

Contenido

Configuración Unity.....	7
1. ¿Cómo se configura el Editor Unity?.....	7
2. ¿Cómo se crea un Project (Proyecto)?	7
3. ¿Cómo importamos Assets (Recursos)?.....	9
4. La interfaz del Editor Unity.....	11
4.1. La ventana Project (Proyecto)	11
4.2. La ventana Console (Consola)	11
4.3. La ventana Hierarchy (Jerarquía)	12
4.4. La vista Scene (Escena)	12
4.5. La ventana Game (Juego)	12
4.6. La ventana Inspector	13
5. La barra de herramientas y la interfaz de usuario Navigation (Navegación).....	15
5.1. La barra de herramientas	15
5.2. Los botones Play (Reproducción)	15
5.3. Cómo manipular objetos.....	15
5.4. Cómo navegar con el ratón	17
6. Las disposiciones de pantallas o Layouts	17
7. Resumen.....	18
El personaje del jugador: primera parte	19
1. ¿Cómo se guarda una Scene (escena)?.....	19
2. ¿Cómo se agrega un personaje modelo?	20
3. ¿Qué son los Prefabs?	25
4. ¿Cómo transformar el personaje en un Prefab?.....	26
5. ¿Cómo animar tu personaje?	28
6. ¿Como se crea el Animator Controller (Controlador de animación)?	28
7. ¿Cómo se configuran las animaciones?	31
8. ¿Cómo se crean las Animator transitions (transiciones de animación)?	33
9. ¿Cómo se añaden Conditions (condiciones) a tu transición?	35
10. ¿Cómo se le asigna un Animator Controller al Prefab JohnLemon?	36
11. ¿Cómo logras que tu personaje reaccione a la física?	37
12. ¿Qué es Root Motion (Movimiento de raíz)?.....	39
13. ¿En qué consiste el bucle Update (Actualizar)?	40
14. ¿Cómo puedes arreglar los movimientos de John Lemon?	40
15. El sistema de coordenadas de Unity	42

16.	Posiciones y rotación.....	43
17.	¿Cómo añadirle un Collider (Colisionador) a John Lemon?	44
18.	Resumen.....	46
	El personaje del jugador: Segunda parte	47
1.	Continuación del personaje del jugador	47
2.	¿Cómo puedes crear tu primer script (PlayerMovement)?	47
3.	¿Cómo creaste esa variable?.....	52
4.	¿Cómo podemos hacer un vector?	53
5.	Configura los valores de tu variable	54
6.	¿Cómo se configura el componente Animator (animador)?	56
7.	¿Cómo se identifica si hay entradas del teclado del jugador o no?.....	56
8.	¿Cómo se crea una variable para guardar una referencia hacia el componente Animator (Animador)?	57
9.	¿Cómo se hace una referencia que apunte hacia el componente Animator?.....	58
10.	¿Cómo se configura la referencia que apunta hacia el componente Animator?	59
	¿Cuál es el significado de esta referencia?	60
	¿Cómo se configura el isWalking parámetro de tipo Animator?	60
11.	¿Cómo puedes crear una rotación para tu personaje?.....	61
	¿Cómo crear una variable turnSpeed?.....	62
	¿Cómo puedes calcular el forward vector (vector delantero) de tu personaje?	62
12.	¿Cómo se ajusta la variable turnSpeed (Velocidad de giro)?	63
	¿Cómo crear y guardar una rotación?.....	64
13.	¿Cómo aplicar movimiento y rotación a tu personaje?	65
14.	Movimiento	66
15.	Rotación	66
16.	¿Cómo puedes hacer cambios a tu método Update (actualizar)?.....	67
17.	¿Cómo puedes poner a prueba los cambios que has hecho?	69
	¿Cómo se le añade el script PlayerMovement (Movimiento del jugador) a John Lemon? 69	
18.	¿Cómo se ajustan las configuraciones de tu vista Game (Juego)?.....	70
19.	Resumen.....	71
	El entorno.....	72
1.	¿Cómo se añade el entorno?	72
	¿Cómo se posiciona el personaje del jugador?	73
2.	¿Cómo se ilumina el entorno?	73
3.	¿Cómo se cambia la Luz direccional (Directional Light)?	73
4.	¿Cómo se crea un efecto de iluminación global con Lightmapping?.....	77

5. ¿Cómo se añade una malla de navegación o Navigation Mesh?	80
¿Cómo se designa un GameObject como algo estático?	80
6. ¿Cómo se crea la Malla de navegación o Nav Mesh?	82
7. Resumen.....	84
La cámara	85
1. Explore el componente de la cámara.....	85
2. ¿Cómo funciona Cinemachine?.....	86
3. Configure una cámara virtual con Cinemachine	87
4. Cambie la configuración del componente de la cámara virtual Cinemachine	88
5. Agregar efectos de posprocesamiento	94
6. Crear una capa de posprocesamiento	95
7. Mejora la calidad de imagen con Anti-aliasing	98
8. Crear (Post-Processing Volume) volumen de postprocesamiento	99
9. Agregue el efecto de gradación de color (Color Grading Effect)	100
10. Añade el efecto Bloom	103
11. Agregue el efecto de ambient occlusion	104
12. Agregue el efecto de viñeta (Vignete)	105
13. Agregue el efecto de (lens distortion) distorsión de lente	106
14. Resumen	108
Terminando el juego	109
1. Configurar la interfaz de usuario.....	109
2. Configure el lienzo.....	112
3. Estira la imagen	114
4. Explore el componente Rect Transform	115
5. Configure el componente Rect Transform	117
6. Agregar una imagen de victoria	119
7. Agregar un componente de grupo de lienzo	122
8. Crear un disparador de final de juego.....	123
9. Crear un nuevo script.....	125
10. Inicie la secuencia de comandos GameEnding	126
11. Agregar un método de actualización	127
12. Escriba el bloque de código de declaración if	129
13. Establezca las variables para su secuencia de comandos GameEnding	132
14. Resumen.....	133
Enemigos, Parte 1: Observadores estáticos.....	134
Objetivo	134

1.	Configuración del prefabricado de gárgola	134
2.	Animar la gárgola	136
3.	Agregar un colisionador a la gárgola.....	137
4.	Crea un disparador para simular la línea de visión de la gárgola.....	138
5.	Escriba un script de observador personalizado	141
6.	Agregue una clase para detectar el personaje del jugador	142
7.	Comprueba que la línea de visión del enemigo está despejada	144
8.	Revise su secuencia de comandos GameEnding	148
9.	Crea dos formas de terminar el nivel	149
10.	Permitir al jugador reiniciar el nivel	152
11.	Completa tu Prefab de gárgola	155
12.	Resumen.....	158
	Enemigos, Parte 2: Observadores dinámicos.....	159
	Objetivo	159
1.	Configurar el fantasma prefab	159
2.	Animar al fantasma	160
3.	Agregar un colisionador al fantasma.....	161
4.	Agregue un componente Rigidbody al Ghost GameObject.....	163
5.	Haz del fantasma un observador	163
6.	Configure un componente Nav Mesh Agent.....	166
7.	Crear un nuevo script WaypointPatrol.....	169
8.	Establezca el destino del agente de malla de navegación	170
9.	¿Cómo funcionan las matrices?	171
10.	Aregar destinos adicionales.....	171
11.	Asigne la referencia del agente de malla de navegación al fantasma prefabricado	173
12.	Coloca fantasmas en tu escena	174
13.	Crear y posicionar los puntos de referencia fantasma	175
14.	Coloca las gárgolas en tu escena.....	176
15.	Limpiar la jerarquía	176
16.	Resumen.....	179
	Audio	180
1.	Audio	180
2.	Una cartilla sobre audio en la unidad	180
3.	¿Qué es el audio no diegético?	181
4.	Crea fuentes de audio para tu juego.....	181
	7. Regrese a su secuencia de comandos GameEnding.....	186

8. Actualice su secuencia de comandos GameEnding para reproducir audio	187
13. Actualice su script PlayerMovement para reproducir audio	193
14. Ajuste su método para usar las nuevas variables	193
16. Actualice el prefabricado JohnLemon	197
19. Corrige la dirección del efecto de sonido fantasma	201
Construir, ejecutar, distribuir	204
1. Ajuste la configuración básica del reproductor	204
2. Ajuste la configuración de resolución y presentación	205
3. Construye tu juego	206
4. ¿Qué produce Unity?	208
5. Resumen	208

Resumen

¡Bienvenidos al Paseo encantado de John Lemon: Proyecto para principiantes en 3D! En este proyecto no solo descubrirás cómo crear un juego de sigilo — cada uno de los 10 tutoriales también explica los principios detrás de cada paso. No se necesita experiencia previa, lo cual hace que el juego John Lemon's Haunted Jaunt (El paseo encantado de John Lemon) sea el comienzo perfecto de tu viaje con Unity.

John Lemon

Versiones de Unity recomendadas

2018.3 - 2019.1

Configuración Unity

1. ¿Cómo se configura el Editor Unity?

¡Bienvenida al proyecto para principiantes en 3D: El paseo encantado de John Lemon! En este proyecto, no solo descubrirás **cómo** crear un juego de sigilo, sino que también cada uno de los 10 tutoriales explica **el principio detrás de cada paso que vayas a tomar**. No se requiere experiencia previa, lo que hace que El paseo encantado de John Lemon sea el juego perfecto para emprender tu viaje con Unity.

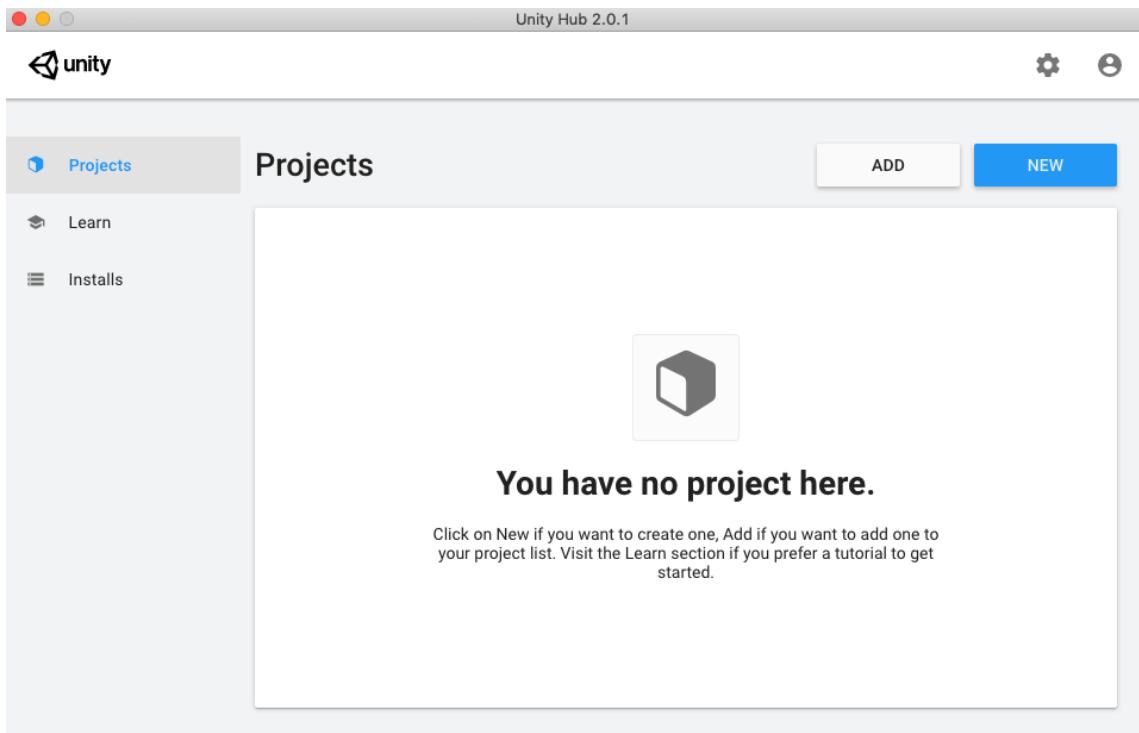
En este primer tutorial explorarás el Editor Unity y te prepararás para comenzar a hacer tu propio juego de sigilo.

Antes de que puedas crear un nuevo Project (Proyecto) necesitas tener la versión 2019.1 del Editor Unity instalada. Para hacerlo puedes instalar el Unity Hub (Centro Unity) visitando [Descarga la versión personal de Unity](#) en el sitio web de Unity.

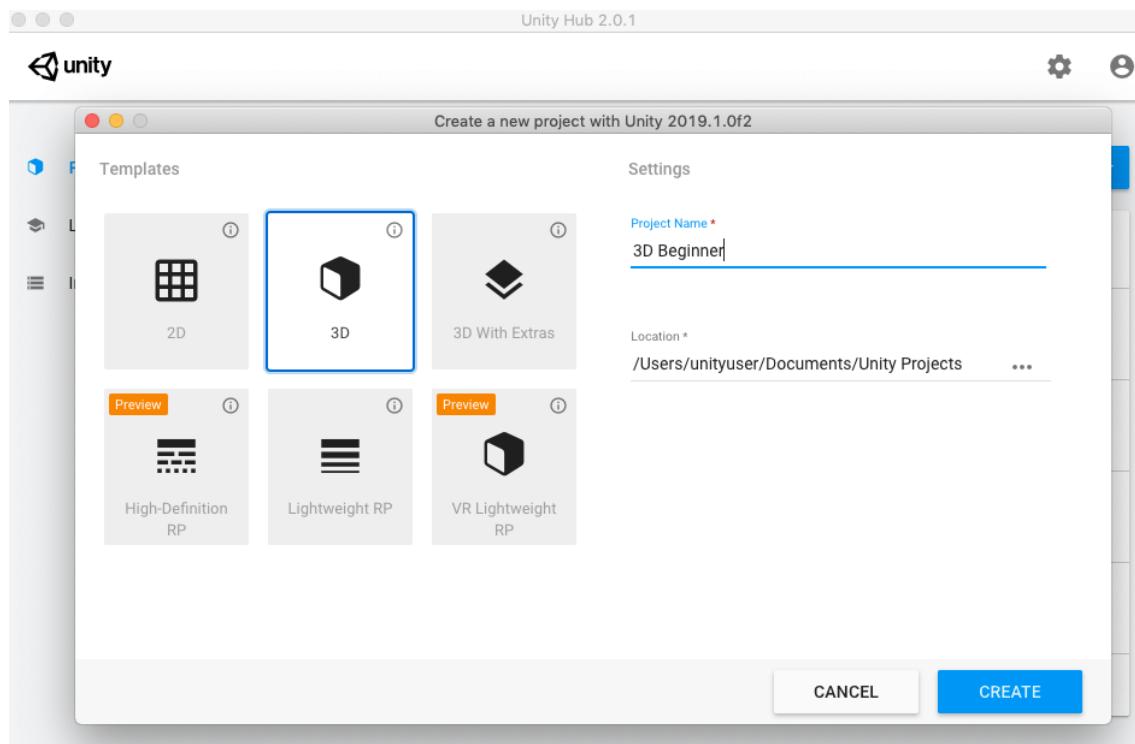
2. ¿Cómo se crea un Project (Proyecto)?

Para crear un nuevo Project (Proyecto) para tu juego;

- Abre el Unity Hub (Centro Unity)
- Inicia sesión con tu cuenta.
- Haz clic en el botón **New** (Nuevo) en la esquina superior derecha.



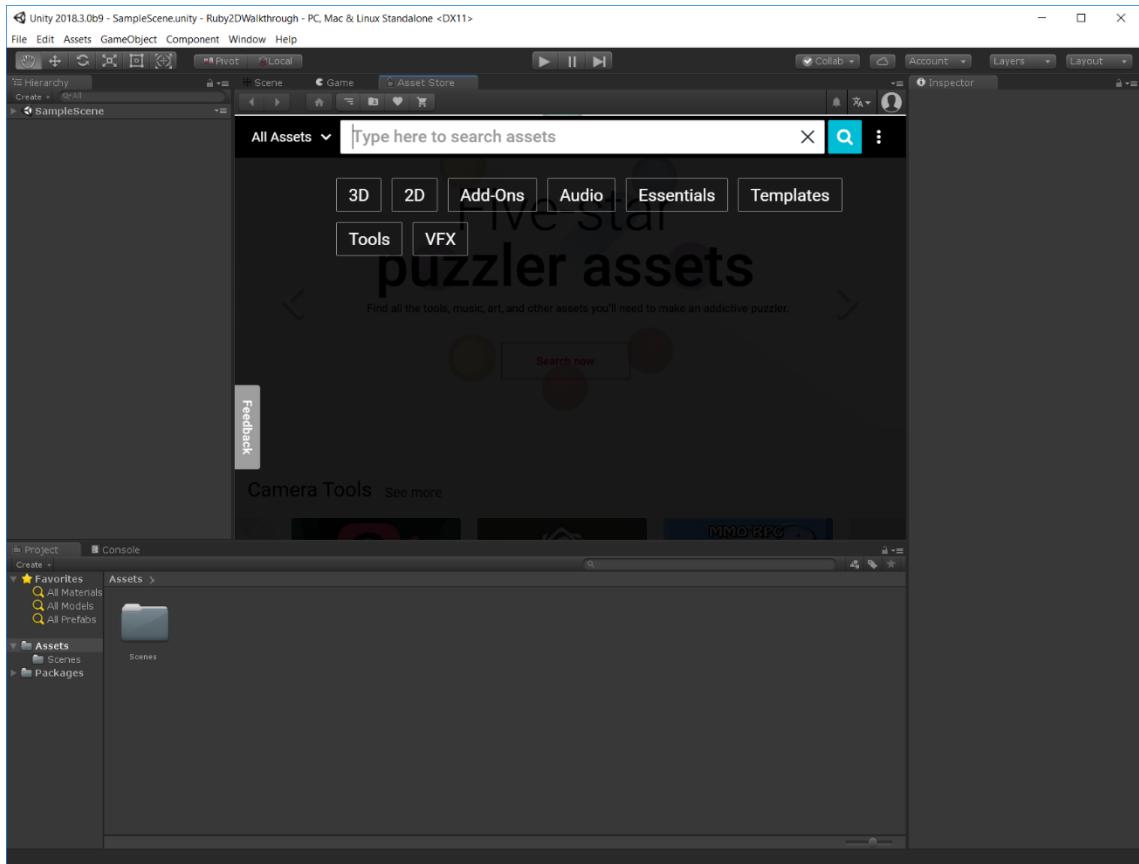
- Ingresa un **Project name** (Nombre del proyecto) — vamos a ponerle el nombre «Principiante 3D».
- Configura la **versión de Unity** a **2019.1**. (Se recomienda versión 2018.2 o superior)
- Escoge la carpeta en la que quieras guardar tu Project (Proyecto).
- Asegúrate de que el formato sea **3D**.
- Haz clic en **Create project (Crear Proyecto)**.



3. ¿Cómo importamos Assets (Recursos)?

Antes de que explores el Editor de Unity, vamos a importar los archivos Assets (Recursos) que necesitarás para este Project (Proyecto). Todos estos archivos están en la Unity Asset Store (Tienda de recursos de Unity), la cual le permite a los creadores de Assets (Recursos) proveer herramientas o archivos a otros usuarios de Unity. Para acceder a la tienda ve a **Window > General > Asset Store**.

La Asset Store (Tienda de recursos) se abrirá dentro de tu Editor.



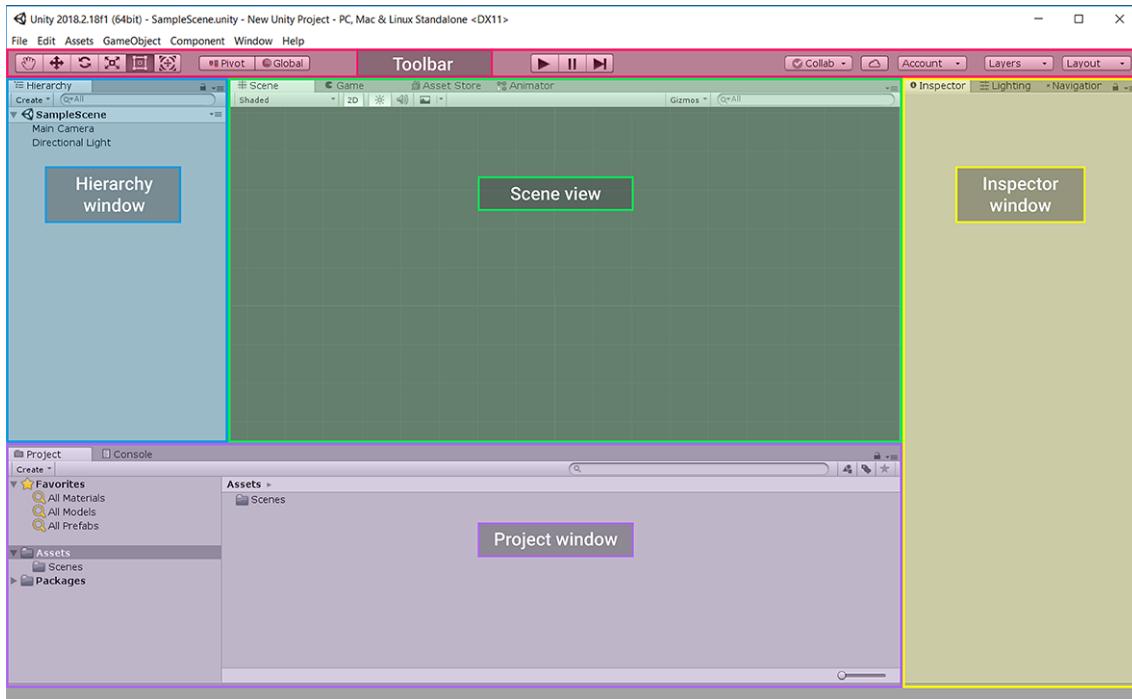
Los pasos a seguir para cargar los Assets (Recursos) dentro de la Scene (Escena):

1. En la barra de búsqueda ingresa "**3D Beginner: Tutorial Resources**" y haz clic en el resultado de la búsqueda.
2. En la página 3D Beginner: Tutorial Resources (Principiante en 3D: Recursos para el tutorial), haz clic en **Download** y espera que termine la descarga.
3. Haz clic en **Import. (Importar)**.
4. Una casilla de diálogo aparecerá diciendo que tus Project Settings (Configuración del proyecto) serán reemplazados. Esto es lo que necesitas—haz clic en **Import** (Importar) para continuar. Esta acción abre la ventana Import Unity Package (Paquete de Importación de Unity).
5. Haz clic en **Import (Importar)** para traer estos archivos a tu Unity Project (Proyecto de Unity).

¡Eso es todo! Ahora que estás lista para comenzar, vamos a explorar para entender cómo funciona el Editor.

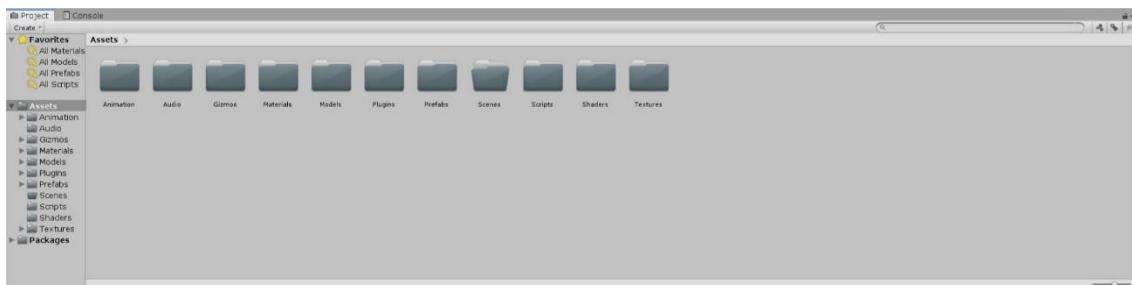
4. La interfaz del Editor Unity

El Editor Unity puede ser intimidante o abrumante cuando estás comenzando a explorarlo, pero esta guía te ayudará a familiarizarte con ello fácilmente. Si alguna vez te olvidas en dónde algo está ubicado algo, puedes regresar a este tutorial en cualquier momento.



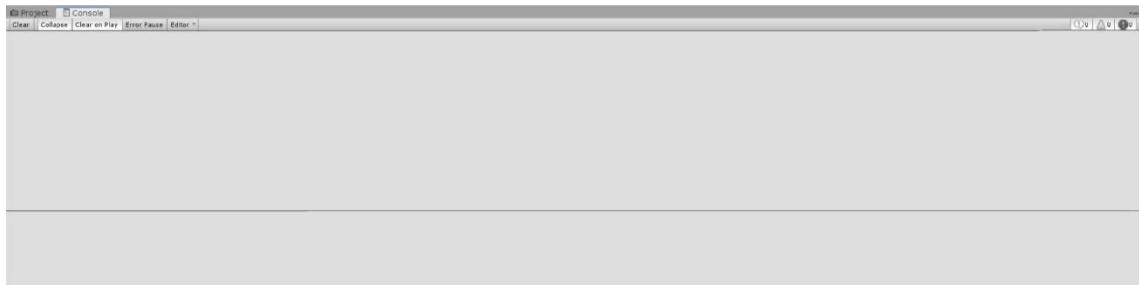
4.1. La ventana Project (Proyecto)

La ventana Project (Proyecto) enumera todos los archivos y directorios en tu Project (Proyecto) actual. Estos archivos incluyen todas las imágenes, sonidos, modelos 3D y otros artículos que se usan en tu Project. Se los conoce en conjunto como **Assets (Recursos)**.



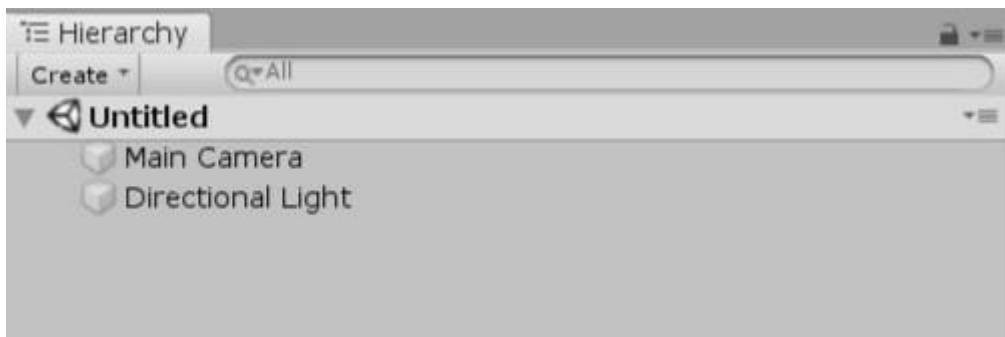
4.2. La ventana Console (Consola)

La ventana Console (Consola) te mostrará las advertencias y errores que está produciendo tu juego y puede darte información útil para corregir esos errores. Por defecto, la pestaña de la ventana Console está junto a la de la ventana Project. Puedes moverla al arrastrando y soltando la pestaña para anclar la ventana junto a la ventana Project.



4.3. La ventana Hierarchy (Jerarquía)

En Unity los juegos están compuestos de Scenes (Escenas). Considera que una Scene es un nivel en tu juego o un entorno diferente. En cada Scene, tienes una lista de objetos que tiene un lugar en esa Scene (por ejemplo los personajes y la decoración). Estos objetos se llaman**GameObjects (Objetos de juego)** en Unity.

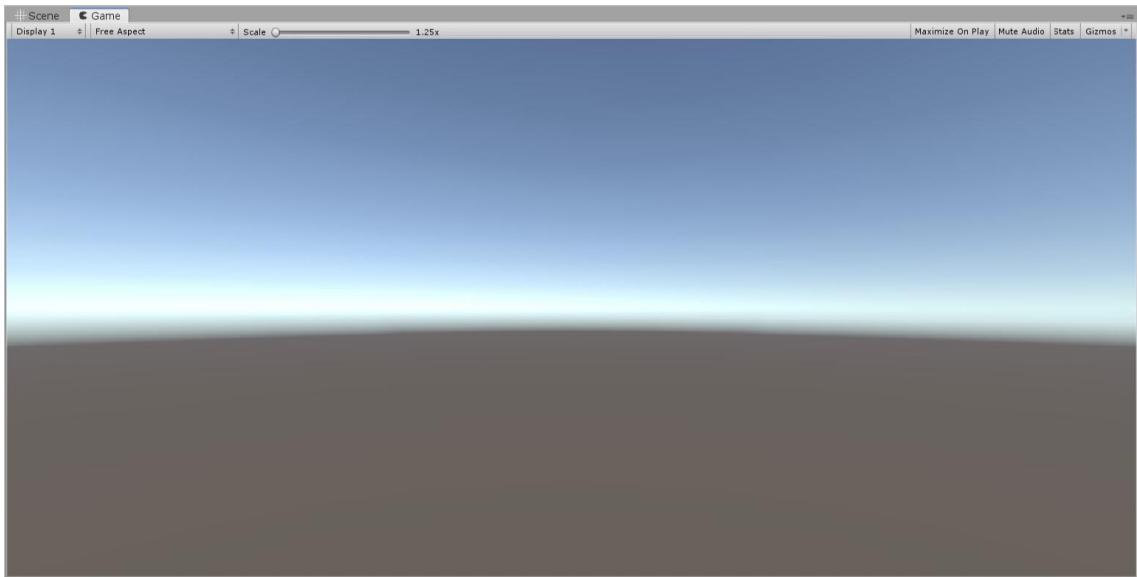


4.4. La vista Scene (Escena)

La vista Scene (Escena) es una ventana donde podemos ver una presentación preliminar en vivo de lo que hay en la Scene. Muestra todas las Scenes que están cargadas actualmente y todos los GameObjects en la ventana Hierarchy. Puedes usar esta ventana para ubicar y mover GameObject en tu Scene. Hacer clic en GameObject en la vista Scene va a resaltarlo en la ventana Hierarchy.

4.5. La ventana Game (Juego)

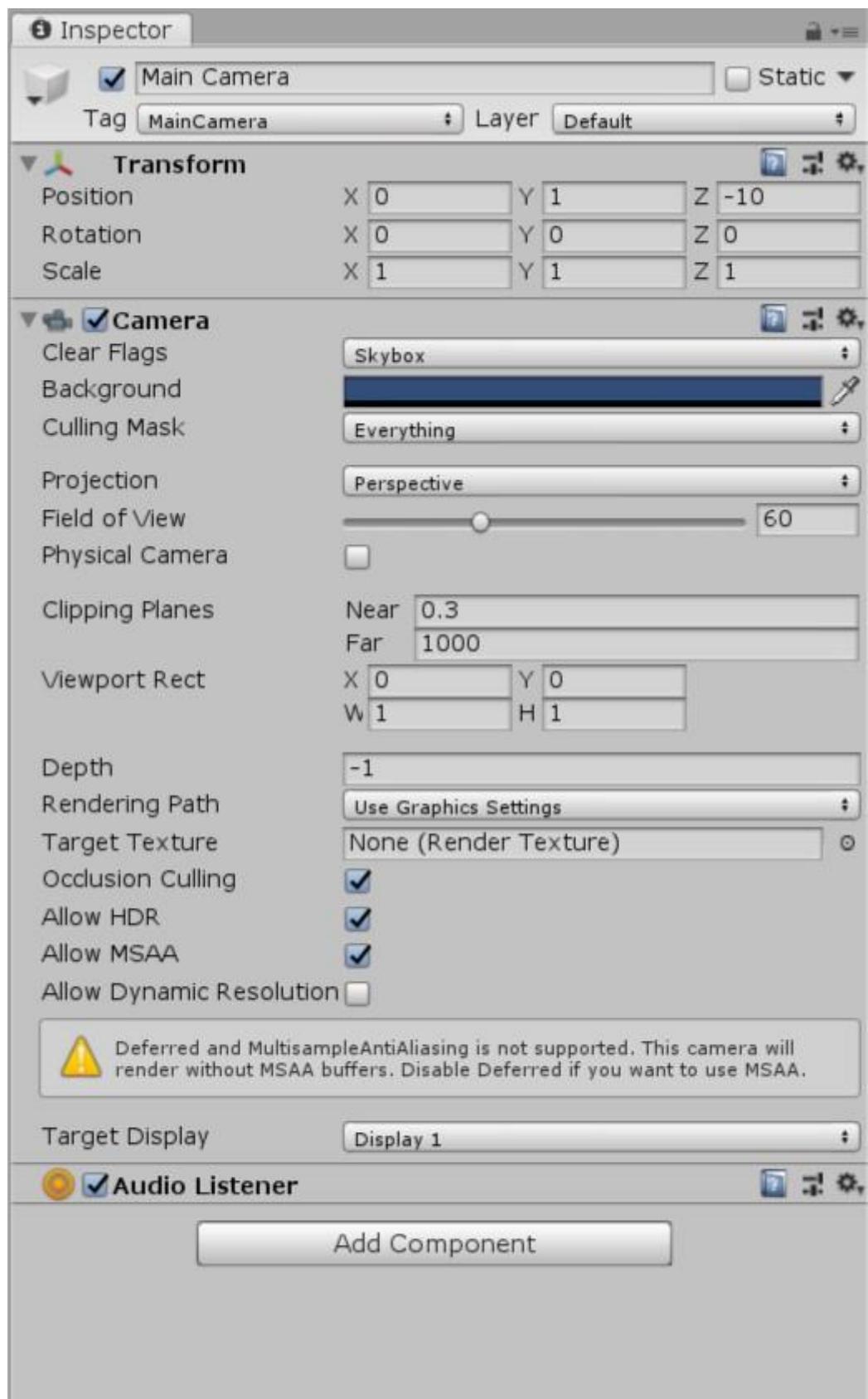
La vista Game está escondida en forma de pestaña junto a la vista Scene por defecto. Mientras la vista Scene te permite mover GameObjects y echarle un vistazo a la Scene entera, la vista Game te muestra lo que el jugador verá cuando esté jugando el juego. Esto es determinado por la que está viendo la cámara dentro de tu Scene. En este momento no hay nada visible en la Scene, por lo tanto no puedes ver nada en la imagen de ejemplo:



La ventana Game es la vista que se mostrará cuando estés testeando tu juego dentro del Editor.

4.6. La ventana Inspector

Cuando seleccionas un GameObject en otra ventana, la ventana Inspector mostrará todos los datos relacionados con ese objeto. Unity usa un **modelo Object - Component (Objeto - Componente)**, lo que significa que puedes añadir diferentes componentes a GameObjects para cambiar sus características. Por ejemplo, un componente Camera (Cámara) le permite al jugador tener una vista dentro de la escena.



