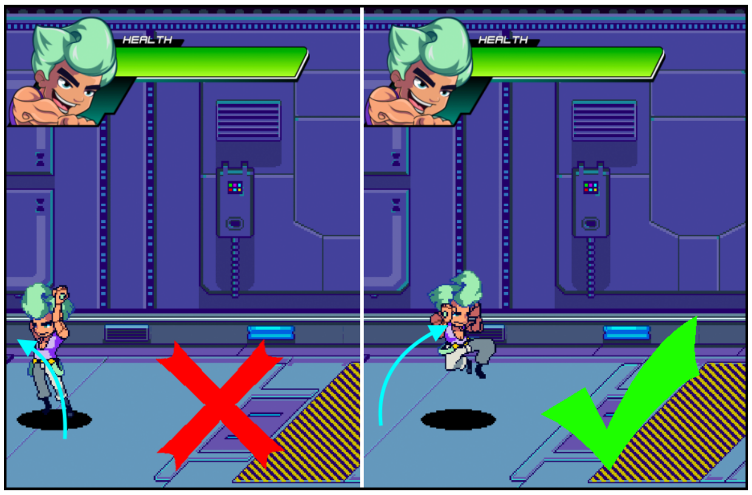
For now, just stick to the known bugs. If you feel the urge, you can come back later and see if you're able to replicate and correct any other peculiar behavior. Feel free to come talk about it in the forums!

Grounded jump bug

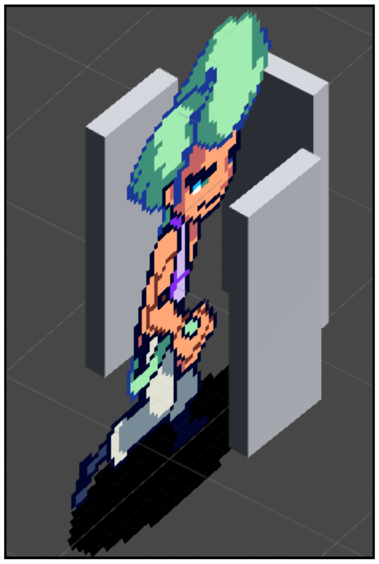
Currently, the hero has issues when jumping into a wall. Replicate it by holding the **left directional button** and walking toward the left-side wall in the game. While still holding the left button, press the **jump button**. The jump animation plays, but the hero doesn't gain any vertical height.



So you know the bug presents itself when the hero jumps toward a wall while adjacent to it, which means you can fix it by implementing logic that prevents jumping toward an adjacent wall but allows jumping in the opposite direction.



Trigger colliders that surround the hero on the front, near and far sides will address the underlying problem. When he's up next to a wall, the collider will aid with disabling the hero's ability to jump toward it. Below, three gray boxes depict the colliders you'll create:



Create a new **C# script** named **JumpColliderItem.cs** in the **Scripts** folder and open it. Replace its contents with this block:

```
using UnityEngine;
using System.Collections;

public class JumpColliderItem : MonoBehaviour {

  public int isTriggeredCount = 0;

  void OnTriggerEnter(Collider collider){
    isTriggeredCount++;
  }

  void OnTriggerExit(Collider collider){
    isTriggeredCount--;
  }
}
```

In here, you handle colliders that dare to overlap any walls. When an overlap occurs, `OnTrigger` increments or decrements the value of `isTriggeredCount`.

Save the script and create another **C# script** named **JumpCollider.cs**. Replace its contents with this:

```
using UnityEngine;
using System.Collections;

public class JumpCollider : MonoBehaviour {

  //1
  public JumpColliderItem frontCollider;
  public JumpColliderItem farCollider;
  public JumpColliderItem nearCollider;

  //2
  public bool CanJump(Vector3 direction, Vector3 frontVector ){
    if (direction.z > 0 && farCollider.isTriggeredCount > 0) {
      return false;
    } else if (direction.z < 0 && nearCollider.isTriggeredCount > 0) {
      return false;
    } else if (frontVector.x < 0 && direction.x < 0 &&
frontCollider.isTriggeredCount > 0) {
      return false;
    } else if (frontVector.x > 0 && direction.x > 0 &&
frontCollider.isTriggeredCount > 0) {
      return false;
    }
    return true;
  }
}
```

The `JumpCollider` class helps determine if a jump in a given direction is possible.

1. References to three `JumpColliderItem` colliders for the hero — one in front and one on each side.

2. `CanJump` tests if a jump's direction has a `JumpColliderItem` and that the value of `isTriggeredCount` is greater than zero. If true, it blocks the jump by returning `false`, otherwise the jump is allowed by returning `true`.

Save the script and open **Hero.cs** so you can disable jumping when the JumpCollider's `CanJump` method returns false.

Add the following variable to the top of the class, below the `gameManager` variable:

```
public JumpCollider jumpCollider;
```

Next, in the `Update` method, locate this `if` condition:

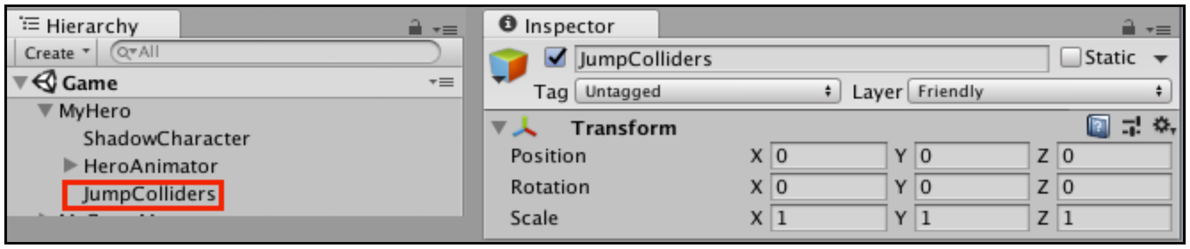
```
if (canJump && jump && !isKnockedOut &&
```

And replace it with this:

```
if (canJump && jump && !isKnockedOut &&
jumpCollider.CanJump(currentDir,frontVector) &&
```

You've added an additional condition to check the `CanJump` method from `jumpCollider` before allowing the hero to jump.

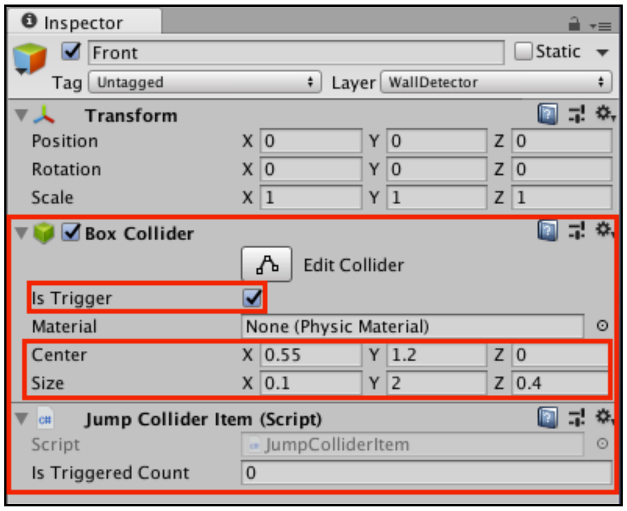
Next, **save** the script and return to Unity. Open the **Game** scene and add an **Empty GameObject** child to **MyHero** and name it **JumpColliders**. Reset its **Transform**.



Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

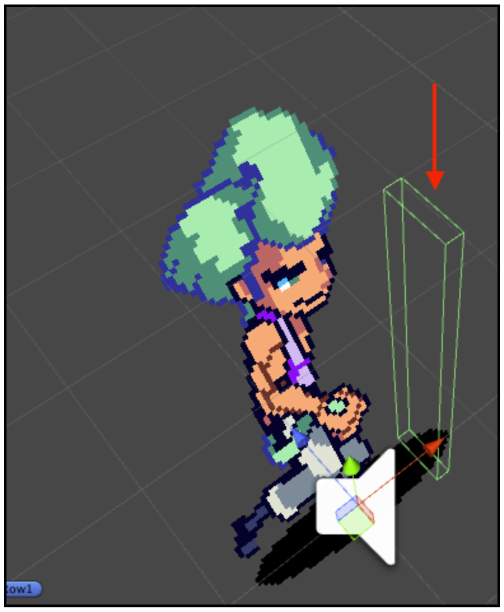
Then, add an **Empty GameObject** child to **JumpColliders** and name it **Front**. Reset its **Transform** and add a **Box Collider** to it.

Check **Is Trigger**, set its **Center** to `(X:0.55, Y:1.2, Z:0)` and **Size** to `(X:0.1, Y:2.0, Z:0.4)`. Also add a **JumpColliderItem** component to this GameObject.

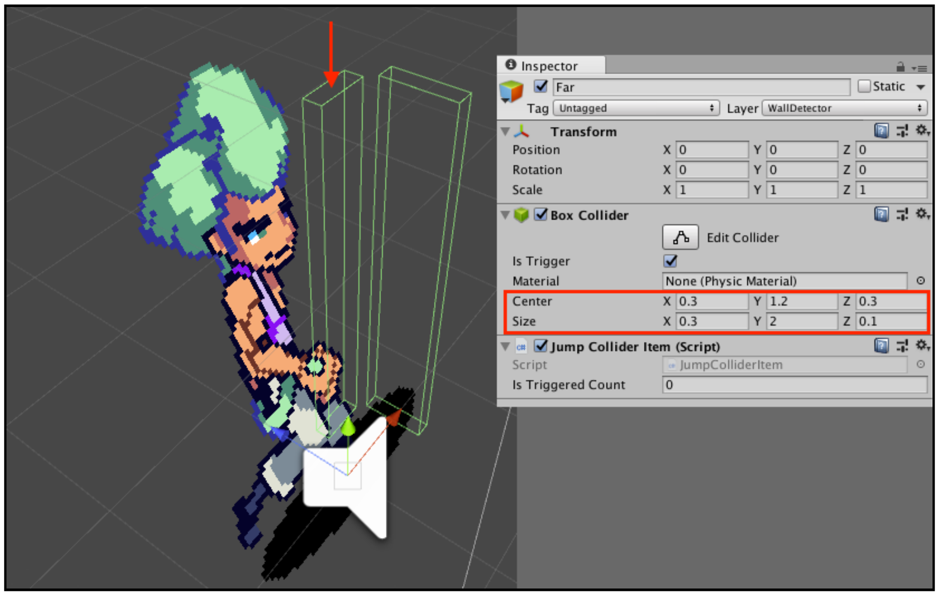


```
Inspector
[✓] Front                                   [ ] Static
Tag  Untagged                 Layer  WallDetector
Transform
Position            X 0       Y 0       Z 0
Rotation            X 0       Y 0       Z 0
Scale               X 1       Y 1       Z 1
[✓] Box Collider
                    Edit Collider
Is Trigger          [✓]
Material            None (Physic Material)
Center              X 0.55    Y 1.2     Z 0
Size                X 0.1     Y 2       Z 0.4
Jump Collider Item (Script)
Script              JumpColliderItem
Is Triggered Count  0
```

This should add the collider in front of the hero in the Scene view.



Duplicate the **Front** GameObject and rename the copy **Far**. Reposition it by setting its **Box Collider's Center** to `(X:0.3, Y:1.2, Z:0.3)` and **Size** to `(X:0.3, Y:2.0, Z:0.1)`. This will position the box collider on the far side of the hero.



Inspector

Far — Static

Tag Untagged — Layer WallDetector

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Box Collider

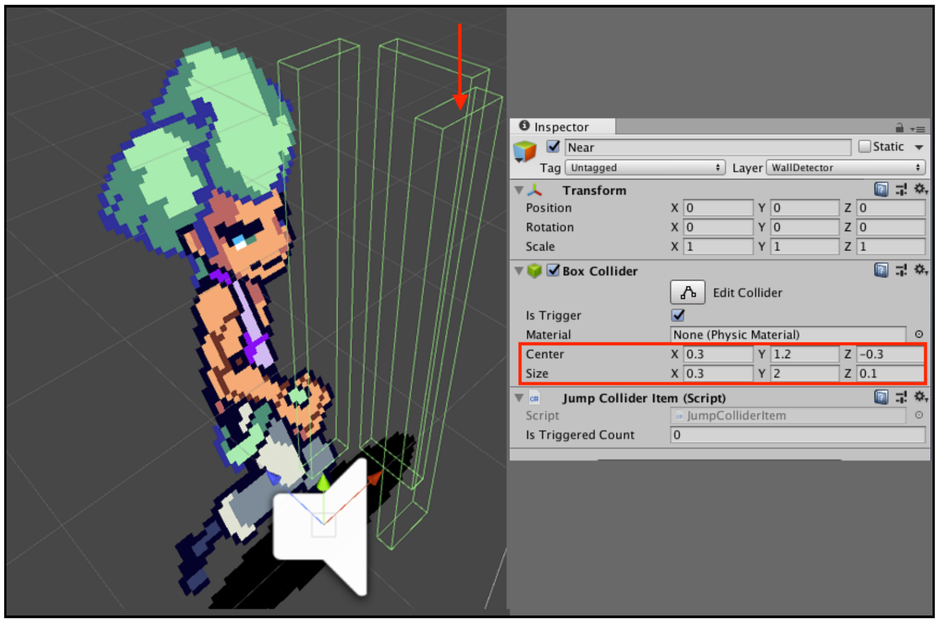
Edit Collider

Is Trigger	✓		
Material	None (Physic Material)		
Center	X 0.3	Y 1.2	Z 0.3
Size	X 0.3	Y 2	Z 0.1

Jump Collider Item (Script)

Script	JumpColliderItem
Is Triggered Count	0

Duplicate the **Far** GameObject and rename it to **Near**. Set its Box Collider’s **Center** to `(X:0.3, Y:1.2, Z:-0.3)` and **Size** to `(X:0.3, Y:2.0, Z:0.1)`. This will create the box collider on the near side of the hero, on the side closer to the camera.



Inspector

Near — Static

Tag Untagged — Layer WallDetector

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Box Collider

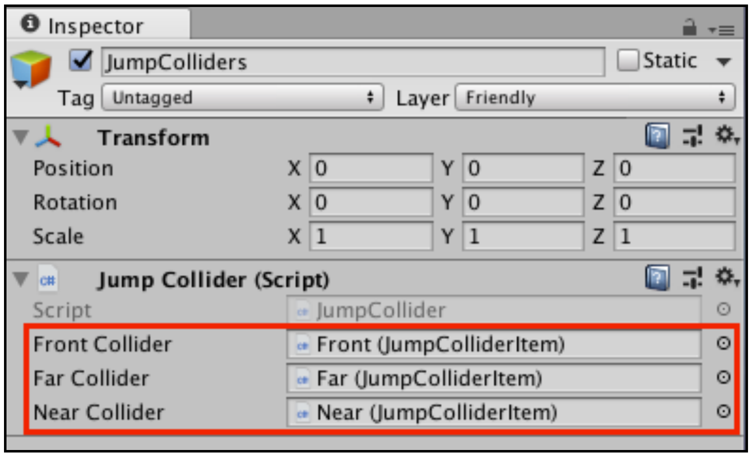
Edit Collider

Is Trigger	✓		
Material	None (Physic Material)		
Center	X 0.3	Y 1.2	Z -0.3
Size	X 0.3	Y 2	Z 0.1

Jump Collider Item (Script)

Script	JumpColliderItem
Is Triggered Count	0

Select **JumpColliders** and add a **JumpCollider** component to it. Set its **Front Collider** to **Front**, **Far Collider** to **Far** and **Near Collider** to **Near**.