Todos los GameObjects empiezan con un componente Transform (Transformación) que te permite especificar su posición y rotación en la Scene.

Otros componentes son opcionales y puedes añadirlos cuando sean necesarios.

5. La barra de herramientas y la interfaz de usuario Navigation (Navegación)

5.1. La barra de herramientas

La barra de herramientas incluye una gama de botones con herramientas útiles que te ayudan a diseñar y a testear tu juego.



5.2. Los botones Play (Reproducción)



Play (Jugar/Reproducir)

Se usa Play (Jugar/Reproducir) para testear la Scene que está cargada en este momento en la ventana Hierarchy y la cual te permite testear tu juego en vivo en el Editor.

Pause (Pausar)

Pause (Pausar), como lo habrás adivinado, te permite pausar el juego que está rodando en la ventana Game. Esto te ayuda a visualizar problemas o asuntos relacionados a la jugabilidad del juego que de otra manera no serías capaz de ver.

Step (Marco por Marco)

Step (Marco por marco) se usa para revisar la Scene pausada marco por marco. Esto funciona muy bien cuando estás buscando cambios en vivo en el mundo del juego en los que ayudaría mucho verlos en tiempo real.

5.3. Cómo manipular objetos



Estas herramientas mueven y manipulan los GameObjects en la ventana Scene. Puedes hacer clic en los botones para habilitarlos o usar una tecla de acceso rápido.

La herramienta Hand (Mano)

Puedes usar esta herramienta para mover tu Scene de un lado al otro dentro de la ventana.

Puedes usar el botón de en medio en el ratón para acceder a esta herramienta.



Tecla de acceso rápido: Q

La herramienta Move (Mover)

Esta herramienta te permite seleccionar artículos y moverlos individualmente.



Tecla de acceso rápido: W

La herramienta Rotate (Rotar)

Selecciona artículos y rótales con esta herramienta.



Tecla de acceso rápido: E

La herramienta Scale (Poner a escala)

Como te habrás dado cuenta, con esta herramienta puedes seleccionar artículos y ponerlos a la escala que quieras.



Tecla de acceso rápido: R

La herramienta Rect Transform (Transformación rectangular)

Esta herramienta hace muchas cosas. Combina las acciones de mover, rotar y poner a escala en una sola herramienta especializada para objetos 2D y UI (Interfaz de Usuario).



Tecla de acceso rápido: T

Rotar, Mover o Poner a escala.

Una vez más, esta herramienta hace muchas cosas. También te permite mover, rotar o poner a escala GameObject, pero está especializada para objetos en 3D.



Tecla de acceso rápido: Y

Otra combinación de acceso rápido muy útil que tal vez querrás recordar es la tecla **F**, la cual te permite enfocarte en un objeto seleccionado. Si te olvidas dónde está un GameObject en tu Scene, selecciónalo en Hierarchy, mueve tu cursor sobre la ventana Scene y pulsa **F** para centrarlo en ella.

5.4. Cómo navegar con el ratón

Cuando estás en la ventana Scene, también puedes:

- hacer clic para seleccionar tu GameObject en la Scene;
- hacer clic con el botón medio del ratón para mover el visor de la cámara de Scene usando la herramienta mano.
- hacer clic derecho y rotar la cámara de la vista Scene usando el modo flythrough (vuelo de pájaro) una variación de la herramienta mano. Mientras haces esto también puedes mover la cámara hacia la izquierda y la derecha usando A y D, hacia adelante y hacia atrás usando W y S, y hacia abajo o hacia arriba usando Q y E.

Para obtener más información sobre cómo mover GameObjects en la vista Scene, consulta [Scene View Navigation \(Navegación de la vista Scene\)](#).

6. Las disposiciones de pantallas o Layouts

Disposiciones de pantallas o *layouts* por defecto

Tú puedes disponer la posición de tu Editor Unity en distintas maneras. Cada disposición de pantalla o *layout* tiene sus propias ventajas y tú encontrarás la que funcione mejor para ti.

Para cambiar la disposición de tu pantalla selecciona **Window > Layouts (Ventana > Disposición de pantalla)** (o usa el menú desplegable Layout en la parte superior derecha del Editor). Puedes escoger entre las disposiciones de pantalla siguientes:

2 by 3 (2 por 3)

Esta disposición de pantalla te permite ver tu vista Scene y tu vista Game juntas con tu Inspector. La vista Project y la vista Hierarchy están ubicadas en columnas a la derecha.

4 Split (División en 4)

Esta disposición de pantalla es muy útil para ver a los modelos 3D desde diferentes ángulos.

Tall (Alta)

Esta disposición de pantalla hace que la vista Scene y el Inspector sean muy claros y agradables a la vista.

Wide (Ancha)

Esta disposición de pantalla empuja las ventanas Hierarchy, Project y Assets hacia el fondo del Editor.

Default (Por defecto)

Esta disposición de pantalla hace que tu vista Scene o tu vista Game tomen la mayor parte del Editor y también que haya espacio para las ventanas Assets, Hierarchy e Inspector en ambos lados. Se trata de adaptar la disposición de pantalla a las preferencias personales de los usuarios. Todos los tutoriales para John Lemon's Haunted Jaunt usan la disposición de pantalla Default con la ventana Console al costado de la ventana Project (como has visto en la sección de este tutorial que habla sobre la ventana Console).

7. Resumen

Ahora que sabes un poco más sobre la manera en la que el Editor Unity está organizado —cuando los tutoriales se refieran a la ventana Project, sabrás exactamente donde buscar para encontrarla. También has añadido Assets (Recursos) para tu juego al Proyect (proyecto). En el tutorial siguiente comenzarás a trabajar en tu juego al empezar a crear tu personaje jugador.

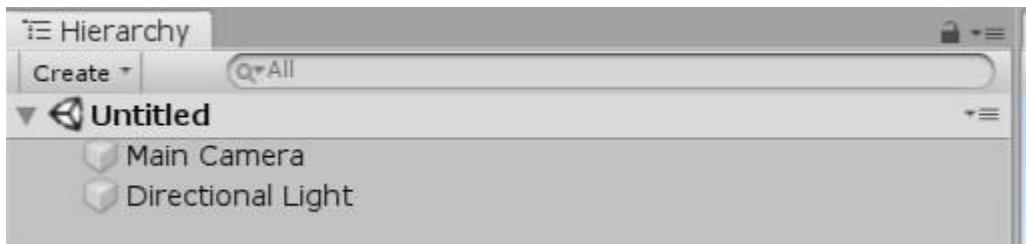
El personaje del jugador: primera parte

1. ¿Cómo se guarda una Scene (escena)?

En el tutorial anterior exploraste la disposición, o *layout*, del Editor Unity. También descubriste que una Scene (escena) está formada de GameObjects (objetos de juego) los cuales tienen componentes que especifican cómo funcionan en un juego.

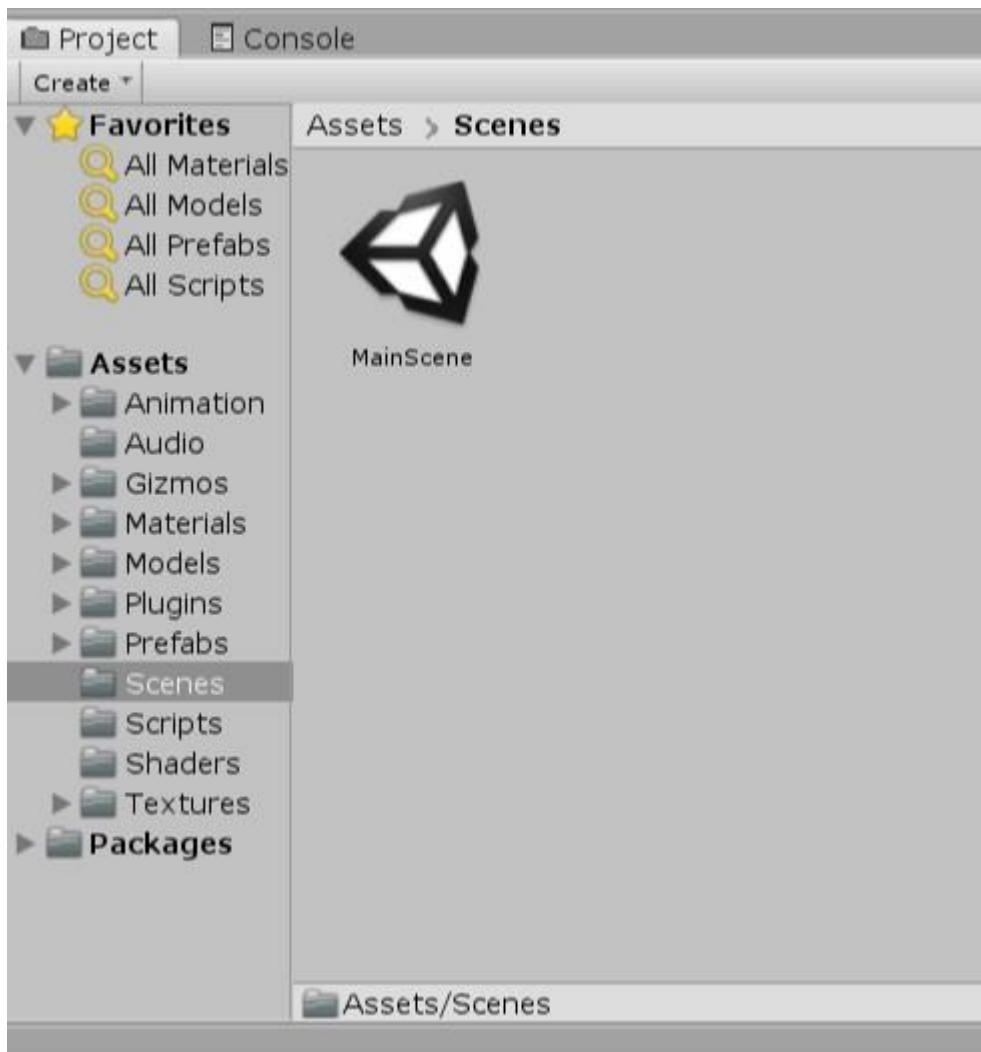
¡Es hora de agregar GameObjects a tu propia Scene!

Cuando se crea un nuevo Project (proyecto), Unity automáticamente creará una Scene vacía llamada «Untitled» (sin título).



Ve a **File > Save** (Archivo > Guardar) o pulsa Ctrl + S y luego selecciona dónde quieras que se guarde el archivo. Ya existe una carpeta llamada Scenes, entonces vamos a guardar esta Scene en esa carpeta con el nombre de «**MainScene**» (escena principal).

Verás que esto crea un Scene Asset (recurso de escena) llamado MainScene en la ventana Project.



Ahora tienes una Scene en donde trabajar. Usaremos esta Scene para el resto de los tutoriales. ¡Recuerda pulsar Ctrl/Cmd + S a menudo para guardar los cambios que has hecho!

2. ¿Cómo se agrega un personaje modelo?

En el tutorial anterior importaste todos los Assets que vas a necesitar para hacer este juego. El primer Asset que usarás es un **3D model** (modelo 3D).

La mayoría de las cosas que verás en un juego 3D son modelos, lo que incluye los personajes, el entorno y el decorado. Usualmente se crean fuera de Unity y luego se importan y se les da funcionalidad para crear un juego. Para usar estos modelos necesitas agregarlos como GameObjects (esto se invoca creando un **caso** del modelo).

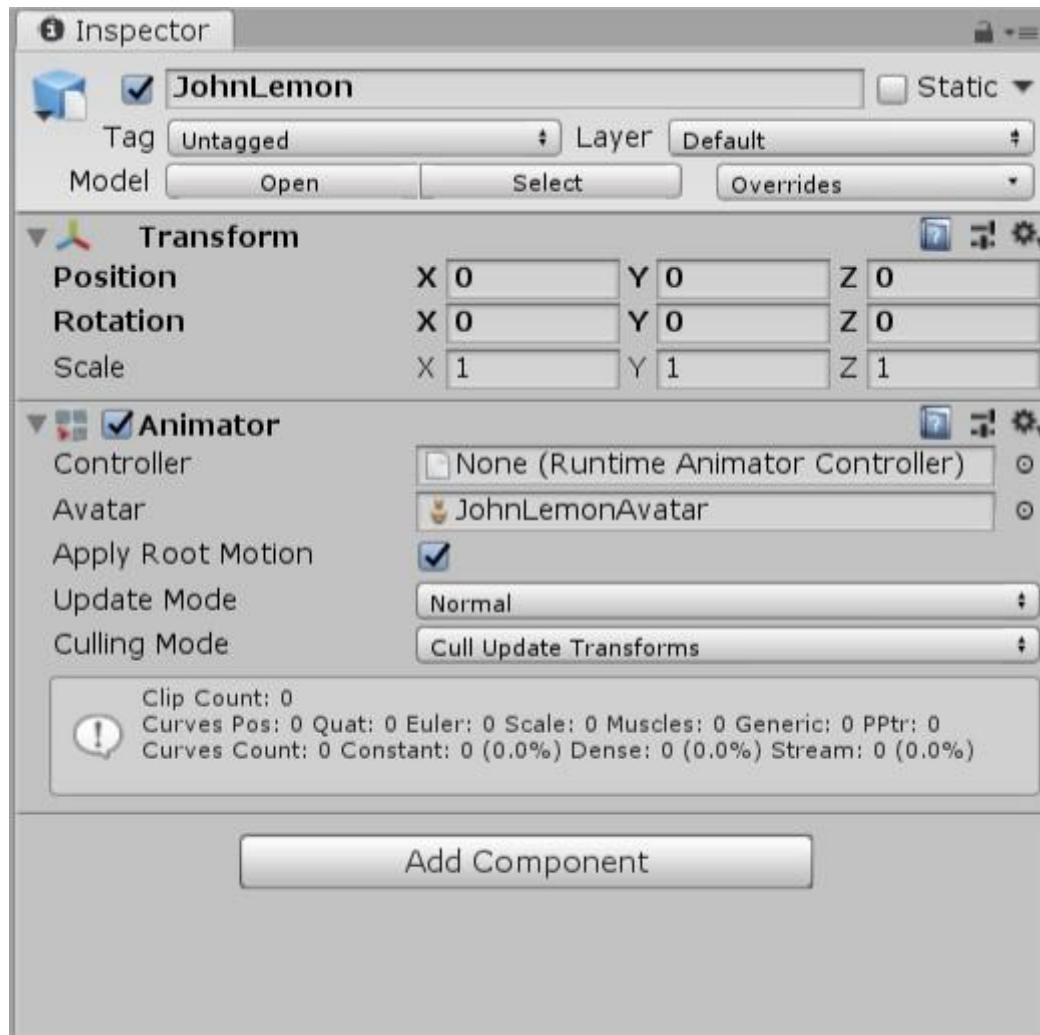
Vamos a añadir John Lemon, el personaje del jugador.

- En la ventana Project, ve a **Assets > Models > Characters** (Recursos > Modelos > Personajes) y encuentra el modelo llamado **JohnLemon**.

- Arrastra el modelo desde la ventana Project a la **vista Scene**. Esto te permite escoger exactamente dónde quieras ubicar al modelo. (También puedes arrastrar modelos a la Hierarchy para crear GameObjects en la posición por defecto).
- Con tu cursor sobre la vista Scene, pulsa **F** para enfocar.

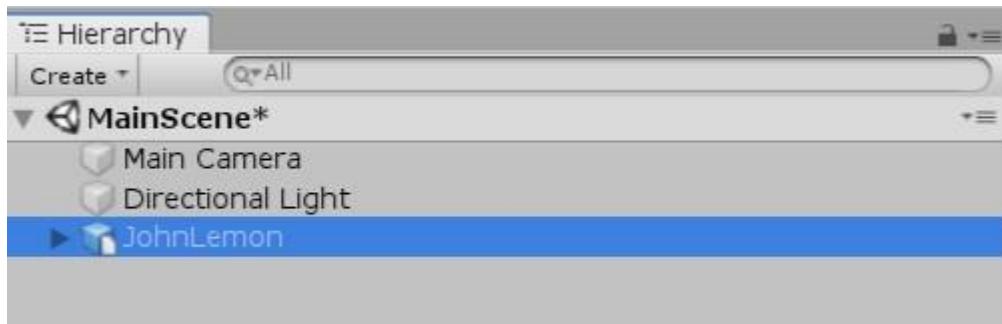
¡Ahora deberías ver el personaje del jugador en la escena! También puedes encontrar información sobre los GameObjects que acabas de exemplificar en el **Inspector**. Actualmente tiene dos componentes:

- un componente **Transform**, lo cual significa que tiene una locación y tamaño en la Scene;
- un componente **Animator**, lo cual significa que puede ser animado.



Pero este personaje tiene mucho más que esas herramientas. Por ejemplo, puedes ver el personaje del jugador en la vista Scene, pero ninguno de estos componentes justifica esto.

- En la **ventana Hierarchy** ubica el GameObject John Lemon.



Hay una flecha junto al nombre del GameObject —esto significa que un GameObject tiene hijos (jerárquicos).

- Haz clic en la flecha para expandir JohnLemon y ver sus hijos.

Este GameObject tiene dos hijos: otro GameObject llamado JohnLemon y otro llamado Root. Selecciona el GameObject hijo llamado **JohnLemon**.

Inspector

JohnLemon Static

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

Skinned Mesh Renderer

Rendering Layer Mask	1
Renderer Priority	0
Quality	Auto
Update When Offscreen	<input type="checkbox"/>
Skinned Motion Vectors	<input checked="" type="checkbox"/>
Mesh	JohnLemon
Root Bone	Hips (Transform)

Edit Bounds

Bounds

Center	X -0.3211945	Y -0.0003232	Z 0.0055948
Extent	X 0.7135001	Y 0.7288001	Z 0.2176

Light Probes Blend Probes

Reflection Probes Simple

Anchor Override None (Transform)

In Deferred Shading, all objects receive shadows and get per-pixel reflection probes.

Cast Shadows On

Receive Shadows

Motion Vectors Per Object Motion

Materials

Dynamic Occluded

JohnToon
Shader Custom/ToonShader

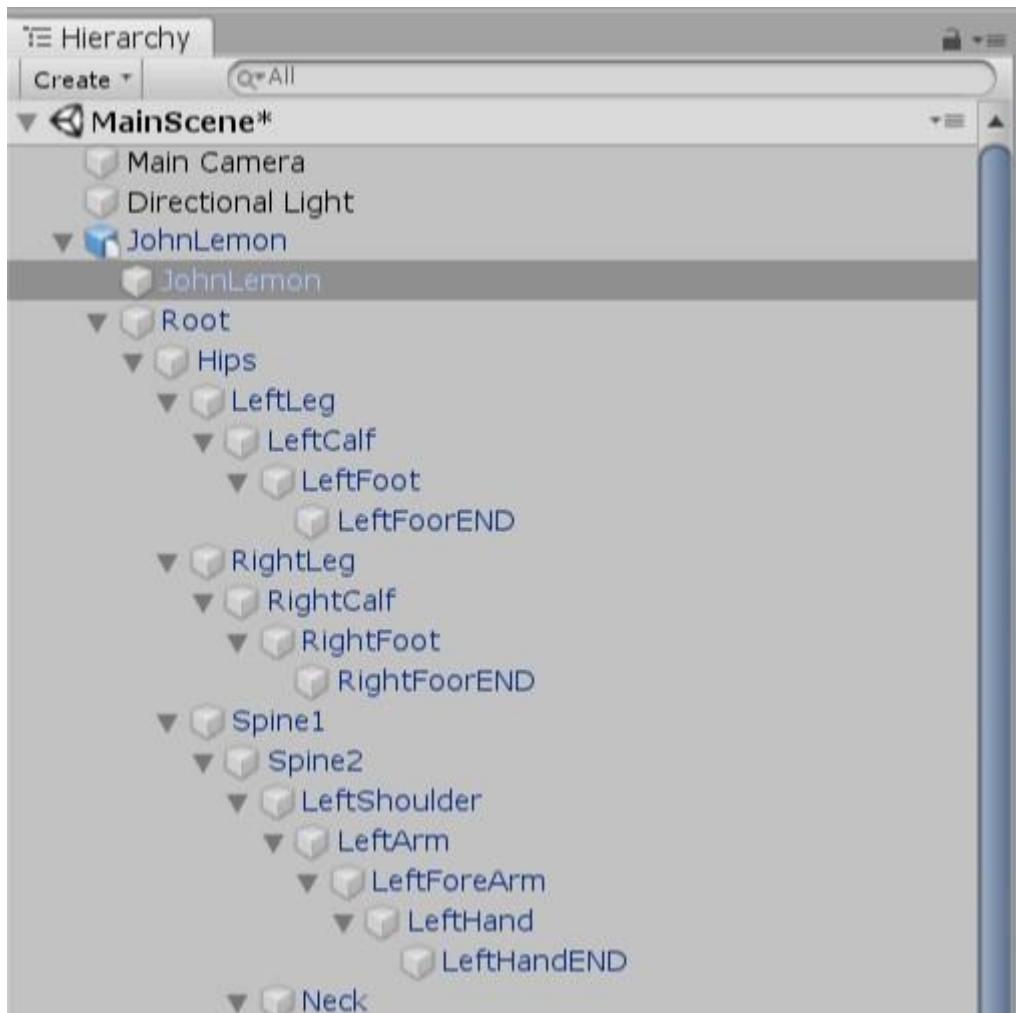
JohnPBR
Shader Standard

Add Component

Este GameObject tiene un componente llamado **Skinned Mesh Renderer** (Reproductor de malla de piel) Esto es lo que te permite ver al personaje.

Los modelos están constituidos de una malla de triángulos y un Mesh Renderer «reproduce» esa malla de manera que puedes verla. El Skinned Mesh Renderer es un tipo especial de Mesh Renderer (Reproductor de malla) que permite que la malla cambie de forma basada en las posiciones y rotaciones de todos los huesos de un modelo. Estos huesos son GameObjects hijos del modelo —los huesos de JohnLemon son todos hijos del GameObject **Root**.

- En la ventana Hierarchy selecciona el GameObject **Root**. Pulsa Alt (Windows) o Option (macOs) y haz clic en la flecha a la izquierda de su nombre para expandir todos sus GameObjects hijos.

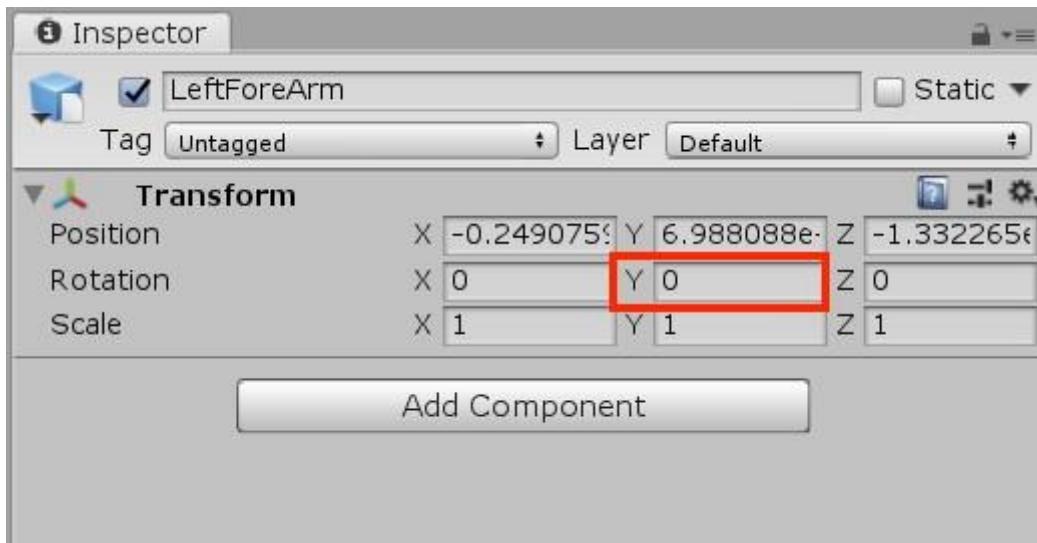


Cada uno de estos GameObjects representa una parte del cuerpo de JohnLemon.

- Haz clic en el botón **Play** en la barra de herramientas para entrar al modo Play.



- En la ventana Hierarchy selecciona el GameObject llamado **LeftForeArm**. En su componente Transform, encuentra el campo **Rotation** (rotación) y configura el valor de Y a **90**.



En la vista Scene deberías ver que el brazo izquierdo de John Lemon se ha doblado 90 grados. Fundamentalmente así es como animarás a JohnLemon: el componente Animator en el GameObject padre del personaje cambiará la rotación de todos los componentes Transform de los GameObjects que conforman los huesos y todos estos cambios juntos animarán el personaje.

- Sale del modo Play pulsando el botón Play otra vez. Ten en cuenta que cualquier cambio que hagas mientras estés en el modo Play no serán guardados al salir.
- Colapsa todos los GameObjects nuevamente pulsando Alt (Windows) o Option (macOS) y haciendo clic en la flecha junto al GameObject padre JohnLemon.Windows.

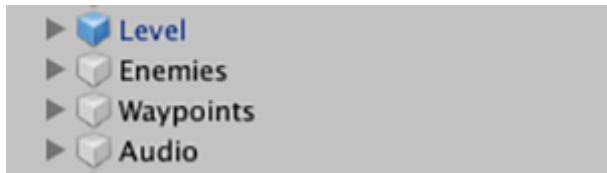
Antes de que te adentres mucho en la creación del personaje, vamos a echarle un vistazo a los Prefabs y cómo te puede ayudar a crear tu juego.

3. ¿Qué son los Prefabs?

Los Prefabs son un tipo de Asset (recurso) especial que representa un GameObject o un conjunto de GameObjects con componentes que ya están configurados. Son como un plano/plantilla que puedes utilizar para hacer casos de la misma cosa fácilmente. Cada caso de un Prefab está vinculado al Asset del Prefab; por lo tanto, el cambiar el Asset cambiará todas las versiones de los Prefabs en todas las Scenes.

El primer uso de este sistema en tu Project (proyecto) será hacer que el personaje sea un Prefab. Esto significa que, si continúas y haces múltiples niveles para el juego, no necesitarás rehacer un JohnLemon para cada nivel —puedes solo exemplificar un nuevo Prefab.

Los Prefabs pueden ser identificados en la ventana Hierarchy a través de su nombre en azul y su ícono:



Pero ¡espera un segundo! El GameObject JohnLemon tiene un nombre azul y un ícono en forma de cubo azul, pero hay un pedazo de papel blanco sobre el cubo. ¿Ya es el GameObject JohnLemon un Prefab?

Es algo parecido. En Unity los modelos funcionan como Prefabs de solo lectura. Son planos para crear casos de ese modelo, pero el plano mismo no se puede cambiar.

4. ¿Cómo transformar el personaje en un Prefab?

Necesitas poder cambiar cosas en este Prefab, entonces vamos a convertirlo en uno que puedas ajustar:

- Arrastra el GameObject desde la ventana Hierarchy a la carpeta **Assets > Prefabs** (Recursos > Prefabs) en la **ventana Project**. Una casilla de diálogo aparecerá preguntándote si quieres hacer un Original Prefab (Prefabricado original) o una Prefab Variant (Variante del prefabricado). Selecciona el **Original Prefab** (Prefab original).



Arrastrar un GameObject de esta manera hace un Prefab sin importar a qué carpeta sea arrastrado. Para mantener los Projects organizados es muy útil guardar todos los Prefabs en la carpeta Prefabs.

- Ahora que el Prefab John Lemon ha sido creado, cualquier cambio que hagas a ese Prefab será reflejado en los casos del Prefab John Lemon dentro de la Scene. Para hacer cambios a un Prefab necesitas abrir el Prefab para editarlo en el modo Prefab. Antes de que lo hagas, guarda la Scene pulsando Ctrl + S (Windows) o Cmd + S (macOS).

¡Ahora puedes abrir el Prefab JohnLemon!



- En la ventana Inspector haz clic en el botón Open Prefab (abrir el Prefab).

Ahora el Editor Unity está en el modo Prefab. Este modo te saca fuera de la Scene que estabas editando antes y te pone en una Scene temporal que solo tiene el Prefab. La vista Scene ha cambiado un poco: en la parte superior hay una nueva barra que dice **Scenes | JohnLemon** a la izquierda y tiene una casilla con el nombre de Auto Save (autoguardar) a la derecha.



Scenes | JohnLemon es la migra de pan para el Prefab que está siendo editado actualmente en el modo Prefab. En este caso solo estás editando el Prefab JohnLemon y el regresar a donde estabas desde aquí te llevaría a la Scene que estabas editando.

- Deshabilita la casilla Auto Save (habilitar esto te restará velocidad). Un botón Save (guardar) aparecerá de tal manera que puedas guardar manualmente cualquier cambio que hagas al Prefab.
- La ventana Hierarchy también tiene una nueva barra en la parte superior.



A la izquierda hay una flecha, si haces clic en la flecha ahora te llevaría de regreso a MainScene (Escena principal).

A través de este Proyecto para principiantes en 3D vas a editar Prefabs y la mayoría de las veces lo harás entrando en el modo Prefab. Acostumbrarse a

cambiar entre la edición de GameObjects en la Scene y la edición de Prefabs es fundamental para usar Unity.

Ahora estás listo para hacer los primeros cambios a tu Prefab JohnLemon al animarlo.

5. ¿Cómo animar tu personaje?

Tu personaje del jugador va a tener dos animaciones diferentes en este juego: una animación de andar para cuando el jugador se mueva y una animación en reposo para cuando no se esté moviendo.

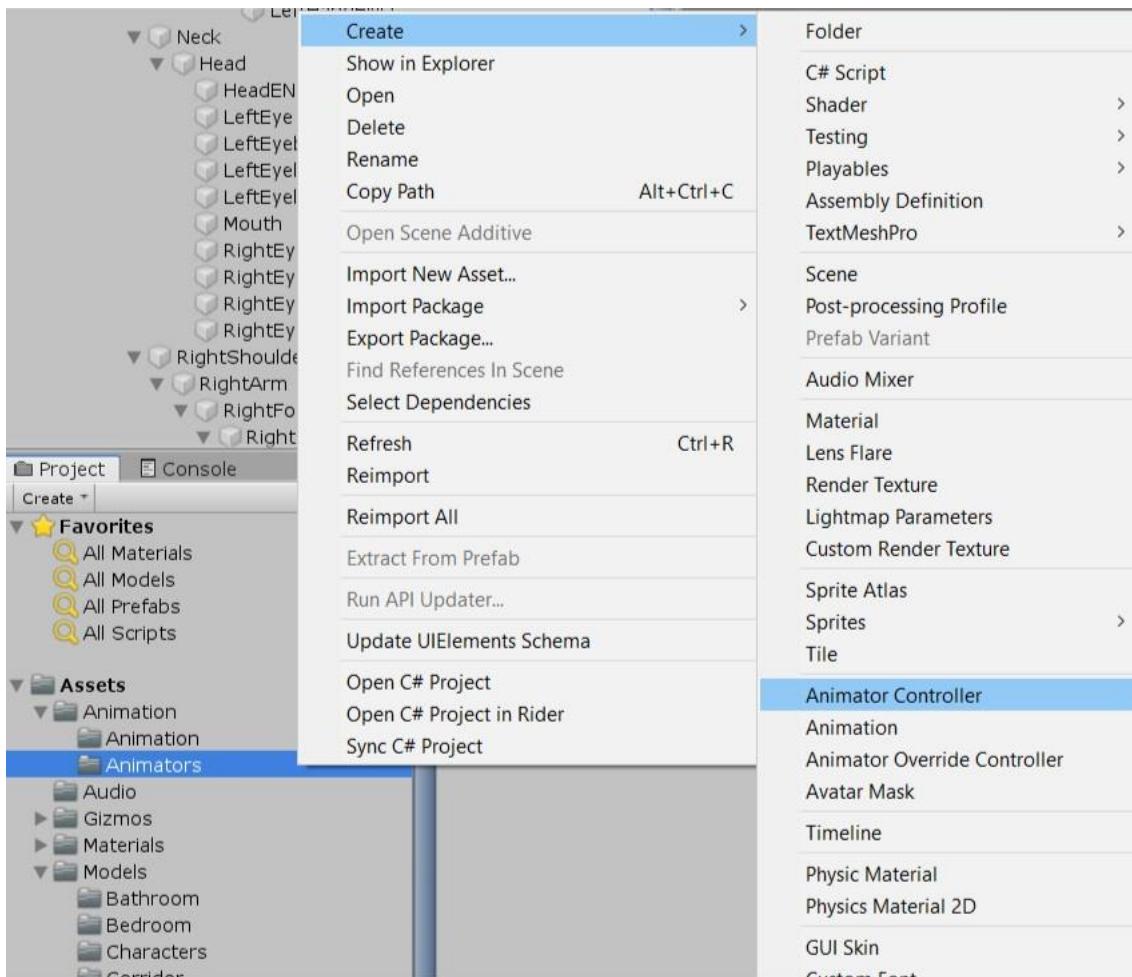
Selecciona el GameObject JohnLemon y échale un vistazo a su componente Animator en el Inspector. La primera propiedad se llama **Controller** (controlador). Esto lleva una referencia a un tipo de Asset llamado un Animator Controller, el cual vas a usar para lograr que JohnLemon se mueva.

Los Animator Controllers (Controladores de animación) contienen una **máquina de estado** la que determina qué animación el componente Animator debería configurar para su jerarquía en cualquier tiempo determinado. Esta animación se base en clips animados que han sido configurados en el Animator Controller.

6. ¿Como se crea el Animator Controller (Controlador de animación)?

Primero, vamos a hacer un Animator Controller (controlador de animación)

- En la ventana Project, encuentra la carpeta **Assets > Animation > Animator** (Recursos > Animación > Animador). Haz clic y selecciona **Create > Animator Controller** (Crear > Controlador de animación).



- Ponle el nombre «JohnLemon» al AnimatorController, luego haz clic doble en él para editarlo en la **ventana Animator**. La ventana Animator tiene dos secciones principales:

La ventana Animator tiene dos secciones principales:

- Un panel para editar Animator Layers (capas de animación) y Animator Parameters (parámetros de animación) a la izquierda.
- Un área la cual muestra la máquina de estado misma a la derecha.