Inspector

JumpColliders Static

Tag Untagged Layer Friendly

Transform

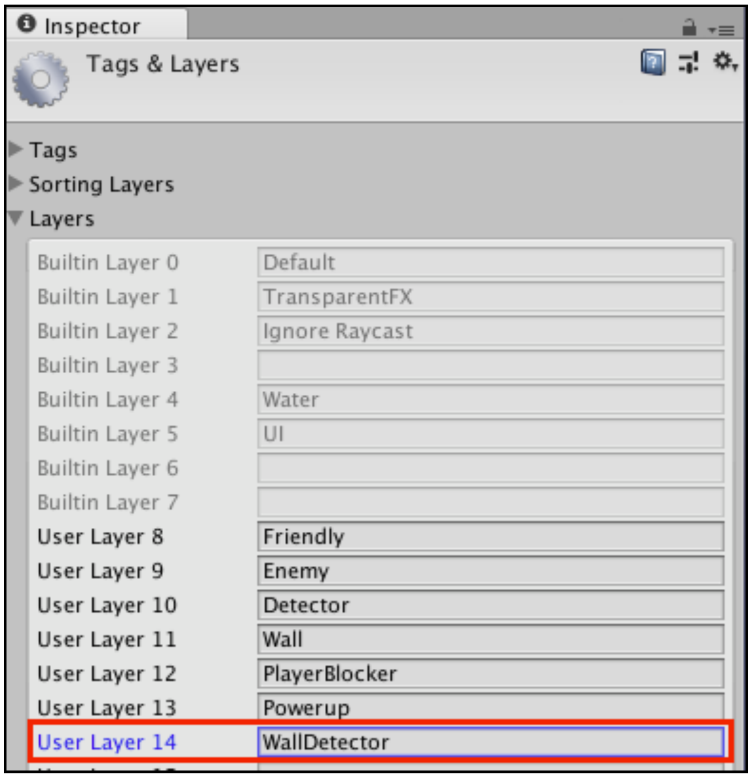
	X		Y		Z	
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

Jump Collider (Script)

Script	JumpCollider
Front Collider	Front (JumpColliderItem)
Far Collider	Far (JumpColliderItem)
Near Collider	Near (JumpColliderItem)

Your JumpColliderItems should have their own layer that only detects layers that block the player.

Open the **Tags and Layers** window by selecting **Edit** \ **Project Settings** \ **Tags and Layers** from the top menu. Add a new layer named **WallDetector**.

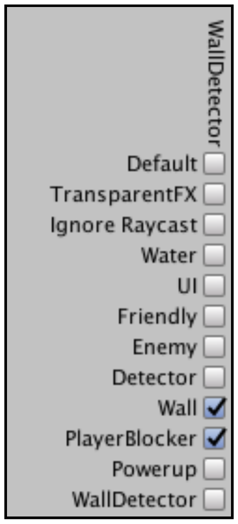

Inspector

Tags & Layers

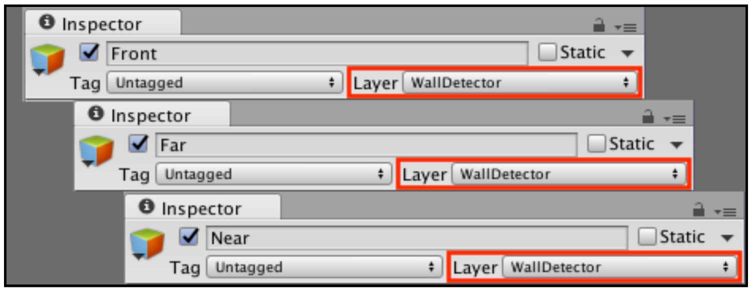
Tags
Sorting Layers
Layers

Builtin Layer 0	Default
Builtin Layer 1	TransparentFX
Builtin Layer 2	Ignore Raycast
Builtin Layer 3	
Builtin Layer 4	Water
Builtin Layer 5	UI
Builtin Layer 6	
Builtin Layer 7	
User Layer 8	Friendly
User Layer 9	Enemy
User Layer 10	Detector
User Layer 11	Wall
User Layer 12	PlayerBlocker
User Layer 13	Powerup
User Layer 14	WallDetector

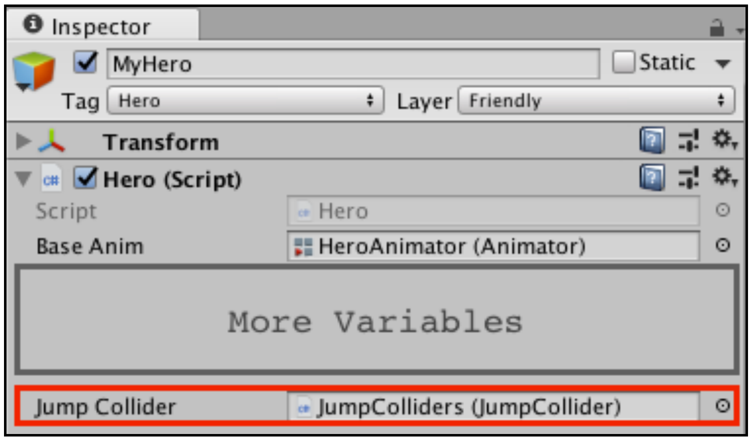
Next, open the **Physics** settings by selecting **Edit** \ **Project Settings** \ **Physics** in the top menu. Check the boxes to make **WallDetector** collide only with **Walls** and **PlayerBlocker**.



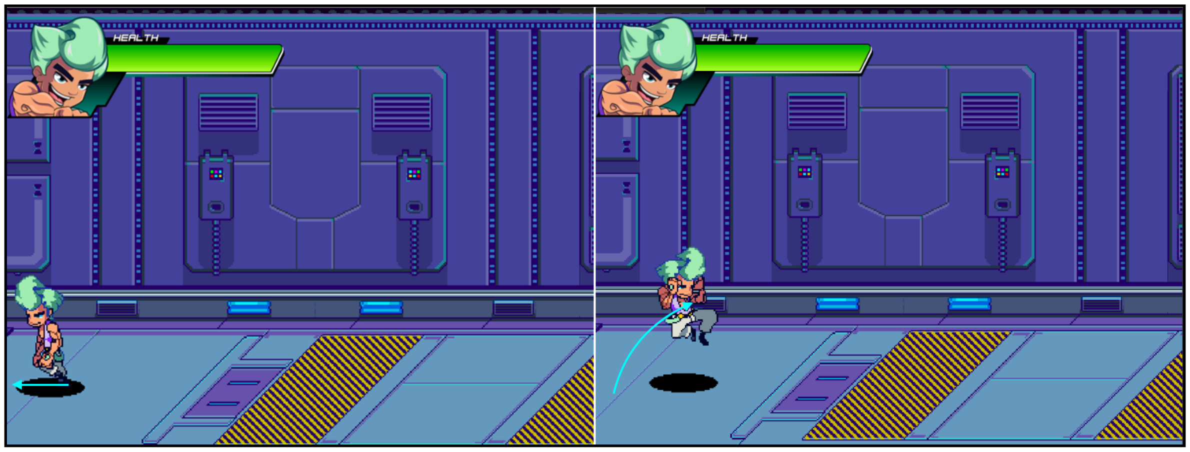
Select the **Front**, **Far** and **Near** in the Hierarchy and set their layers to **WallDetector**. This completes the collider assembly!



Set the Hero component's **Jump Collider** in MyHero to the **JumpColliders** GameObject you created earlier.



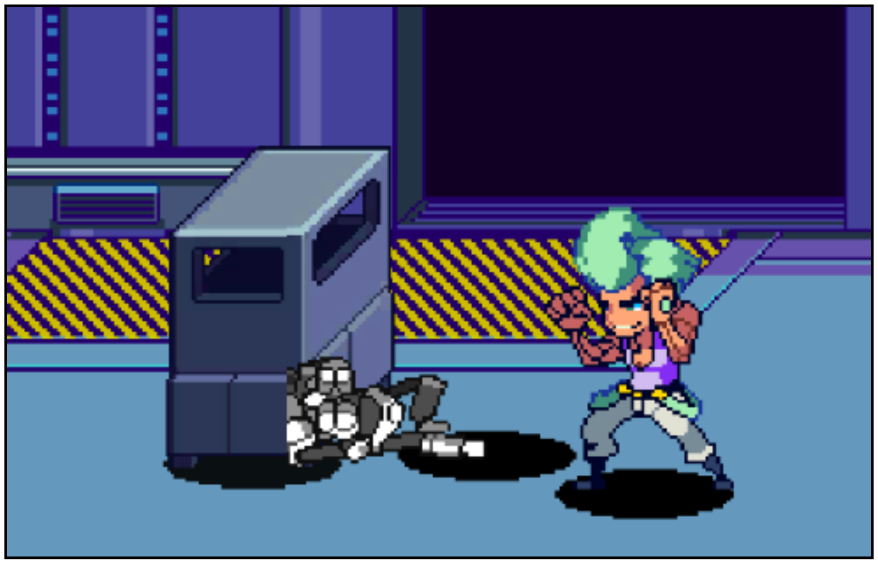
Save the scene and the project and try to replicate the bug. No go, right?



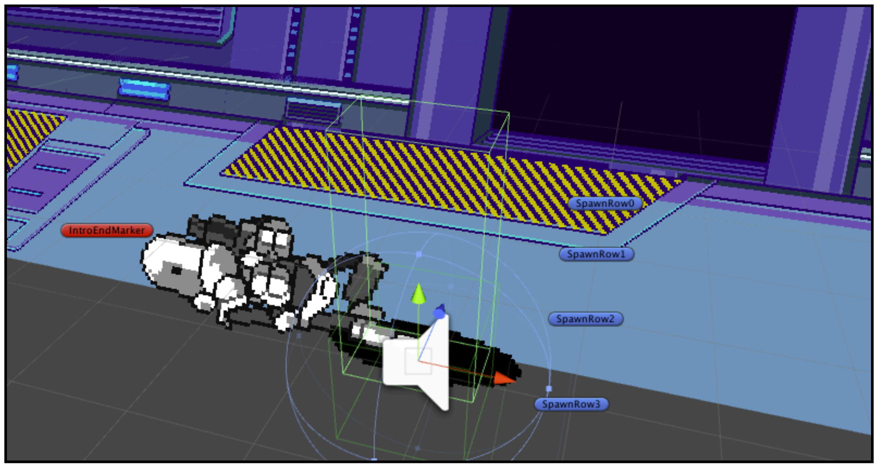
Ker-smack! One bug squashed.

Sprite sandwich

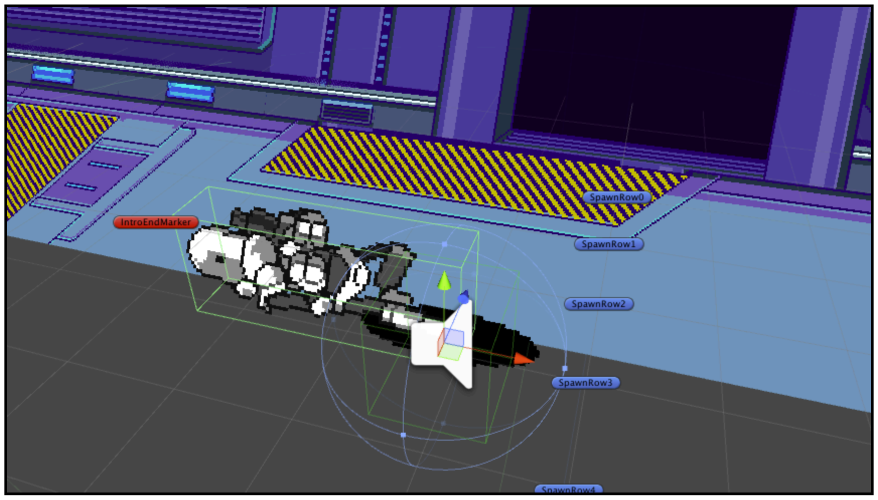
One of the more unfortunate bugs occurs when an actor goes down near a trash can and finds itself sandwiched between two layers of sprites.



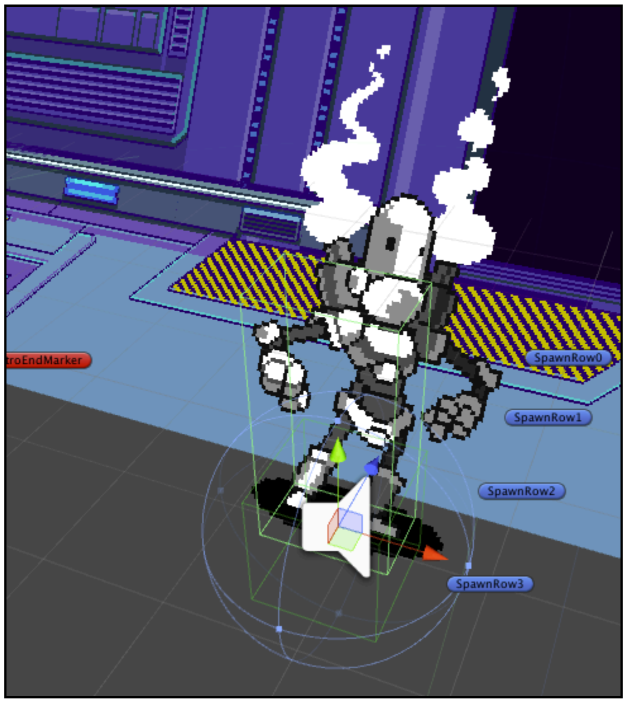
Although amusing, this is a noticeable fault In the game. Adjusting the box collider when an actor is knocked down will address the underlying issue.



Currently, it's not approximating an actor lying down. A more appropriate approximation is the box collider below:



However, when the robot gets back up, it should return to its proper, upright state.



To achieve this, you'll need your own logic to change the box collider's center and size variables when an Actor is knocked down.

Create a new **C# script** named **ActorCollider.cs** in the **Scripts** folder and open it. Replace its contents with this:

```
using UnityEngine;
using System.Collections;
//1
[RequireComponent(typeof(BoxCollider))]
public class ActorCollider : MonoBehaviour {

  //2
  public Vector3 standingColliderCenter;
  public Vector3 standingColliderSize;

  public Vector3 downColliderCenter;
  public Vector3 downColliderSize;

  private BoxCollider actorCollider;

  void Awake() {
    actorCollider = GetComponent<BoxCollider> ();
  }

  //3
  public void SetColliderStance(bool isStanding) {
    if (isStanding) {
      actorCollider.center = standingColliderCenter;
      actorCollider.size= standingColliderSize;
    } else {
      actorCollider.center = downColliderCenter;
      actorCollider.size= downColliderSize;
    }
  }
}
```

The `ActorCollider` toggles its `BoxCollider` position and size by using the `SetColliderStance` method.

1. The ActorCollider requires a `BoxCollider` component for it to work, which is assigned by using the `GetComponent<BoxCollider>()` method in the `Awake` method.
2. The ActorCollider needs values for the BoxCollider's `center` and `size` when the Actor is standing or when it is knocked down. This is stored in the `standingColliderCenter`, `standingColliderSize`, `downColliderCenter` and `downColliderSize` variables.
3. The `SetColliderStance` assigns the appropriate values to the boxCollider when this method is called. If the parameter `isStanding` is false, it uses the `downColliderCenter` and `downColliderSize` values. Otherwise, the values of `standingColliderCenter` and `standingColliderSize` are used.

Save the script and open **Actor.cs**. Add this below the `audioSource` variable:

```
protected ActorCollider actorCollider;
```

Your new reference to the ActorCollider will be attached as a component to any GameObjects using the Actor class.

Append the following to the end of the `Start` method:

```
actorCollider = GetComponent<ActorCollider> ();
actorCollider.SetColliderStance (true);
```

Here you're setting the reference to the sibling ActorCollider component using the `GetComponent<ActorCollider>()` method. You're also setting the initial stance to standing by calling `SetColliderStance` with a value of `true`.

Next, add the following to the end of the `Die` method:

```
actorCollider.SetColliderStance (false);
```

This will adjust the ActorCollider to the down stance when the `Die` method is called.

Find this `yield return` statement in the `KnockdownRoutine` coroutine:

```
yield return new WaitForSeconds (1.0f);
```

Replace it with this:

```
actorCollider.SetColliderStance (false);
yield return new WaitForSeconds (1.0f);
actorCollider.SetColliderStance (true);
```

This will adjust the ActorCollider to the down stance before the coroutine pauses. Then, when the actor finally gets back up, the ActorCollider is set to the standing stance again.

Save the script and open **Robot.cs**. You might remember that you implemented an override of the `KnockdownRoutine` in this class — you'll need to add the ActorCollider methods to it.

In the `KnockdownRoutine` coroutine, find this:

```
yield return new WaitForSeconds (2.0f);
```

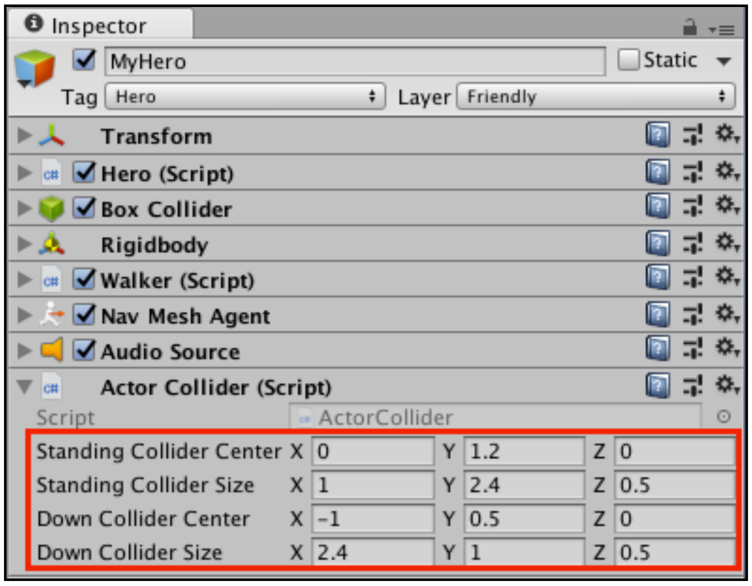
Replace it with:

```
actorCollider.SetColliderStance (false);
yield return new WaitForSeconds (2.0f);
actorCollider.SetColliderStance (true);
```

That's all you need to do to adjust the robot's collider so it handles being knocked down.

Save the script and return to Unity to add ActorCollider components to Actor instances. Select **MyHero** in the Hierarchy and add an **ActorCollider** component to it. Set the values like so:

- Standing Collider Center: `(X:0, Y:1.2, Z:0)`
- Standing Collider Size: `(X:1, Y:2.4, Z:0.5)`
- Down Collider Center: `(X:-1, Y:0.5, Z:0)`
- Down Collider Size: `(X:2.4, Y:1, Z:0.5)`

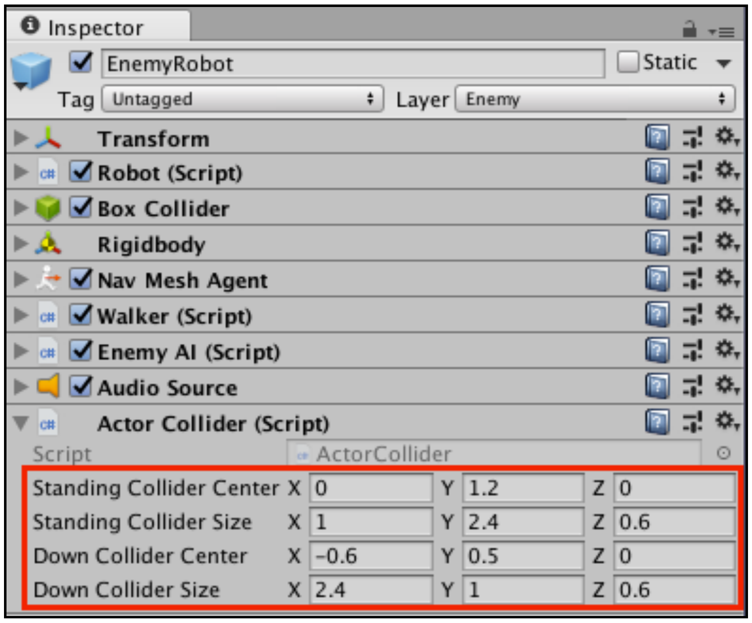


MyHero
Tag Hero
Layer Friendly
Script ActorCollider

	X	Y	Z
Standing Collider Center	0	1.2	0
Standing Collider Size	1	2.4	0.5
Down Collider Center	-1	0.5	0
Down Collider Size	2.4	1	0.5

Select the **EnemyRobot** prefab in the **Prefabs** folder and add an **ActorCollider** to it. Set these values:

- Standing Collider Center: `(X:0, Y:1.2, Z:0)`
- Standing Collider Size: `(X:1, Y:2.4, Z:0.6)`
- Down Collider Center: `(X:-0.6, Y:0.5, Z:0)`
- Down Collider Size: `(X:2.4, Y:1, Z:0.6)`

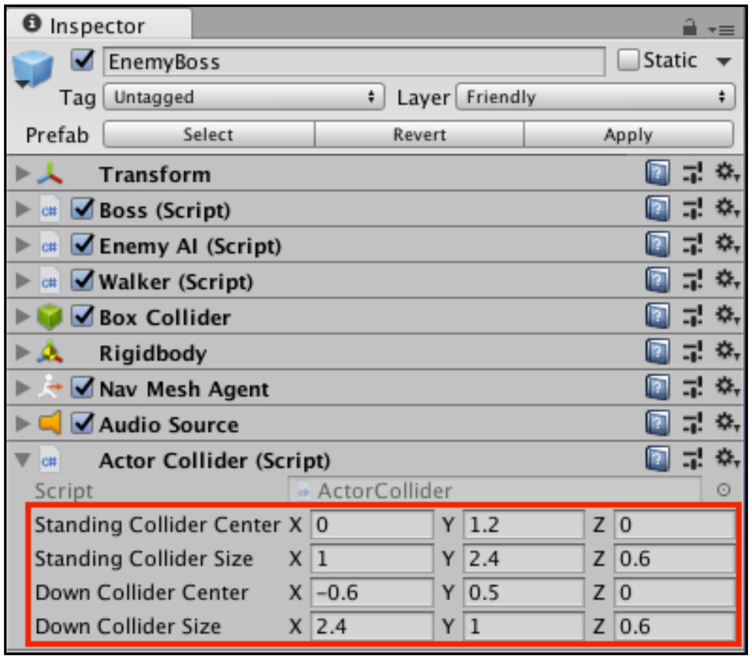


EnemyRobot
Tag Untagged
Layer Enemy
Script ActorCollider

	X	Y	Z
Standing Collider Center	0	1.2	0
Standing Collider Size	1	2.4	0.6
Down Collider Center	-0.6	0.5	0
Down Collider Size	2.4	1	0.6

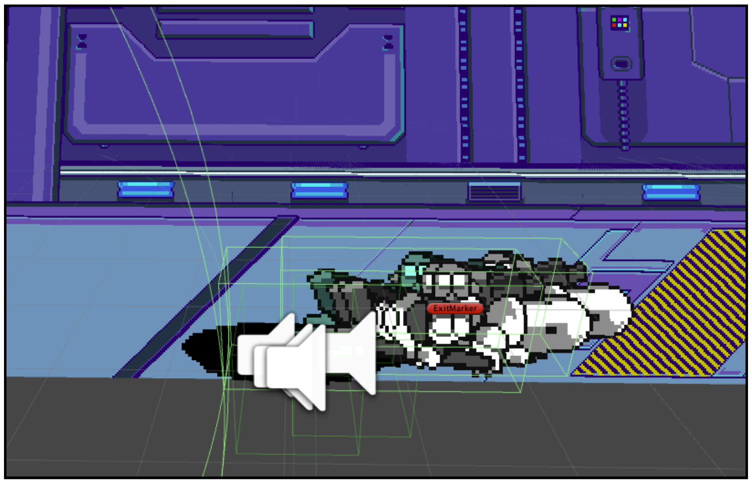
Move on to the **EnemyBoss** prefab from the same folder and add an **ActorCollider** to it. Modify it as shown below:

- Standing Collider Center: `(X:0, Y:1.2, Z:0)`
- Standing Collider Size: `(X:1, Y:2.4, Z:0.6)`
- Down Collider Center: `(X:-0.6, Y:0.5, Z:0)`
- Down Collider Size: `(X:2.4, Y:1, Z:0.6)`

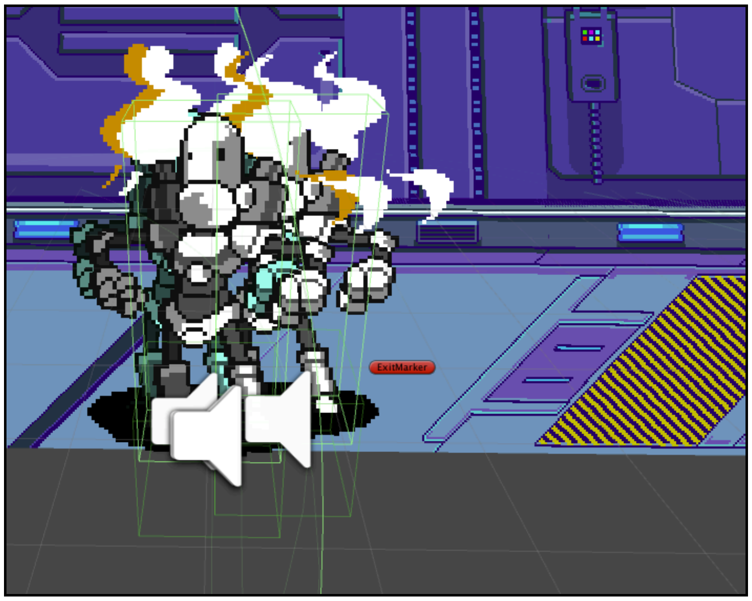


Standing Collider Center	X	0	Y	1.2	Z	0
Standing Collider Size	X	1	Y	2.4	Z	0.6
Down Collider Center	X	-0.6	Y	0.5	Z	0
Down Collider Size	X	2.4	Y	1	Z	0.6

Save the scene, save the project and click **Play**. Notice that when you knock enemies down, their colliders adjust to the lying down state.



When they get back up, the collider reverts to the standing values.



This should prevent the knocked down actor sprites from getting tangled up with the trash. Great job, the bug is now fixed. The game itself looks great! The main menu needs some love too.

Main menu polishes

You could really do a lot to the MainMenu scene, but we will keep the scope for the menu simple and impactful: fixing the sound effect cutoff bug when the player taps to play, and adding a bit of pop to the text.

You already have a transition from the MainMenu scene to the Game scene but it will not play the full sound effect whenever the game loads too fast. You might not notice it at first, but it becomes apparent after you’ve played at least one time because the Game scene is cached in memory to speed up loading.

An easy fix is to delay when the Game scene loads by a few seconds, allowing the blip sound effect to play all the way through.

First, open the **MainMenu** scene from the **Scenes** folder.

Then open **MainMenu.cs** from the **Scripts** folder to modify the manner in which the Game scene loads.

Add the following to the file:

```
using System.Collections;
```

This will enable the script to use Coroutines.

Add the following variable:

```
private Coroutine loadingRoutine;
```

This will hold a reference to the scene load coroutine.

Add this new method to the class:

```
private IEnumerator LoadGameScene(float delayDuration) {
  yield return new WaitForSeconds (delayDuration);
  GameManager.CurrentLevel = 0;
  SceneManager.LoadScene("Game");
}
```

This coroutine will delay the loading of the scene by the duration of the parameter `delayDuration`.

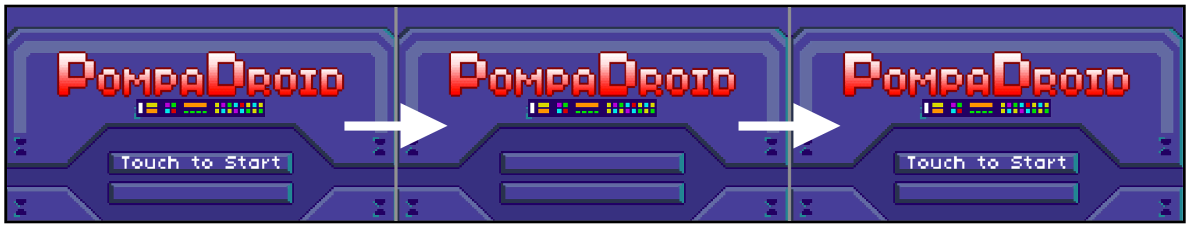
Replace the contents of the `GoToGame` method with this:

```
if (loadingRoutine == null) {
  loadingRoutine = StartCoroutine (LoadGameScene (2.0f));
}
```

This new `GoToGame` code will load the Game scene after a 2-second delay, and `if (loadingRoutine == null)` also prevents multiple calls to the `LoadGameScene` coroutine.

The sound bug should be squashed sufficiently now!

Next, to add a splash of polish to the main menu, you'll make the "Touch To Start" text flicker.