Haz clic en la pestaña **Parameters** (Parámetros) en la parte superior izquierda de la ventana Animator.

La máquina de estado del Animator Controller toma decisiones basadas en los valores actuales de sus **Animator Parameters** (parámetros de animación). Estos Animator Parameters tienen valores configurados por un *script*. Necesitarás un parámetro por cada variable independiente que pueda afectar la animación que el personaje está reproduciendo.

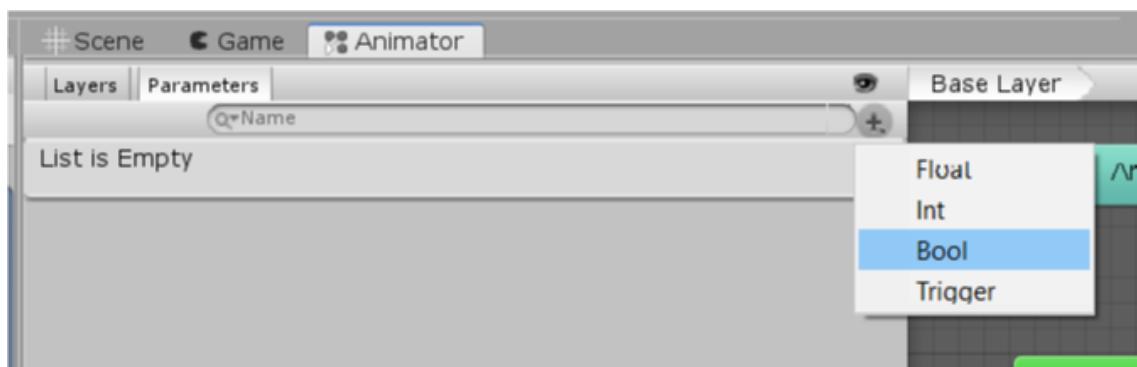
JohnLemon tendrá dos animaciones: una animación en reposo para cuando no se está moviendo y una animación andante para cuando se mueve. Entonces hay dos estados en los que puede estar: caminando o en reposo. Hay cuatro tipos de parámetros:

Hay cuatro tipos de parámetros:

- Un parámetro **float** tiene el valor de una variable de punto flotante (un número con decimales).
- Un parámetro **int** tiene el valor de un entero (un número sin decimales).
- Un parámetro **bool** tiene el valor de un booleano (el que puede ser true —verdadero— o false —falso—).
- Un parámetro **trigger** es un tipo especial de parámetro que no contiene un valor —causa un cambio de una animación a otra.

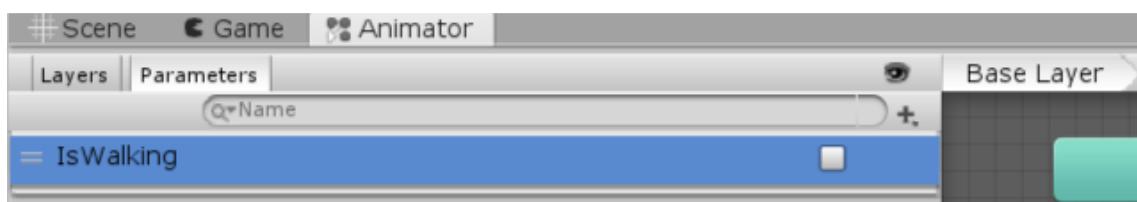
En este caso sabes que el personaje está caminando o está inmóvil; por lo tanto, un parámetro **bool** es lo más lógico.

- Haz clic en el botón y selecciona **Bool** del menú desplegable y crea un nuevo Bool Animator Parameter (Parámetro de animación booleano).



- Ponle al nuevo Animator Parameter el nombre de «**IsWalking**» (está caminando). Es importante que escribas y uses las mayúsculas de manera exacta —entenderás por qué en el próximo tutorial cuando escribas tu primer script.

A la derecha del nombre del parámetro IsWalking hay una casilla deshabilitada.



Este es el valor por defecto del parámetro. Sin el ingreso de datos del jugador tu personaje va a estar inmóvil; por lo tanto, IsWalking debería ser falso. Esto significa que no tienes que cambiar nada aquí.

7. ¿Cómo se configuran las animaciones?

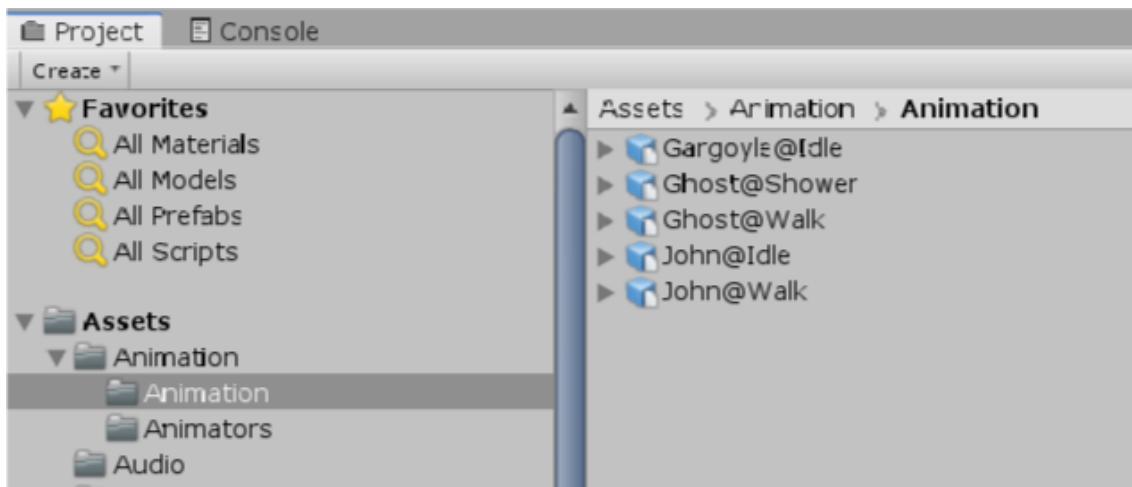
Ahora tienes una manera de determinar qué animación reproducir, pero no hay ninguna animación lista aún —vamos a configurarlas.

- En la ventana Project ve a la carpeta **Assets > Animation > Animation window** (Recursos > Animación > Ventana Animación) .

Actualmente las animaciones tienen iconos mostrando a lo que se parecen. Sin embargo, en este caso será más fácil si puedes ver los Assets en un listado. Encuentra el deslizador en la parte inferior derecha de la ventana Project y arrástralo hacia la izquierda, hasta que llegues al final.



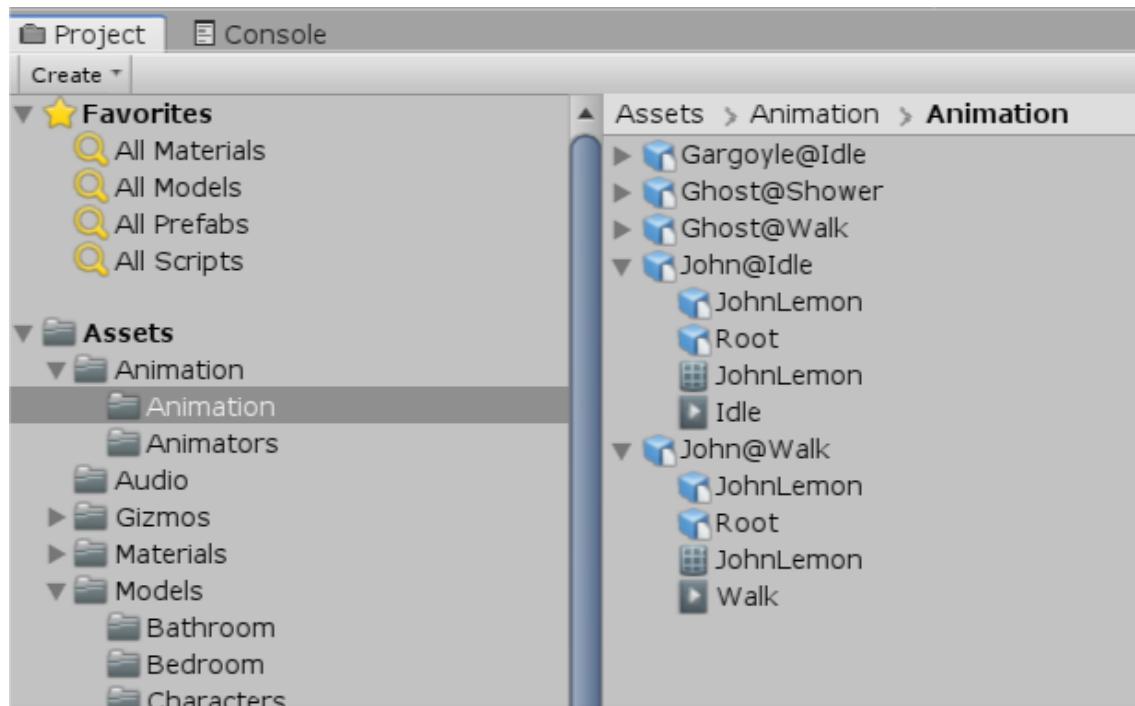
Deberías ver cinco modelos, dos de los cuales comienzan con **John@**. Esta manera de nombrar te permite saber que estas son animaciones para JohnLemon.



Expand both **John@Idle** and **John@Walk**, so you can see their sub-Assets.

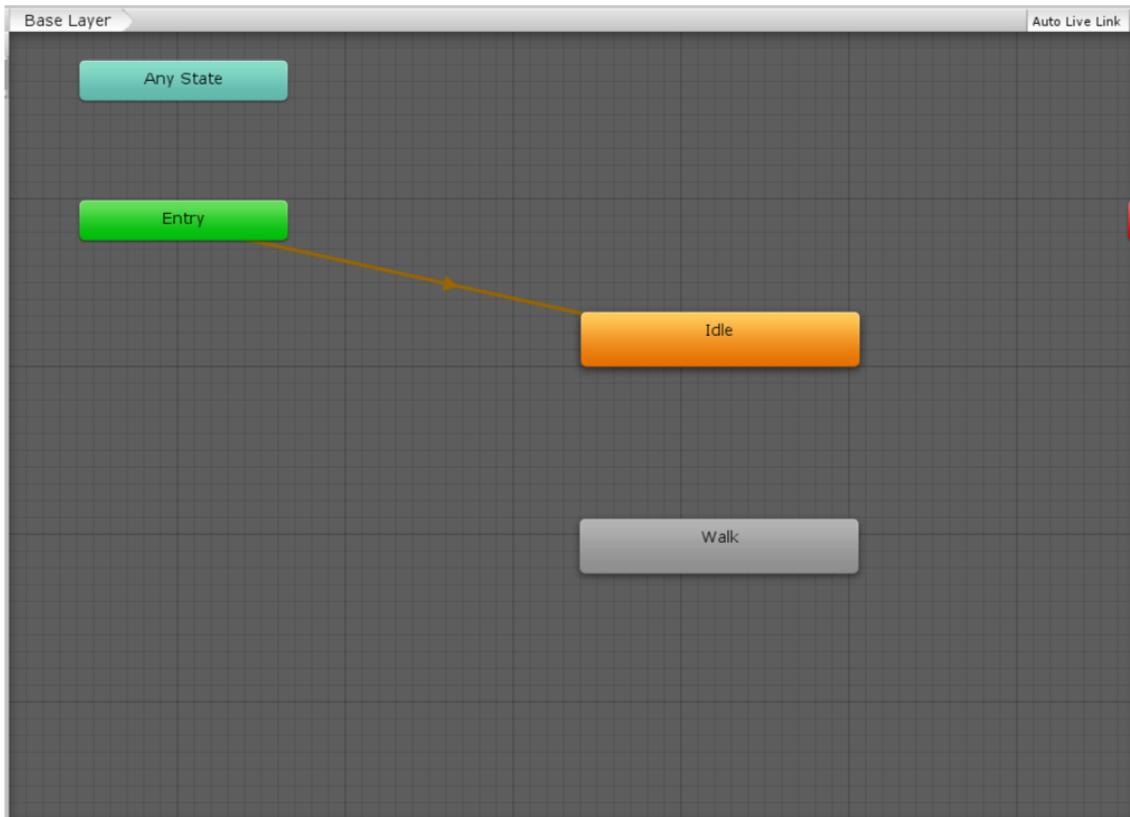
Los subrecursos muestran los hijos del GameObject padre (recuerda, los modelos son ejemplificados como GameObjects), las mallas y las animaciones. Para estos modelos, las animaciones son Idle (en reposo) y Walk (caminar) respectivamente.

- Para usar esas animaciones en tu Animator Controller arrástralas desde la ventana Project a la ventana Animator. Comienza con **Idle**.



- Las animaciones existen en un Animator Controller en **Animator States** (estados de animación). Cuando arrastras las animaciones Idle y Walk, el Animator Controller crea dos estados que los contiene y les pone el nombre de las animaciones.

Un Animator State es parte de la máquina de estado que contiene un Animator Controller. La máquina de estado contiene lógica, la cual determina qué estado es el actual. El estado actual luego determina la animación que se está reproduciendo.



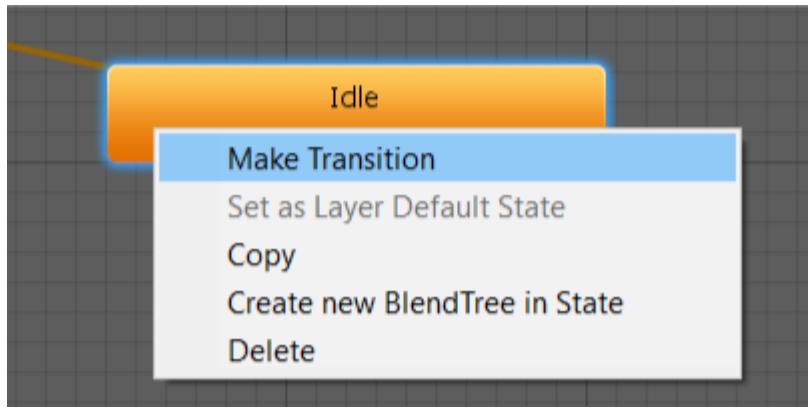
El estado por defecto se muestra en **anaranjado**. En este caso el estado por defecto es **Idle** porque lo arrastraste primero. El estado por defecto puede cambiarse al hacer clic en un estado y seleccionando Set As Layer Default State (Configurar como el estado de la capa por defecto).

Tu máquina de estado ahora tiene dos estados dentro de ella, pero no tiene ninguna lógica que define que estado debe reproducir. Actualmente la máquina de estado comenzará en el estado por defecto y no cambiará nunca; por lo tanto, JohnLemon siempre estará en reposo. Para añadir un poco de lógica necesitas crear **Animator Transitions** (Transiciones de animación).

8. ¿Cómo se crean las Animator transitions (transiciones de animación)?

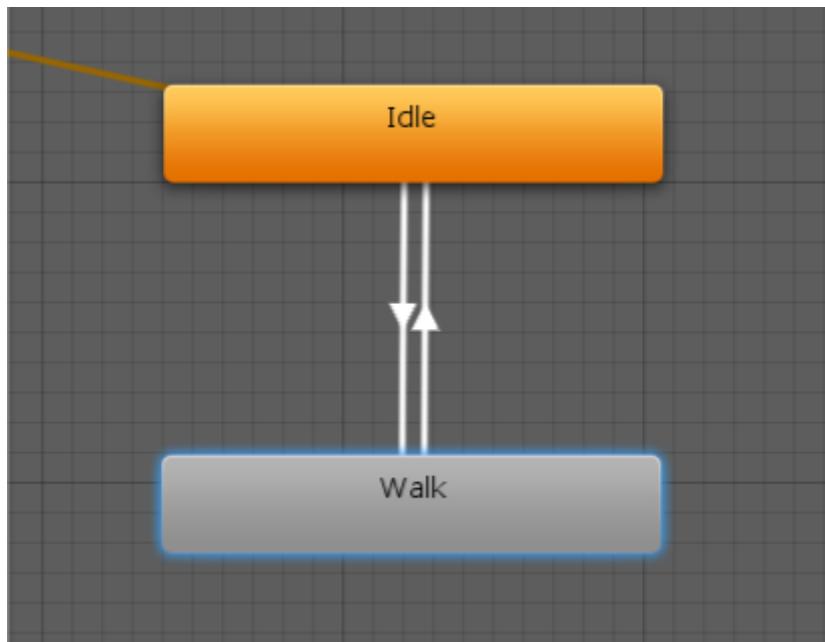
Para crear Animator transitions (transiciones de animación):

- Haz clic en el estado Idle y selecciona Make Transition. Esto empieza una transición que sigue el cursor del ratón.



Para terminar de crear la transición haz click en el estado Walk.

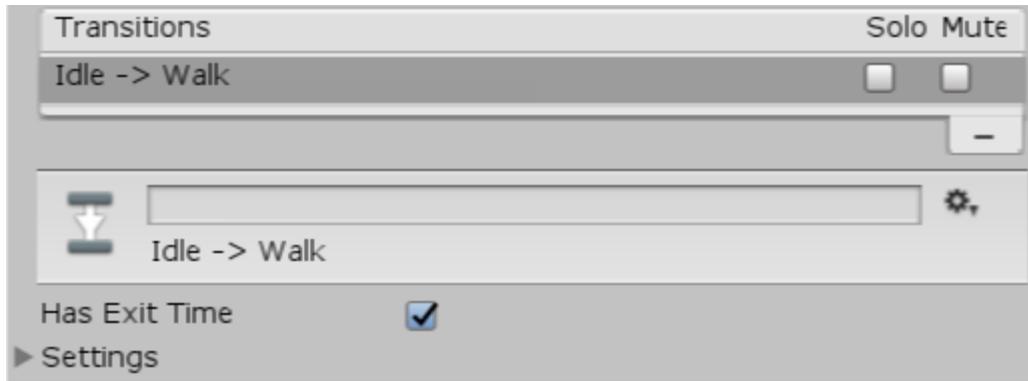
- Tu personaje necesita poder regresar de un estado de reposo a uno donde camine; por lo tanto, repite este proceso para crear una transición de Walk a Idle. La máquina de estado debería parecerse a esto:



- La máquina de estado tiene una manera de hacer una transición desde las dos animaciones, pero aún no sabe cuándo hacerla. ¿Recuerdas haber creado el parámetro IsWalking? Esto es lo que usarás para determinar si la máquina de estado debe cambiar de animación o quedarse en la misma.

Selecciona la transición de **Idle** a **Walk** haciendo clic en la línea de conexión.

- En el Inspector, échale un vistazo a la configuración para esta transición. Lo primero que necesitas cambiar es la **casilla Has Exit Time**.



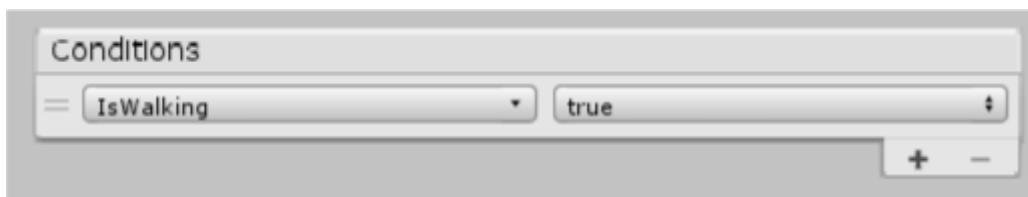
Si Has Exit Time es verdadero (si la casilla está habilitada) luego después de que cierto tiempo haya pasado, la transición será tomada y la máquina de estado reproducirá el siguiente estado. En este juego es importante cuando se toma la transición; por lo tanto, **deshabilita** la casilla **Has Exit Time** (Tiene tiempo de salida).

Un poco más abajo de la ventana Inspector verás una advertencia que dice: «Transition needs at least one condition or an Exit Time to be valid, otherwise it will be ignored» (La transición necesita al menos una condición o un Tiempo de salida para que sea válida, de otro modo será ignorada). Porque acabas de deshabilitar la casilla Has Exit Time actualmente no hay una razón por la que se deba tomar esta transición. Necesitas darle una añadiendo una **Condition** (condición).

9. ¿Cómo se añaden Conditions (condiciones) a tu transición?

To add Conditions to your transition:

- Haz clic en el botón debajo del botón debajo de lista (actualmente vacía) Conditions.
- Se requiere la transición Idle to Walk cuando el personaje está caminando —es decir cuando **IsWalking** es **verdadero**.



La Condition por defecto creada es realmente lo que necesitas aquí: si IsWalking es verdadero, JohnLemon debería transicionar de Idle a Walk. ¡No tienes que cambiar nada!

Luego, necesitas configurar la transición de Walk a Idle. Esta transición tiene requisitos similares.

- Selecciona la transición de **Walk** a **Idle** en la ventana Animator. **Deshabilita la casilla Has Exit Time**—como antes, no quieras que el estado cambie después de cierto tiempo determinado, solo cuando lo quieras.
- Ahora que la casilla Has Exit time está deshabilitada necesitas crear otra Condition.

Esta Condition también debería verificar IsWalking, pero esta vez el valor debería ser configurado como falso. Si el personaje no está caminando entonces debería transicionar de Walk a Idle.

¡El Animator Controller está listo!

10. ¿Cómo se le asigna un Animator Controller al Prefab JohnLemon?

Todavía necesitas decirle al JohnLemon Prefab que este es el Animator Controller que debe usar:

- En la ventana Project, ve a Assets > Animations > Animators (Recursos > Animations > Animators) y encuentra el Animator Controller JohnLemon.
- Selecciona el GameObject JohnLemon en la ventana Hierarchy y luego arrastra el Animator Controller a la propiedad Controller de su componente Animator en el Inspector. Este es un paso muy importante sin el cual JohnLemon no podrá moverse.
- Haz clic en el botón Save en la parte superior derecha de la ventana Scene (tal vez necesites seleccionar la pestaña Scene para hacerlo).

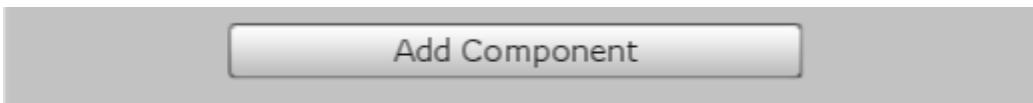
Ahora todos los casos del Prefab John Lemon han sido actualizados para que incluyan tus cambios.

11. ¿Cómo logras que tu personaje reaccione a la física?

Tu personaje va a estar deambulando alrededor de una casa embrujada con diferentes habitaciones y corredores que explorar. ¡Desde que no es un fantasma, es importante asegurarnos de que no pueda caminar a través de paredes!

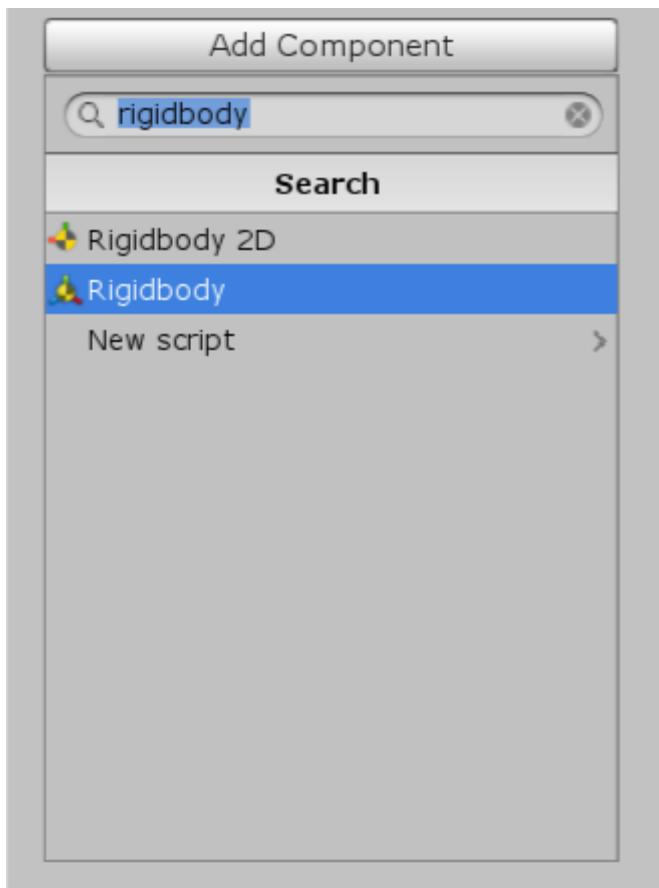
Para hacerlo, tienes que asegurarte de que el JohnLemon Prefab reacciona a la **física**

- Abre el Prefab JohnLemon para editar en el modo Prefab. En el Inspector encontrarás dos componentes: un Transform y un Animator. Para reaccionar a la física tu personaje necesita dos componentes más: un Rigidbody (cuerpo rígido) y un Collider (colisionador). Un componente **Rigidbody** marca un GameObject como algo que es parte del sistema físico que puede moverse. Ya que definitivamente quieres que tu personaje pueda moverse y queremos que choquen en contra de paredes (por lo tanto, ser parte del sistema físico), el GameObject JohnLemon necesita un componente Rigidbody.
- Haz clic en el botón **Add Component** (Añadir componente).

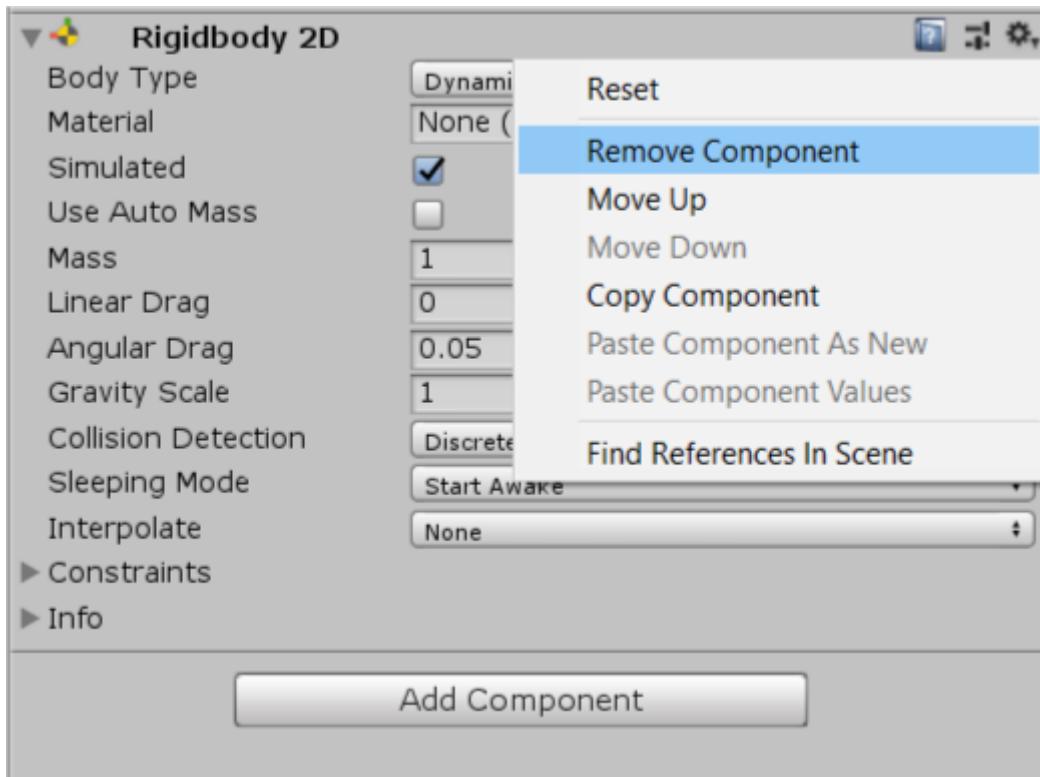


Add Component

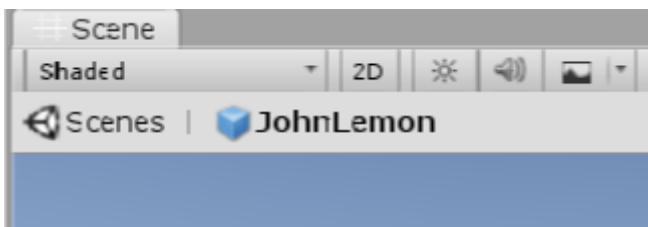
- En la barra de búsqueda que aparecerá ingresa «Rigidbody» y selecciónalo de las opciones que aparecerán.



Este es un proyecto en 3D; por lo tanto, asegúrate de que hagas clic en Rigidbody y **no** en Rigidbody 2D. Si accidentalmente añades un componente, puedes removerlo usando el menú de contexto del componente —el engranaje en la parte superior derecha del componente.



- Guarda lo que has hecho hasta ahora en el Prefab JohnLemon para asegurarte de que no pierdas nada.
- Desde que le has dado un componente Rigidbody al Prefab JohnLemon, va a reaccionar a los efectos físicos como la gravedad. Vamos a entrar al modo Play y verlo en acción. Regresa a la MainScene haciendo clic en **Scenes** en la parte superior de la vista Scene, luego pulsa el botón Play en la barra de herramientas.



Mmmm, eso no está bien: el personaje se cae una distancia corta y luego se detiene. Pulsa el botón Play otra vez para salir del modo Play.

Esto está siendo ejemplificado por el Animator. La tercera propiedad en el componente Animator es **Apply Root Motion** (Aplicar movimiento de raíz), la cual está habilitada actualmente.

12. ¿Qué es Root Motion (Movimiento de raíz)?

Las animaciones se usan para mover y rotar todos los GameObjects en una jerarquía en particular. La mayoría de estos movimientos y rotaciones se hacen relativos a sus padres; sin embargo, el GameObject padre de la jerarquía no tiene un parent y por lo tanto sus movimientos no son relativos. Este GameObject padre también se llama Root (raíz) y por ende su movimiento se llama Root Motion (Movimiento de la raíz).

Apply Root Motion está habilitado en tu componente Animator de tal manera cualquier movimiento de la animación será aplicado a cada marco. Desde que el Animator está reproduciendo Idle no hay ningún movimiento; por ende, el Animator no aplicará ningún movimiento. Entonces ¿por qué se mueve el GameObject JohnLemon? Esto es gracias al **modo Update** (actualizar) del Animator.

13. ¿En qué consiste el bucle Update (Actualizar)?

Los juegos funcionan de manera similar a las películas y a la televisión: se muestra una imagen en una pantalla y esta imagen cambia muchas veces por segundo dando la ilusión de movimiento. Estas imágenes se llaman marcos o fotogramas. El proceso de dibujar estos marcos en la pantalla se llama **rendering** (representar o renderizar). En las películas o en la televisión la próxima imagen que se muestra en la pantalla a menudo está predefinida, pero en los juegos la imagen siguiente puede variar extremadamente porque el usuario tiene influencia en lo que ocurre luego. Cada imagen necesita resolverse de acuerdo en el ingreso del usuario —y desde que esta variante puede pasar cada milisegundo, la programación que resuelve lo que se muestra también trabaja a esa velocidad. Esto se llama el **bucle Update** (actualizar).

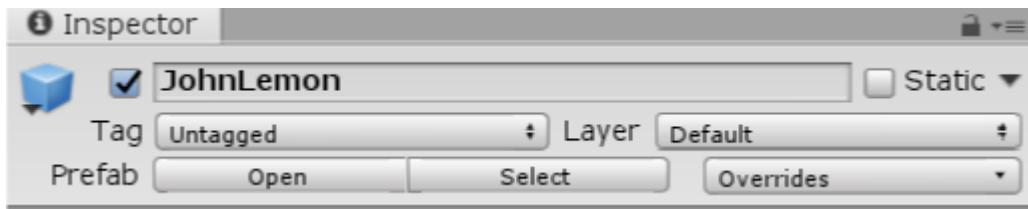
Cada vez que se muestra un marco, una lista de cosas se muestran en orden. Todo lo que necesitas saber por ahora es que componentes personalizados tienen un método **Update** que se invoca y una nueva imagen se reproduce en la pantalla. Estos Updates varían in longitud, dependiendo en la complejidad de las computaciones y las reproducciones. Sin embargo, hay otro bucle separado que ejecuta todas las operaciones de física. Este bucle no varía cuán a menudo se actualiza y por lo tanto se llama **FixedUpdate** (Actualización fija).

Un componente Animator puede cambiar cuando ejecuta su actualización. Por defecto ejecuta esto de acuerdo con la reproducción. Esto significa que el Animator está moviendo al personaje en **Update** y el **Rigidbody** está moviendo el personaje simultáneamente en **FixedUpdate**. Esto es lo que está causando tu problema y puede ser remediado fácilmente.

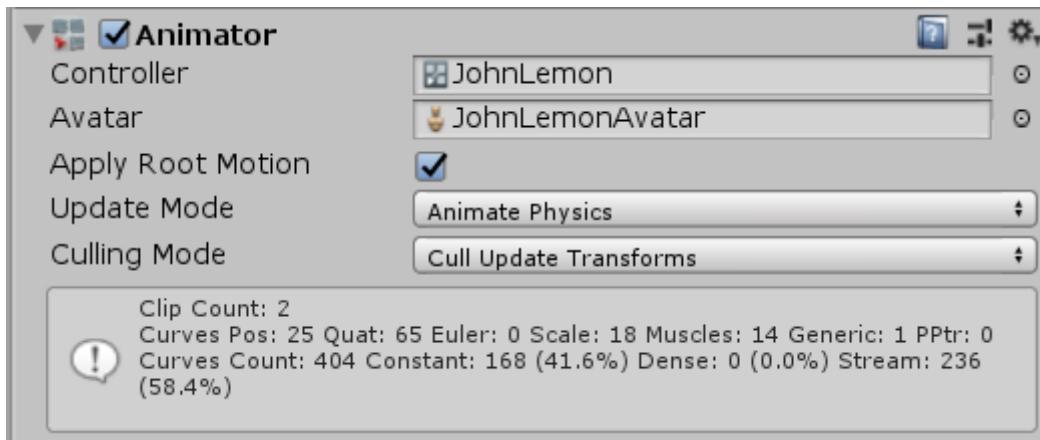
14. ¿Cómo puedes arreglar los movimientos de John Lemon?