Create a new **C# script** in the **Scripts** folder and name it **ImageFlicker.cs**. Replace its contents with this code block:

```
using UnityEngine;
//1
using UnityEngine.UI;
using System.Collections;

//2
[RequireComponent(typeof(Image))]
public class ImageFlicker : MonoBehaviour {

  //3
```

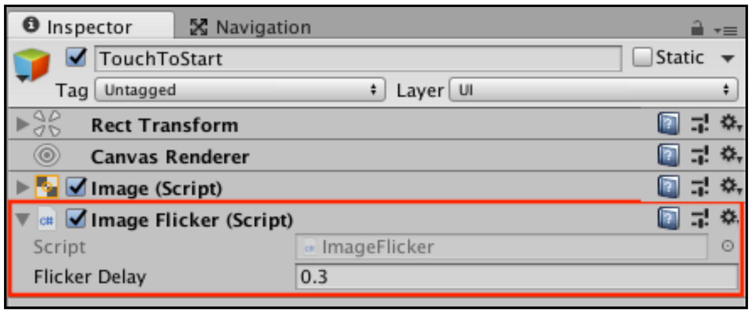
```
    private bool isShown = true;
    public float flickerDelay = 0.3f;
    private Image image;

    //4
    void Start () {
      image = GetComponent<Image> ();
      InvokeRepeating ("ToggleImage", flickerDelay, flickerDelay);
    }

    //5
    void ToggleImage() {
      image.enabled = isShown;
      isShown = !isShown;
    }
  }
```

The ImageFlicker class flickers the Image component to which it is assigned.

1. The `UnityEngine.UI` namespace is here so you can access the `Image` component.
2. ImageFlicker requires an image that will flicker.
3. Declare the variables of the ImageFlicker here, most notably a `flickerDelay` variable that holds the flicker speed.
4. Use the `GetComponent<Image>()` method to assign the image. `InvokeRepeating` behaves like a coroutine, except that this method calls `ToggleImage` for every `flickerDelay` duration forever, or until the `Invoke` is cancelled.

That finishes up this script. **Save** it and return to Unity.

Select the **TouchToStart** child of **Canvas** and add an **ImageFlicker** component to it.



Save the scene and the project and click **Play**. You should notice now that the "Touch To Start" text flickers. Also, when you start the game, the initial blip sound effect plays for a full two seconds before the game loads the Game scene.

Wow, PompaDroid is looking great! The game is essentially finished.

You can start it from the main menu and play through the whole game without having to stop and tinker with anything. When the enemy gets too strong and you lose, don’t worry, just return to the main menu and start again. Play as many times as you like!

Where to go from here?

Put away your polishing rags and puff your pompadour, friend! You made a lot of little changes and your game is almost ready to ship. In this chapter, you:

- Added background music and sound effects.
- Added intro and outro text, along with a splash screen at loading.
- Squashed a bunch of bugs…*ker-splat*!
- Polished a few things here and there to add quality to the final product.

Up next, you’ll deploy the game to smartphones and tablets. Get ready to enjoy PompaDroid on mobile devices!

Chapter 12: Running on Mobile Devices

Unity offers a plethora of build targets, ranging from desktop platforms (Windows, OSX, Linux, etc.) to consoles (PS4, Xbox One, Nintendo Switch, etc.) to mobile (iOS, Android, etc.). No wonder so many developers build games on Unity!

In this chapter, the main focus will be creating a working build that will run on both iOS and Android. Both mobile platforms are extremely popular. As a matter of fact, together they have 98% of the mobile market share.

You'll do the following in this chapter:

- Add code to support the back button.
- Set up an Android build and run the game on an Android device.
- Set up an iOS build, create an Xcode Project and run the game on an iOS device.

Prerequisites: To complete all the steps in this chapter, as written, you'll need: An Android device and an iOS device, a Mac running the latest version of Xcode, and an Apple developer account, which is required to build to a device

If you do not have a Mac, you won't be able to build the game for iOS. But it's very similar to building for Android, so you're not really losing out. If you're lacking devices, go ahead and use simulators so you can learn the process — you can always do it again when you have devices.

Time to build!

Running on Android

Android, a Java-based operating system (OS) for mobile phones dominates the global market, making it the most popular mobile OS. It also has an adorable robot for a mascot:

Android releases new versions periodically, usually with names that sound better suited to late-night munchies than some OS. At the time of writing, the latest version is Android 8.0 Oreo. PompaDroid will support **Android 4.0 Ice Cream Sandwich** through **Android 8.0 Oreo**.

You can create an Android project from Unity running on macOS or Windows. Each OS has slightly different labels for some of next steps, but the process is similar.

Before running on a smartphone, you'll first implement the **Back** button for Android.

Supporting the back button

The back button returns the user to the previous screen, for example, backing out of menus, or going back to the previous view in an app, etc.

You want it so that when the player is on the main menu, pressing Back exits the game. And when the game is running in the game scene, the back button leads to the main menu.

Start by opening **MainMenu.cs**. Add the following `Update` method:

```
void Update() {
  if (Input.GetKeyDown(KeyCode.Escape)) {
    Application.Quit();
  }
}
```

The code uses the `Update` method to detect a key press from the `Input` class.

Specifically, it checks if the player pressed the `KeyCode.Escape` key — the equivalent to the back button on Android. If true, the game receives a signal to quit using the `Application.Quit()` line.

Save the script and open **GameManager.cs**. Insert the following code at the end of the `Update` method.

```
if (Input.GetKeyDown(KeyCode.Escape)) {
  //1
  SceneManager.LoadScene("MainMenu");
}
```

This code block is similar to the code added in the `MainMenu` class, except the back button loads the `MainMenu` scene.

Save the script and return to Unity. Click **Play** to run the game. Press the **Esc** button on your keyboard to simulate the back button. The game should return to the main menu while in a level and close when from the main menu.

Great, the game is now ready to build for Android. Next up is setting up your Android software development kit (SDK).

Setting up the Android SDK

To create Android builds for PompaDroid, you'll need:

- The latest version of **Unity** with the **Android Module** installed.
- **Java Development Kit** (JDK)
- **Android Command Line Tools**.
- An Android device, running Android 4.0 (Ice Cream Sandwich) or later.

Installing Java Development Kit (JDK)

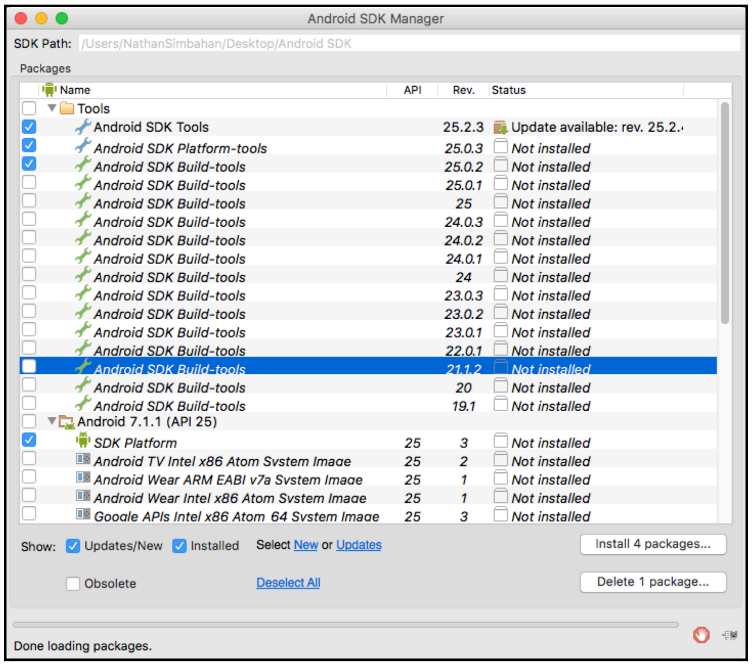
Download the Java Development Kit and install it in your system. (http://www.oracle.com/technetwork/java/javase/downloads/index.html) [http://www.oracle.com/technetwork/java/javase/downloads/index.html].

Once installed, tell Unity where to find the JDK by going to **Unity \ Preferences** from the top menu if on macOS or **Edit \ Preferences** on Windows. Set the **JDK** path in the External Tools tab to where the JDK is installed.

Installing Android command line tools

Go to this website to install Android command line tools: (http://developer.android.com/sdk/index.html)[http://developer.android.com/sdk/index.html]. You do **not** need to download Android Studio to create a build for Android, just the Android SDK.

Extract the ZIP file to a location of your choosing. Go to that location. Open the **tools** folder then open the file named **android** to display the **Android SDK Manager** window.


Android SDK Manager

SDK Path: /Users/NathanSimbahan/Desktop/Android SDK

Packages

	Name	API	Rev.	Status
☐	Tools			
✓	Android SDK Tools		25.2.3	Update available: rev. 25.2.-
✓	Android SDK Platform-tools		25.0.3	Not installed
✓	Android SDK Build-tools		25.0.2	Not installed
☐	Android SDK Build-tools		25.0.1	Not installed
☐	Android SDK Build-tools		25	Not installed
☐	Android SDK Build-tools		24.0.3	Not installed
☐	Android SDK Build-tools		24.0.2	Not installed
☐	Android SDK Build-tools		24.0.1	Not installed
☐	Android SDK Build-tools		24	Not installed
☐	Android SDK Build-tools		23.0.3	Not installed
☐	Android SDK Build-tools		23.0.2	Not installed
☐	Android SDK Build-tools		23.0.1	Not installed
☐	Android SDK Build-tools		22.0.1	Not installed
☐	Android SDK Build-tools		21.1.2	Not installed
☐	Android SDK Build-tools		20	Not installed
☐	Android SDK Build-tools		19.1	Not installed
☐	Android 7.1.1 (API 25)			
✓	SDK Platform	25	3	Not installed
☐	Android TV Intel x86 Atom System Image	25	2	Not installed
☐	Android Wear ARM EABI v7a System Image	25	1	Not installed
☐	Android Wear Intel x86 Atom System Image	25	1	Not installed
☐	Google APIs Intel x86 Atom 64 System Image	25	3	Not installed

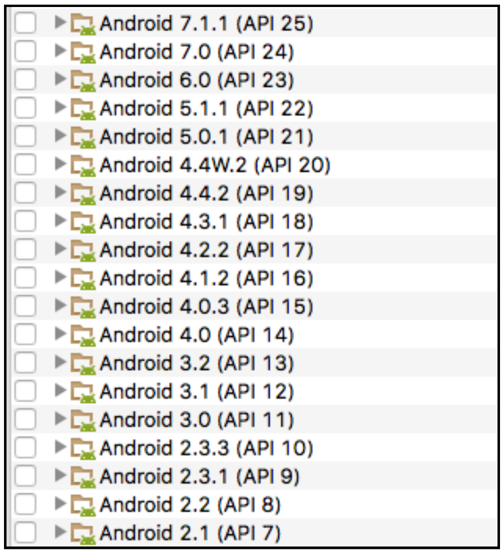
Show: ✓ Updates/New ✓ Installed Select New or Updates Install 4 packages...

☐ Obsolete Deselect All Delete 1 package...

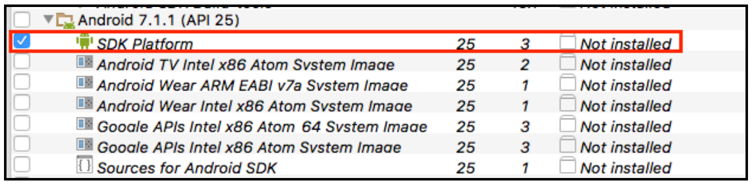
Done loading packages.

Gather the tools you need from the **Tools** tab: check **Android SDK Tools**, **Android SDK Platform-tools** and **Android SDK Build-tools**.

Next up, you’ll download the SDK for the desired Android version.

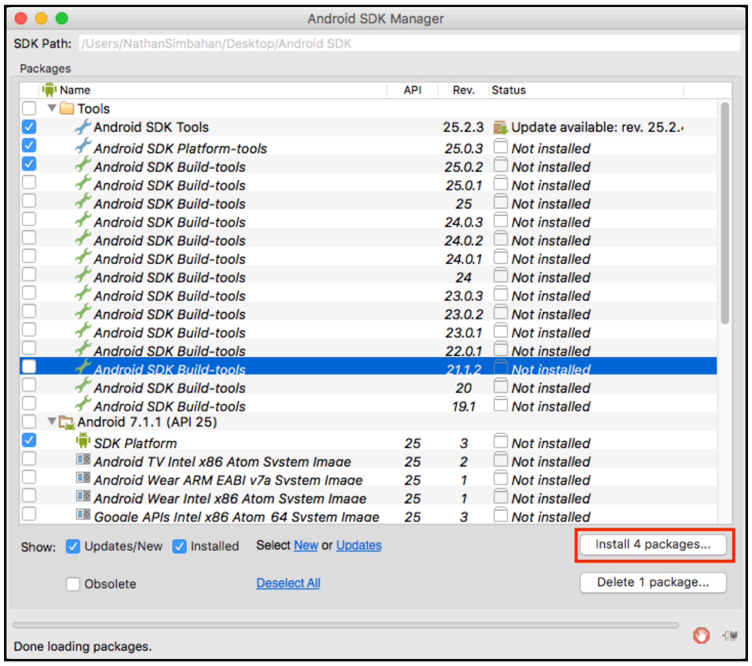

☐ Android 7.1.1 (API 25)
☐ Android 7.0 (API 24)
☐ Android 6.0 (API 23)
☐ Android 5.1.1 (API 22)
☐ Android 5.0.1 (API 21)
☐ Android 4.4W.2 (API 20)
☐ Android 4.4.2 (API 19)
☐ Android 4.3.1 (API 18)
☐ Android 4.2.2 (API 17)
☐ Android 4.1.2 (API 16)
☐ Android 4.0.3 (API 15)
☐ Android 4.0 (API 14)
☐ Android 3.2 (API 13)
☐ Android 3.1 (API 12)
☐ Android 3.0 (API 11)
☐ Android 2.3.3 (API 10)
☐ Android 2.3.1 (API 9)
☐ Android 2.2 (API 8)
☐ Android 2.1 (API 7)

Uncheck all other components except the **SDK Platform** that matches your device — this is simply to reduce the time you’re waiting for the download.



	Name	API	Rev.	Status
☐	Android 7.1.1 (API 25)			
✓	*SDK Platform*	25	3	Not installed
☐	*Android TV Intel x86 Atom System Image*	25	2	Not installed
☐	*Android Wear ARM EABI v7a System Image*	25	1	Not installed
☐	*Android Wear Intel x86 Atom System Image*	25	1	Not installed
☐	*Google APIs Intel x86 Atom_64 System Image*	25	3	Not installed
☐	*Google APIs Intel x86 Atom System Image*	25	3	Not installed
☐	*Sources for Android SDK*	25	1	Not installed

Click the **Install X packages...** button at the bottom-right corner of the SDK manager window.