Primero regresemos a editar el Prefab:

- En la ventana Inspector haz clic en el botón Prefab **Open** (abrir el Prefab)



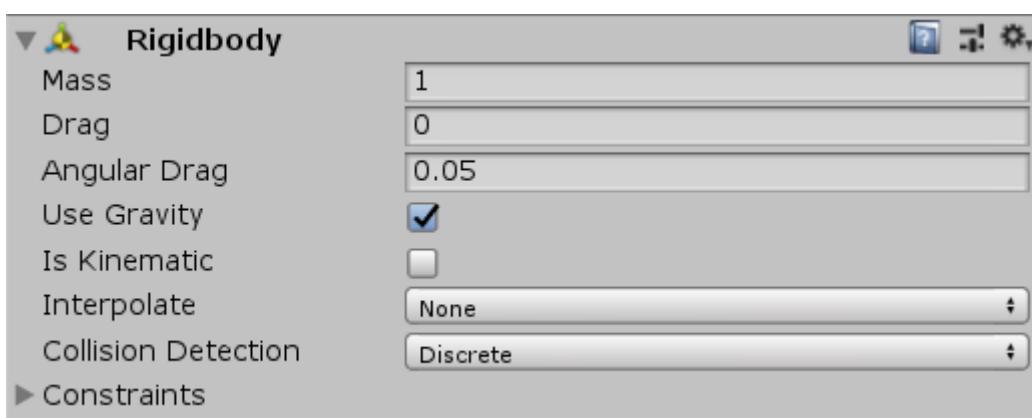
- Ve al componente Animator. En el menú desplegable Update Mode Property, selecciona **Animate Physics**.



Este cambio causará que el Animator mueva al personaje a tiempo con la física. Ahora no debe haber competición entre los bucles de actualización para mover al personaje y reaccionará a la física como es debido.

- Ya has visto que las animaciones no tienen ningún movimiento vertical de raíz. Hay algunas maneras en las que se puede detener.

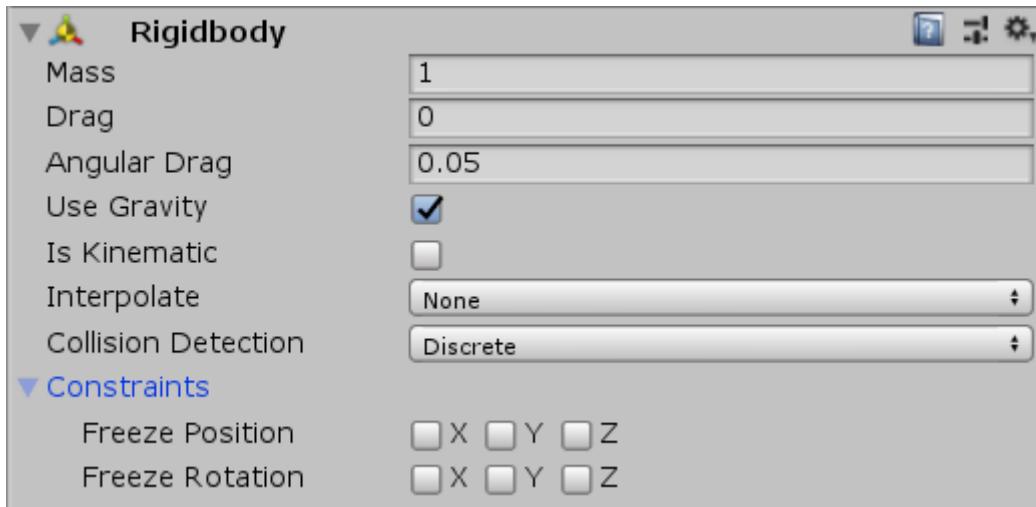
La opción más obvia es deshabilitar Use Gravity (Usar gravedad) en el componente Rigidbody.



Esto no permitiría que el personaje se caiga, pero no es exactamente lo que necesitas. Si deshabilitas Use Gravity entonces el personaje, si un choque lo empuja hacia abajo no caería o

flotaría si un choque lo empuja hacia arriba. Definitivamente no quieres que esto ocurra, entonces necesitas **limitar el movimiento del personaje usando la física**.

- Encuentra el componente Rigidbody y haz clic en la flecha para expandir la propiedad **Constraints** (restricciones).

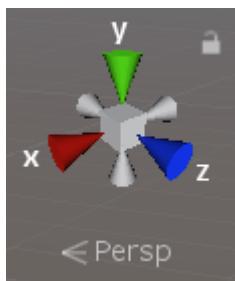


Estas propiedades Constraint restringen la dirección en la que el Rigidbody puede moverse.

Vamos a explorar el sistema de coordenadas de Unity para entender exactamente cómo funcionan.

15. El sistema de coordenadas de Unity

Las posiciones y las direcciones en Unity trabajan en coordenadas 3D representadas por las letras x, y y z —juntas estas forman un vector. Descubrirás más sobre los vectores en el próximo tutorial. Una Scene tiene una definición global de estas direcciones de tal manera que puedas verlas en la parte superior derecha de la ventana Scene.



Las flechas de colores representan la dirección positiva de cada eje y las líneas grises opuestas representan la dirección negativa:

- el eje horizontal (o de las equis) es rojo.
- el eje vertical (o de las yes) es verde.
- el eje z es azul.

16. Posiciones y rotación

Todos los GameObjects sin padres jerárquicos en la escena tienen **posiciones** relativas al origen de la escena (0, 0, 0). Todos los GameObjects con padres tiene posiciones relativas a su parente. Sin embargo, desde que los GameObjects pueden rotarse, la posición relativa (o local) de un hijo será en las coordenadas en rotación de su parente. Los Rigidbody Constraints se refieren a estas coordenadas locales.

Vamos a echarle un vistazo a los ejes locales del GameObject JohnLemon:

1. Selecciona el GameObject JohnLemon y luego échale un vistazo a la vista Scene.



Verás que:

- el eje horizontal (rojo) está apuntando hacia la derecha del personaje

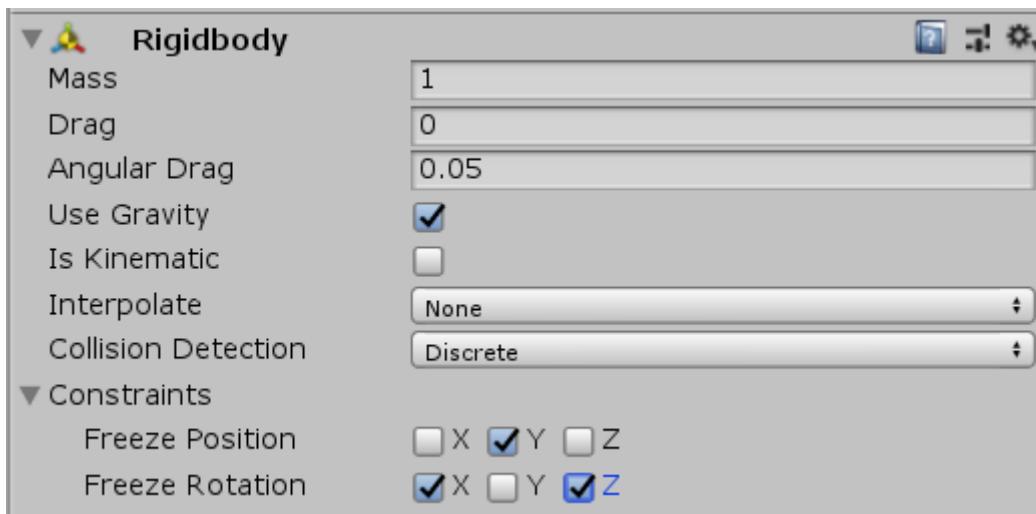
- el eje vertical (verde) está apuntando hacia arriba
- el eje z (azul) está apuntando hacia adelante

Entonces, ¿qué limitaciones debería tener tu personaje? Tu personaje necesita poder moverse hacia adelante y de un lado al otro; por ende, no hay necesidad de limitar su posición x o z. Sin embargo, es importante que no se mueva hacia arriba o hacia abajo. Para eliminar esa posibilidad habilita la casilla **Freeze Position Y** (Congelar la posición Y).

- Rotation (rotación) se refiere a la rotación alrededor de un eje específico. Por ejemplo, si el personaje fuera rotado alrededor de su eje horizontal entonces parecería que estuviera acostado supino o prono:

Si fuera rotado alrededor de su eje Z, parecería que estaba acostado de lado:

John Lemon no se debería mover en ninguna de estas direcciones; por lo tanto, habilita las casillas **Freeze Rotation X y Z**. John Lemon necesita darse la vuelta; por lo tanto, mantén la casilla **Freeze Rotation Y** habilitada.



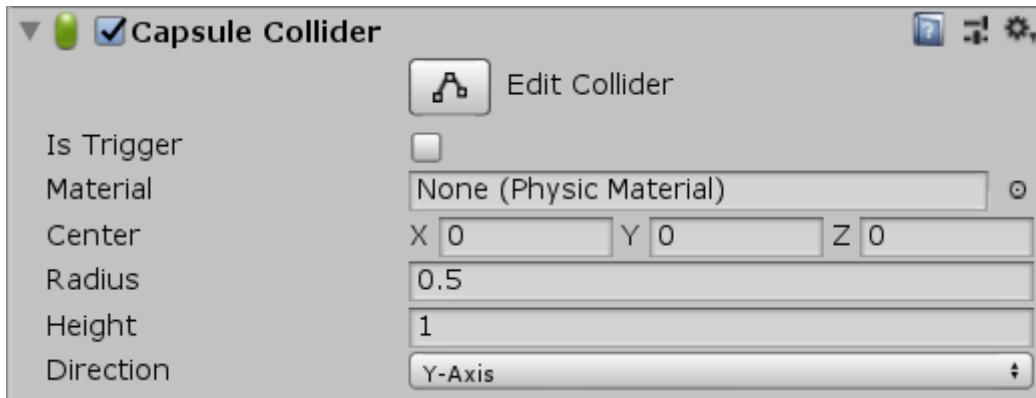
El componente Rigidbody está configurado y tu personaje reaccionará al sistema de física. Sin embargo, en realidad no tiene una presencia física en la Scene todavía. Nada puede chocarse contra él y no puede chocar contra nada. Para que tenga una presencia física en la escena necesita un **Collider** (Colisionador).

17. ¿Cómo añadirle un Collider (Colisionador) a John Lemon?

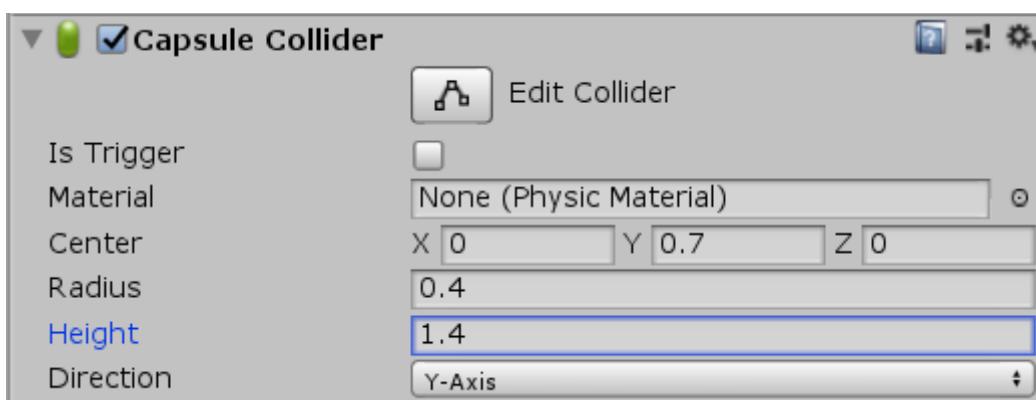
Un Collider es un término general que cubre un número de componentes distintos. Los Colliders definen la forma de un objeto con el propósito de colisiones físicas, lo que le permite

a tu personaje golpear cosas y que lo golpeen. Hay componentes Collider de muchas formas distintas, pero la más simple que le queda bien a JohnLemon es el **Capsule Collider** (Colisionador en forma de cápsula).

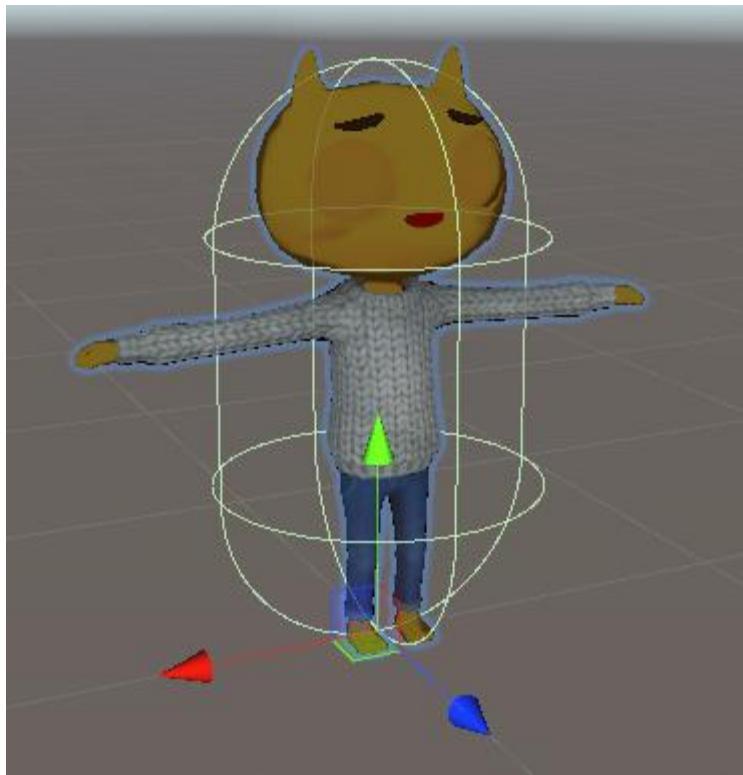
- Haz clic en el botón Add Component (Añadir componente) en la ventana Inspector y encuentra el Capsule Collider. Selecciona la opción **Capsule Collider**, asegurándote de que no hagas clic en Capsule Collider 2D.



- Actualmente el Capsule Collider realmente no cubre el modelo; por lo tanto, las colisiones en el juego no funcionarían bien —necesitas cambiarlo de tamaño y de posición para que sea aproximadamente de la forma de JohnLemon. Cambia la propiedad **Height** (Altura) del Capsule Collider a **1,4** y la propiedad **Center** (Centro) a **(0, 0,7, 0)**. Estos cambios significan que el medio del colisionador está a la mitad de su altura medida desde el suelo y desde que JohnLemon mide aproximadamente 1,4 metros (o 140 cms.) la forma lo cubre bastante bien.
- Todavía está un poquito ancho —cambia la propiedad **Radius** (Radio) del Capsule Collider a **0,4**.



¡Parece que logramos que la cápsula sea del tamaño correcto!



Ahora tienes un personaje que reaccionará a la física tal como lo necesitas, chocando contra las paredes, pero sin caerse.

18. Resumen

En este tutorial empezaste a trabajar con el personaje para tu juego. Ahora tienes una ligera idea del poder de los Prefab y has creado un conjunto de reglas simples que gobiernan cómo se comportará JohnLemon. ¡En el siguiente tutorial explorarás la última parte del rompecabezas que forma el personaje al escribir tu propio script!

El personaje del jugador: Segunda parte

1. Continuación del personaje del jugador

En el tutorial previo comenzaste a trabajar en el Prefab JohnLemon añadiendo un sistema para animarlo y componentes para hacerlo trabajar con el sistema de física. ¡Ahora vas a crear un componente personalizado: tu primer *script*!

¿Qué es un script?

Un script es un documento de texto que contiene una serie de instrucciones para la computadora. Comúnmente a estas instrucciones se les llama código. Las instrucciones están escritas de tal manera que la computadora pueda entenderlas, en este caso usando el lenguaje de programación llamado **C#** (Si sharp).

C# define la manera en la que las instrucciones están escritas y algunas de las palabras que se usan. Afortunadamente, las palabras que se usan a menudo tienen el mismo significado en C# del que tienen en inglés. Por ejemplo, la primera palabra con la que nos toparemos en C# es «**using**» (usando)—esto quiere decir que el script está escrito usando código de algún otro lugar. Otro ejemplo es «**public**» (público), lo cual significa que cualquier cosa puede acceder a algo. Hay demasiados ejemplos de analizar, pero en estos tutoriales explorarás cada uno de ellos cuando se presenten.

Todos los scripts que escribirás para este proyecto tomarán la forma de **MonoBehaviours** (Comportamientos únicos). MonoBehaviours son tipos especiales de *scripts* que pueden ser anexados a GameObjects, tal como se puede hacer con los componentes. Esto se debe a que son un tipo específico de componente que tú mismo puedes escribir.

Los scripts comparten algunas similitudes con los Prefabs (Prefabricados):

- Un script se crea como un Asset (recurso), igual como se hace con un Prefab.
- Añadir un script a un GameObject como un componente es realmente una exemplificación del script, al igual que el añadir un Prefab a una escena es una exemplificación de ese Prefab.

Sin embargo, los scripts son diferentes en muchos aspectos ¡Vamos a entrar en materia para aprender más sobre esto!

2. ¿Cómo puedes crear tu primer script (PlayerMovement)?

Primero, crea tu nuevo script como un Asset (Recurso):

1. Encuentra la carpeta **Assets > Scripts** (Recursos > *Scripts*) en la ventana Project (Proyecto). Haz clic en la carpeta y escoge **Create > C# Script** (Crear > *Script* en C#). Ponle el nombre «PlayerMovement» a tu script.

Ojo: Los scripts que se van a usar como un componente necesitan que su *asset* tenga el mismo nombre que el **nombre de la clase** en el mismo script. Cuando Unity crea un archivo *script*, le da el nombre de una clase que es igual al primer nombre que se le dio al Asset. Aun así, cuando se cambia el nombre del Asset, el nombre de la clase no cambia.

2. Selecciona tu scrip y mira la ventana Inspector. Deberías ver el siguiente código:

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

public class PlayerMovement : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

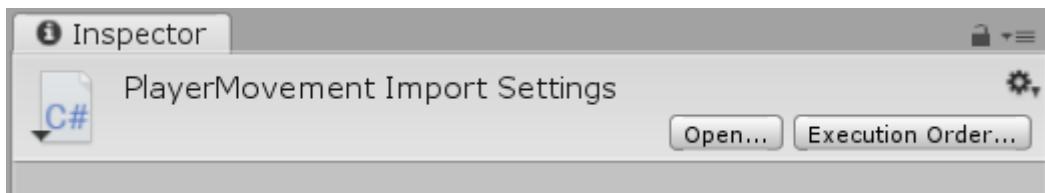
    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

Encuentra la línea que comienza con «**public class PlayerMovement**». Esto es lo que define el nombre de la clase. Si tu *script* no dice PlayerMovement, borra el *script* Asset y luego crea uno nuevo llamado PlayerMovement.

3. Ya que has creado tu *script* Asset, ábrelo para editarlo. Puedes hacer doble clic en el Asset o hacer clic en el botón «Open...» en la ventana Inspector.



La edición de *scripts* no se hace dentro de Unity —en su lugar los *scripts* se abren en otro programa llamado Visual Studio. Una vez que esté abierto podrás editarlos.

Analicemos el *script* por defecto

Vamos a analizar el *script* por defecto que puedes ver en Visual Studio ahora.

1. Las tres primeras líneas de código son:

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

Se les llama **using directives** (directrices de uso). Te permiten usar código implementado en otro lado dentro de este archivo *script*. Por ejemplo, la palabra «MonoBehaviour» en la línea siguiente no se podría usar si no tuvieras «using UnityEngine» aquí. En la mayoría de los casos, las directrices de uso que Unity incluye por defecto son suficientes; por ende, no necesitas preocuparte mucho acerca de ellas.

2. La línea siguiente es:

```
public class PlayerMovement : MonoBehaviour
```

Esto es el comienzo de una **declaración de clase** o *class declaration*. Las clases son como planes para ejemplificaciones que se llaman Objetos u *Objects*. Cuando una clase es ejemplificada (por ejemplo, al anexar un *script* como un componente a un GameObjet), esa ejemplificación se llama un **objeto**. Los objetos son los cimientos del código ¡y existen a través de todo el trabajo que ya has hecho!

Las clases que ya sabes y con las que estás familiarizado incluyen Animator, Rigidbody, GameObject y Transform. Ya están en tu código. Otra manera en la que podemos pensar sobre las clases es que son fábricas en las que se toma una materia prima, se hace algo con ella y que luego produce un resultado. Regresaremos a la analogía sobre las fábricas con otros elementos de código.

3. La línea siguiente es simplemente una llave abierta: {

En esta serie de tutoriales las llamaremos **llaves**. Las llaves son un elemento esencial del lenguaje C# porque definen **bloques de código**. Los bloques de código son líneas de código que existen entre una llave de apertura y una de cierre. Las llaves siempre deben estar en pares.

Para la declaración del bloque de código, la llave de cierre está al fondo, pero también hay dos bloques de código más dentro de la declaración. Los dos bloques de código que están contenidos dentro de la clase están sangrados. La sangría no es algo técnicamente necesario, pero es muy útil para definir dónde el código empieza y dónde termina.

4. La siguiente línea es:

```
// Start is called before the first frame update
```

Cualquier texto que esté precedido de una barra doble, como esta, se llama un comentario. Un comentario es cualquier cosa que quieras que sea ignorada completamente por la computadora. En la mayoría de los casos funcionan como una especie de etiqueta para explicar algo sobre el código que los rodea. En este caso, el comentario da una explicación breve de lo que está declarado debajo suyo.

5. Después del comentario está el comienzo de la primera declaración de método:

```
void Start()
```

Vamos a regresar a la analogía de la fábrica. Si una clase es una fábrica; entonces, un método es una máquina dentro de esa fábrica. Los métodos pueden asimilar datos, ejecutar una operación y luego emitir (o regresar) datos.

Todas las declaraciones de métodos tienen el mismo formato:

- Primero declaran el **tipo de retorno** o *return type*. Esto es el tipo de datos que se va a emitir una vez que el método acabe. En este ejemplo el tipo de retorno es **void**. Esta es una palabra especial de C# que significa «nada»—el método no retorna nada, literalmente.
- Después del tipo de dato viene el **nombre del método**, en este caso **Start** (comienzo).
- Después del nombre hay un par de paréntesis. En esta serie de tutoriales los llamaremos paréntesis. Entre los paréntesis, los métodos tienen una oportunidad de declarar qué tipo de datos quieren asimilar. Estos trozos de datos se llaman **parámetros**. Ya que no hay nada dentro de los paréntesis, no hay ningún parámetro declarado.

Estos tres elementos (el tipo de retorno, el nombre y los parámetros) forman la **firma** de un método. En la mayoría de los casos, un método puede tener cualquier firma que quieras. Sin embargo, las clases MonoBehavior pueden usar algunos métodos especiales los que necesitan tener firmas específicas. Estos métodos especiales no necesitan invocarse desde tu código. En su lugar Unity los invoca en momentos específicos. El método Start es un ejemplo de esos métodos especiales. Se invoca tan pronto como el GameObject en el que está comienza, lo cual es usualmente tan pronto como empieza la Scene. Esto lo hace la opción ideal para hacer cosas que no quieras repetir, como la configuración.

6. Después de la firma del método Start hay un bloque de código. Esto define todo el código que se ejecuta cuando se invoca el método. El invocar un método es cómo se usa el método (harás esto más adelante). Para regresar a la analogía de la fábrica: la declaración del método es cómo funciona la máquina en la fábrica e invocar el método es como realmente se usa la máquina.

A la firma de declaración de método seguida del bloque de código se los conoce juntos como una **definición de método**. Los términos «declaración de método» y «definición de método» se intercambian mucho cuando se habla de C# porque ocurren al mismo tiempo. La diferencia realmente solo importa en otros lenguajes (tal como C++).

7. Ahora es un momento oportuno para hablar sobre el orden en el que se escriben las cosas en una clase. Siguiendo con la analogía sobre las fábricas y las máquinas, no importa dónde estén las máquinas dentro de tu fábrica, pero el orden en el cual las máquinas ejecutan sus operaciones es muy importante.

En C# no importa en qué orden los métodos se declaran en una clase, pero el orden en el que los métodos se ejecutan sus operaciones importa mucho.

8. Las dos últimas partes del *script* son otro comentario y otra definición de método.

El método **Update** es otro método especial para MonoBehaviors. Se invoca en cada marco, antes de algo sea reproducido en la pantalla.

Ahora que tienes un entendimiento básico de algunas de las partes del *script* por defecto, vamos a personalizar este *script* para tu juego.

3.¿Cómo podemos crear variables para los ejes horizontal y vertical?

Tu *script* PlayerMovement necesita tomar el ingreso o las entradas que hace el usuario y traducirlo al movimiento del personaje.

Lo primero que necesitas hacer es obtener algunos datos del sistema de ingresos de Unity (Unity's input system). El *script* necesita verificar lo que ocurren con los ingresos todo el tiempo y desde que Update se invoca en cada marco, tiene sentido verificar los ingresos allí. Pero ¿qué es lo que específicamente necesita verificar?

Tiene sentido mover al personaje usando las **teclas WASD** o las **flechas**; por lo tanto, el *script* necesita verificar los valores de esas teclas específicas en el teclado. Podría verificar si alguna de esas teclas está siendo pulsada individualmente o no y decidir lo que el personaje debería hacer (o no), pero hay una alternativa que lo hará un poco más fácil.

Unity tiene un **administrador de ingresos** (input manager) que define varios botones y ejes que pueden encontrarse a través de su nombre. Por ejemplo, tiene un eje llamado **Horizontal**, el cual es representado por las teclas A y D y las flechas izquierda y derecha. Entonces, al verificar eso la computadora del jugador podría decidir si el personaje se mueve hacia la izquierda o la derecha.

Escribe el código para crear tus variables

¡Vamos a empezar! Añade la línea siguiente dentro de los corchetes del método Update:

```
float horizontal = Input.GetAxis ("Horizontal");
```

Debería verse así

```
// Update is called once per frame
void Update()
{
    float horizontal = Input.GetAxis ("Horizontal");
}
```

Si las clases son fábricas y los métodos son máquinas en esas fábricas, las **variables** son cajas que contienen las cosas en esas fábricas. Es decir, las variables son una manera de guardar datos. Los datos que necesitas guardar es el valor del ingreso del eje horizontal. En Unity los ejes de ingreso regresan un número entre -1 y 1 —este tipo de dato se llama **dato de punto flotante** o float. Un dato de punto flotante representa un número con decimales.

Hay algunas partes importantes sobre la sintaxis o estructura de esta línea de código:

- En C#, un **signo igual** significa que se debe asignar lo que sea que esté a la derecha (el resultado de un método) a la variable a la izquierda (una variable nueva de tipo float).
- El **punto** entre Input y GetAxis le permite a la computadora acceder algo dentro del objeto anterior (GetAxis es un método dentro de Input; por lo tanto, para ir de Input a GetAxis se usa un punto).
- El código de C# está constituido de cosas llamadas instrucciones. Cada instrucción contiene una o más instrucciones para la computadora y pueden considerarse oraciones. Los dos puntos marcan el final de la instrucción y funciona como un punto al final de una oración.

3. ¿Cómo creaste esa variable?

Vamos a explorar lo que hace tu código más detalladamente:

Tienes la clase Input y estás navegando a través de la clase para encontrar el método que se llama GetAxis. Entonces estás invocando ese método al poner paréntesis tras su nombre. No obstante; al contrario de Start y Update, GetAxis tiene un **parámetro** —un dato que necesita para ejecutar su tarea. Específicamente, GetAxis necesita el nombre del axis para el cual está tratando de encontrar un valor. Estás tratando de encontrar el valor del eje horizontal aquí; por lo tanto, se lo has dado como un parámetro.

El tipo de dato para este tipo de información se llama una **cadena** o *string*. Esto se refiere a una cadena de caracteres, por ejemplo, una palabra o una oración. Al poner marcas de

discurso alrededor de la palabra Horizontal, le estás diciendo a la computadora que debe tratarla como una cadena.

Una vez que la computadora ha obtenido los valores del eje necesitará guardarlos en algún lugar. A la izquierda de la invocación del método has introducido una nueva variable llamada horizontal y la has configurado a que sea igual al valor que se encuentra de GetAxis.

Luego, añade otra línea de código para encontrar el valor del eje llamado **Vertical** y guardarlo en una variable llamada **vertical**.

El método Update debería verse así

```
// Update is called once per frame
void Update()
{
    float horizontal = Input.GetAxis ("Horizontal");
    float vertical = Input.GetAxis ("Vertical");
}
```

Ahora tienes los valores para los ejes horizontal y vertical. El siguiente paso es combinarlos dentro de un **vector**, de esa manera pueden usarse para cambiar posición.

4. ¿Cómo podemos hacer un vector?

En Unity, el espacio en 3D se representa con tres coordenadas las cuales hacen un vector. El tipo de dato que representa un vector en Unity se llama **Vector3**. Esto es lo que representa la posición de un GameObject y necesitarás hacer un Vector3 que representa ese cambio. El movimiento que el usuario representa es fundamental para esta clase y lo podrías necesitar para otras cosas también.

Con esto en mente, es importante considerar el **ámbito** de la variable que necesites crear.

El ámbito de la variable es el área de código donde puede utilizarse. Usualmente esto es tan simple como el bloque de código en el cual está declarado. Por ejemplo, ambas variables de punto flotante (floats) que acabas de declarar (horizontal y vertical) están en el ámbito para el método Update entero porque ese es el bloque de código donde fueron declaradas. Se dice que son **locales** para el método Update.

Si quieres usar una variable en muchos métodos diferentes, puedes crear variables fuera del ámbito del método. En cambio, estas son locales a la **clase**.

Añade la siguiente línea sobre las definiciones de los métodos:

```
Vector3 m_Movement;
```

Tu *script* debería verse así:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    Vector3 m_Movement;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        float horizontal = Input.GetAxis ("Horizontal");
        float vertical = Input.GetAxis ("Vertical");
    }
}

```

La nueva línea de código está instruyendo a la computadora que cree una variable tipo Vector3 llamada **m_Movement** que puedes usar donde quieras en la clase PlayerMovement (Movimiento del jugador).

Convenciones para la nomenclatura

¿Pero qué significa esa `_m` al principio del nombre? Esto es parte de algo que se llama una **convención de nomenclatura**. Las convenciones de nomenclatura se usan para identificar un objeto o la clase de un objeto. En este proyecto usarás las convenciones internas de Unity. Todas las variables comienzan con una letra minúscula pero las palabras que le siguen comienzan con una mayúscula —esto se llama **camelCase** o caja camello.

La excepción a esto son las **variables de un miembro privado**, las cuales comienza con el prefijo `m_` y todas las palabras comienzan con una letra mayúscula. Esto se llama **PascalCase**. Las variables de instancia o miembro de dato son aquellas que pertenecen a una clase antes que a un método específico. La parte `m_` de una variable privada de instancia viene de que son variables «miembro».

5. Configura los valores de tu variable

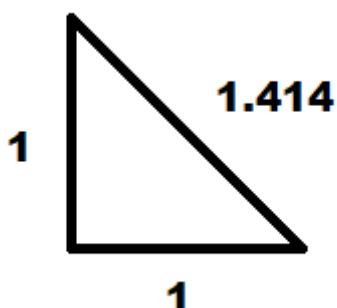
Ya que tienes una variable para guardar el movimiento de tu personaje, necesitas configurar un valor para ella. Desde que esto puede cambiar en cada marco, necesitas configurarla cada marco —deberías hacerlo en el método Update.

En el método Update, después de que las variables vertical y horizontal hayan sido creadas, añade la línea siguiente:

```
m_Movement.Set(horizontal, 0f, vertical);
```

Los vectores en un espacio de 3D tienen tres valores —el método **Set** (configurar) le asigna un valor a cada uno. Tiene tres parámetros, uno para cada coordenada del vector. El vector de movimiento ahora tiene un valor del ingreso horizontal en el eje horizontal, 0 en el eje y el ingreso verticales en el eje z. También hay una f después del 0 en el segundo parámetro, la cual le dice al computador que trate este número como un dato de punto flotante o *float*.

Ahora necesitas resolver un pequeño problema. El vector de movimiento está compuesto de dos números que pueden tener un valor máximo igual a 1. Si el valor de ambos es 1, la longitud del vector, (que también se conoce como su magnitud) será más de 1. Esta es la relación entre los lados de un triángulo que se describe en el teorema de Pitágoras.



Esto significa que tu personaje se moverá más rápido diagonalmente que si lo hace a lo largo de un solo eje. Para asegurarte de que esto no ocurra, necesitas asegurarte de que el vector de movimiento siempre tenga la misma magnitud. Puedes hacer esto al **normalizarlo**. Normalizar un vector significa mantener la misma dirección del vector, pero cambiar su magnitud a 1.

Añade el *script* siguiente debajo de la línea que escribiste anteriormente para llamar a un método en el mismo vector:

```
m_Movement.Normalize();
```

Tu *script* completo debería verse así:

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

```

public class PlayerMovement : MonoBehaviour
{
    Vector3 m_Movement;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        float horizontal = Input.GetAxis ("Horizontal");
        float vertical = Input.GetAxis ("Vertical");

        m_Movement.Set(horizontal, 0f, vertical);
        m_Movement.Normalize ();
    }
}

```

Este ejemplo incluye un espacio entre obtener las entradas y configurar el vector de movimiento. Esto no es importante, solo hace que se vea más ordenado

6. ¿Cómo se configura el componente Animator (animador)?

Ya que has creado un vector de movimiento, hay otras tareas que necesitan hacerse en cada marco. Necesitas decirle a la computadora que haga lo siguiente:

1. Dile al componente Animator si el personaje está caminando o no.
2. Obtén una rotación para el personaje desde una entrada del jugador (de manera similar a cómo obtuviste su movimiento).
3. Aplica movimiento y rotación a tu personaje

Vamos a comenzar con el componente Animator.

7. ¿Cómo se identifica si hay entradas del teclado del jugador o no?

Si hay alguna entrada del jugador en el teclado entonces tu personaje debería caminar y si no hay entonces debería quedarse quieto.

1. Para comenzar, necesitas escribir una línea de código que determine si hay entradas horizontales o no. Añade el siguiente código después de la línea que normaliza el movimiento del vector:

```
bool hasHorizontalInput = Mathf.Approximately (horizontal,  
0f);
```

Aquí estás creando una **variable tipo «bool»** (la que puede ser true o false) y le estás dando el nombre de `hasHorizontalInput` (Tiene entrada horizontal). Luego estás configurándola a que sea igual al valor que regresa un método. Este método se llama `Approximately` (aproximadamente) y es de la clase `Mathf`. Toma dos parámetros de tipo punto flotante y regresa una variable tipo «bool» —verdadero si los dos puntos flotantes son aproximadamente iguales y falso si no lo son. Por lo tanto; en este escenario, el método regresará el valor true si la variable `horizontal` es aproximadamente cero.

¡Pero espera! Hay otro carácter en esta línea con el que no te has topado antes: el signo de exclamación de cierre en frente de la invocación del método. Este es el **operador lógico de negación** e invierte una variable tipo `bool` poniendo `true` a `false` (verdadero a falso) y `false` a `true` (falso a verdadero). Esto significa que `hasHorizontalInput` se configura a `true` (verdadero) cuando la variable `horizontal` no es aproximadamente igual a 0. Es decir, `hasHorizontalInput` es verdadero cuando `horizontal` no es cero.

2. Pero no solo te debe importar el eje horizontal. Añade una línea similar para el eje vertical:

```
bool hasVerticalInput = Mathf.Approximately (vertical, 0f);
```

Esta línea está haciendo exactamente lo mismo, pero para el eje vertical.

3. Ahora sabes que cuando obtienes entradas en los ejes, necesitas combinarlos en una sola variable tipo «bool». Añade la siguiente línea de código:

```
bool isWalking = hasHorizontalInput || hasVerticalInput;
```

Esta línea crea otra variable tipo `bool` llamada `isWalking`, la cual está configurada a los valores de `hasHorizontalInput` o `hasVerticalInput`. Las dos líneas verticales son los **operadores lógicos OR**. Esto compara la variable `bool` en cada lado. Si uno de ellos, o si ambos son verdaderos; entonces, es igual a `true`. De otra manera es igual a `false`. Eso significa que si `hasHorizontalInput` o `hasVerticalInput` son verdaderos; entonces, `isWalking` es `true` y si no lo son, es `false` o `falso`.

8. ¿Cómo se crea una variable para guardar una referencia hacia el componente Animator (Animador)?

Luego, necesitas decirle al componente Animator si el personaje debería estar caminando o no usando la variable tipo «bool» que acabas de crear. Para hacerlo, necesitas acceder al componente Animator.

Pero, espera un momento, ¿por qué necesitas hacer algo especial para acceder al componente Animator cuando no necesitaste hacerlo para invocar métodos en Input o Mathf? Esto se debe a que los métodos Input y Mathf son estáticos.

Los métodos estáticos son métodos que se invocan en el tipo de una clase en vez de uno de los casos de esa clase. No hay necesidad de tener un solo caso de la clase Input para determinar el valor de los ejes porque Input es un concepto más global. Por definición, los métodos que obtienen esos valores son estáticos. De la misma manera, la clase Mathf está llena de métodos auxiliares (métodos que ayudan a otro método a ejecutar su tarea) los que no involucran ningún dato específico para un caso específico de Mathf; por lo tanto, esos métodos también son estáticos.

Sin embargo, toma en consideración tu variable m_Movement. Tuviste que configurar valores específicos para ese caso en particular de Vector3; por lo tanto, esos métodos no son estáticos. The important thing to remember is that static methods are called using a type name and non-static methods (or 'instance' methods) are called using an instance name.

9. ¿Cómo se hace una referencia que apunte hacia el componente Animator?

Antes de que puedas acceder al componente *Animator* necesitas configurar una referencia que apunte hacia ello y esto se obtiene a través del método **GetComponent** (Obtener componente). Esta referencia se usará a través de la clase y no solo en un solo método. Tiene sentido mantenerlo como una variable de instancia (igual que el vector de movimiento); de tal manera que su ámbito sea local a la clase.

En la parte superior de la clase, encima de la declaración del vector de movimiento, pero debajo de la declaración de la clase, añade la siguiente línea

```
Animator m_Animator;
```

El *script* debería verse así:

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

public class PlayerMovement : MonoBehaviour
{
    Animator m_Animator;
```

```

    Vector3 m_Movement;

    void Start ()
    {
    }

    void Update ()
    {
        float horizontal = Input.GetAxis ("Horizontal");
        float vertical = Input.GetAxis ("Vertical");

        m_Movement.Set(horizontal, 0f, vertical);
        m_Movement.Normalize ();

        bool hasHorizontalInput = !Mathf.Approximately
(horizontal, 0f);
        bool hasVerticalInput = !Mathf.Approximately (vertical,
0f);
        bool isWalking = hasHorizontalInput || hasVerticalInput;
    }
}

```

Ojo: En este ejemplo los comentarios encima de los métodos Start y Update no han sido removidos. Si quieras podrías hacerlo, pero no es necesario —los comentarios no afectan el código en manera alguna.