Android SDK Manager

SDK Path: /Users/NathanSimbahan/Desktop/Android SDK

Packages

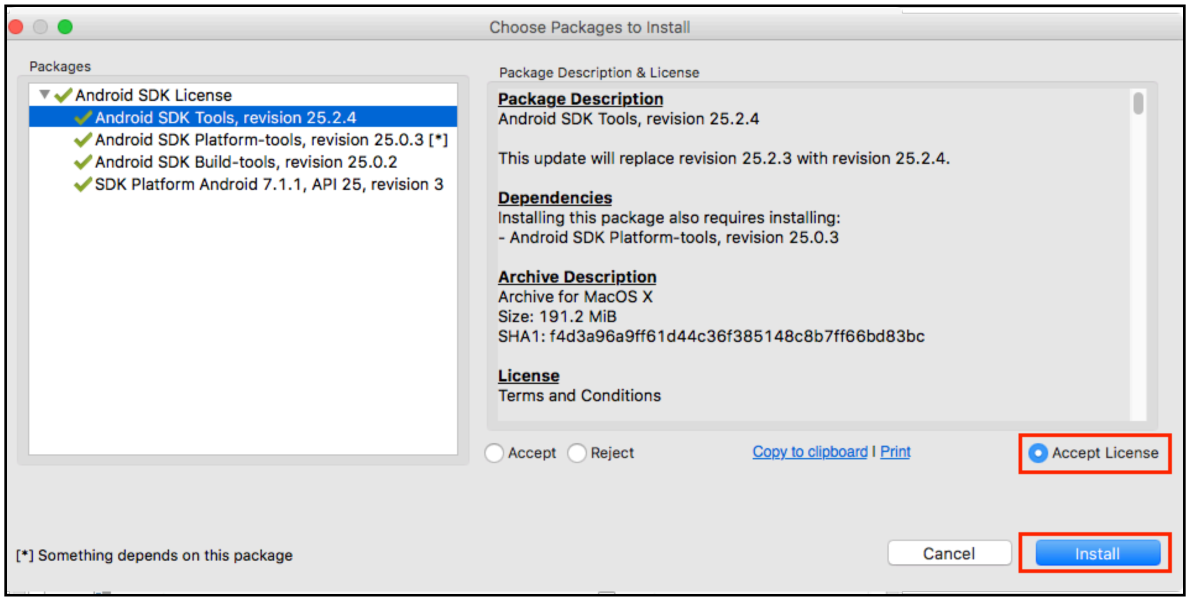
	Name	API	Rev.	Status
☐	Tools			
✓	Android SDK Tools		25.2.3	Update available: rev. 25.2.4
✓	*Android SDK Platform-tools*		25.0.3	Not installed
✓	*Android SDK Build-tools*		25.0.2	Not installed
☐	*Android SDK Build-tools*		25.0.1	Not installed
☐	*Android SDK Build-tools*		25	Not installed
☐	*Android SDK Build-tools*		24.0.3	Not installed
☐	*Android SDK Build-tools*		24.0.2	Not installed
☐	*Android SDK Build-tools*		24.0.1	Not installed
☐	*Android SDK Build-tools*		24	Not installed
☐	*Android SDK Build-tools*		23.0.3	Not installed
☐	*Android SDK Build-tools*		23.0.2	Not installed
☐	*Android SDK Build-tools*		23.0.1	Not installed
☐	*Android SDK Build-tools*		22.0.1	Not installed
☐	*Android SDK Build-tools*		21.1.2	Not installed
☐	*Android SDK Build-tools*		20	Not installed
☐	*Android SDK Build-tools*		19.1	Not installed
☐	Android 7.1.1 (API 25)			
✓	*SDK Platform*	25	3	Not installed
☐	*Android TV Intel x86 Atom System Image*	25	2	Not installed
☐	*Android Wear ARM EABI v7a System Image*	25	1	Not installed
☐	*Android Wear Intel x86 Atom System Image*	25	1	Not installed
☐	*Google APIs Intel x86 Atom_64 System Image*	25	3	Not installed

Show: ✓ Updates/New ✓ Installed Select New or Updates — Install 4 packages...

☐ Obsolete Deselect All — Delete 1 package...

Done loading packages.

Accept the licenses for all packages.



Choose Packages to Install

Packages

- ✓ Android SDK License
  - ✓ Android SDK Tools, revision 25.2.4
  - ✓ Android SDK Platform-tools, revision 25.0.3 [*]
  - ✓ Android SDK Build-tools, revision 25.0.2
  - ✓ SDK Platform Android 7.1.1, API 25, revision 3

Package Description & License

Package Description
Android SDK Tools, revision 25.2.4

This update will replace revision 25.2.3 with revision 25.2.4.

Dependencies
Installing this package also requires installing:
- Android SDK Platform-tools, revision 25.0.3

Archive Description
Archive for MacOS X
Size: 191.2 MiB
SHA1: f4d3a96a9ff61d44c36f385148c8b7ff66bd83bc

License
Terms and Conditions

○ Accept ○ Reject Copy to clipboard | Print [● Accept License]

[*] Something depends on this package

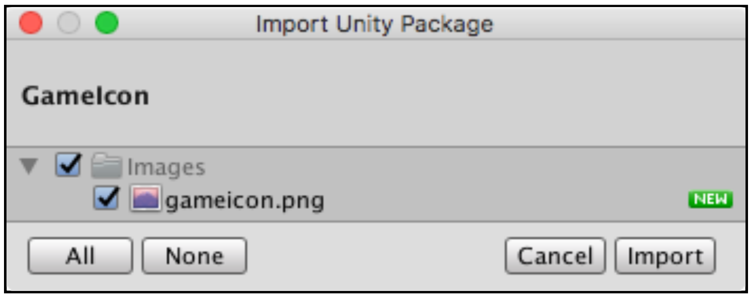
Cancel Install

Once you have accepted, just walk away and let the download happen. Go for a quick walk or something!

Building for Android

Welcome back. If your download is done, you're ready to make Android builds!

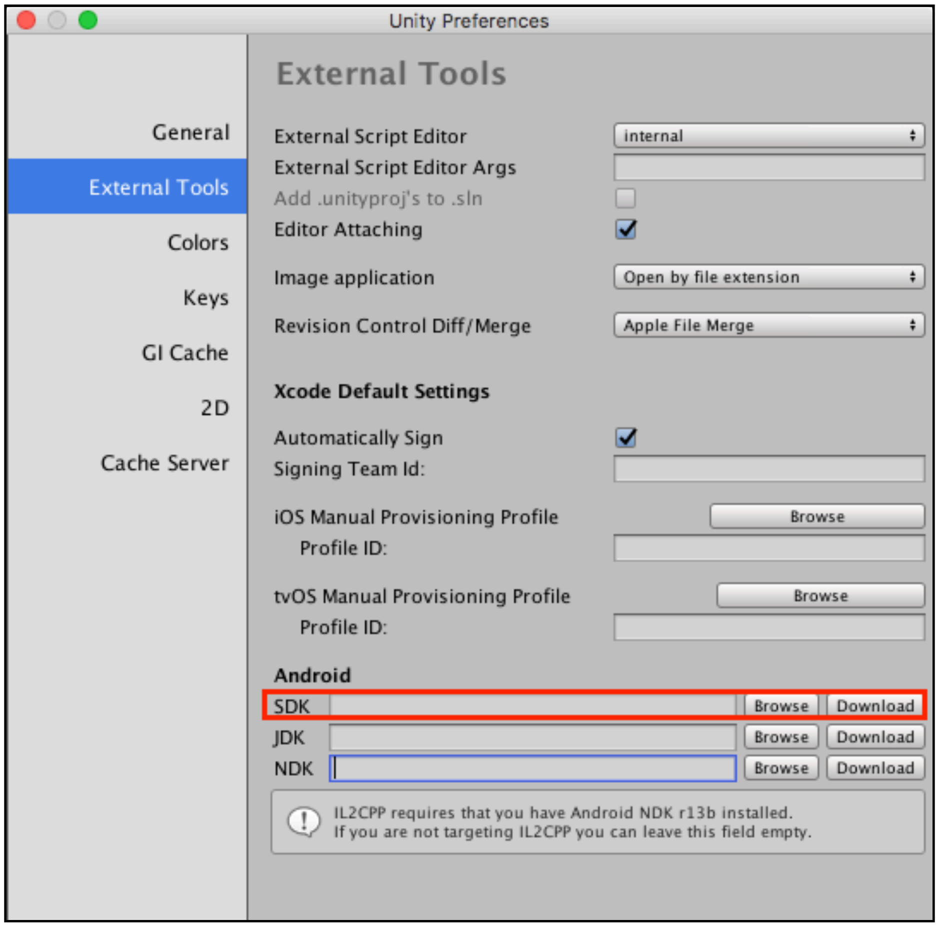
Import **GameIcon.unitypackage** from the **Unity Packages** folder into the project — it contains the game icon sprite.



The icon is simply the hero sprite on a blue background.

Select **Unity** \ **Preferences** from the top menu if on macOS or **Edit** \ **Preferences** on Windows. Select the **External Tools** tab.

In the Android section, click the top **Browse** button next to the SDK field and navigate to the folder where you extracted the command line tools.



External Tools

External Script Editor: internal
External Script Editor Args
Add .unityproj's to .sln
Editor Attaching ✓
Image application: Open by file extension
Revision Control Diff/Merge: Apple File Merge

Xcode Default Settings

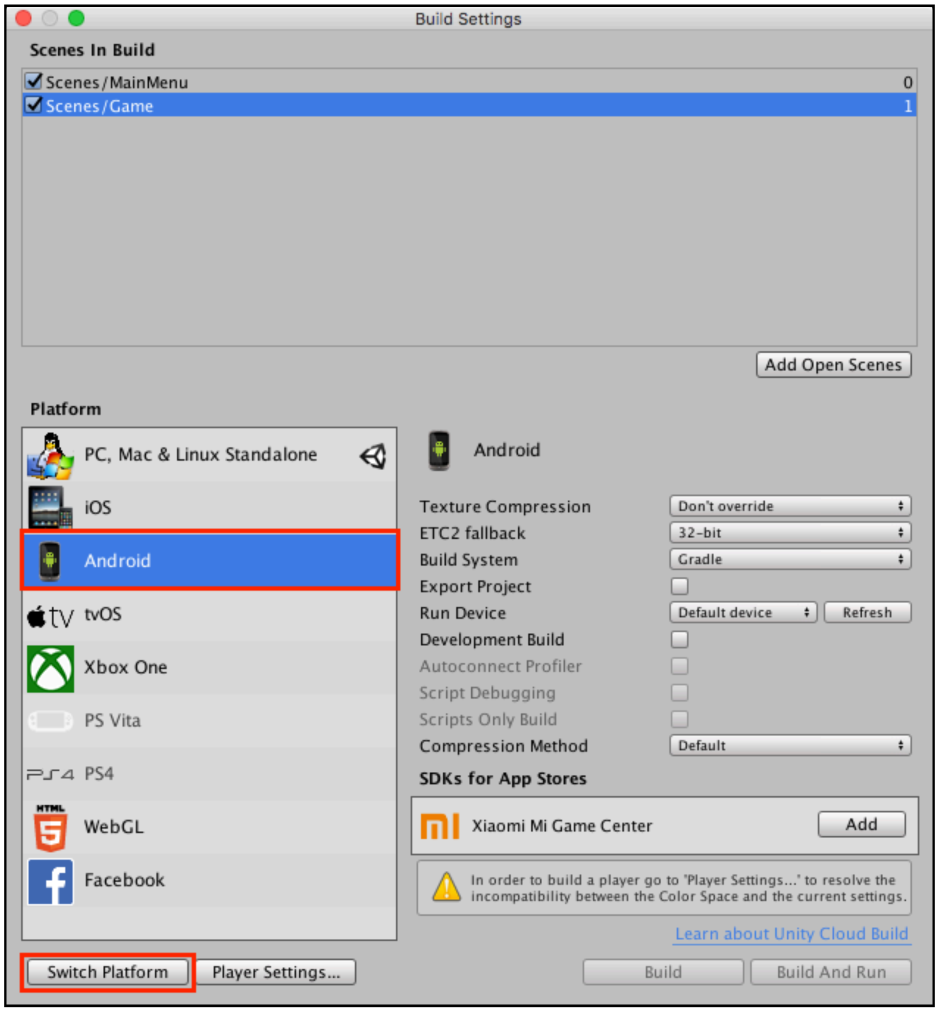
Automatically Sign ✓
Signing Team Id:
iOS Manual Provisioning Profile
Profile ID:
tvOS Manual Provisioning Profile
Profile ID:

Android

SDK
JDK
NDK

IL2CPP requires that you have Android NDK r13b installed.
If you are not targeting IL2CPP you can leave this field empty.

Close the **Preferences** window and open **Build Settings** by selecting **File \ Build Settings** from the top menu. Select **Android** in the list and click the **Switch Platform** button.

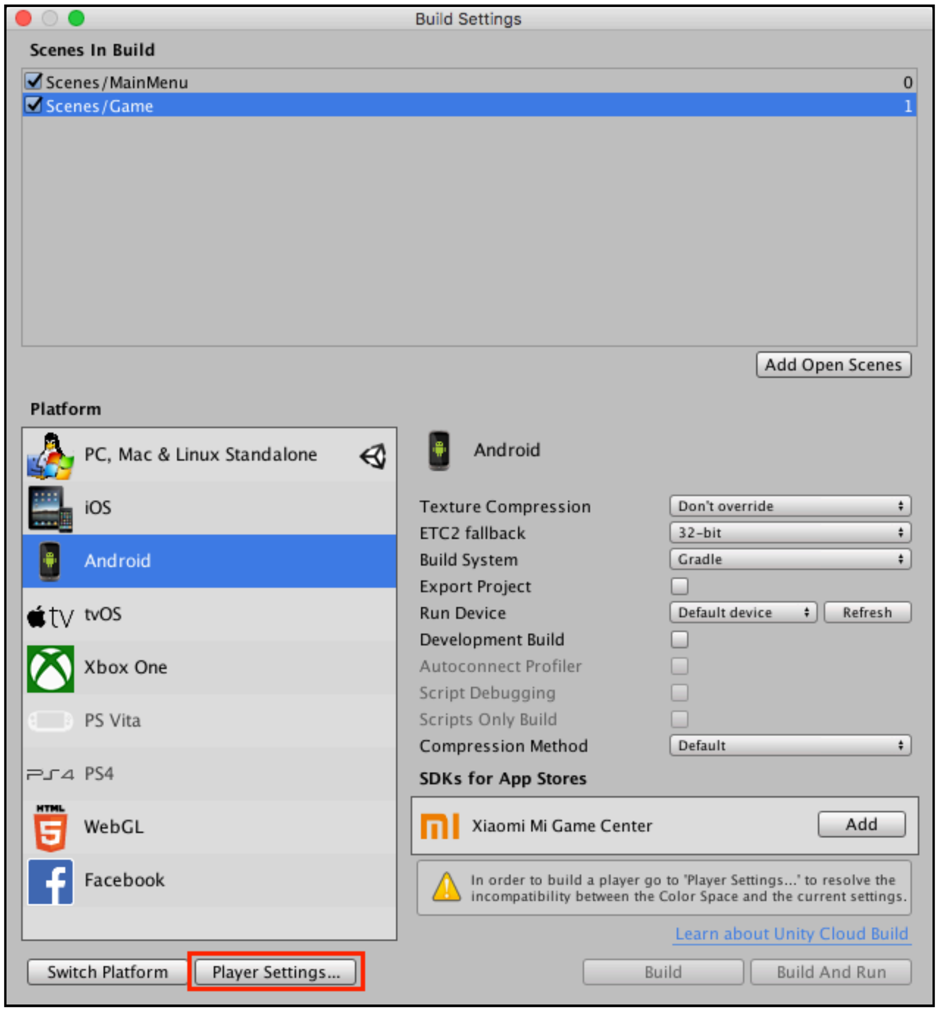


Texture Compression	Don't override
ETC2 fallback	32-bit
Build System	Gradle
Export Project	☐
Run Device	Default device / Refresh
Development Build	☐
Autoconnect Profiler	☐
Script Debugging	☐
Scripts Only Build	☐
Compression Method	Default

In order to build a player go to 'Player Settings...' to resolve the incompatibility between the Color Space and the current settings.