10. ¿Cómo se configura la referencia que apunta hacia el componente Animator?

Has creado una variable para guardar una referencia al componente *Animator*, pero hasta ahora no has configurado algún valor para esta variable y está vacía. Cuando una variable está vacía de esta manera en C#, se dice que es **null** (nula). Cuando pueda haber una referencia, pero no hay una, la referencia es **null**.

No quieres que haya una referencia nula aquí y por eso necesitas configurar la referencia de manera adecuada. También necesitas poder acceder al componente *Animator* en cualquier método; por lo tanto, sería bueno que configures la referencia tan pronto como sea posible.

Uno de los métodos que se invocan primero en un MonoBehaviour es el método Start que exploraste anteriormente en este tutorial; de tal manera, tiene mucho sentido que configures la referencia allí. Agrega la línea siguiente al método Start:

```
m_Animator = GetComponent<Animator> ();
```

El método debería verse así:

```
void Start ()  
{  
    m_Animator = GetComponent<Animator>();  
}
```

Esta línea de código usa sintaxis que ya hemos visto y otra que es nueva:

- La variable a la que estás asignándole valor está a la izquierda.
- El nombre de un método está a la derecha (pero no tiene nada escrito antes).
- Hay signos de mayor que y menor que alrededor del Animator, antes de los paréntesis que has visto antes.
- La línea termina con un punto y coma.

¿Cuál es el significado de esta referencia?

Vamos a explorar tu código:

Primero, ¿por qué no hay una clase antes de GetComponent (Obtener componente)? Cuando añadiste esto antes, estabas accediendo métodos en otros objetos (por ejemplo, el método Normalize en el vector de movimiento). Sin embargo, GetComponent es algo a lo que ya tienes acceso, es parte del MonoBehaviour y ya que tu clase es un MonoBehaviour, ya tienes acceso a ella.

Luego, los símbolos mayor que y menor que. Estos han sido añadidos porque el método GetComponent es **genérico**. Un método genérico es aquel que tiene dos configuraciones diferentes de parámetros: **parámetros normales** y **parámetros de tipo**. Los parámetros enlistados entre los símbolos de mayor y menor que son los parámetros de tipo.

En este escenario, GetComponent necesita saber qué **tipo** de componente buscas. Si buscas un componente Animator, el tipo de parámetro es Animator. La línea de código dice, «obtén una referencia que apunte hacia el componente de tipo “Animator” y asígnaselo a la variable llamad `m_Animator`».

¿Cómo se configura el `isWalking` parámetro de tipo Animator?

Ahora que tienes una referencia que apunta hacia el componente Animator, puedes usarla para configurar el `IsWalking` parámetro de tipo Animator que creaste en el tutorial anterior.

Añade la línea de código siguiente debajo de la creación de la variable `isWalking` en el método `Update`:

```
    m_Animator.SetBool ("IsWalking", isWalking);
```

Este código invoca el **método SetBool** usando la referencia al componente Animator que acabas de configurar. El primer parámetro es el nombre del parámetro Animator al que quieres asignarle un valor y el segundo es el valor que les quieres asignar. Es importante que

prestes atención a la ortografía y el uso de mayúsculas y que los copies exactamente; de otra manera, el método no sabrá a cuál de los parámetros Animator debe asignarle un valor.

El script debería verse así:

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

public class PlayerMovement : MonoBehaviour
{
    Animator m_Animator;
    Vector3 m_Movement;

    void Start ()
    {
        m_Animator = GetComponent<Animator>();
    }

    void Update ()
    {
        float horizontal = Input.GetAxis ("Horizontal");
        float vertical = Input.GetAxis ("Vertical");

        m_Movement.Set(horizontal, 0f, vertical);
        m_Movement.Normalize ();

        bool hasHorizontalInput = !Mathf.Approximately
(horizontal, 0f);
        bool hasVerticalInput = !Mathf.Approximately (vertical,
0f);
        bool isWalking = hasHorizontalInput || hasVerticalInput;
        m_Animator.SetBool("IsWalking", isWalking);
    }
}
```

¡Esto es todo! Has configurado el valor del parámetro Animator.

11. ¿Cómo puedes crear una rotación para tu personaje?

Vamos a echarle un vistazo a la tarea que necesita completarse en cada marco: la creación de una rotación para tu personaje.

En este juego el personaje debería poder caminar solo hacia el frente y necesita mirar hacia la misma dirección en la que se mueve. Sin embargo, si John Lemon se volteó rápidamente hacia

la dirección que quieras se verá un poco raro; por ende, necesitas que sea un poco más despacio. Pero ¿cuánto deberías retardarlo?

Una manera más fácil de plantear esta cuestión es considerar la velocidad en la que el personaje debe voltearse.

Para ver esta velocidad, necesitas crear una nueva variable.

¿Cómo crear una variable turnSpeed?

Añade la siguiente línea a la parte superior de la clase, encima de la variable de instancia:

```
public float turnSpeed;
```

Vamos a explorar este código.

Has añadido la palabra **public** antes de la declaración de la variable. En Unity, las variables de instancia públicas aparecen en la ventana Inspector y pueden ser modificadas.

También has usado la caja de camello (en vez de PascalCase, con su prefijo m_). Esto se debe a que la variable es pública y las convenciones de nomenclatura de Unity usan este formato para las variables de instancia públicas. Las convenciones de nomenclatura pueden ser muy útiles, pero no hay una razón técnica por la que las adoptamos.

¿Cómo puedes calcular el **forward vector** (vector delantero) de tu personaje?

Recuerda que necesitas que el personaje mire hacia la dirección de su movimiento. Todos los componentes Transform tienen un vector delantero y un buen paso intermedio sería calcular lo que quieras que sea el **forward vector (vector delantero)** de tu personaje.

Añade la línea siguiente de código al final del método Update:

```
Vector3 desiredForward = Vector3.RotateTowards  
(transform.forward, m_Movement, turnSpeed * Time.deltaTime, 0f);
```

- Es una línea de código muy larga y parece muy complicada, pero incluye muchas cosas familiares. Vamos a analizarla paso por paso:
- Este código crea una variable Vector3 llamada desiredForward (dirección delantera deseada).
- Esto configura el retorno de un método que se llama RotateTowards (rotar hacia), el cual es un método estático de la clase Vector3. RotateTowards toma cuatro parámetros —los primeros dos son de tipo Vector3 y son los vectores que se rotan desde y hacia un punto respectivamente.
- El código empieza con transform.toward (transformar hacia) y apunta hacia la variable m_Movement. Transform.forward es un acceso rápido al componente Transform y obtiene su vector delantero.

- Los siguientes dos parámetros son la cantidad entre el vector de arranque y el vector meta: primero, el cambio en el ángulo (en radianes) y luego el cambio en magnitud. Este código cambia el ángulo al multiplicarlo por **turnSpeed** (velocidad de giro) * **Time.deltaTime** (tiempo por tiempo delta) y la magnitud por 0.

Time.deltaTime es el tiempo que ha pasado desde el marco anterior (puedes pensar sobre esto como el tiempo entre marcos). Entonces ¿por qué necesitas multiplicar **turnSpeed** por eso?

El método **Update** se invoca en cada marco y si tu juego está rodando a 60 marcos por segundo; entonces, eso significa que este método será invocado 60 veces en un segundo. Habría un cambio muy pequeño en cada invocación; de tal manera que, a lo largo de 50 marcos tu obtendrás el cambio que quieras por segundo. ¿Pero qué pasa si un juego rueda a 30 marcos por segundo? Solo la mitad de las invocaciones del método se harían en ese tiempo; por lo tanto, solo la mitad del giro habría sucedido. No quieres que el número de marcos por segundo afecte cuán rápido gire tu personaje —eso no sería nada bueno.

¿Qué pasaría si en vez de un cambio por marco, estás lidiando con un cambio por segundo? Eso haría las cosas mucho más fáciles. Para hacerlo, necesitas multiplicar cualquier cambio que quieras por segundo por la cantidad de tiempo que toma un marco. Esto es exactamente lo que hace esta línea de código.

12. ¿Cómo se ajusta la variable **turnSpeed** (Velocidad de giro)?

La variable **turnSpeed** es el ángulo en radianes que quieras que el personaje gire por segundo. Esto se multiplica por **Time.deltaTime** para obtener lo que el personaje debe girar en este marco. Los radianes son una medida diferente para un ángulo. Son similares a los grados, pero son una medida más natural. Un círculo tiene 2π dentro; por lo tanto, aproximadamente 6. Tu personaje siempre va a tomar el giro más corto; entonces, lo máximo que tu personaje giraría serían **3 radianes**.

Dado que, una **turnSpeed** de 3 significa que le toma a tu personaje cerca de un segundo para hacer un giro completo. Eso es muy lento. Una velocidad de giro de 5 significaría que le toma cerca de medio segundo para un giro completo, lo que es muy lento. Vamos a probar un valor de 20 y ver qué te parece —la puedes cambiar luego, si quieres.

Cambia la línea que declara la variable **turnSpeed** en la parte superior de la clase a:

```
public float turnSpeed = 20f;
```

¡Ahora tenemos un vector para la dirección en la que quieras que mire tu personaje!

¿Cómo crear y guardar una rotación?

Luego, necesitas usar el vector para obtener una rotación y guardarla; de esa manera, puedes usarla cuando quieras. Vas a guardarla al igual que lo hiciste con el vector de movimiento; por ende, tiene sentido declarar esa variable aquí.

Añade la siguiente línea debajo de la línea que declara el Vector3 llamado m_Movement:

```
Quaternion m_Rotation = Quaternion.identity;
```

Los **quaternions** (**cuaternarios**) son una manera de guardar rotaciones; ellos evaden algunos de los problemas que conlleva guardar rotaciones como un vector 3D. Este tutorial no los explora en detalle —el saber qué son y que se usan para guardar rotaciones es todo lo que necesitas saber por el momento.

Les has dado al Quaternion un valor por defecto de Quaternion.identity. Normalmente, las variables que son parte de la clase (variable de instancia o miembros) en vez de ser parte de un método específico son configuradas a su valor por defecto cuando se crea un caso de la clase. Por ejemplo, el valor por defecto de un Vector3 es tener los valores de X, Y y Z configurados a 0. Ocurre lo mismo con un Quaternion. Sin embargo, mientras que tiene sentido tener un vector cero (ya que indica que no hay movimiento), un Quaternion cero no tiene mucho sentido. Configurar este valor de Quaternion.identity simplemente le da un valor que indica que no hay rotación, lo que tiene mucho más sentido como valor por defecto.

Ya que has creado una variable de rotación, ahora puedes configurarla. Añade la siguiente línea debajo de la creación de la variable desiredForward:

```
m_Rotation = Quaternion.LookRotation (desiredForward);
```

Esta línea simplemente invoca el método LookRotation y crea una rotación que mira en la dirección del parámetro que se ha pasado.

Tu script debería verse así:

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

public class PlayerMovement : MonoBehaviour
{
    public float turnSpeed = 20f;

    Animator m_Animator;
    Vector3 m_Movement;
    Quaternion m_Rotation = Quaternion.identity;

    void Start ()
    {
```

```

        m_Animator = GetComponent<Animator>();
    }

    void Update ()
    {
        float horizontal = Input.GetAxis ("Horizontal");
        float vertical = Input.GetAxis ("Vertical");

        m_Movement.Set(horizontal, 0f, vertical);
        m_Movement.Normalize ();

        bool hasHorizontalInput = !Mathf.Approximately
(horizontal, 0f);
        bool hasVerticalInput = !Mathf.Approximately (vertical,
0f);
        bool isWalking = hasHorizontalInput || hasVerticalInput;
        m_Animator.SetBool("IsWalking", isWalking);

        Vector3 desiredForward = Vector3.RotateTowards
(transform.forward, m_Movement, turnSpeed * Time.deltaTime, 0f);
        m_Rotation = Quaternion.LookRotation (desiredForward);
    }
}

```

13. ¿Cómo aplicar movimiento y rotación a tu personaje?

¡Casi has terminado! El último paso es aplicar el movimiento y la rotación a tu personaje. Hay muchas maneras en las que puedes hacerlo, pero desde que el personaje necesita ser parte del sistema de física, necesitas mover el Rigidbody en vez de usar cualquier otra técnica.

Para hacerlo, necesitas una referencia al componente Rigidbody. Puedes obtenerlo de la misma manera que lo hiciste para obtener el componente Animator.

1. Añade la línea de código siguiente a tu **script** inmediatamente después de la declaración de la variable Animator:

```
Rigidbody m_Rigidbody;
```

2. Añade otra línea de código justo después de configurar la referencia a la variable Animator:

```
m_Rigidbody = GetComponent<Rigidbody> ();
```

Ahora que tienes la referencia al Rigidbody, vamos a pensar sobre los detalles que conlleva el mover un personaje animado. El personaje tiene una animación Walk (caminar) muy divertida y sería bueno usar el movimiento de raíz para ella. No obstante, la animación no tiene ningún

giro y si tratas de girar el Rigidbody en el método Update, podría ser ignorado por la animación (lo que podría resultar en que el personaje no gire cuando debe hacerlo).

Lo que realmente necesitas es un poco del movimiento de raíz de la animación, pero no todo; específicamente, necesitas aplicar el movimiento, pero no la rotación. Entonces ¿cómo cambiar la manera en la que el movimiento de raíz se aplica desde el Animator? Afortunadamente, los MonoBehaviours tienen un método especial que puedes usar para cambiar cómo se aplica el movimiento de raíz desde el Animator.

Declara un nuevo método debajo del método Update:

```
void OnAnimatorMove ()  
{  
}  
}
```

Este método te permite aplicar el movimiento de raíz como quieras aplicarlo, lo que significa que el movimiento y la rotación pueden aplicarse por separado.

14. Movimiento

Vamos a comenzar con movimiento. Añade la línea siguiente al método OnAnimatorMove:

```
m_Rigidbody.MovePosition (m_Rigidbody.position + m_Movement *  
m_Animator.deltaPosition.magnitude);
```

Esta es otra línea de código que parece un poco complicada; pero otra vez, hay muy poco que realmente sea nuevo para ti aquí.

Primero, estás usando tu referencia al componente Rigidbody para invocar su método MovePosition (mover posición) y pasándolo en un solo parámetro: su nueva posición. La nueva posición comienza en la posición actual del Rigidbody y luego añades un cambio a eso —el vector de movimiento multiplicado por la magnitud de deltaPosition (posición delta) del Animator. Pero ¿qué significa eso?

La posición delta o deltaPosition del Animator es el cambio de posición debido al movimiento que habría sido aplicado a este marco. Estás tomando la magnitud de eso (su longitud) y lo multiplicas por el vector del movimiento el cual está en la dirección actual en la que queremos que se mueva el personaje.

15. Rotación

Luego, aplica la rotación. Añade la línea siguiente justo debajo de la invocación MovePosition (mover posición) en el método OnAnimatorMove (cuando se mueva el animador...):

```
m_Rigidbody.MoveRotation (m_Rotation);
```

Esto es muy similar a la invocación de MovePosition, excepto que se aplica a la rotación. Esta vez no estás aplicando un cambio a la rotación, estás configurando la rotación directamente.

¡Esta es la última línea de código en tu primer *script*! Pero hay algo más que deberías ajustar.

16. ¿Cómo puedes hacer cambios a tu método

Update (actualizar)?

En el tutorial anterior aprendiste acerca del bucle Update o actualizar (que se usa para reproducción) y el bucle FixedUpdate o actualización fija (el mismo que corre las operaciones que usan física). Te has asegurado de que el Animator corra a tiempo con el bucle de física para evitar conflictos entre la física y la animación. No obstante, ahora estás ignorando el movimiento de raíz usando OnAnimatorMove (cuando se mueva el animador...). Esto significa que OnAnimatorMove realmente será invocado a tiempo con la física y no con la reproducción como tu método Update.

El vector de movimiento y rotación se configura en Update. Si se invoca primero OnAnimatorMove luego tendrás un problema porque un Quaternion (cuaternario) sin un valor no tiene ningún sentido.

Para asegurarte de que el vector de movimiento y rotación esté configurado para que funcione en conjunto con OnAnimatorMove, cambia tu método Update a uno que sea un método FixedUpdate de la manera siguiente:

```
void FixedUpdate ()
```

Este es otro método especial que Unity invoca automáticamente, pero este funciona en conjunto con la física. En vez de invocarse en cada marco, FixedUpdate se invoca antes de que el sistema de física resuelva cualquier colisión y otras interacciones que hayan ocurrido. Por defecto se invoca 50 veces por segundo.

¡Eso es todo! El *script* completo debería verse así:

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

public class PlayerMovement : MonoBehaviour
{
    public float turnSpeed = 20f;

    Animator m_Animator;
```

```

        Rigidbody m_Rigidbody;
        Vector3 m_Movement;
        Quaternion m_Rotation = Quaternion.identity;

        void Start ()
        {
            m_Animator = GetComponent<Animator> ();
            m_Rigidbody = GetComponent<Rigidbody> ();
        }

        void FixedUpdate ()
        {
            float horizontal = Input.GetAxis ("Horizontal");
            float vertical = Input.GetAxis ("Vertical");

            m_Movement.Set(horizontal, 0f, vertical);
            m_Movement.Normalize ();

            bool hasHorizontalInput = !Mathf.Approximately
(horizontal, 0f);
            bool hasVerticalInput = !Mathf.Approximately (vertical,
0f);
            bool isWalking = hasHorizontalInput || hasVerticalInput;
            m_Animator.SetBool ("IsWalking", isWalking);

            Vector3 desiredForward = Vector3.RotateTowards
(transform.forward, m_Movement, turnSpeed * Time.deltaTime, 0f);
            m_Rotation = Quaternion.LookRotation (desiredForward);
        }

        void OnAnimatorMove ()
        {
            m_Rigidbody.MovePosition (m_Rigidbody.position +
m_Movement * m_Animator.deltaPosition.magnitude);
            m_Rigidbody.MoveRotation (m_Rotation);
        }
    }
}

```

Ojo: En C# el orden en el que se declaran los métodos en una clase no importa; por ende, tus métodos tal vez estén ordenados de manera diferente.

Guarda tu *script* y felícítate a ti mismo por haber hecho un buen trabajo. Es hora de regresar a Unity y poner a prueba lo que has hecho.

17. ¿Cómo puedes poner a prueba los cambios

que has hecho?

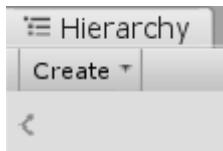
¿Cómo se le añade el *script* PlayerMovement (Movimiento del jugador) a John Lemon?

Una vez que hayas regresado a Unity, necesitas añadir el botón Add Component (Añadir componente) a la ventana Inspector, pero también lo puedes hacer de otra manera :

1. selecciona el GameObject John Lemon;
2. en la ventana Inspector, haz clic en la opción Edit Prefab (Editar el prefab) para entrar al modo Prefab;
3. en la ventana Project (Proyecto), ve a **Assets > Scripts** (Recursos > Scripts) y encuentra el **script PlayerMovement** (Movimiento del jugador);
4. arrastra el *script* PlayerMovement desde la ventana Project a la ventana Inspector;
5. si tu modo Prefab no está configurado a Auto Save (autoguardar), pulsa el botón Save (guardar);



6. haz clic en la flecha «regresar» y regresa siguiendo la migra de pan del Prefab;



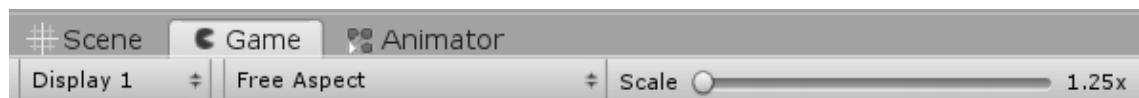
7. si la MainScene (Escena principal) no está cargada, ve a la ventana Project y cárgala haciendo doble clic en el Scene Asset (Recurso escena) en la carpeta **Assets > Scenes** (Recursos > Escenas);
8. selecciona el GameObject JohnLemon dentro de la Scene —verás que tiene todos los componentes y configuraciones que le diste a su Prefab.

18. ¿Cómo se ajustan las configuraciones de tu vista Game (Juego)?

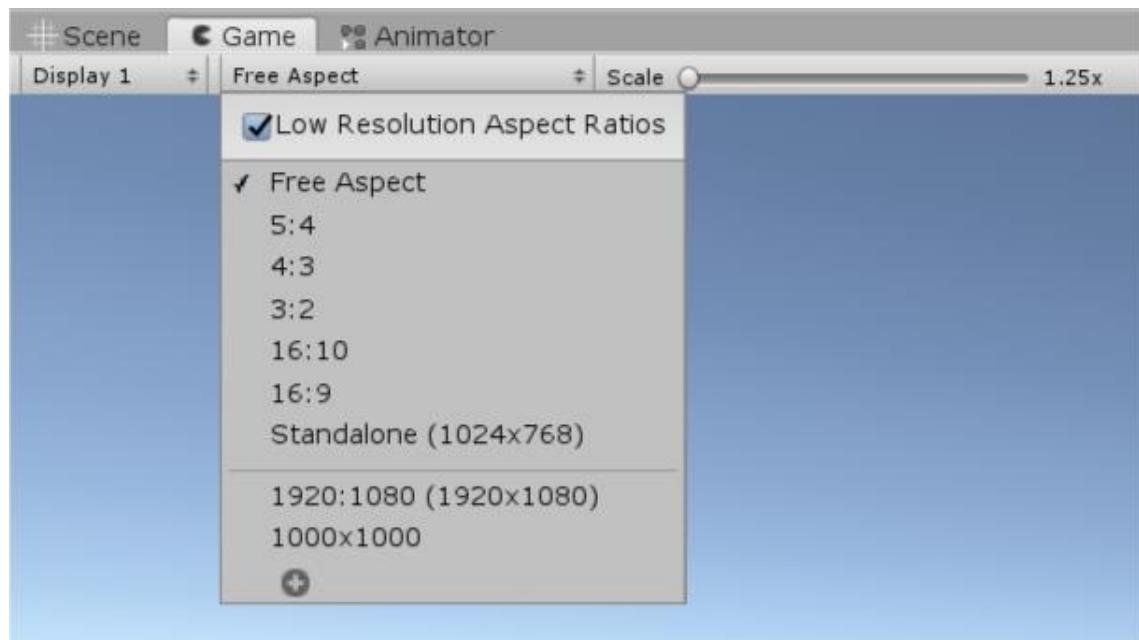
Antes de que pongas a prueba el Prefab, tal vez necesites ajustar algunas de las configuraciones para la vista Game (Juego). Si tienes un monitor de alta resolución, Unity automáticamente pone la ventana Game a escala para mejorar el rendimiento. Lo hace al configurar la escala de la ventana Game.

Actualmente, tu Scene (Escena) no es lo suficientemente complicada como para que el rendimiento sea algo de qué preocuparse; por lo tanto, si tienes este problema, lo vamos a resolver ahora.

1. Selecciona la pestaña de la ventana Game. Puedes ver si tu ventana Game ha sido puesta a escala en la parte superior.



2. Haz clic en el menú desplegable de proporción, el mismo que actualmente dice **Free Aspect** (Aspecto libre).



3. Deshabilita la casilla **Low Resolution Aspect Ratios** (Proporciones de baja resolución) y luego cambia el deslizador de escala a 1x. Si tu no puedes deshabilitar esta casilla (o si ya está deshabilitada), no te preocunes. Esto no afectará el juego cuando esté terminado y debería configurarse automáticamente para tu monitor.

¡La ventana Game ahora está como debería verse para que puedas poner a prueba JohnLemon! Haz clic en el botón Play para comenzar y usa las flechas en tu teclado para que puedas mover a John Lemon.

Ojo: Si recibes un mensaje de error de compilación, ¡no dejes que cunda el pánico! Regresa y verifica tu código con el ejemplo muy cuidadosamente. Es muy fácil cometer errores, especialmente cuando acabas de escribir tu primer *script*. Asegúrate de que guardes cualquier enmienda que hagas a tu código y luego ponlos a prueba en el modo Play otra vez.

19. Resumen

Durante este tutorial escribiste tu primer *script* y exploraste algunos de los conceptos fundamentales del escribir código en Unity. Si no tienes experiencia codificando y has entendido todo fácilmente, ¡qué bien! Pero, si tuviste dificultades con algo la primera vez que lo trataste de hacer, no te preocupes. Puede tomar un poco de tiempo y práctica llegar a sentirse cómodo o cómoda con estos conceptos.

En el tutorial siguiente, crearás el entorno y el sistema de iluminación para tu juego; de esa manera, JohnLemon tendrá algo muy espeluznante del que escapa

El entorno

Resumen

En el tutorial anterior, acabas de construir el personaje del jugador. Este personaje se puede mover, pero actualmente no tiene nada qué explorar. Ahora estás listo para añadir el entorno a tu Scene (Escena), iluminarla correctamente y añadir una Malla de navegación o Navigation Mesh para ayudar a los enemigos que eventualmente van a deambular por los pasillos de tu casa embrujada.

1. ¿Cómo se añade el entorno?

En el tutorial anterior, acabas de construir el personaje del jugador. Este personaje se puede mover, pero actualmente no tiene nada qué explorar. Ahora estás listo para añadir el entorno a tu Scene (Escena), iluminarla correctamente y añadir una Navigation Mesh (Malla de navegación) para ayudar a los enemigos que eventualmente van a deambular por los pasillos de tu casa embrujada.

Construir entornos en Unity es un proceso que puede variar mucho dependiendo en el tipo de juego que estás haciendo. Generalmente esto implica exemplificar modelos múltiples al arrastrarlos desde la ventana Project (Proyecto) a la ventana Hierarchy (Jerarquía). El entorno usualmente necesita una presencia física dentro de la escena; por lo tanto, se añade Colliders (Colisionadores) a cada uno de los modelos. A menudo, estos elementos son Prefabs (Prefabricados); entonces, no tendrás que agregar Colliders nuevos cada vez que añadas un elemento al entorno.

Crear los entornos para juegos puede ser un proceso muy laborioso y ya has practicado estos procesos cuando configuraste el personaje del jugador. Ya hemos creado el entorno para este juego para que no tengas que hacer algo muy repetitivo. Todo lo que necesitas hacer es exemplificarlo:

1. En la ventana Project ve a la carpeta **Assets > Scenes** (Recursos > Escenas) y haz clic doble en MainScene (Escena principal) para cargarla.
2. Navega a **Assets > Prefabs** (Recursos > Prefabricados) y selecciona el **Level Prefab** (Nivel prefabricado)
3. Arrastra el **Level Prefab** desde la ventana Project a la jerarquía para exemplificarlo.

El Level Prefab está constituido de muchos modelos, entre ellos paredes, suelos, puertas, muebles y decoraciones. Si quieres ver cómo algo está armado, simplemente puedes expandir su jerarquía en la ventana Hierarchy.

¿Cómo se posiciona el personaje del jugador?

Ahora que tiene un entorno dentro de la Scene, necesitas colocar a JohnLemon de tal manera que el personaje empieza el juego en el lugar correcto:

1. En la ventana Hierarchy, selecciona el GameObject JohnLemon
2. En el Inspector, configura la propiedad Position (Posición) del componente Transform (Transformar) a (-9,8, 0, -3,2).
3. Ve a **File > Save** y guarda tus cambios a la Scene.
4. Entra al modo Play para ver tus cambios. La cámara no está exactamente donde necesita estar, pero ahora hay un nivel completo para que JohnLemon explore y del que pueda escapar.

Este entorno es genial, pero se ve un poco plano. El siguiente paso es ajustar la iluminación para la Scene.

2. ¿Cómo se ilumina el entorno?

La iluminación tiene un gran impacto en el ambiente. Cambiar la iluminación en tu juego es el primer paso hacia crear un entorno realmente macabro que los jugadores puedan explorar.

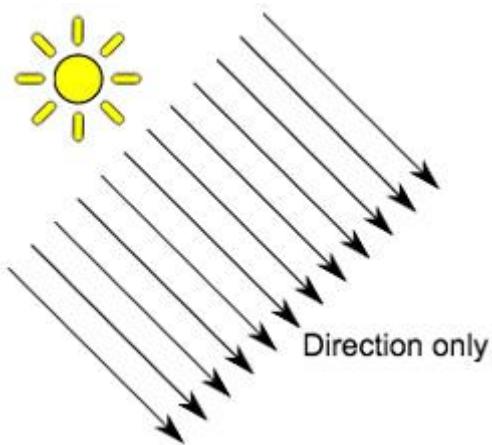
Las luces en Unity son simplemente GameObjects con un componente Light (luz) anexado. El componente Light contiene configuraciones que te permiten cambiar el color y la intensidad de una luz, y también el tipo de luz y otros controles más complejos para las sombras y Lightmapping.

Cada Scene nueva en Unity viene con una Luz direccional o **Directional Light**, uno de los dos GameObjects que se crea por defecto. Tu Scene también tiene una serie de luces extra que fueron agregadas por el Level Prefab.

Vamos a comenzar con la Luz direccional o Directional Light.

3. ¿Cómo se cambia la Luz direccional (Directional Light)?

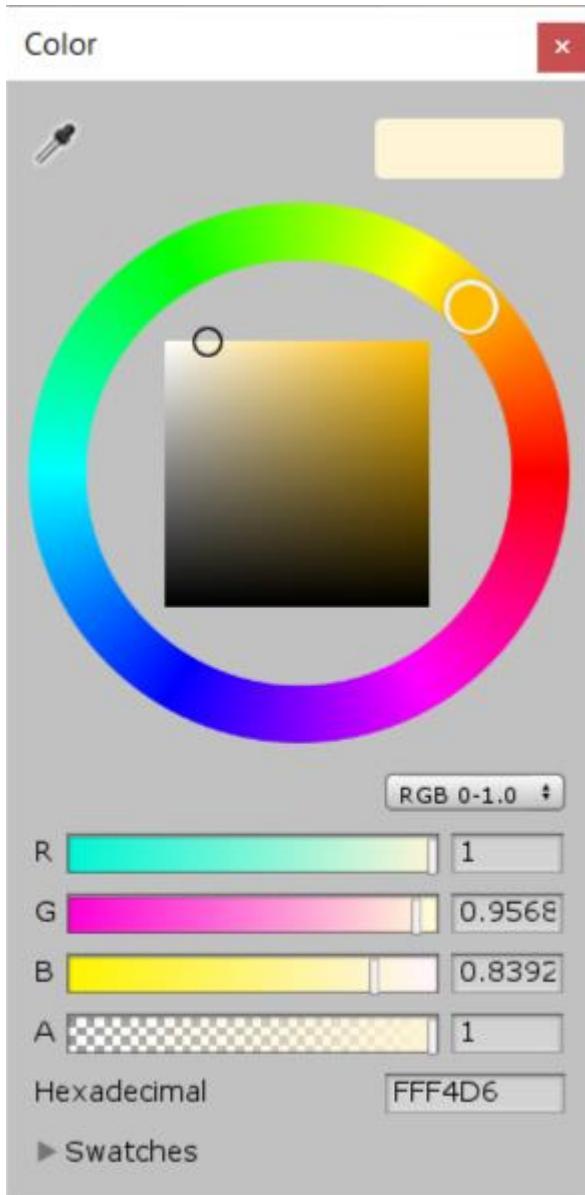
La Luz direccional (Directional Light) en tu Scene (Escena) está configurada al tipo **Directional** o **Direccional Type**. Las luces de tipo direccional imitan objetos luminarios muy distantes, como el sol. Pueden colocarse en cualquier lugar en la escena y rotarse al ángulo en el que la luz afecta la escena.



Vamos a alterar la luz direccional para que se vea más como la luz de la luna y para que cree unas sombras interesantes:

1. En la ventana Hierarchy, selecciona el GameObject **Directional Light**.
2. Vas a cambiar algunas de las propiedades del componente Light, comenzando con el Color, pero primero necesitas entender cómo funcionan los colores.

En el Inspector, haz clic en la propiedad Color. Hacer clic en cualquier propiedad Color en Unity va a abrir la **ventana para escoger colores** o **Color picker window**:



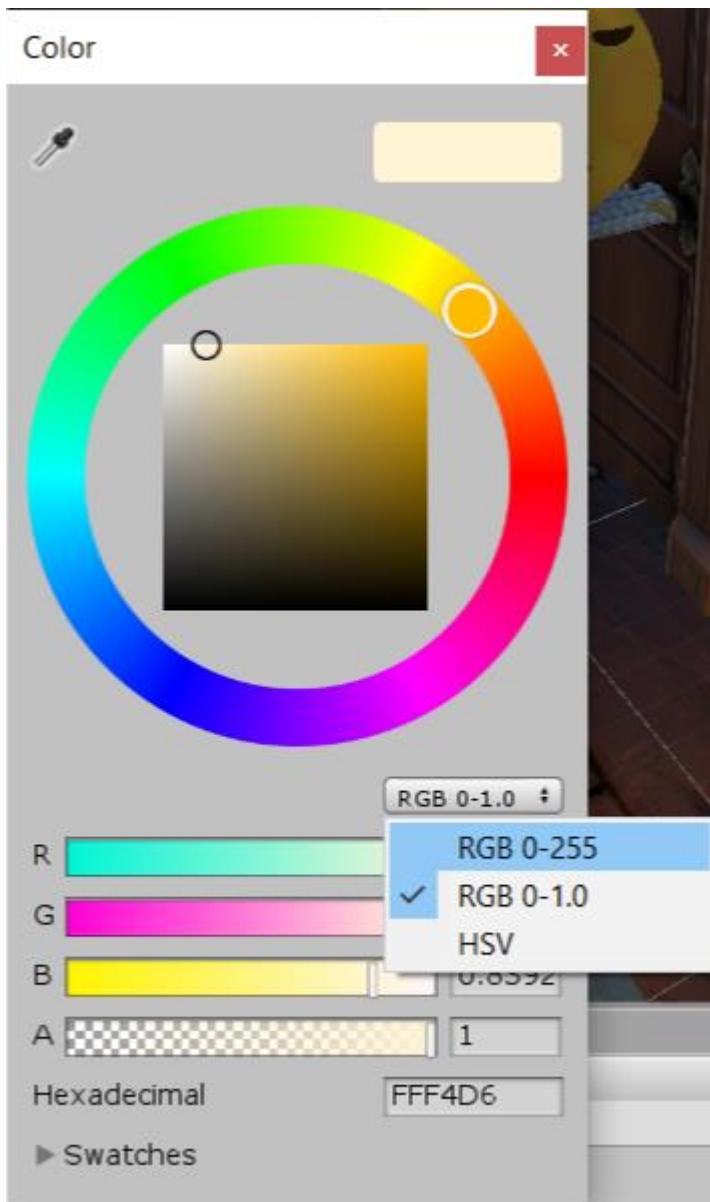
3. La ventana para escoger colores te da un número de opciones para configurar los valores de un color.

Cada color está constituido de cuatro canales:

- rojo,
- verde,
- azul,
- Alpha (alfa)

La cantidad de cada uno de estos canales que constituyen un color puede representarse en distintas maneras. Puedes escoger la representación usando el menú desplegable debajo del círculo de colores.

Verifica que el menú desplegable esté configurado a RGB 0-255. Esta representación significa que cada canal está representado por un entero entre 0 y 255.



4. Configura la propiedad Color del componente Light a (225, 240, 250). Esto creará una luz de un azul muy tenue en vez de una amarilla.
5. En el componente Light, incrementa la propiedad Intensity (Intensidad) a 2.
6. Ahora puedes mejorar la calidad de las sombras tenebrosas en la casa embrujada. En el componente Light, encuentra la propiedad **Realtime Shadows** (Sombras en tiempo actual). Cambia su propiedad Resolution (Resolución) de Use Quality Settings (Usar configuraciones de calidad) a **Very High Resolution** (La resolución más alta).
7. Configura el Bias de las propiedades Realtime Shadows y Normal Bias a 0,01.

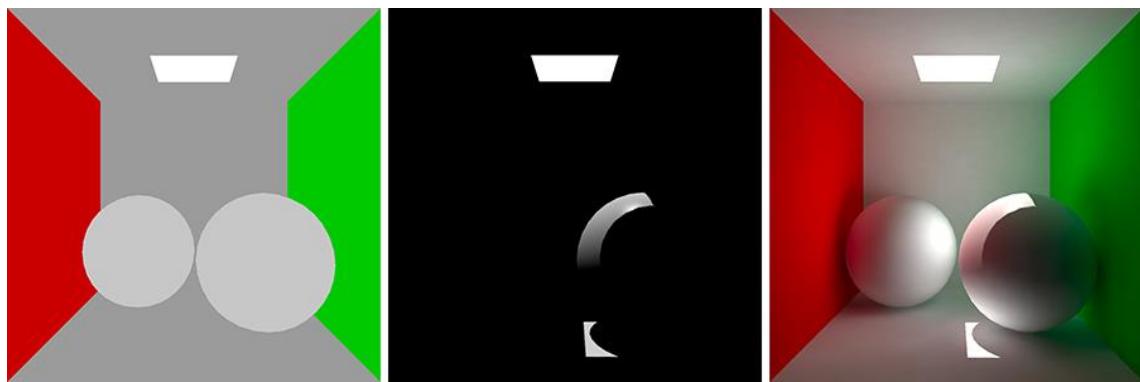
8. En el componente Transform (Transformación), configura la propiedad Rotation (rotación) a (30, 20, 0). Esto va a rotar la luz de tal manera que brille a través de las ventanas de la casa.

9. Guarda tu Scene.

¡La escena ya se ve mucho más tenebrosa que antes!

4. ¿Cómo se crea un efecto de iluminación global con Lightmapping?

Hay dos tipos de iluminación en Unity que te ayudan a simular el comportamiento de las luces en el mundo real: iluminación directa e indirecta. La iluminación directa viene desde una fuente específica, como el sol (la luz direccional en tu escena). La iluminación indirecta es la iluminación adicional que ocurre cuando la luz directa rebota en otras superficies.



La misma Escena (Scene): Sin iluminación (izquierda), solo con iluminación directa (centro) y con iluminación indirecta global (derecha). Presta atención a cómo los colores se transfieren cuando la luz «rebota» entre superficies, dándole un efecto mucho más realístico.

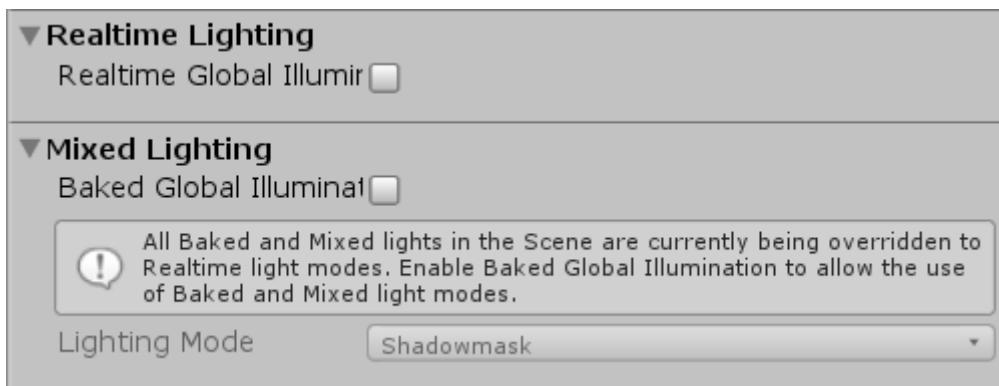
Este efecto se crea comúnmente en Unity usando la herramienta **Global Illumination Lightmapping** o Iluminación global con Lightmapping. Lightmapping imita el choque y rebote de luz en la escena y la escribe en un recurso o Asset guardado en el proyecto. Esto toma algo de tiempo, pero puede resultar en escenas que parecen muy realísticas.

Para ahorrar algo de tiempo puedes usar una aproximación para crear un efecto similar:

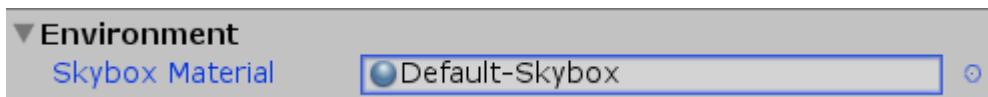
1. Ve al menú Window (ventana) y selecciona Rendering > Lighting Settings (Reproducción > Configuraciones de iluminación). Haz clic y arrastra la etiqueta con el nombre y estaciónala junto a la ventana Inspector.

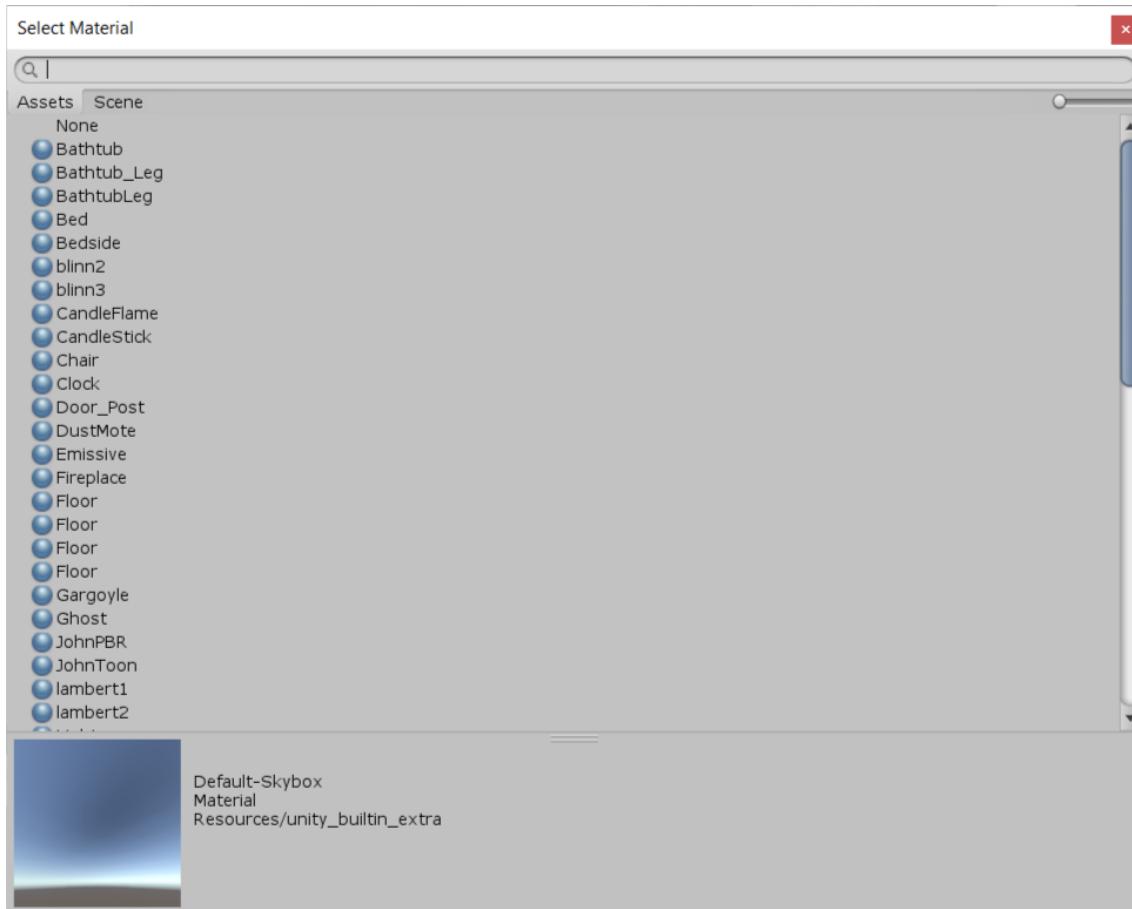
La ventana Lighting Settings (Configuraciones para la iluminación) es el punto de control principal para las herramientas de iluminación global de Unity para la escena cargada actualmente. También contiene las configuraciones para la iluminación del entorno (Environment Lighting), reflejos (Reflections) y niebla (fog). The Lighting Settings window is the main control point for Unity's Global Illumination features for the currently loaded scene. It also contains the settings for Environment Lighting, Reflections and Fog

2. En la sección Realtime Lighting (Iluminación en tiempo actual) deshabilita la casilla **Realtime Global Illumination** (Iluminación global en tiempo actual). En la sección Mixed Lighting (Iluminación mixta) deshabilita la casilla Baked Global Illumination (Iluminación global incrustada).



3. En la sección Environment (entorno), usa el botón en forma de círculo en la extrema derecha de la propiedad **Skybox Material** (Material para la bóveda celestial) para configurarla a None (una referencia tipo null). No necesitas un cielo realístico para este juego y vas a usar iluminación alternativa.





4. En la sección Environment (entorno), configura la fuente de luz para el entorno o **Environment Light Source** a **Gradient** (gradiente).

El gradiente está separado en tres campos de colores:

- **Sky** o cielo, el mismo que controla la luz del entorno que viene desde arriba de la escena.
- **Equator** o ecuador, el mismo que controla la luz que viaja desde el horizonte hacia el medio de la escena.
- **Ground** o suelo, el mismo que controla la luz que viene desde debajo de la escena.

Puedes considerar el gradiente como una gran esfera que envuelve a toda la escena.

5. Configura el color del cielo o Sky a un gris más claro (170, 180, 200). Esto incrementará el resplandor de los pisos y objetos como camas y mesas.

6. Configura el color del ecuador o Equator a un gris azulado (90, 110, 130). Esto incrementará el resplandor de las paredes y los ornamentos.

7. Configura el color del piso o Ground a negro (0, 0, 0). El color del piso puede agregar una luz que apunta hacia arriba que crearía un efecto de iluminación global muy agradable. No obstante, este efecto sería demasiado resplandeciente para la casa embrujada que estás creando.

8. Guarda tu escena.

Acabas de completar la iluminación básica para tu juego y la has usado para mejorar su ambiente tenebroso. Luego, crearás una malla que ayudará a los fantasmas a moverse a través del entorno de tu juego.

5. ¿Cómo se añade una malla de navegación o Navigation Mesh?

Todos sabemos que las casas embrujadas tienen fantasmas que deambulan en los pasillos. ¡La tuya no debería ser diferente!

Para ayudar a nuestros fantasmas a navegar a través de la casa, Unity tiene un sistema integrado llamado **NavMesh** o malla de navegación. En el segundo tutorial aprendiste que una malla o Mesh es un conjunto de triángulos que calzan el uno con el otro para definir una forma. Esta malla permite que JohnLemon sea reproducido en la pantalla. La malla de navegación o NavMesh es una forma invisible sobre el suelo que define un área en la que los GameObjects seleccionados se pueden mover.

Entonces, ¿cómo decides qué áreas se pueden mover y cuáles no?

[¿Cómo se designa un GameObject como algo estático?](#)