This will re-import all assets to the game, since each platform has unique specifications for asset compression protocols.

Once re-importing is finished, select **Player Settings** in the **Build Settings** window to open PompaDroid's settings.



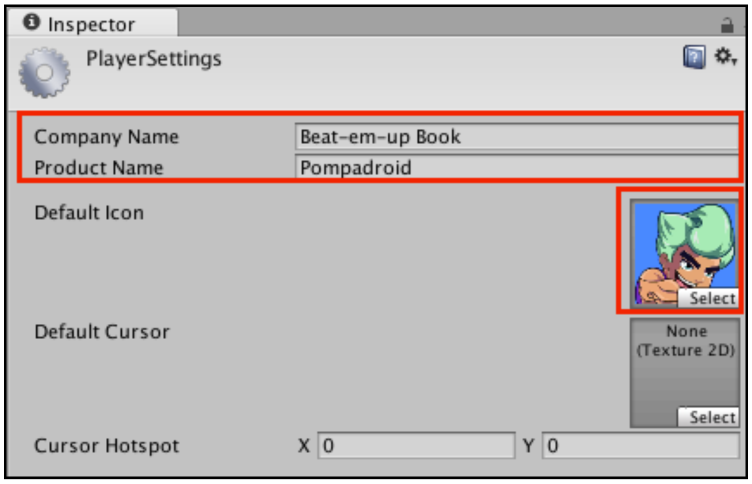
Texture Compression: Don't override
ETC2 fallback: 32-bit
Build System: Gradle
Export Project: ☐
Run Device: Default device — Refresh
Development Build: ☐
Autoconnect Profiler: ☐
Script Debugging: ☐
Scripts Only Build: ☐
Compression Method: Default

In order to build a player go to 'Player Settings...' to resolve the incompatibility between the Color Space and the current settings.

First, you'll need to assign a new, unique **bundle identifier** for your project. Bundle identifiers typically follow this format:

```
com.<COMPANY NAME>.<PRODUCT NAME>
```

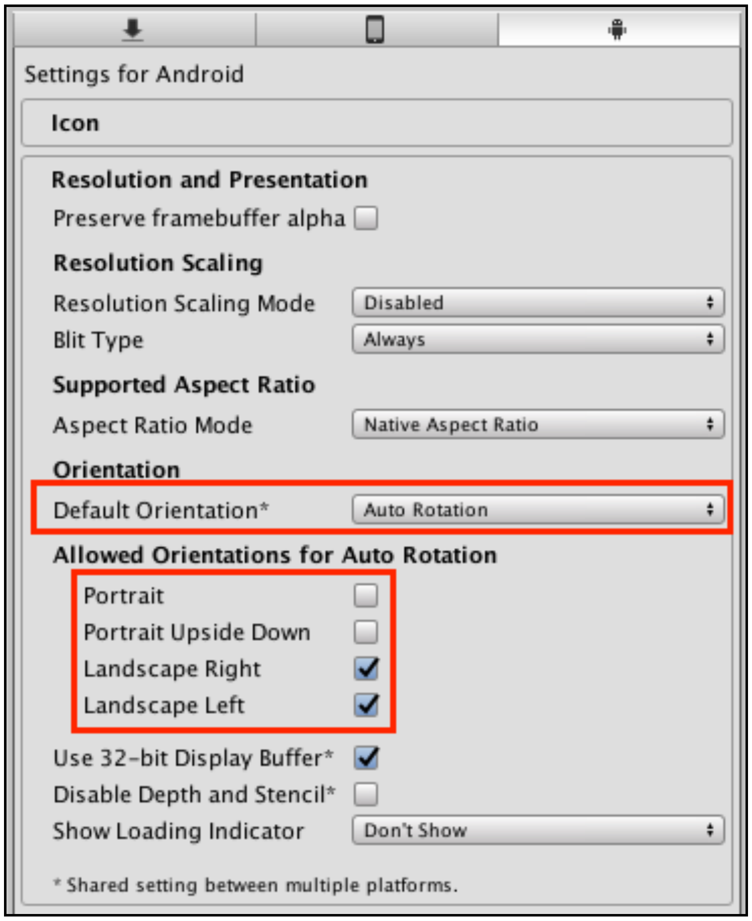
In the **PlayerSettings**, set **Company Name** to **Beat-em-up Book**, **Product Name** to **Pompadroid**, and select **gameicon** as the **Default Icon**.



Inspector
PlayerSettings

Company Name: Beat-em-up Book
Product Name: Pompadroid
Default Icon: Select
Default Cursor: None (Texture 2D) Select
Cursor Hotspot: X 0 Y 0

Next, in the **Resolution and Presentation** settings for Android, set **Default Orientation** to **Auto Rotation**, uncheck **Portrait** and **Portrait Upside Down** but keep both **Landscape Right** and **Landscape Left** checked. This game should only run in landscape, but it should support both variants.



Settings for Android

Icon

Resolution and Presentation
Preserve framebuffer alpha ☐

Resolution Scaling
Resolution Scaling Mode: Disabled
Blit Type: Always

Supported Aspect Ratio
Aspect Ratio Mode: Native Aspect Ratio

Orientation
Default Orientation*: Auto Rotation

Allowed Orientations for Auto Rotation
Portrait ☐
Portrait Upside Down ☐
Landscape Right ✓
Landscape Left ✓

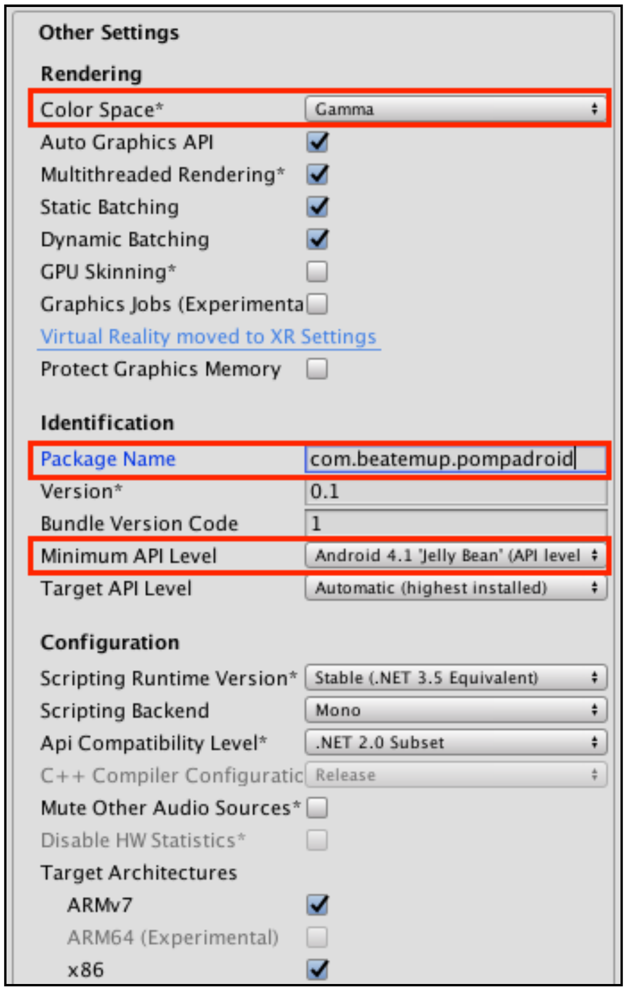
Use 32-bit Display Buffer* ✓
Disable Depth and Stencil* ☐
Show Loading Indicator: Don't Show

* Shared setting between multiple platforms.

In the **Other Settings** tab, set **Bundle Identifier** to **com.beatemup.pompadroid** or any bundle ID of your choice.

> **Note**: It's important that this is a unique identifier, so apps on your Android device are not in conflict with each other.
>
> For demonstration purposes, this project will use the bundle identifier **com.beatemup.pompadroid**, but feel free to use your own Bundle Identifier when building for your own use.

Set the **Minimum Version** to the lowest version you want to support. In the following example, the device is running on Android 5.0, so that's what I've chosen.



Other Settings

Rendering

Color Space*	Gamma
Auto Graphics API	✓
Multithreaded Rendering*	✓
Static Batching	✓
Dynamic Batching	✓
GPU Skinning*	☐
Graphics Jobs (Experimenta	☐
Virtual Reality moved to XR Settings	
Protect Graphics Memory	☐

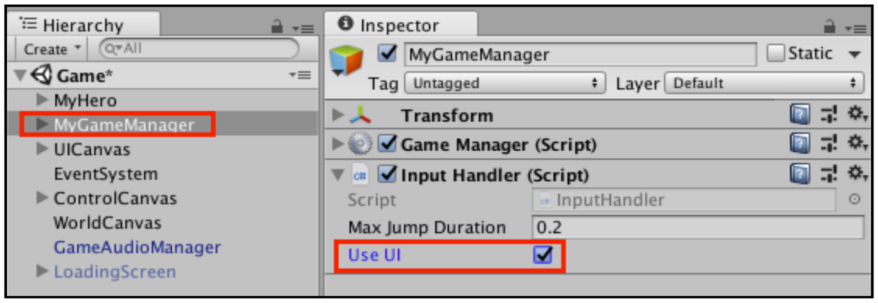
Identification

Package Name	com.beatemup.pompadroid
Version*	0.1
Bundle Version Code	1
Minimum API Level	Android 4.1 'Jelly Bean' (API level
Target API Level	Automatic (highest installed)

Configuration

Scripting Runtime Version*	Stable (.NET 3.5 Equivalent)
Scripting Backend	Mono
Api Compatibility Level*	.NET 2.0 Subset
C++ Compiler Configuratic	Release
Mute Other Audio Sources*	☐
Disable HW Statistics*	☐
Target Architectures	
ARMv7	✓
ARM64 (Experimental)	☐
x86	✓

Okay, you're set to run on Android. **Save** the project and open the **Game** scene. Enable **Use UI** for **MyGameManager** so the game can use touch controls.



Save the scene and the project.

Setting up a test device

Your next step is setting up a test device, which you need to do before creating a build.

Up first is enabling **Developer Options** on your Android device, if you haven't already. Follow these steps:

1. On your Android device, navigate to **Settings** \ **About Phone** or **Settings** \ **About Tablet**.
2. Scroll to **Build Number** and tap it seven times. A toast message will appear, confirming that you are now a developer. The exact location of the **Build Number** entry varies from device to device but should be around the **About Phone** or **About Tablet** settings.
3. Navigate to **Settings** \ **Developer options** \ **Debugging** and check **enable USB debugging**.

Your device is now ready to receive an Android build! Plug it in to the computer.

Build and run

In the editor, open **Build Settings** again and make sure that **MainMenu** scene and **Game** scene are included in the build. Also, make sure that the target platform is Android.

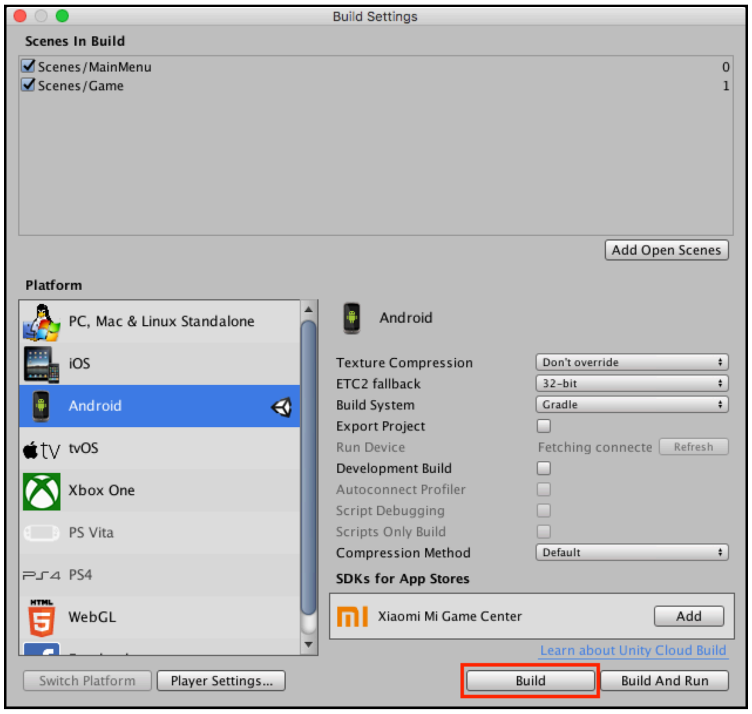
Click the **Build and Run** button and set a suitable location for the Android Application Package (APK).

Hurry up and wait for the build and copy process to complete — now is a fine time to break away and do something besides stare at a screen.

Build and run: there's another way

Check for the game on your device. It should be there, but if not or it's not working right, there is an alternative approach.

Click the **Build** button and locate a suitable place to save your APK.



Scenes In Build
- ✓ Scenes/MainMenu 0
- ✓ Scenes/Game 1

Platform
- PC, Mac & Linux Standalone
- iOS
- Android
- tvOS
- Xbox One
- PS Vita
- PS4
- WebGL

Android

Texture Compression	Don't override
ETC2 fallback	32-bit
Build System	Gradle
Export Project	☐
Run Device	Fetching connecte
Development Build	☐
Autoconnect Profiler	☐
Script Debugging	☐
Scripts Only Build	☐
Compression Method	Default

SDKs for App Stores
- Xiaomi Mi Game Center

Transfer the APK file to your device and install it by locating it in a file manager and tapping it.

PompaDroid should definitely be there now.

Play on, dear reader. Play on! It should be the same with the exception of the touchscreen controls. Tap the **back** button on the device while you're playing. Tap it again to test if it exits the game.

Congratulations! If you've never built a game before, this has to be a pretty special moment. You did it! That is indeed PompaDroid running on Android!