Cuando se identifica un GameObject como algo estático o **Static**, el sistema de navegación de Unity asume que no se moverá. El entorno de tu juego está constituido de muchos GameObject con muchos componentes de reproducción de mall o Mesh Renderer. La combinación de todas las mallas de los componentes de reproducción de malla o Mesh Renderer Components cuyos GameObjects han sido designados como estáticos forma la base para la malla de navegación o NavMesh.

Para identificar los GameObjects del entorno como estáticos:

- 1.** En la jerarquía o Hierarchy, selecciona el GameObject **Level** (nivel);
- 2.** En el inspector, habilita la casilla **Static**;



3. Una casilla de diálogo va a aparecer preguntándote si quieres habilitar las casillas estáticas para todos los objetos hijos también. Selecciona, **Yes, change children** (Sí, cambia los hijos).



4. Ahora el GameObject **Level** (nivel) y todos sus GameObjects hijos están designados como estáticos, pero necesitas establecer una excepción. Hay un **Ceiling Plane** o plano del techo en el diseño del nivel, el mismo que se usa para crear sombras. Si incluyes esto, los fantasmas podrían terminar caminando en el techo. Aunque suena muy tenebroso, eso no va a funcionar en este juego.

En la ventana Hierarchy expande el GameObject **Level** y sus hijos.

5. Ve a **Level > Corridors > Dressings > Ceiling Plane** (Nivel > Pasillos > Decoración> Plano del techo) y selecciona el GameObject **Ceiling Plane** o plano del techo.



6. En el Inspector, deshabilita la casilla **Static** (estático).

7. Ve a **File > Save** (Archivo > Guardar) y guarda tu escena.

Tal vez habrás notado que estás haciendo estos cambios dentro de la escena en vez de en el Prefab Level. Es decir, estás ignorando la configuración del Prefab sin

cambiarla. Esto es muy útil cuando quieras hacer cambios menores a un patrón estándar

6. ¿Cómo se crea la Malla de navegación o Nav Mesh?

El proceso de crear una malla de navegación o Navmesh se llama **baking**. Esto se hace desde la ventana Navigation (navegación).

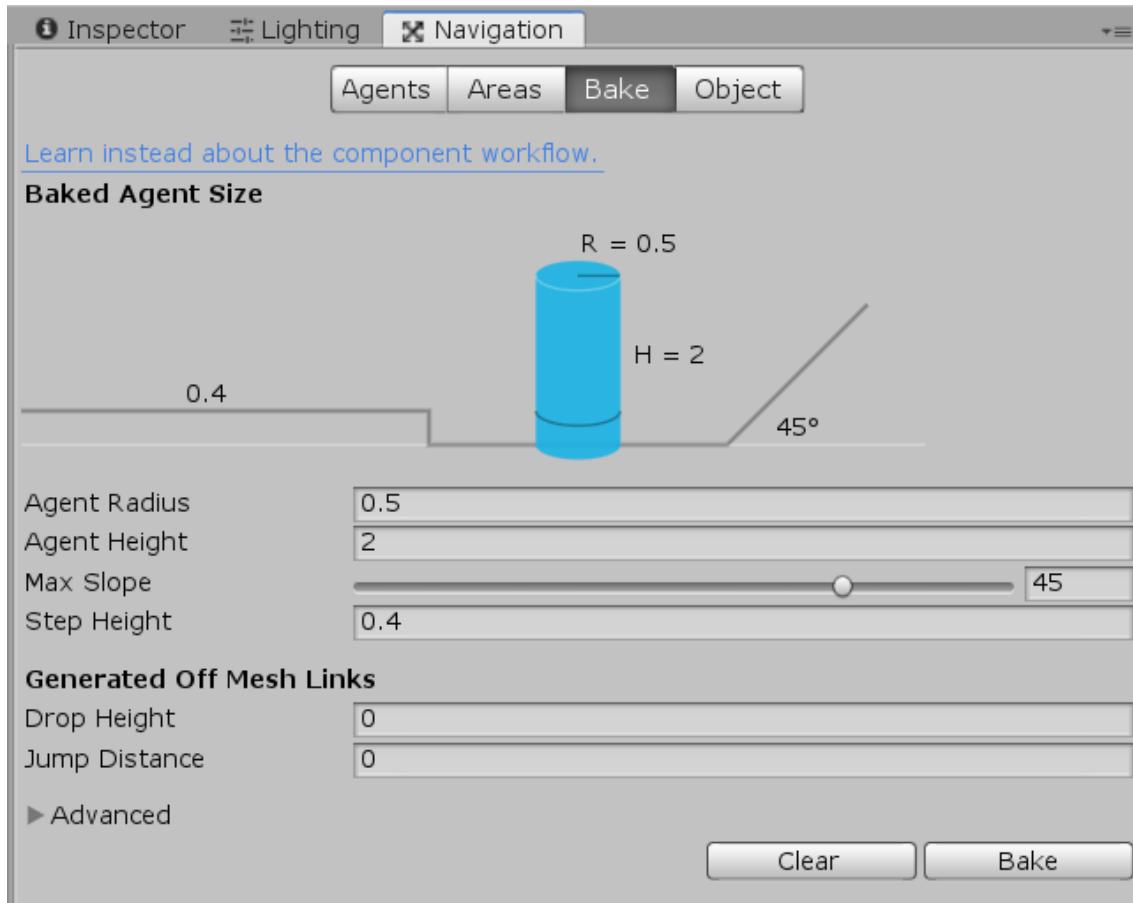
Para crear tu Navmesh:

1. En la barra de menú ve a Window > AI > Navigation (Ventana > Inteligencia artificial > Navegación) para abrir la ventana Navigation. La ventana debería anclarse a la ventana Inspector. Si no lo hace, arrástrala y ánclala allí.
2. Hay cuatro pestañas en la parte superior de la ventana Navigation: **Agents** (agentes), **Areas** (áreas), **Bake** (crear un NavMesh) y **Object** (objeto).



Selecciona la pestaña **Bake**.

3. La configuración de Bake controla los detalles de cómo se construirá la malla de navegación o Navmesh. Las primeras configuraciones se refieren a los agentes que van a atravesar la malla (los fantasmas que deambularán) o los **Navmesh Agents**. Se refieren específicamente al tamaño de los agentes y el terreno en el que pueden moverse.



La única configuración que necesitas ajustar para tu juego es el **Agent Radius** (Radio del Agente). Los fantasmas que están deambulando dentro de la casa embrujada serán más pequeños que los por defecto.

Cambia el **Agent Radius** a 0,25.

4. Selecciona el botón **Bake** en la parte inferior derecha de la ventana.

El proceso de crear un Navmesh o Baking puede tomar de unos segundos a unos cuantos minutos, dependiendo del poder de procesamiento de tu computadora. Cuando termina, el entorno en la ventana Scene estará cubierto por una **malla celeste**. Esta es el área del entorno en la que los fantasmas podrán deambular.



La malla de navegación o Navmesh solo será visible cuando la ventana Navigation está abierta y activa (si cambias a la pestaña Inspector, la malla desaparecerá de la vista Scene). No te preocunes si no puedes verlo. Todavía está allí.

5. Recuerda que debes guardar los cambios que has hecho a tu escena para que no pierdas lo que has hecho hasta ahora.

7. Resumen

En este tutorial añadiste un entorno muy interesante del que JohnLemon pueda escaparse. También lo has iluminado de manera apropiada. También creaste un NavMesh; de esa manera, los enemigos serán capaces de deambular por el entorno.

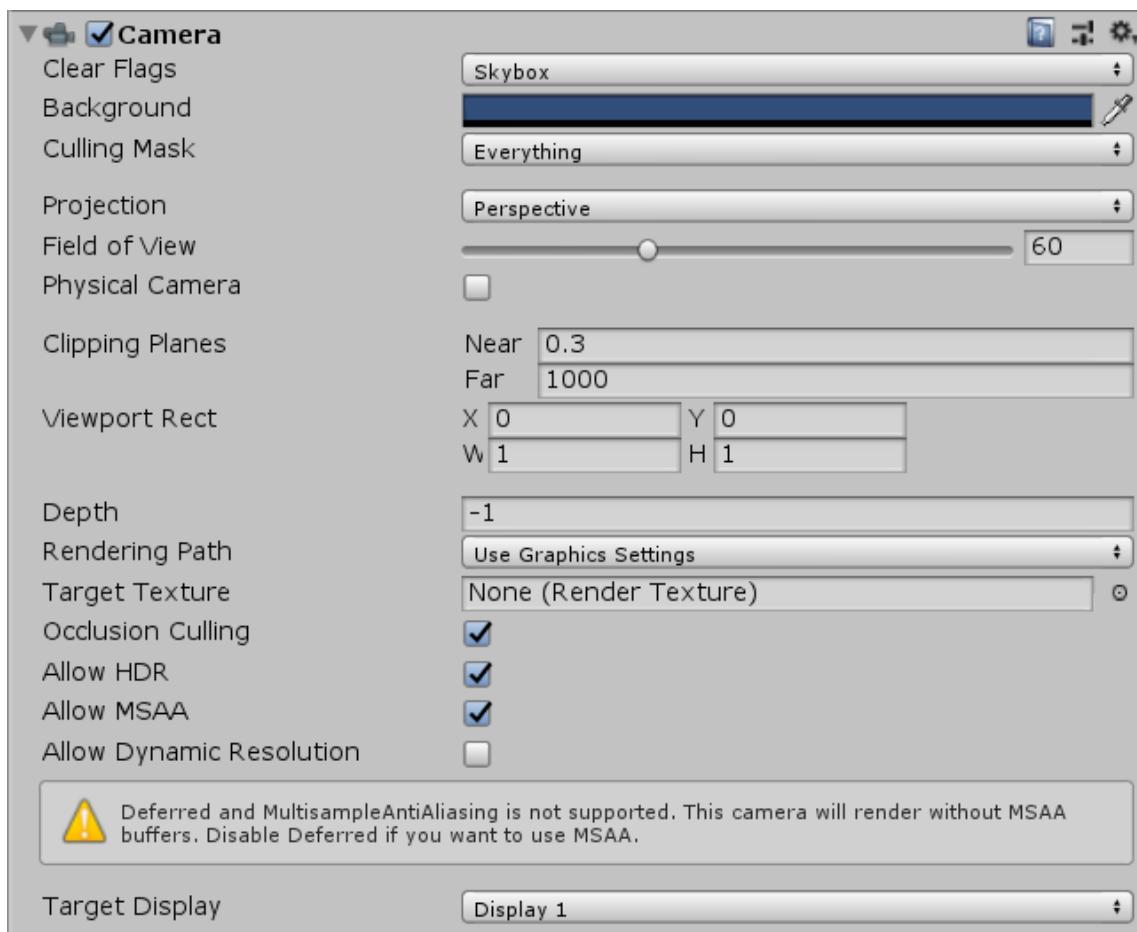
¡Tu juego está empezando a tomar forma y a verse muy bien! Es hora de hacer que la cámara se mueva y de agregar efectos de posprocesamiento que harán que todo se vea mucho mejor en tu escena.

La cámara

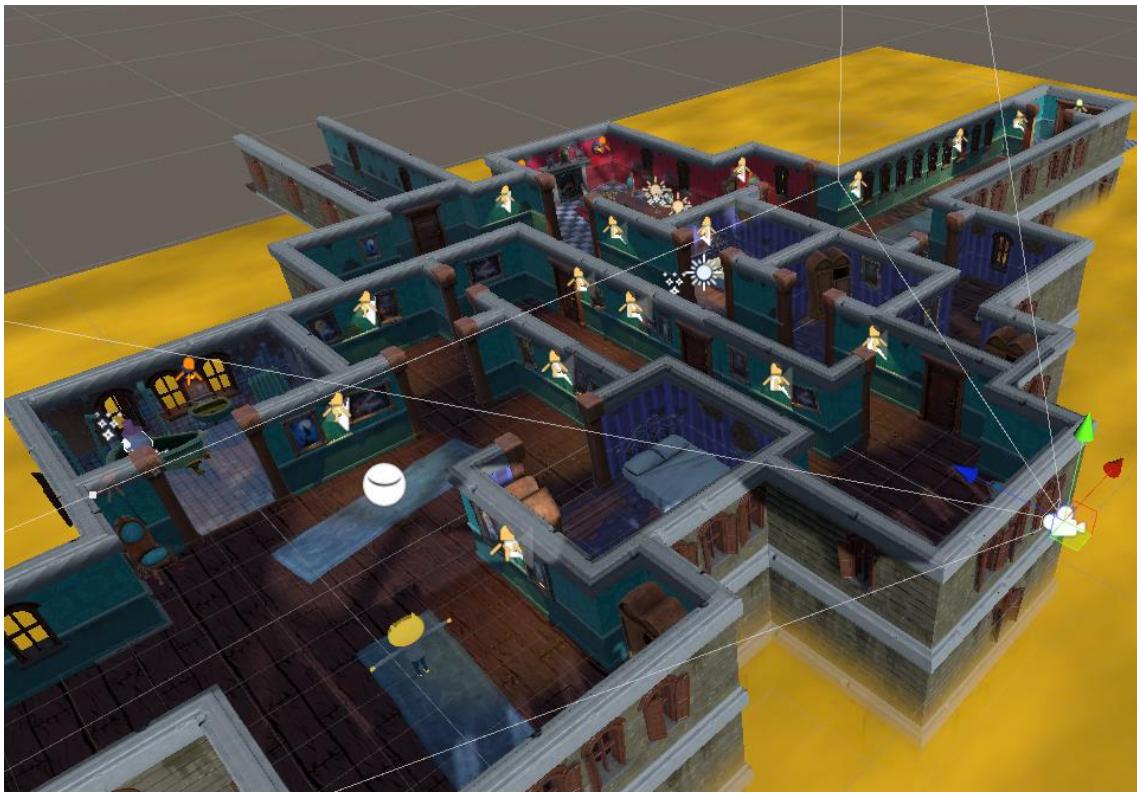
1. Explore el componente de la cámara

En el tutorial anterior, creaste un nivel para tu juego. El personaje ahora puede caminar por la escena, pero esto significa que a veces se aleja de la cámara. En este tutorial, ajustará la configuración de la cámara para asegurarse de que esto no suceda.

Las escenas en Unity están compuestas por GameObjects que a su vez tienen una colección de componentes adjuntos. La forma en que los jugadores ven la escena se construye de la misma manera. Para ver la escena, un GameObject en la escena debe tener un **componente de cámara**. Cuando se crea una nueva escena, se agrega un GameObject llamado Cámara principal que tiene un componente de cámara.



La cámara apunta hacia abajo en el eje **z** de GameObject y se comporta exactamente como todos los demás GameObjects. En la vista de escena, se puede ver un aparato que representa la cámara de **cono truncado**. El frustum es una forma sólida que se parece a una pirámide con la parte superior cortada paralela a la base. Esta es la forma de la región que puede verse y representarse con una cámara de perspectiva.



Cuando haces un juego, tienes algunas opciones para asegurarte de que la cámara siga al personaje del jugador. Una solución sería escribir un script para esto. Sin embargo, Unity tiene una solución integrada al problema: **Cinemachine**.

2. ¿Cómo funciona Cinemachine?

Cinemachine es la respuesta de Unity a todas las cosas relacionadas con las cámaras en los juegos. Un resumen básico de este sistema es el siguiente:

- Se crean una o más cámaras 'virtuales' en una escena.
- Estas cámaras virtuales son administradas por un componente llamado Cinemachine Brain.
- Cinemachine Brain está conectado al mismo GameObject que un componente de la cámara; de forma predeterminada, será el GameObject de la cámara principal.
- Cinemachine Brain gestiona todas las cámaras virtuales y decide qué cámara virtual (o combinación de cámaras virtuales) debe seguir la cámara real.

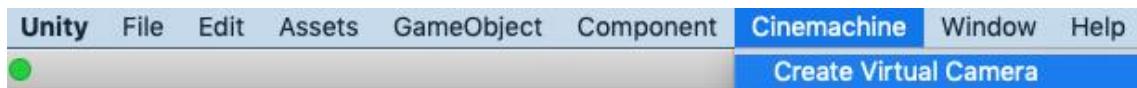
En tu juego, la cámara solo seguirá a JohnLemon, por lo que solo necesitarás una cámara virtual. Te asegurarás de que esta cámara virtual siga a JohnLemon, y luego el GameObject de la cámara principal se apegará a eso.

3. Configure una cámara virtual con Cinemachine

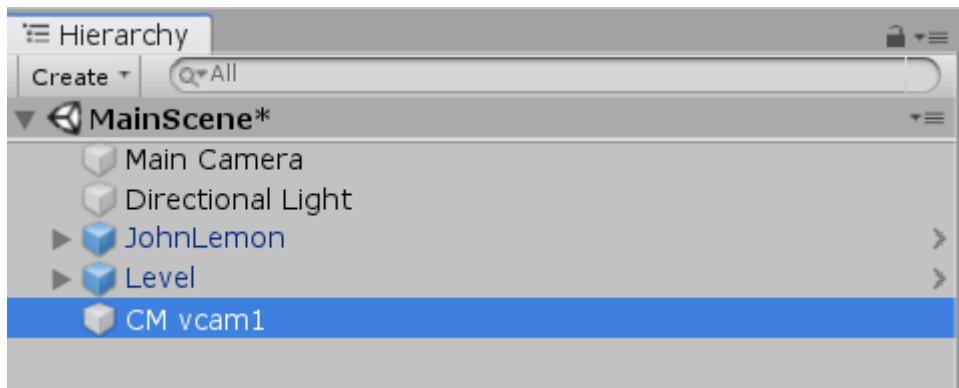
Para configurar una cámara virtual con Cinemachine:

En la ventana Proyecto, vaya a **Activos> Escenas** y haga doble clic en **MainScene**.

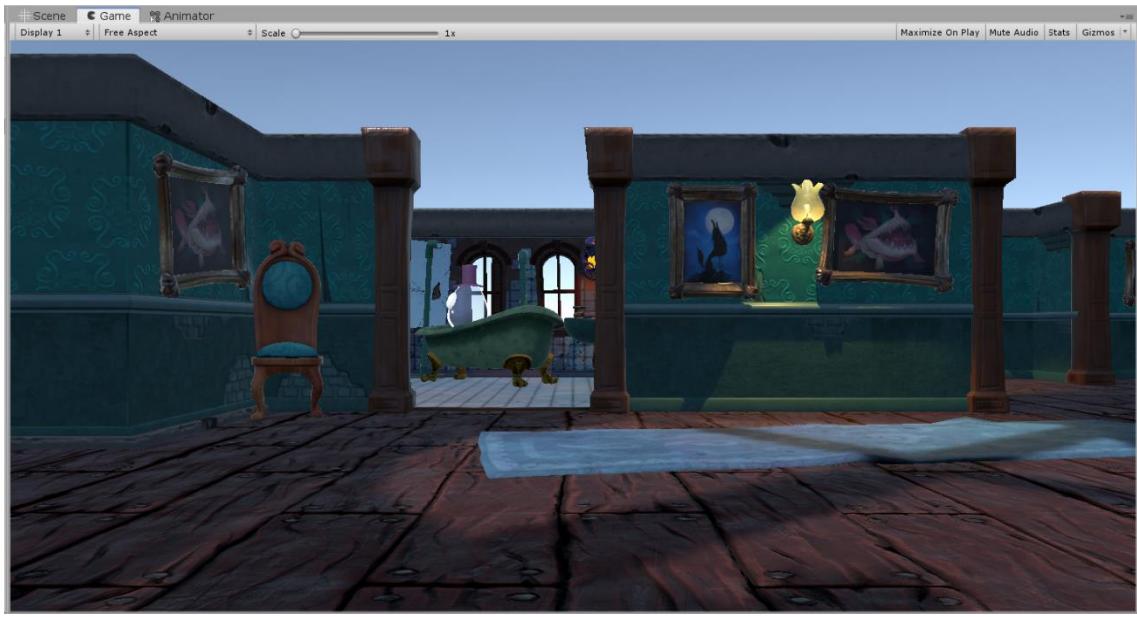
1. En la Jerarquía, seleccione el JohnLemon GameObject.
2. Mueva el cursor sobre la ventana Escena y presione la tecla **F**.
3. En el menú superior, vaya a **Cinemachine> Crear cámara virtual**.



Esto creará un nuevo GameObject en la escena llamado **CM vcam1**:



Esto se creará en el punto de enfoque de la cámara de la ventana de escena, por lo que debería estar justo encima de JohnLemon. Sin embargo, dado que ahora tiene una cámara virtual en la escena, habrá movido la vista de la ventana del juego, por lo que ya no podrá ver a JohnLemon en la ventana del juego.



Para solucionar esto, debe asegurarse de que la cámara virtual rastrea al personaje cambiando la configuración del componente de la cámara virtual Cinemachine.

4. Cambie la configuración del componente de la cámara virtual Cinemachine

Para configurar la cámara virtual para que rastree a JohnLemon:

1. En la Jerarquía, seleccione **CM vcam1** .
2. En el Inspector, busque el componente Cinemachine Virtual Camera. Este componente tiene muchas configuraciones, pero por el momento solo necesita enfocarse en tres secciones: las referencias del objetivo, el cuerpo y el objetivo.

Cinemachine Virtual Camera (Script)

Status: Live Solo

Game Window Guides

Save During Play

Priority 10

Follow None (Transform)

Look At None (Transform)

Standby Update Round Robin

Lens

Field Of View 40

Near Clip Plane 0.1

Far Clip Plane 5000

Dutch 0

Transitions

Body Transposer

⚠ Transposer requires a Follow Target. Change Body to Do Nothing if you don't want a Follow target.

Binding Mode Lock To Target With World Up

Follow Offset X 0 Y 0 Z -10

X Damping 1

Y Damping 1

Z Damping 1

Yaw Damping 0

Aim Composer

⚠ A LookAt target is required. Change Aim to Do Nothing if you don't want a LookAt target.

Tracked Object Offset X 0 Y 0 Z 0

Lookahead Time 0

Lookahead Smoothing 10

Lookahead Ignore Y

Horizontal Damping 0.5

Vertical Damping 0.5

Screen X 0.5

Screen Y 0.5

Dead Zone Width 0

Dead Zone Height 0

Soft Zone Width 0.8

Soft Zone Height 0.8

Bias X 0

Bias Y 0

Noise none

Extensions

Add Extension (select)

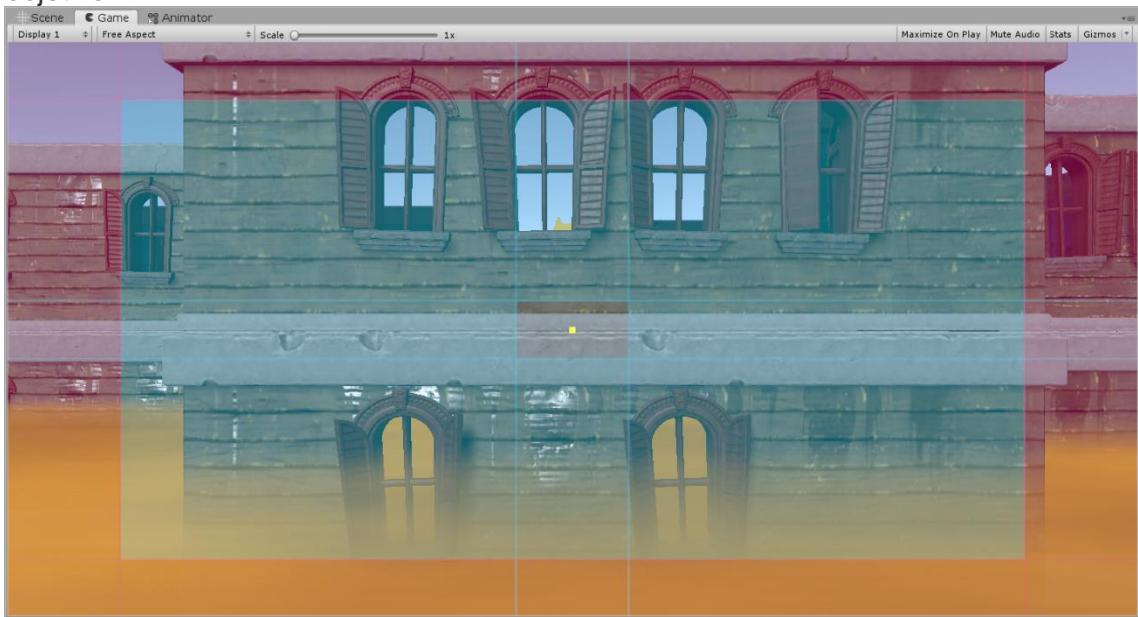
La sección de referencias de destino tiene dos configuraciones: **Seguir** y **Ver**. Estas son referencias opcionales a los componentes Transformar de GameObjects.

Si desea que su cámara virtual se mueva, entonces necesita saber *cómo* moverse. Específicamente, necesita una referencia a una Transformación que seguirá. Del mismo modo, si desea que su cámara virtual gire, entonces necesita saber qué tiene que girar para mirar.

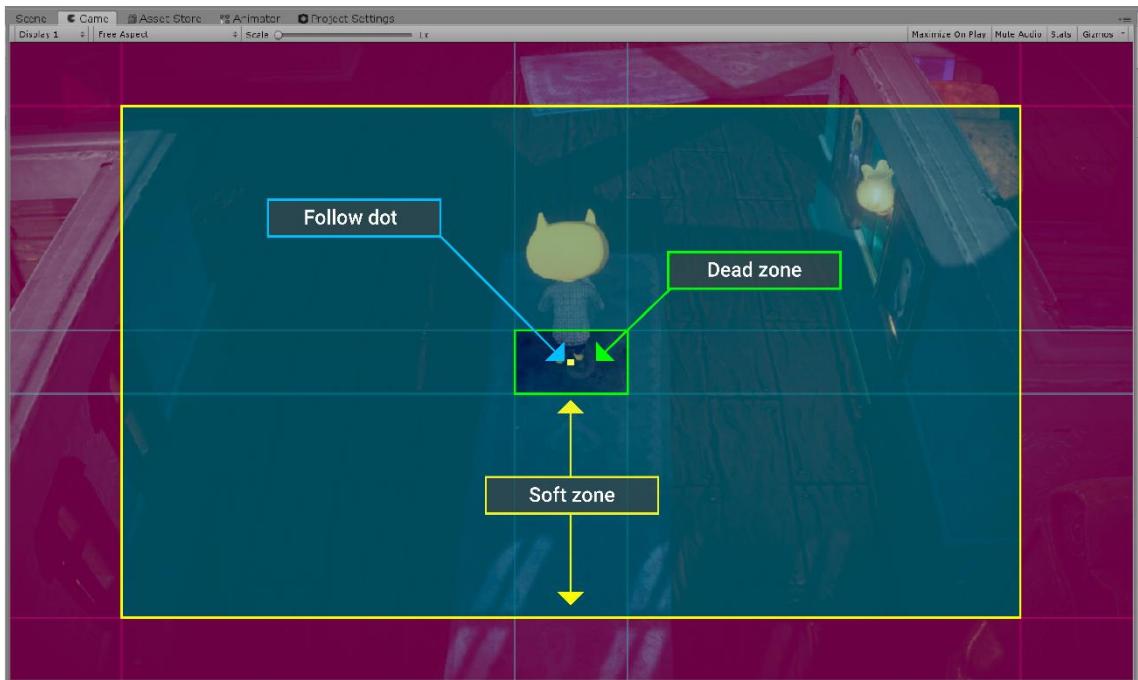
La siguiente sección dos secciones son el **cuerpo** y el **objetivo**. Estos son ajustes de cómo se mueve y gira la cámara, respectivamente. Va a restringir el movimiento de la cámara virtual, por lo que tiene a JohnLemon a la vista todo el tiempo, pero en realidad no mira al personaje.

3. En la sección **Objetivo (Aim)**, cambie el menú desplegable en la parte superior derecha de **Composer** a **No hacer nada**.
4. Arrastre y suelte el objeto de juego JohnLemon desde la ventana Jerarquía en la propiedad Seguir del componente de cámara virtual Cinemachine. Esto cambiará la configuración **Seguir** para hacer referencia a la transformación de JohnLemon.
5. En la sección **Cuerpo (Body)**, cambie el menú desplegable en la parte superior derecha de la sección de **Transposer** a **Framing Transposer**. Cambiar el cuerpo a un transpositor de encuadre le permitirá controlar la posición de la cámara virtual al darle reglas sobre dónde debe estar su objetivo **Seguir** en la pantalla. Para obtener más información sobre los diferentes valores de configuración del cuerpo, puede consultar la documentación de Cinemachine (en la barra de menú superior, vaya a **Cinemachine> Acerca de**). La ventana del juego (Game) ahora debería tener varios cuadros rojos y azules. Estas son guías sobre dónde en la pantalla puede ser el

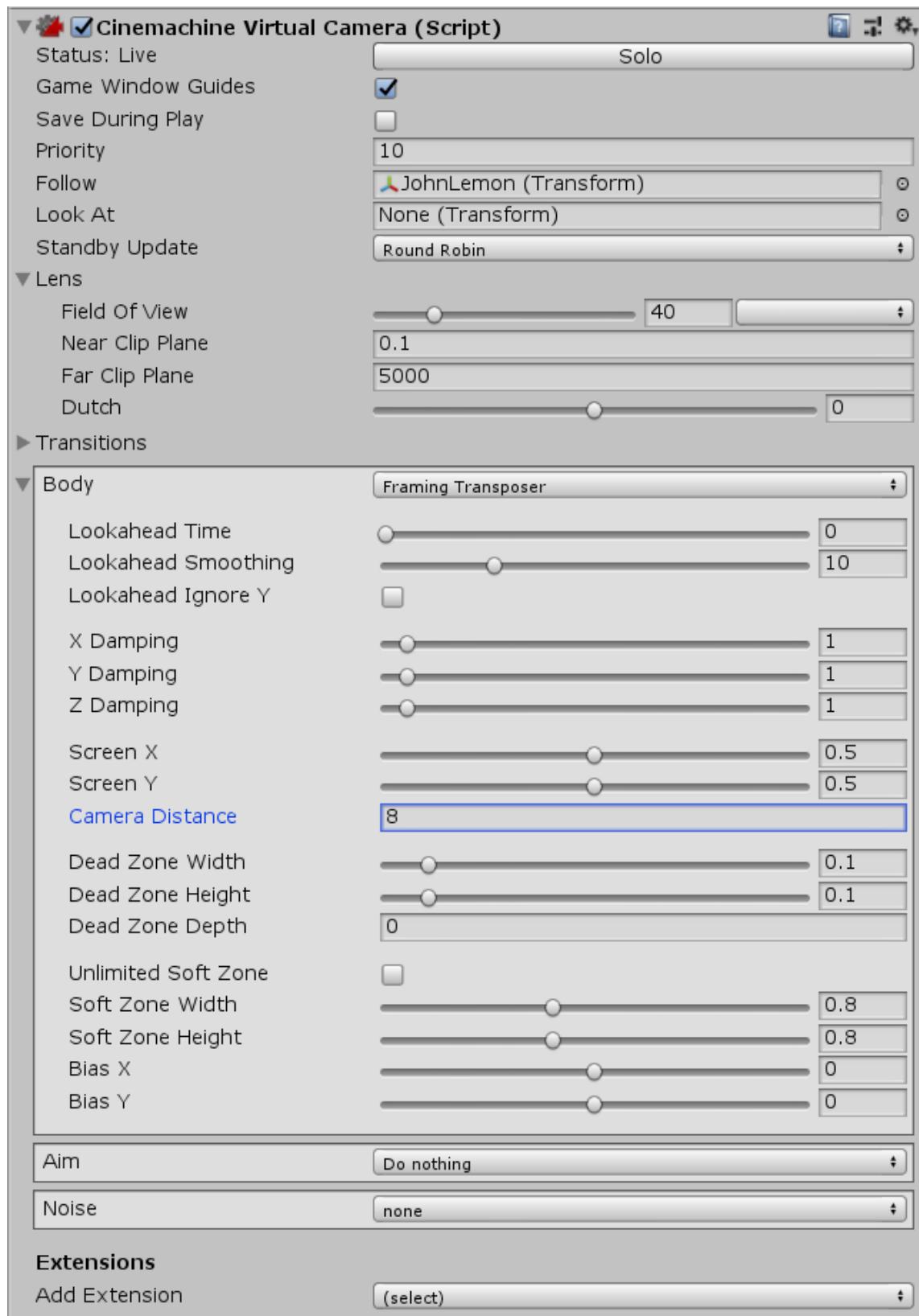
objetivo:



6. Ahora configuremos la cámara virtual en el ángulo correcto. En la Jerarquía, seleccione **CM vam1** GameObject. En el componente Transformar CM vam1 GameObject, establezca la **Rotación** alrededor del eje x en **45**. Ahora la cámara virtual está inclinada hacia abajo y mirando al personaje desde arriba. Este es más o menos el aspecto que estás buscando. ¿Por qué se movió la cámara virtual cuando solo cambiaste la rotación? Este es el poder de Cinemachine! Sabías que necesitabas un ángulo más vertical para nuestra cámara, y la cámara virtual resolvió que si querías que se inclinara hacia abajo, tenía que estar más arriba para tener su objetivo en la pantalla.



7. La mayoría de las configuraciones predeterminadas de Framing Transposer están bien para tu juego. Lo único que necesita cambiar es la configuración de **Distancia de la cámara** : el personaje es demasiado pequeño en la pantalla, por lo que debe mover la cámara virtual un poco más cerca. Cambie la configuración de **Distancia de cámara de 10 a 8** .



8. En la Jerarquía, seleccione **CM vcam1** y cámbiele el nombre a **VirtualCamera**.

9. Anule la selección de VirtualCamera GameObject para eliminar las pautas de la ventana Juego.

10. Guarde la escena yendo a **Archivo> Guardar escena** desde el menú superior, o presionando Ctrl + S (Windows) o CMD + S (macOS).

11. Haz clic en el botón Jugar en la barra de herramientas y prueba el juego hasta ahora. Haga clic en Reproducir nuevamente cuando haya terminado de probar.

Eso es todo: ¡ha configurado la cámara virtual y su movimiento! Ahora puedes comenzar a agregar un toque visual a la cámara para que el juego se vea aún más increíble.

5. Agregar efectos de posprocesamiento

El procesamiento posterior implica la aplicación de filtros y efectos a la imagen del juego antes de que se muestre en la pantalla (similar a los filtros que se pueden agregar a las fotos). En esta sección del tutorial, explorará cómo funciona el procesamiento posterior en Unity y luego lo aplicará a su propio juego.

[¿Cómo funciona el postprocesamiento en Unity?](#)

En general, los efectos de procesamiento posterior se agrupan y se utilizan en diferentes áreas del mundo del juego. Esto significa que cuando la cámara está en un área particular, su conjunto designado de procesos se aplica a la imagen.

Imagine que está en una habitación oscura y luego salga por una puerta a la luz del día. Tus ojos tardarán un tiempo en adaptarse y todo parecerá mucho más brillante de lo que suele ser cuando salgas de la habitación. En Unity, esto podría simularse aplicando diferentes grupos de postprocesos en la sala y fuera de ella. Los grupos de procesos posteriores son activos denominados **perfíles de procesamiento posterior**. Las áreas del mundo del juego que tienen perfiles asignados son Componentes llamados **Volumenes de Post Proceso**.

A veces, diferentes cámaras tienen diferentes postprocesos. Para vincular un conjunto de procesos a una cámara específica, se agrega un componente **Post Process Layer** al mismo GameObject que el componente Camera. Las capas son una forma de organizar GameObjects por comportamiento.

[Obtenga más información sobre el postprocesamiento en Unity](#)

Este tutorial incluye instrucciones paso a paso para agregar efectos de procesamiento posterior específicos en Unity Editor. Si desea explorar estos efectos con más detalle, puede encontrar más información en:

- [Descripción general del postprocesamiento del Manual de Unity](#)
- [La documentación del paquete de posprocesamiento](#)

6. Crear una capa de posprocesamiento

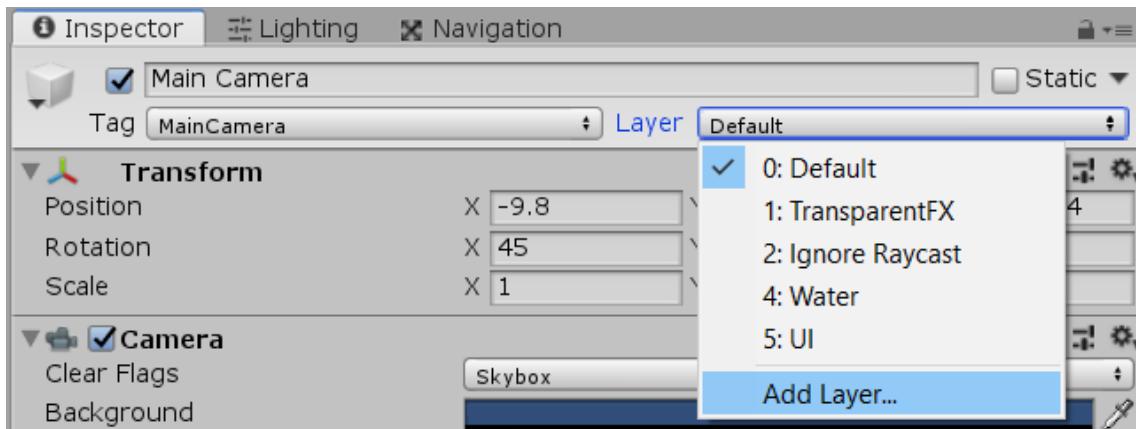
Para crear una capa de posprocesamiento:

1. En la Jerarquía, seleccione el Objeto de juego de la **cámara** principal.
2. En el Inspector, busque la propiedad **Layer** debajo del nombre del GameObject.



Un GameObject solo puede pertenecer a una capa. La cámara principal está en la capa predeterminada. Hay varias opciones por defecto en Unity, pero también puede crear sus propias capas en el Administrador de etiquetas y capas.

3. En el menú desplegable de la propiedad Capa, seleccione **Agregar capa ...**

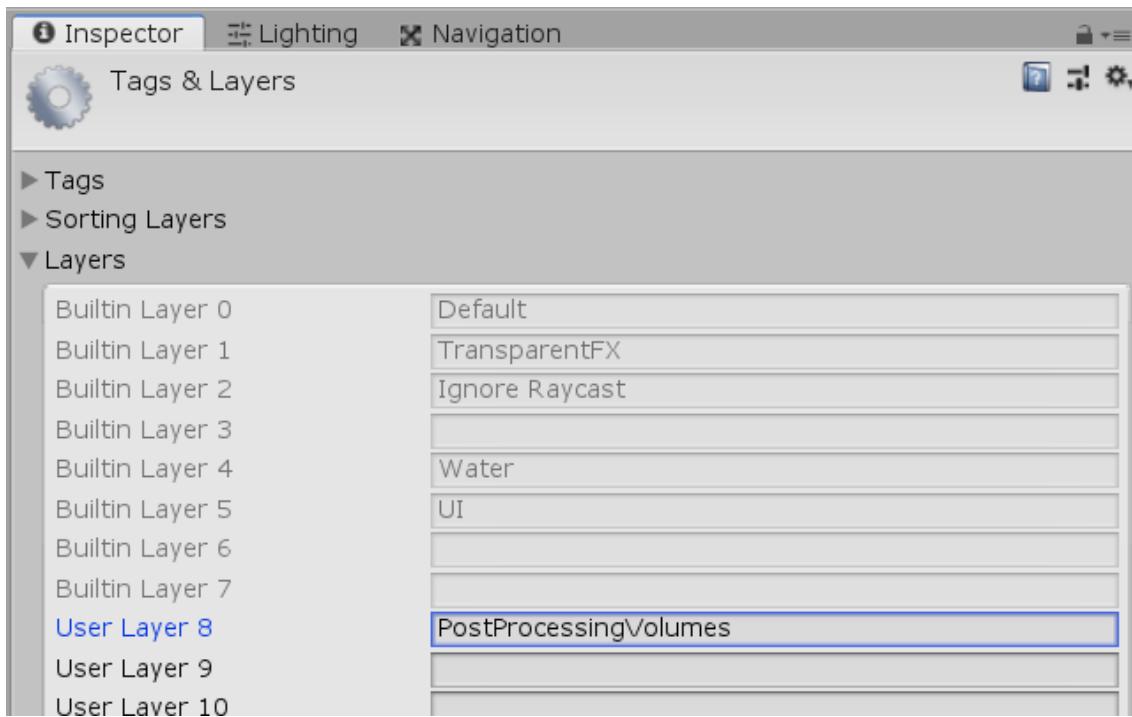


4. En el Administrador de etiquetas y capas hay 32 capas disponibles en (comenzando con 0 y terminando con 31).