Next up, assuming you have the hardware available, you'll build out the game for iOS so it can run on iPhones and iPads.

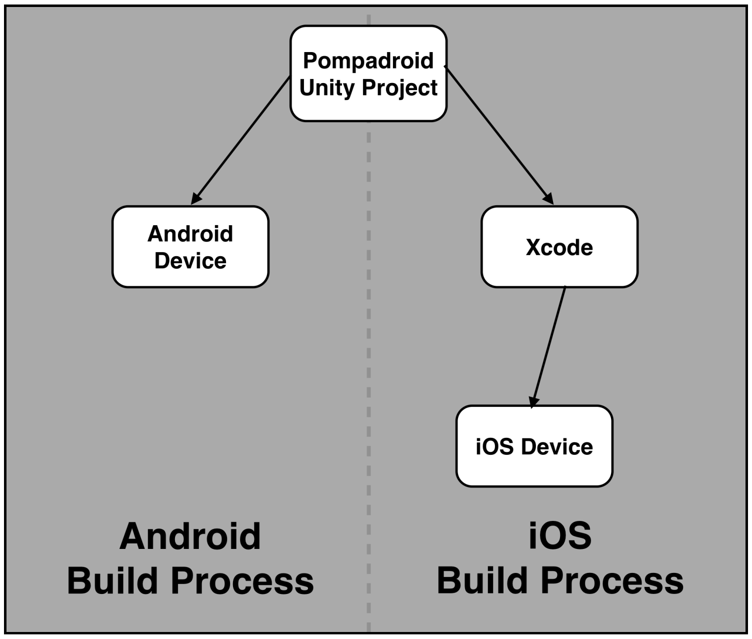
Note: If you don't have a Mac, you're done!! You made it! Congratulations! I think you should literally do a happy dance right now. But if you have a Mac and want to put this awesome game on your iPhone, keep reading. And stop dancing — you've got work to do!

Running on iOS

iOS is the second-most popular OS in terms of market share, so building for this platform is a worthwhile endeavor.

Creating a build for iOS works quite differently from Android. It is a two-step process.

1. Create an Xcode Project
2. Build the game from Xcode



For better or for worse, iOS has no back button. However, there is no need to remove the code that supports the Android back button.

The latest stable version of iOS at the time of writing is iOS 11. This book supports iOS versions 9, 10, and 11.

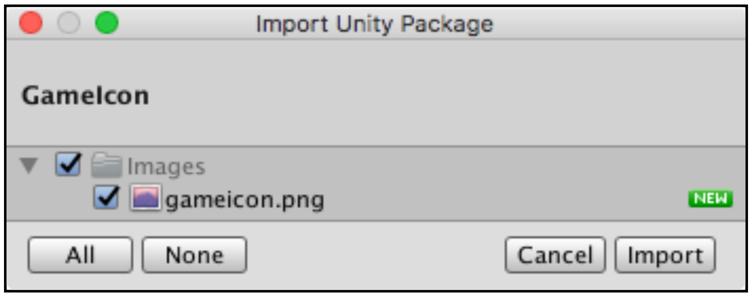
Older devices running iOS 7 or 8 should still be able to run the current project.

To create a build for iOS, and actually load it to your device, you'll need an Apple developer account. If you're not, you will be able to simulate it on your screen, at least.

Here are the other requirements for building PompaDroid on iOS:

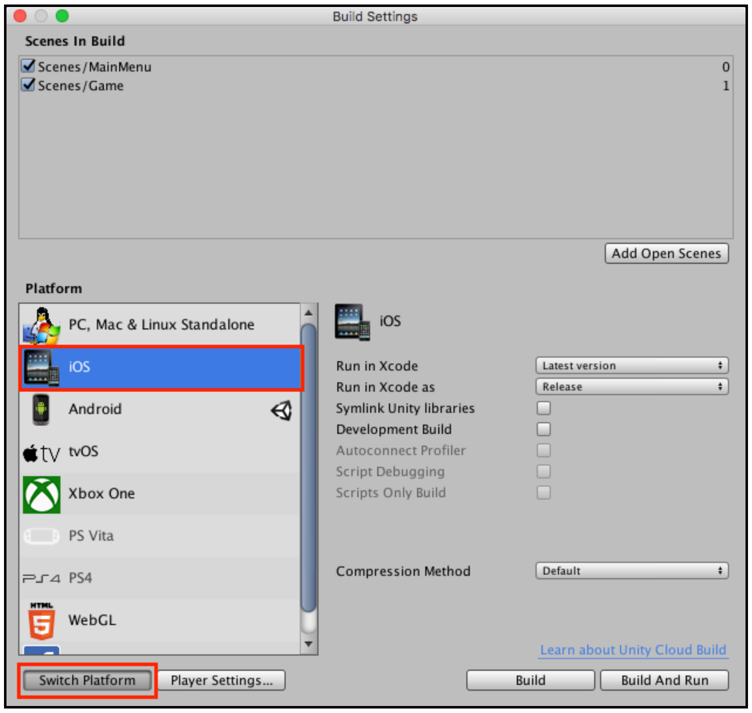
- The latest version of **Unity**, along with the **iOS Module** installed.
- **Xcode 9**, along with a computer running **macOS 10.12.6 (Sierra)** or later.
- An iOS device running **iOS 9** or later.
- An **Apple developer account**.

Import the **GameIcon.unitypackage** from the **Unity Packages** folder into the project to add the game icon sprite to the project.



As it was before, the icon is simply the hero sprite on a blue background.

Open the **Build Settings** by selecting **File \ Build Settings** from the top menu. Select iOS from the list on the bottom-left and click the **Switch Platform** button to switch to iOS.



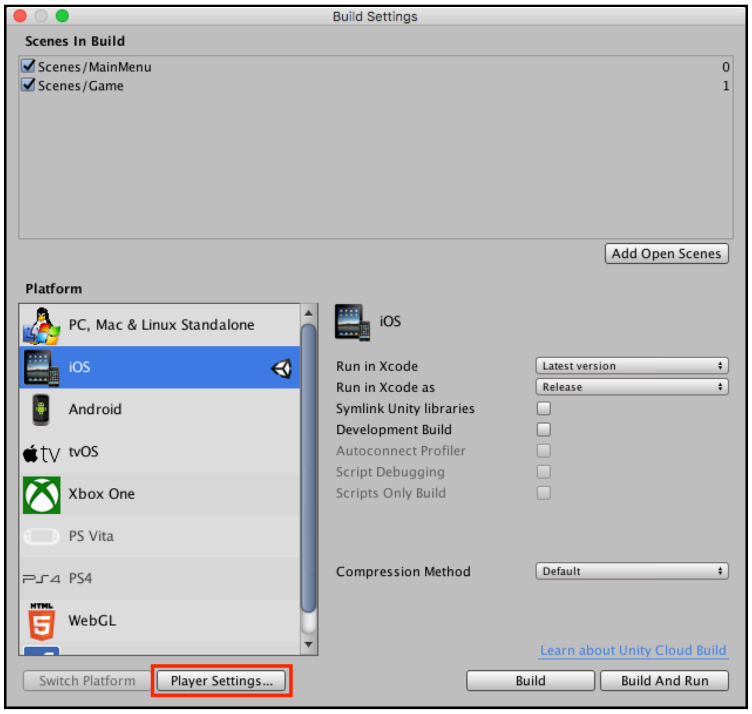
☑ Scenes/MainMenu 0
☑ Scenes/Game 1

iOS

Run in Xcode	Latest version
Run in Xcode as	Release
Symlink Unity libraries	☐
Development Build	☐
Autoconnect Profiler	☐
Script Debugging	☐
Scripts Only Build	☐
Compression Method	Default

This will re-import all assets in the game, since each platform has different requirements for compression.

Once that is done, select **Player Settings** in the **Build Settings** window to open the settings of PompaDroid.



☑ Scenes/MainMenu 0
☑ Scenes/Game 1

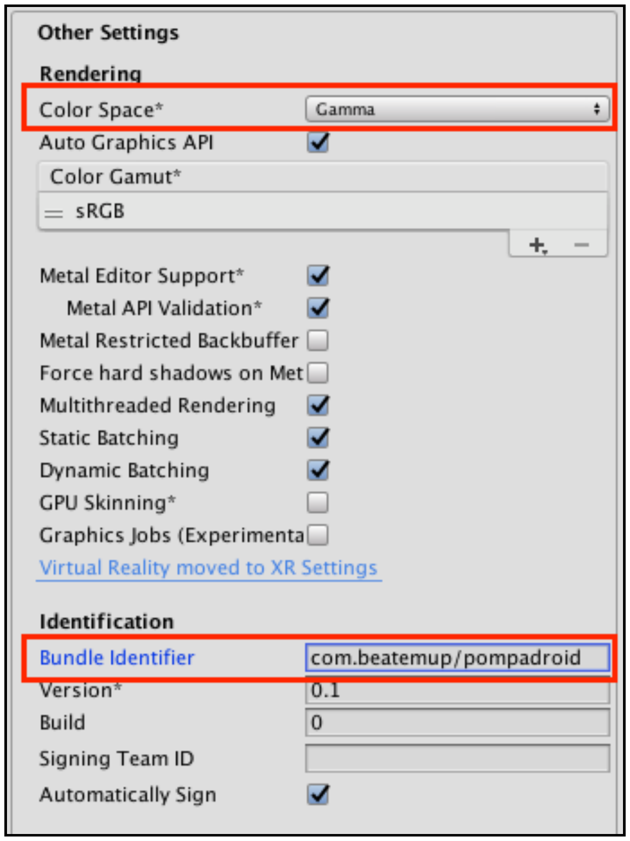
iOS

Run in Xcode	Latest version
Run in Xcode as	Release
Symlink Unity libraries	☐
Development Build	☐
Autoconnect Profiler	☐
Script Debugging	☐
Scripts Only Build	☐
Compression Method	Default

First, you'll need to assign a unique **Bundle Identifier** for your project. Bundle Identifiers usually use the following format:

```
com.<COMPANY NAME>.<PRODUCT NAME>
```

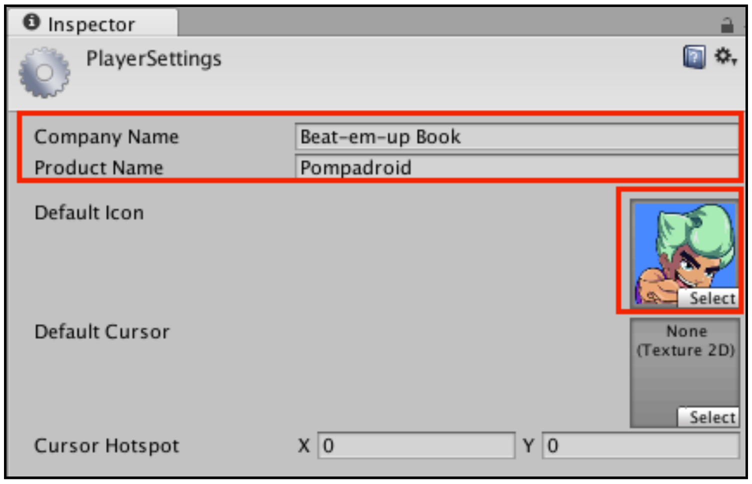
In the **Other Settings** tab, set **Color Space** to **Gamma**, **Bundle Identifier** to **com.beatemup.pompadroid** or any bundle ID of your choosing. Leave the other settings to their default values.


Other Settings
Rendering
Color Space* Gamma
Auto Graphics API ✓
Color Gamut*
sRGB
Metal Editor Support* ✓
Metal API Validation* ✓
Metal Restricted Backbuffer ☐
Force hard shadows on Met ☐
Multithreaded Rendering ✓
Static Batching ✓
Dynamic Batching ✓
GPU Skinning* ☐
Graphics Jobs (Experimenta ☐
Virtual Reality moved to XR Settings
Identification
Bundle Identifier com.beatemup/pompadroid
Version* 0.1
Build 0
Signing Team ID
Automatically Sign ✓

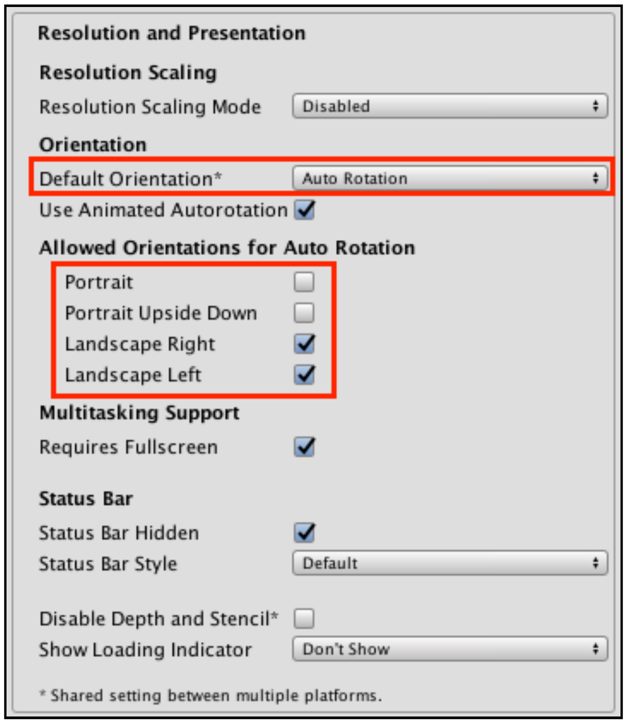
Note: Unique is a critical requirement for a bundle identifier because it ensures apps on your iOS device are never in conflict.

For demonstration purposes, this project will use the bundle identifier **com.beatemup.pompadroid**, but feel free to use your own **Bundle Identifier**.

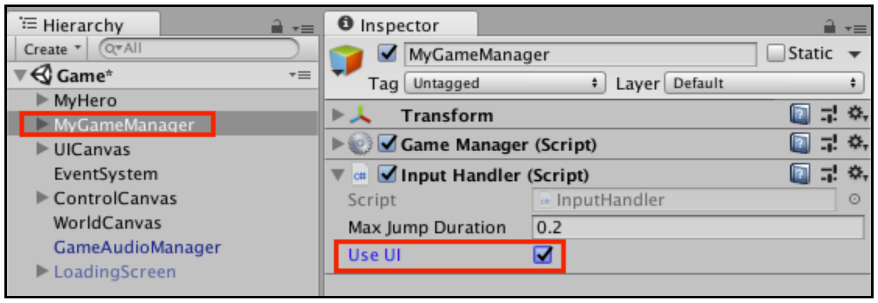
In the **PlayerSettings**, set **Company Name** to **Beat-em-up Book**, **Product Name** to **Pompadroid** and select **gameicon** as the **Default Icon**.


Inspector
PlayerSettings
Company Name: Beat-em-up Book
Product Name: Pompadroid
Default Icon
Default Cursor: None (Texture 2D)
Cursor Hotspot: X 0 Y 0

Next, in the **Resolution and Presentation** settings for iOS, set **Default Orientation** to **Auto Rotation**, uncheck **Portrait** and **Portrait Upside Down** and keep **Landscape Right** and **Landscape Left** checked. This game will run in both landscape mode variants but not in portrait mode.


Resolution and Presentation
Resolution Scaling
Resolution Scaling Mode: Disabled
Orientation
Default Orientation*: Auto Rotation
Use Animated Autorotation ✓
Allowed Orientations for Auto Rotation
Portrait ☐
Portrait Upside Down ☐
Landscape Right ✓
Landscape Left ✓
Multitasking Support
Requires Fullscreen ✓
Status Bar
Status Bar Hidden ✓
Status Bar Style: Default
Disable Depth and Stencil* ☐
Show Loading Indicator: Don't Show
* Shared setting between multiple platforms.

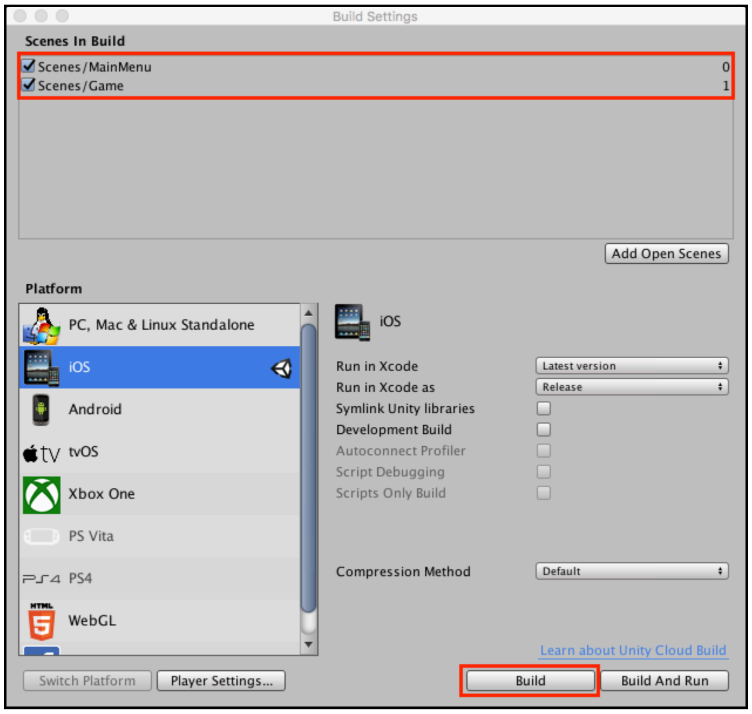
Okay, you should be able to run the app on iOS. **Save** the project and open the **Game** scene. Make sure that **MyGameManager** > **Use UI** is enabled so the game can use the touch controls.



MyGameManager
Tag Untagged Layer Default
Script InputHandler
Max Jump Duration 0.2
Use UI ✓

Save the scene and the project. Open **Build Settings** and make sure that the **MainMenu** and **Game** scenes are part of the build. Confirm that the target platform is **iOS**.

Select **Build** and map to a location to save the project.



✓ Scenes/MainMenu 0
✓ Scenes/Game 1

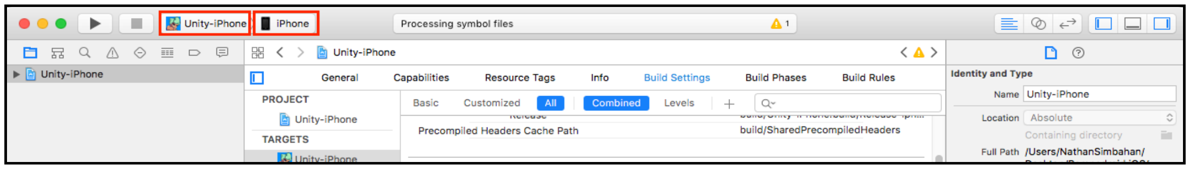
iOS
Run in Xcode: Latest version
Run in Xcode as: Release
Symlink Unity libraries: ☐
Development Build: ☐
Autoconnect Profiler: ☐
Script Debugging: ☐
Scripts Only Build: ☐
Compression Method: Default

Click **Save** and standby for the build progress to complete. Now might be a good time for another quick break!

Once complete, open the **Unity-iPhone.xcodeproj** file that was generated to open Xcode.

If needed, sign into your developer account within Xcode. To do this, select **Xcode \ Preferences** then select the **Accounts** tab.

Plug your iOS device into the computer. Select **Unity-iPhone** at the top-left corner and select your plugged-in test device.



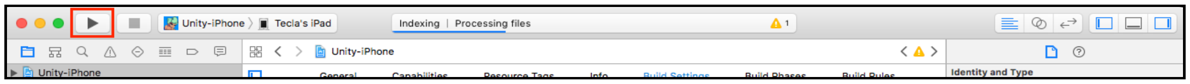
Precompiled Headers Cache Path build/SharedPrecompiledHeaders

Name Unity-iPhone

Location Absolute

Full Path /Users/NathanSimbahan/

Click the **Play** button on the top-left and hold while the build finishes.



Holy coding, Batman! PompaDroid is running on your iOS device! Savor this moment. It's beautiful, just beautiful.

Where to go from here?

In this chapter, you've:

- Implemented support for Android's back button.
- Built and tested the game on an Android device.
- Built and tested the game on an iOS device.

Huge congratulations are in order. You've managed to create your first beat 'em up game! You started out with a blank project and now have a game that runs in the palms of your hands. I hope you feel accomplished, because that was no easy feat!

Although this book and all the assets made your task easier, it was still *you* who put in the time, and now you have gained a ton of knowledge about what actually goes into building a game on Unity. Your life will never be the same!

Turn the page to explore potential next steps for this game.

Appendix: GridSnapper Classes

The GridSnapper class performs multiple tasks that ultimately help you create a tilemap for the game:

- It draws a helpful grid on the scene view.
- It snaps all attached child tile GameObjects to the grid.
- It creates the optimized mesh for the tilemap you are creating.

Let's take a closer look at it and learn more about what it does.

```
using UnityEngine;

//1
[ExecuteInEditMode]
public class GridSnapper : MonoBehaviour {

  //2
  public string filename;
  public bool autoSnapping;

  public Color gridColor = Color.white;

  //3
  public void OnDrawGizmos() {

    float scale = 1000;
    int columns = 100;
    int rows = 100;

    Vector2 offset = new Vector2(0.5f, 0.5f);

    Gizmos.color = gridColor;

    for (int j = -rows; j < rows; j++) {
      Vector3 min = new Vector3(scale + offset.x, j + offset.y, 0);
      Vector3 max = new Vector3(-scale + offset.x, j + offset.y, 0);
```

```
        min = transform.TransformPoint(min);
        max = transform.TransformPoint(max);

        Gizmos.DrawLine(min, max);
      }

      for (int i = -columns; i < columns; i++) {

        Vector3 min = new Vector3(i + offset.x, +scale + offset.y, 0);
        Vector3 max = new Vector3(i + offset.x, -scale + offset.y, 0);

        min = transform.TransformPoint(min);
        max = transform.TransformPoint(max);

        Gizmos.DrawLine(min, max);

      }
    }
  }
```

Quite a bit going on in there — here is what happens in each section.

1. The class has the ExecuteInEditMode attribute, which allows the MonoBehaviour to run in edit mode. This is necessary because the class needs to help design the level while not currently playing the game.
2. The class has three variables, a filename, an auto-snap flag and the color of the grid you want to display on screen.
3. This code block performs the first task of the GridSnapper. It implements the `OnDrawGizmos` method. It traverses all integers within a certain range going vertical and horizontal and draws lines on them. Tadaaaaa! What you have in the end is a grid.

In the next code block, you perform the second task of snapping child objects to a grid.

```
   //1
    void Update() {
      if (autoSnapping) {
        SnapChildren();
      }
    }

    //2
    public void SnapChildren() {
      foreach (Transform child in transform) {

        //do the snapping;
        Vector3 pos = child.localPosition;
        pos.x = Mathf.RoundToInt(pos.x);
        pos.y = Mathf.RoundToInt(pos.y);
        pos.z = Mathf.RoundToInt(pos.z);
        child.localPosition = pos;
      }
    }
```

Here’s what is happening in there:

1. Since the class now runs in edit mode, the `Update` method checks if the autoSnapping is enabled. If true, it calls the `SnapChildren` method.

2. The `SnapChildren` method iterates through all child Transforms of the GridSnapper and rounds its X, Y, and Z localPosition to the nearest integer, thereby snapping the object to the grid.

Finally, you create the optimized mesh in the following block:

```
    public Mesh MakeMesh() {
      Mesh mesh = new Mesh();

      int polygons = transform.childCount;

      Vector3[] vertices = new Vector3[polygons * 4];
      Vector2[] uvs = new Vector2[polygons * 4];
      int[] tris = new int[6 * polygons];

      for (int i = 0; i < polygons; i++) {
        SpriteRenderer spriteRenderer =
  transform.GetChild(i).GetComponent<SpriteRenderer>();

        vertices[i * 4 + 0] = spriteRenderer.transform.localPosition +
  (Vector3)spriteRenderer.sprite.vertices[3];
        vertices[i * 4 + 1] = spriteRenderer.transform.localPosition +
  (Vector3)spriteRenderer.sprite.vertices[1];
        vertices[i * 4 + 2] = spriteRenderer.transform.localPosition +
  (Vector3)spriteRenderer.sprite.vertices[0];
        vertices[i * 4 + 3] = spriteRenderer.transform.localPosition +
  (Vector3)spriteRenderer.sprite.vertices[2];

        uvs[i * 4 + 0] = spriteRenderer.sprite.uv[3];
        uvs[i * 4 + 1] = spriteRenderer.sprite.uv[1];
        uvs[i * 4 + 2] = spriteRenderer.sprite.uv[0];
        uvs[i * 4 + 3] = spriteRenderer.sprite.uv[2];

        tris[i * 6 + 0] = (i * 4) + 0;
        tris[i * 6 + 1] = (i * 4) + 2;
        tris[i * 6 + 2] = (i * 4) + 1;
        tris[i * 6 + 3] = (i * 4) + 2;
        tris[i * 6 + 4] = (i * 4) + 3;
        tris[i * 6 + 5] = (i * 4) + 1;
      }

      mesh.vertices = vertices;
      mesh.uv = uvs;
      mesh.triangles = tris;

      mesh.RecalculateNormals();
      mesh.RecalculateBounds();
```

```
        return mesh;
    }
```

The `MakeMesh` method combines all the child Transforms of the GridSnapper into a single Mesh. It performs all allocations for the number of vertices, UVs and triangles based on the number of children. It also populates it by using the values of each child transform.

And that's it! That's how the GridSnapper works its magic. Pretty cool, right?

Conclusion

Whew! It’s been a long journey, but you made it. You’ve built PompaDroid from scratch, creating all the classes and components for yourself. Not only that, you know your way around Unity.

The real prize here is that you’ve created a framework upon which you can continue to build and learn by adding more features to the game.

Here are a few ideas to get you started:

- Create a varying decision-making AI using the **EnemyAI** class. All you’d need to do is play around with the weights for each possible action for a given situation.
- Create a bigger world by making more **tiled map level prefabs** and adding more **LevelData** Scriptable Objects to the game.
- Add more **EnemyTypes**, more **Powerups**, more **breakable objects** — more of everything, really!

And if I were going to keep going, I’d toy around with these ideas:

- Create more actions—for example, a double-tap up or down to make the hero jump tiles quickly.
- Create character and level selection screens.
- Give the player multiple lives and access to objects that restore hit points.
- Challenge the hero by allowing some droids to self-heal.
- More power-ups and bigger, badder bosses.
- Music and sound effects to suit my taste.

On behalf of the book team, I’d like to say that it has been our pleasure to create this starter kit for you, and we thank you once again for picking up this book! Your support is what allows the team behind raywenderlich.com to perpetually produce useful, timely, and relevant tutorials of all kinds, and for that opportunity, we are truly grateful. Couldn’t do it without you!

— Nathan, Maria Gelyn, Eric, Allen, and Wendy

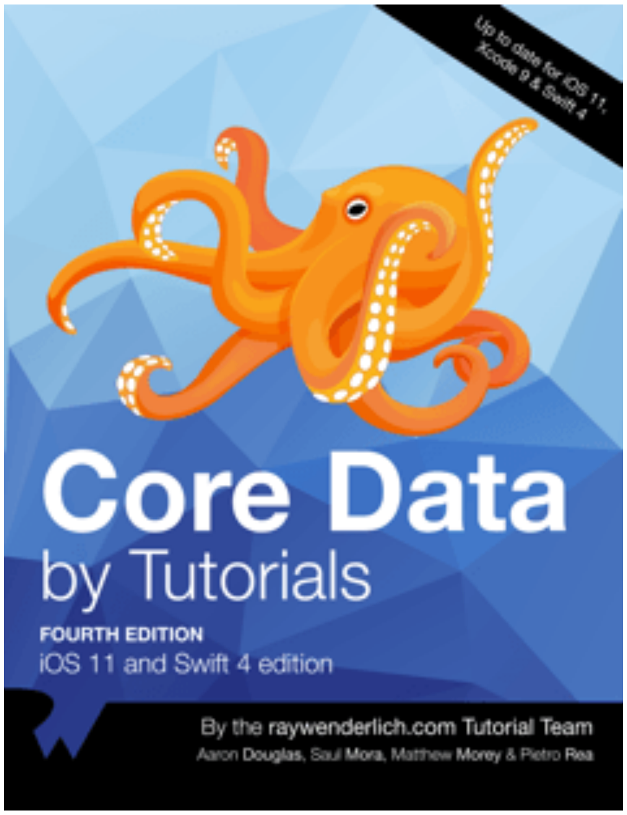
More Books You Might Enjoy

Up to date for iOS 11, Xcode 9 & Swift 4
iOS Apprentice
SIXTH EDITION
Beginning iOS development with Swift 4
Over 10,000 copies sold!
By Matthijs Hollemans & Fahim Farook

Up to date for iOS 11, Xcode 9 & Swift 4
Swift
Apprentice
THIRD EDITION
Beginning programming with Swift 4
By the raywenderlich.com Tutorial Team
Janie Clayton, Alexis Gallagher, Matt Galloway, Eli Ganim, Ben Morrow, Cosmin Pupăză, and Steven Van Impe

Core Data by Tutorials

https://store.raywenderlich.com/products/core-data-by-tutorials


Up to date for iOS 11, Xcode 9 & Swift 4
Core Data
by Tutorials
FOURTH EDITION
iOS 11 and Swift 4 edition
By the raywenderlich.com Tutorial Team
Aaron Douglas, Saul Mora, Matthew Morey & Pietro Rea

This book is for intermediate iOS developers who already know the basics of iOS and Swift 4 development but want to learn how to use Core Data to save data in their apps.

Start with with the basics like setting up your own Core Data Stack all the way to advanced topics like migration, performance, multithreading, and more!