Las primeras ocho capas están reservadas para las capas predeterminadas de Unity, pero las 24 restantes se pueden usar para lo que deseé. Como necesita una Capa para su Volumen de Post Proceso, tiene sentido crear una Capa para todos los Volúmenes de Post Proceso.

5. Agregue " PostProcessingVolumes " como User Layer 8.

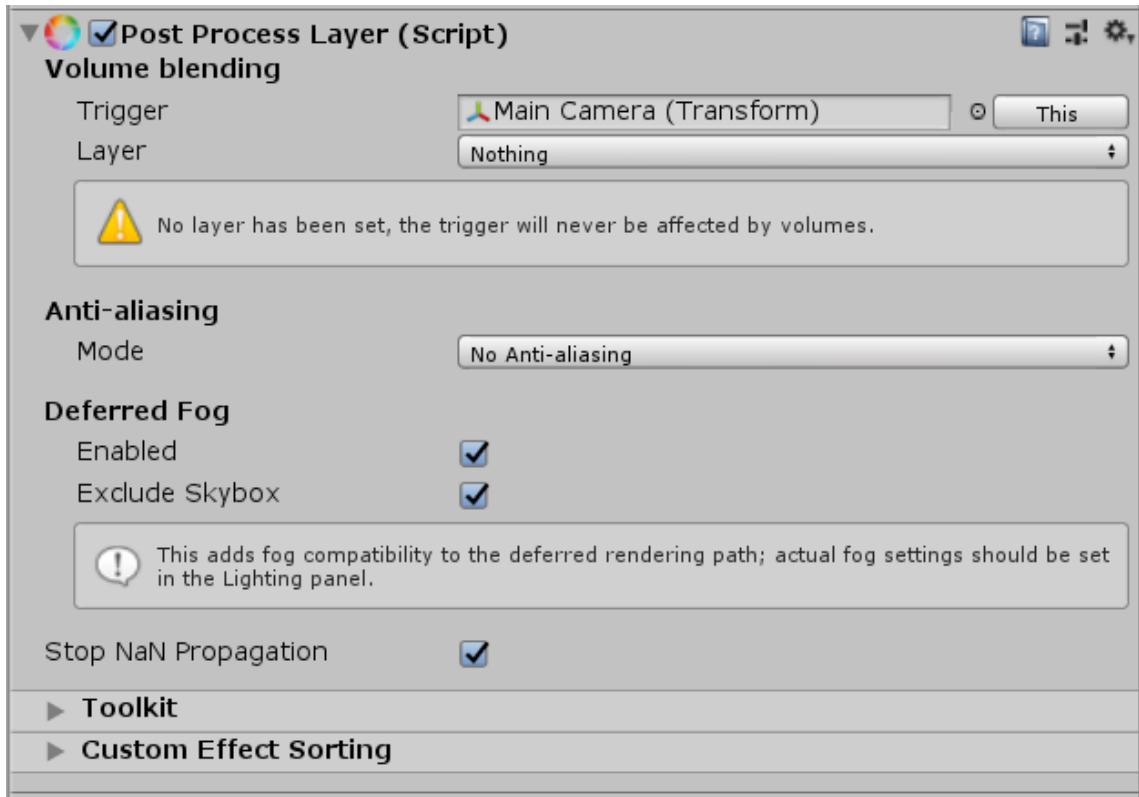


6. En la Jerarquía, seleccione el Objeto de juego de la cámara principal. Seguirá estando en la capa predeterminada:



Esto se debe a que, aunque haya creado una nueva capa, no ha reasignado la Cámara principal a esta.

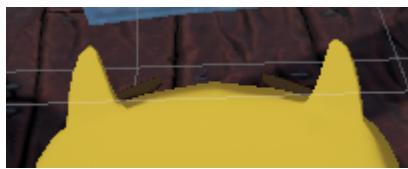
7. Agregue un componente de **Post Process Layer** a la cámara principal.



- Cambie la propiedad Layer de Nothing a **PostProcessingVolumes** usando el menú desplegable.

7. Mejora la calidad de imagen con Anti-aliasing

Antes de continuar para configurar su volumen y perfil, consideremos rápidamente el suavizado. El alias es cuando el borde de un objeto se ve irregular y se puede ver el contorno del píxel:



El suavizado es un efecto de procesamiento posterior que reduce la prominencia de estas líneas irregulares al rodearlas con píxeles de tonos intermedios de color. Existen varios algoritmos diferentes para determinar qué píxeles deben ajustarse y en qué medida: utilizará el más simple y efectivo.

- Asegúrese de que la vista Juego sea visible, para que pueda ver los cambios que está realizando en esta sección del tutorial.

2. En el componente **Post Process Layer**, cambie el menú desplegable de propiedades **Modo** de Sin suavizado a **Suavizado aproximado rápido (FXAA)**.
3. Active la casilla de verificación (**Fast Mode**)**Modo rápido** debajo del menú desplegable. ¡JohnLemon ya se ve mucho mejor!

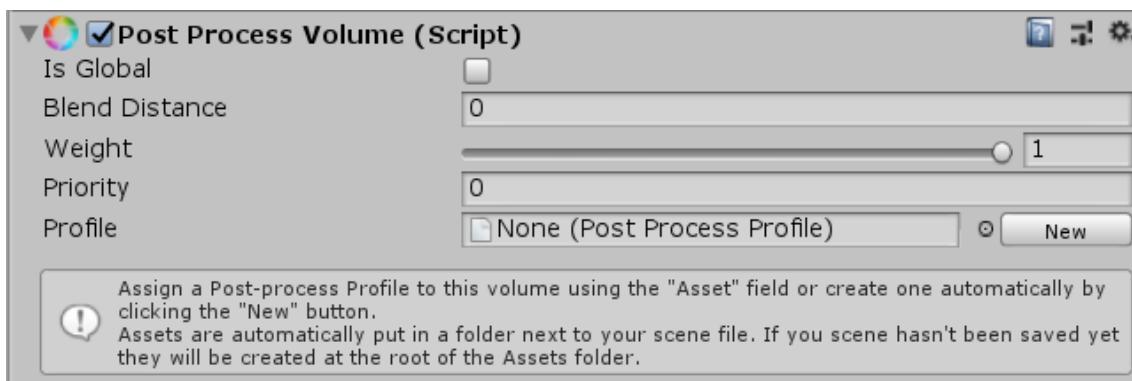


Ahora la capa de proceso posterior está configurada correctamente, puede pasar a crear su volumen.

8. Crear (Post-Processing Volume) volumen de postprocesamiento

Para crear un volumen de procesamiento posterior:

1. En la ventana Jerarquía, go to **Create > Create Empty..**
2. Nombre el nuevo GameObject "**GlobalPost**".
3. En el Inspector, establezca la Capa en **PostProcessingVolumes** .
4. Cambie la propiedad **Posición** del componente Transformar a **(0 , 0 , 0)**. Este GameObject será su **Volumen global de postproceso** . El volumen global funciona como un perfil predeterminado para las cámaras que afecta. Otros volúmenes no globales podrían anular los efectos de este, pero su juego no necesitará ningún volumen adicional.
5. Agregue un componente de volumen de proceso posterior a GlobalPost.



6. Active la casilla de verificación **Is Global** checkbox.

7. Haga clic en el botón **Nuevo** en el extremo derecho de la propiedad **Perfil(Profile)** .

Esto creará un nuevo perfil de postprocesamiento. Su activo se guardará en una nueva carpeta llamada **MainScene_Profiles** en **Assets> Scenes** .

8. Ahora que se ha establecido un Perfil, verá más configuraciones disponibles en el Componente de Volumen del Proceso Posterior. La subsección **Invalidaciones** se refiere a los diferentes efectos que pueden anular la representación normal de la cámara. Por defecto, un perfil no tiene efectos, por lo que no hay nada en la subsección Invalidaciones.

9. Agregue el efecto de gradación de color (Color Grading Effect)

Para agregar el efecto de Calificación de color:

1. Haga clic en el botón **Agregar efecto ...** y seleccione **Unidad> Clasificación de color** . La gradación de color es una forma de cambiar el brillo, el contraste y los colores de la vista renderizada. Puede pensar en ello como aplicar filtros en software de edición de fotos o Instagram. Para que tu juego se vea más espeluznante, vas a ajustar los colores y los niveles de brillo de la vista renderizada.

Nota: La configuración dentro de este componente son accesos directos al activo de perfil; todos los cambios en la subsección Invalidaciones cambiarán el activo del perfil y no afectarán directamente al componente.

2. Seleccione la flecha junto a **Calificación de color** para expandir su configuración.

Color Grading

All None

Mode High Definition Range

Tonemapping
 Mode None

White Balance
 Temperature 0
 Tint 0

Tone
 Post-exposure (EV) 0
 Color Filter HDR
 Hue Shift 0
 Saturation 0
 Contrast 0

Channel Mixer
 Red Red 100
 Green Green 0
 Blue Blue 0

Trackballs

0.00 0.00 0.00 1.00 1.00 1.00 1.00 1.00 1.00

Lift Gamma Gain

Grading Curves

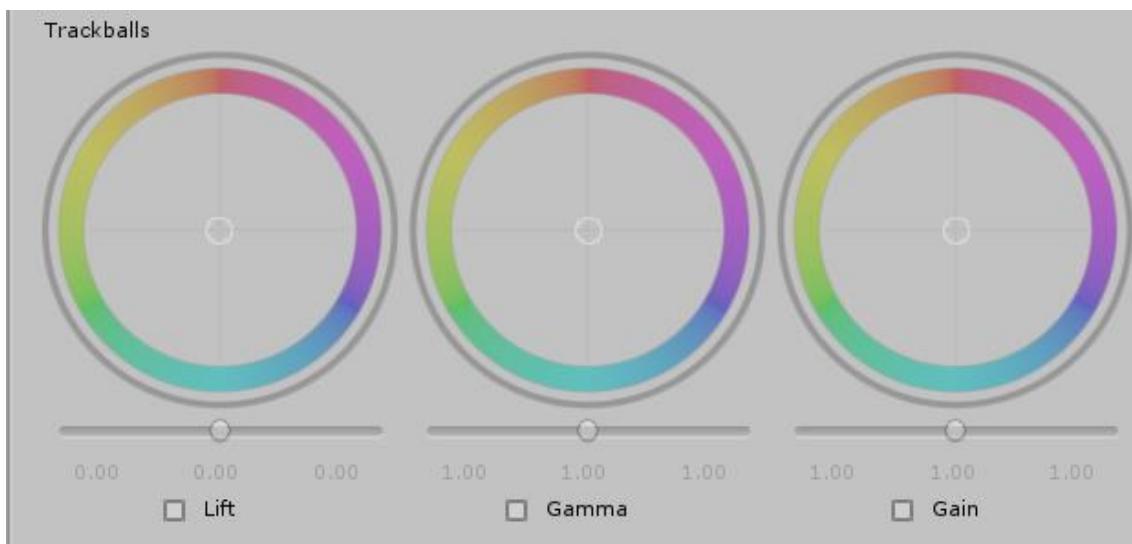
Hue Vs Hue

(Not Overriding)

3. Primero, necesitas **Tonemap** la escena. La asignación de tonos cambia la salida del cuadro final renderizado para que sea mucho más fácil controlar las luces y las sombras. Active la casilla de verificación **Modo** en la subsección **Tonemapping**. Usando el menú desplegable, cambie su propiedad de Ninguno a **ACES**. El mapeo de tonos ACES es una forma estándar de la industria de aplicar el mapeo de tonos. Ayuda a crear un aspecto más cinematográfico para sus proyectos.

4. Notarás que la escena se ve realmente oscura ahora. Esto se debe a que la exposición predeterminada de ACES es cero (para obtener más información, consulte la documentación). Una buena manera de solucionar esto es usar la función de Post-exposición. La exposición posterior es una forma de ajustar la exposición predeterminada. En la subsección **Tono(Tone)**, active la casilla de verificación **Post-exposición (EV)**. Establezca el valor en **1** para alegrar su escena.

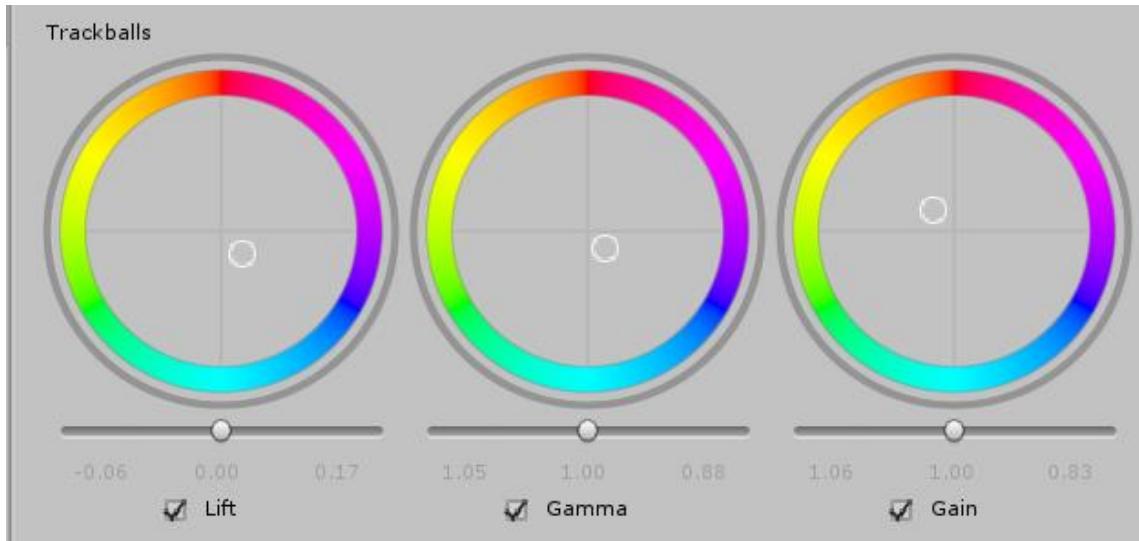
5. También puede hacer que su escena sea un poco más espeluznante cambiando algunos colores. A la mitad de la sección de Calificación de color, encuentre los círculos de color llamados **Trackballs**.



6. Active las casillas **Lift**, **Gamma** and **Gain** checkboxes beneath each Trackball. La elevación afecta los colores de sus sombras, la ganancia cambia los reflejos más brillantes y Gamma cubre todo en el medio (o rango medio) del color de su imagen.

7. Arrastre el círculo en el centro de las Trackballs para ajustar ligeramente el color en su vista de Juego. Arrastrar el:

- **Lift and Gamma Trackballs** ligeramente hacia el azul
- **Gain Trackball** ligeramente hacia el amarillo



Esto agregará profundidad a las sombras y calidez a la iluminación.

8. Contraiga el efecto de Calificación de color.

10. Añade el efecto Bloom

Sus luces son mucho más efectivas, pero todavía no se parecen a las luces reales.

Agregue el efecto Bloom para darles un brillo agradable:

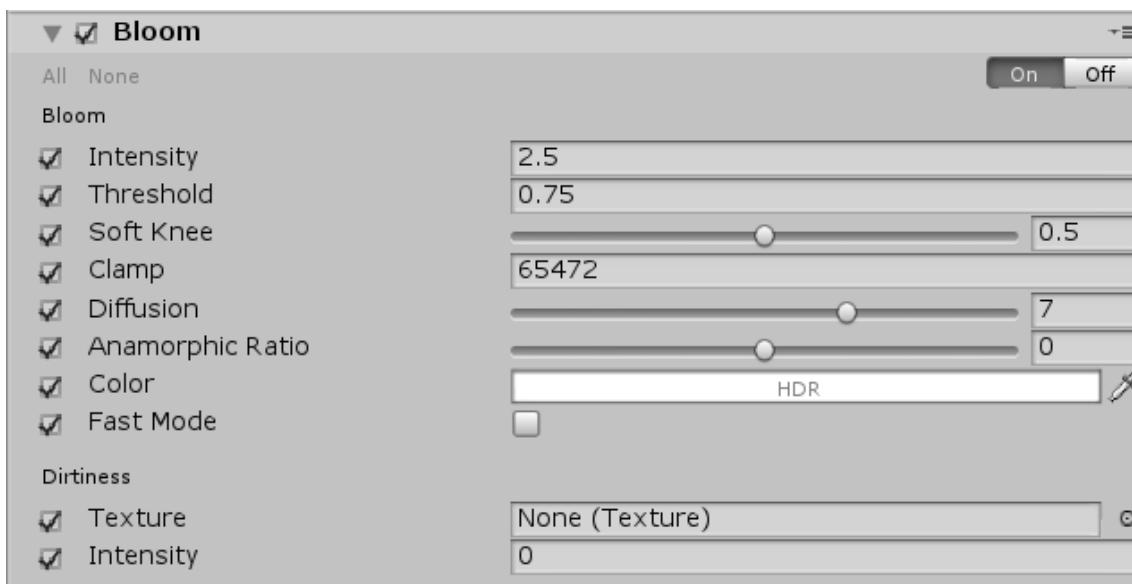
1. Haga clic en el botón **Agregar efecto ...** y seleccione **Unidad> Bloom**. Expande la sección de efecto **Bloom**.



2. Seleccione el acceso directo **Todos** justo debajo del título de la sección: esto habilitará todas las configuraciones para Bloom. Actualmente, nada es realmente brillante, ya que la **intensidad** no es lo suficientemente alta.

3. Establezca la propiedad **Intensidad** en **2.5**, para crear un agradable brillo en las luces.

4. Es posible que vea otros objetos más brillantes que siente que también podrían verse bien con un poco de brillo. Puede agregarlos al efecto Bloom utilizando la propiedad **Umbral (Thershold)**. Bajar este valor agrega menos píxeles brillantes al efecto. Establezca el Umbral en **0.75** para agregar más brillo a la escena.

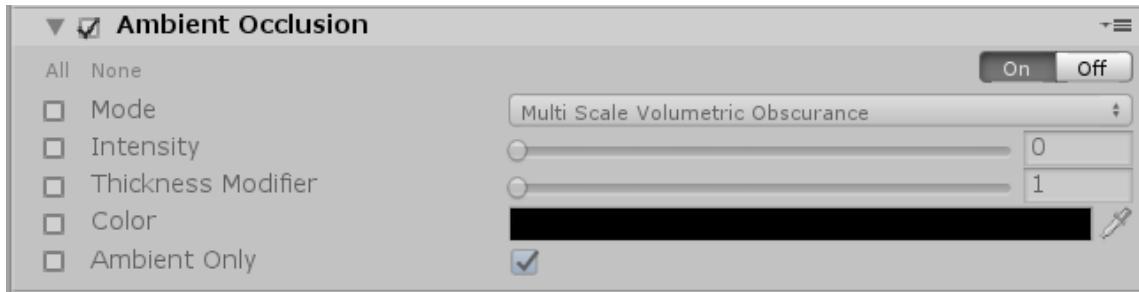


5. Contraer el efecto Bloom.

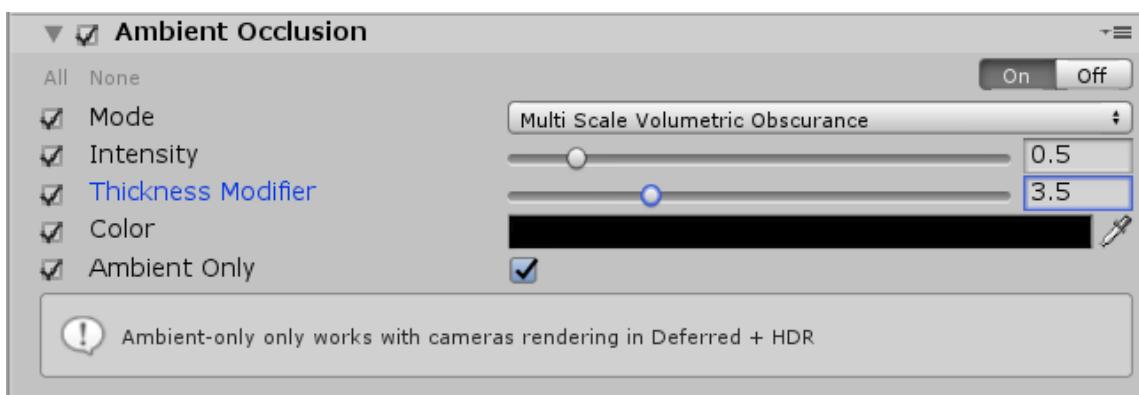
11. Agregue el efecto de ambient occlusion

Las luces ahora se sienten brillantes, pero es importante mantener la atmósfera espeluznante de esta casa embrujada. Una buena forma de oscurecer algunas de esas esquinas y grietas es usar el efecto de **occlusión ambiental**. La oclusión ambiental imita el efecto de la vida real de que la luz no llega a las esquinas más oscuras oscureciendo esas áreas. Para agregar este efecto:

1. Haga clic en el botón **Agregar efecto ...** y seleccione **Unidad> Oclusión ambiental**.
. Expanda la sección Efecto de **occlusión ambiental**.



2. Haga clic en el acceso directo de Ambient Occlusion's **All** para activar todas las funciones que necesita.
3. Al igual que Bloom, deberá establecer la Intensidad antes de que podamos verla funcionar. Establezca la propiedad Intensidad en **0.5** y creará una agradable sombra espeluznante en las esquinas de las habitaciones.
4. Puede extender este efecto aún más a través de la habitación utilizando el **Modificador de espesor (Thickness Modifier)**. Establezca la propiedad Modificador de espesor en **3.5**.



Ahora deberías poder ver un gran efecto sombrío que se extiende detrás de algunos de los otros accesorios de la escena.

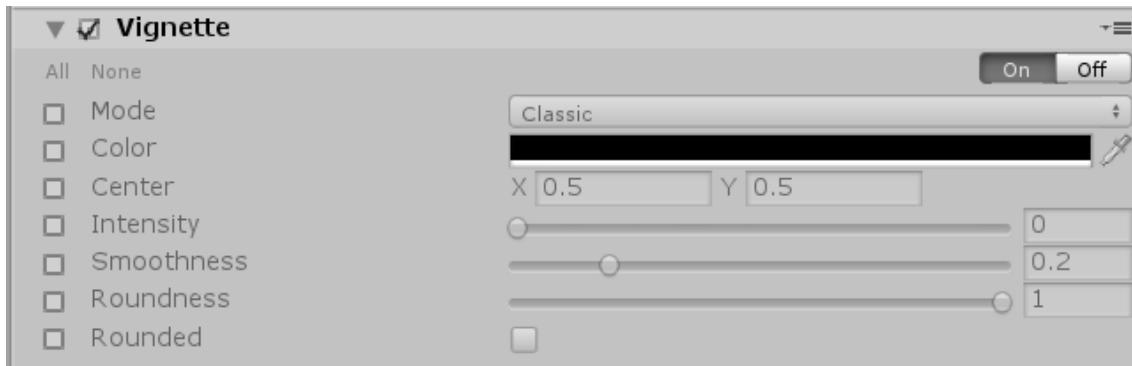
5. Contraer el efecto de oclusión ambiental.

12. Agregue el efecto de viñeta (Vignete)

Varios efectos simulan cómo una cámara real vería la escena. Los últimos efectos cambiarán la forma en que funcionaría la lente de dicha cámara. Primero, agregará el efecto Viñeta para oscurecer los bordes de la lente de la cámara. Esto ayuda a

centrarse en el jugador y hace que el juego se sienta más claustrofóbico. Para agregar este efecto:

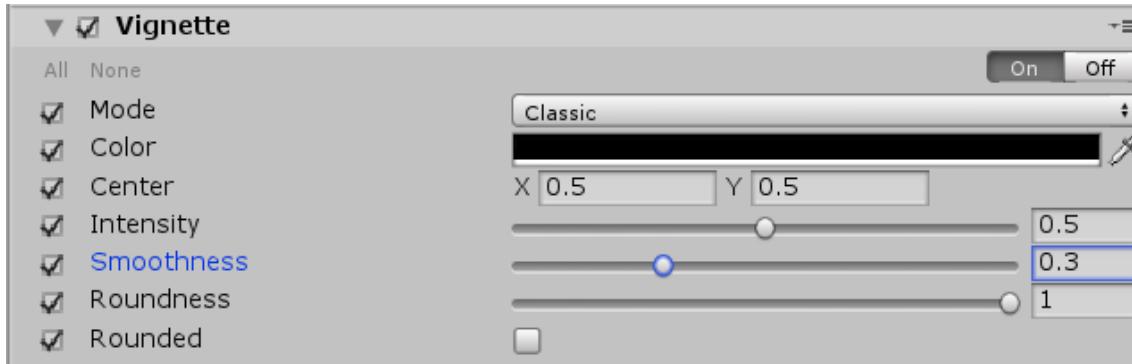
1. Haga clic en el botón **Agregar efecto ...** y seleccione **Unidad> Viñeta**. Expanda la sección de efecto de **viñeta**.



2. Haga clic en el acceso directo **Todos de la** viñeta para activar todas las funciones que necesita.

3. Al igual que Bloom y la oclusión ambiental, el efecto de viñeta necesita una intensidad. Esto determinará qué tan lejos en la pantalla se extiende el efecto. Establezca la propiedad **Intensidad** en **0.5**.

4. Puede hacer que se sienta aún más claustrofóbico y espeluznante al aumentar el suavizado. La propiedad Suavidad determina la distancia sobre la cual el efecto se desvanece hacia el centro de la pantalla. Establezca la propiedad **Suavidad** en **0.3**.



5. Contraiga el efecto Viñeta.

13. Agregue el efecto de (lens distortion) distorsión de lente

El efecto final que necesita agregar es Distorsión de lente. Esto deforma el cuadro final renderizado y ayudará a que parezca que estás viendo el juego a través de una cámara espía. Para agregar este efecto:

1. Haga clic en el botón **Agregar efecto ...** y seleccione **Unidad> Distorsión de lente**.
. Expanda la sección Efecto de **distorsión de lente** .



2. Haga clic de la distorsión de la lente **Todo** acceso directo para activar todas las características que necesita.
3. Al igual que con los otros efectos, la **intensidad** controla la fuerza del efecto. Intenta deslizar la intensidad de un lado a otro para tener una idea del efecto. Luego establezca la propiedad **Intensidad** en **35** .
4. Cuando usa este efecto, a veces puede crear fallas visuales en los bordes de la pantalla donde la imagen se ha deformado. Para solucionar esto, simplemente puede acercar un poco el marco renderizado para eliminar los problemas técnicos de la vista. Establezca la propiedad **Scale** en **1.1** para solucionar esto.



5. Guarde su escena.

¡Has terminado el postprocesamiento!

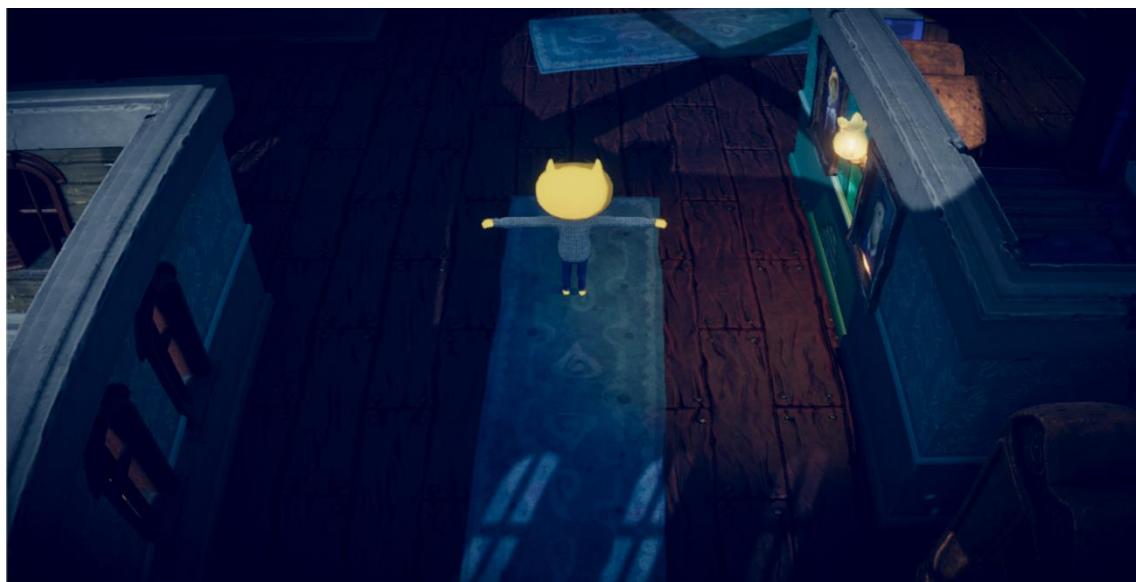
14. Resumen

En este tutorial, trataste muchas imágenes de tu juego, no solo cómo se mueve la cámara, sino también cambiando las imágenes para que tu juego se vea realmente profesional. Mire la diferencia visual que ha hecho:

Antes de:



Después:



En el siguiente tutorial, explorará la interfaz de usuario (interfaz de usuario) de su juego y creará un final para el juego.

Terminando el juego

1. Configurar la interfaz de usuario

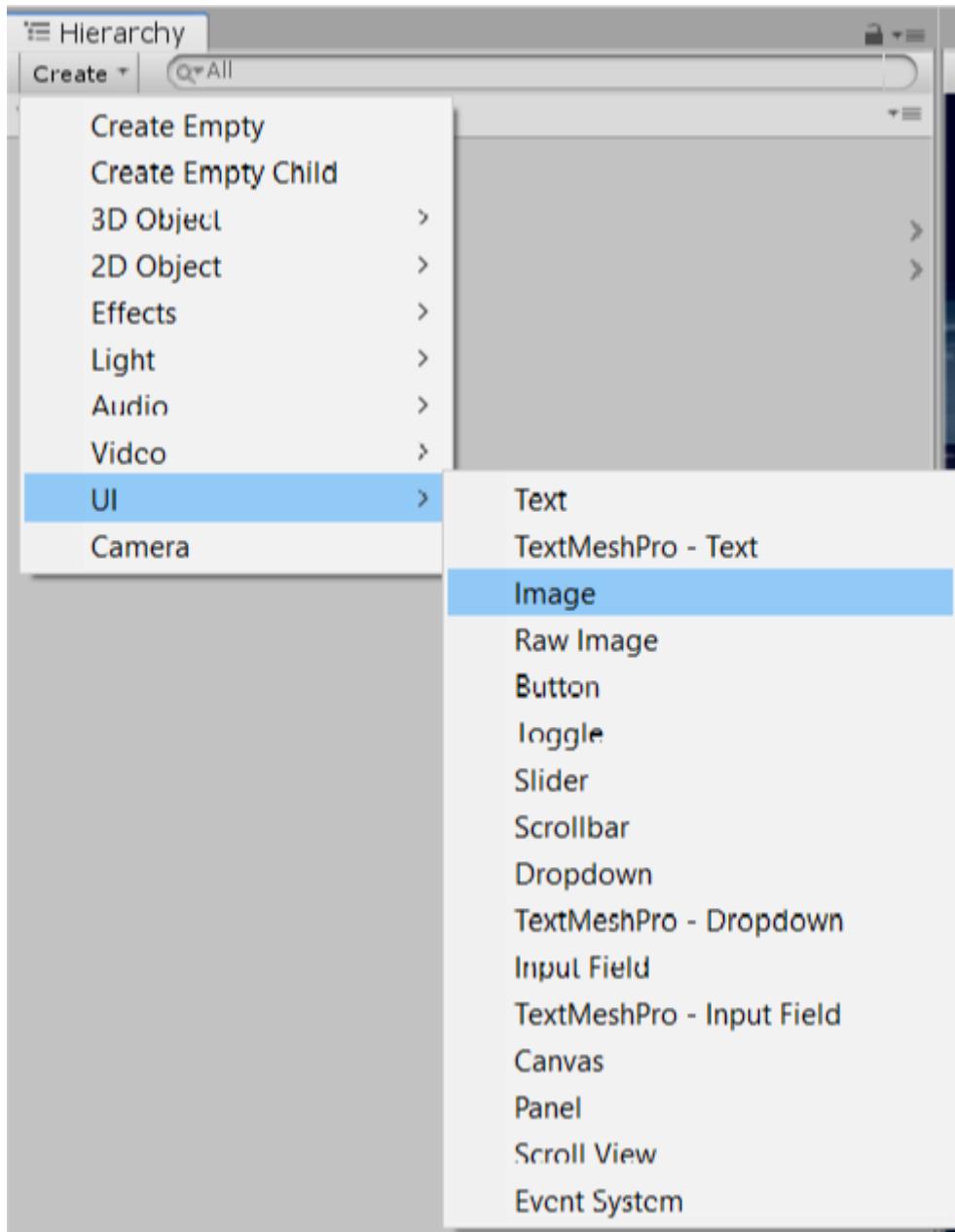
Ya has trabajado mucho para configurar el personaje, el entorno y la cámara para tu juego. A continuación, debe crear un final, para que el juego realmente termine cuando JohnLemon se escape de la casa. Una vez que hayas hecho esto, estarás listo para poblar tu juego con enemigos y hacer algunas mejoras finales.

Antes de comenzar a crear algo, es importante saber lo que está tratando de lograr.

Cuando JohnLemon llega a la salida, el juego debe desvanecerse y salir, pero debes tener cuidado con esto.

Primero, sin embargo, debes hacer que el juego se desvanezca. Para hacerlo, utilizará el sistema **UI (interfaz de usuario)** de Unity .

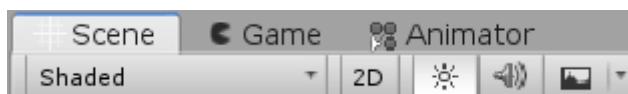
1. En la ventana Jerarquía, haga clic en el botón **Crear** . Esto se puede usar para crear todo tipo de GameObjects básicos.
2. Vaya a **UI> Imagen** . Una imagen de la interfaz de usuario puede extenderse por toda la pantalla del jugador y su opacidad se puede cambiar para crear un efecto de desvanecimiento.
¡Perfecto!



Esto agregará algunos GameObjects nuevos a su escena:

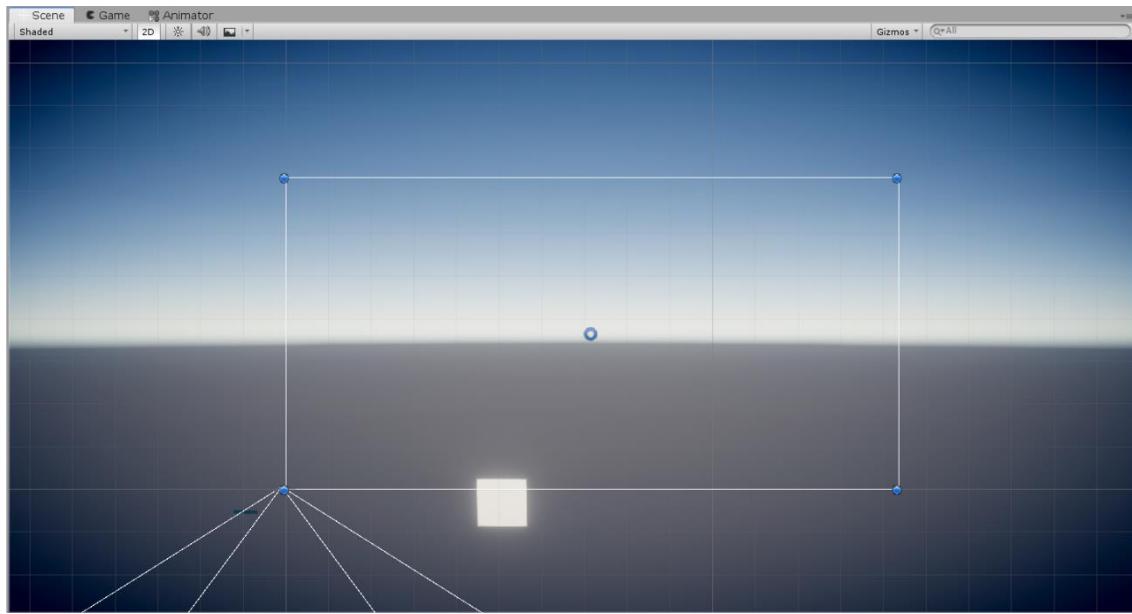


3. En la ventana Escena, haga clic en el botón 2D en la barra superior para habilitar el Modo 2D. Esto te permitirá ver los nuevos GameObjects correctamente.



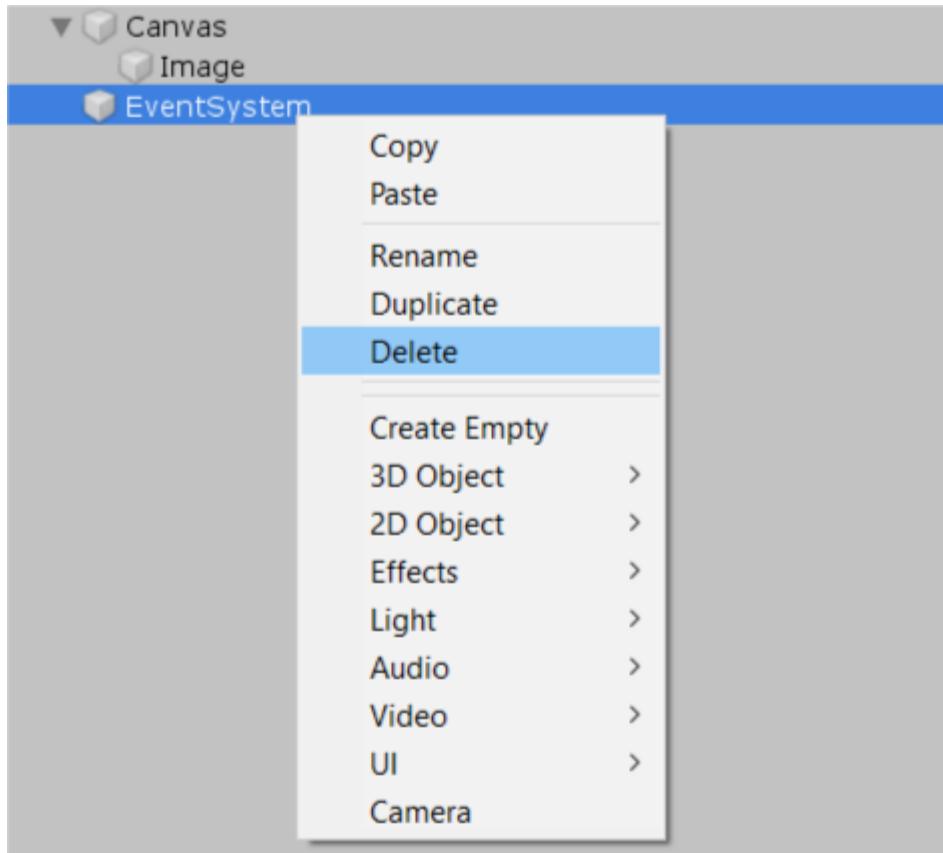
4. En la Jerarquía, seleccione el **Canvas** GameObject. Coloca el cursor sobre la escena y pulse F

5. Amplíe para ver el lienzo y la imagen más de cerca. Puede usar la rueda de desplazamiento o presionar Alt, hacer clic con el botón derecho y arrastrar.



6. En la Jerarquía, seleccione **EventSystem** GameObject. Este GameObject tiene componentes conectados que funcionan juntos para permitir que cualquier elemento de la interfaz de usuario en la pantalla interactúe con la entrada del usuario. Sin embargo, en su juego el jugador no necesita poder interactuar con la interfaz de usuario.

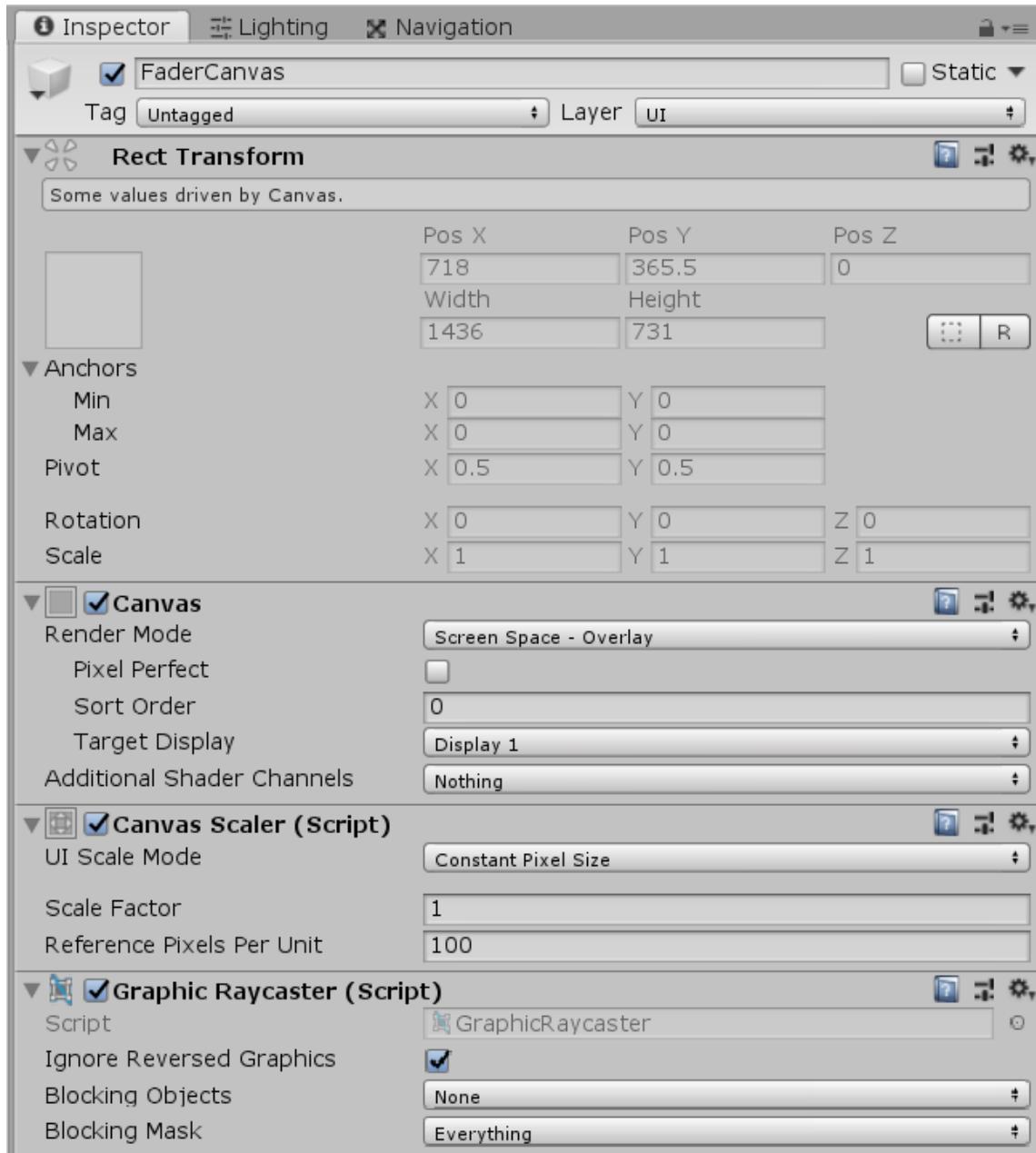
7. Haga clic derecho en EventSystem GameObject y seleccione **Eliminar**.



A continuación, usará los dos GameObjects restantes (Canvas e Image) para crear el efecto de desvanecimiento.

2. Configure el lienzo

1. En la Jerarquía, cambie el nombre del Lienzo a **FaderCanvas** .
2. En el Inspector, eche un vistazo a los componentes conectados a FaderCanvas.



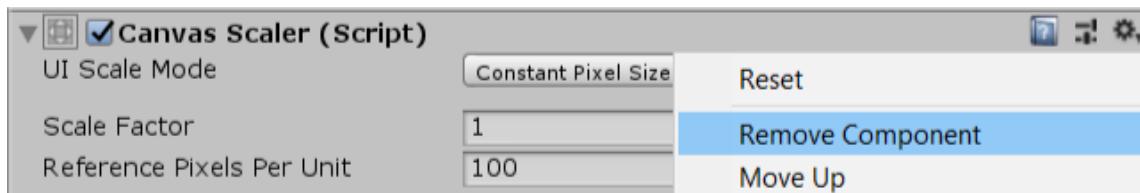
Debes notar que su componente Transformar es diferente a los que has visto antes. Los GameObjects que forman parte del sistema de interfaz de usuario necesitan un mejor control sobre su posición, por lo que tienen un componente **Rect Transform**. Para un lienzo en la raíz de una jerarquía de interfaz de usuario, la configuración de Transformar Rect es de solo lectura.

3. El componente Canvas controla cómo se representan los elementos de la interfaz de usuario que pertenecen a ese Canvas. Esta representación está controlada principalmente por la configuración del **Modo de representación (Render mode)**. Tiene tres modos potenciales:

- Espacio de pantalla superposición (Screen space Overlay), donde el lienzo llena la pantalla y todos los elementos de la interfaz de usuario del lienzo se representan sobre todo lo demás

- Espacio de pantalla cámara (screen space cámara), donde el lienzo llena la pantalla pero se representa en una cámara específica y está sujeta a la distancia de la cámara
- World Space, donde la IU existe en la escena y se representa delante o detrás de otros objetos (por ejemplo, etiquetas de nombre sobre los personajes en un mundo 3D)
- Necesita estirar una imagen en la pantalla y hacer que se procese sobre la parte superior de todo lo demás. Esto significa que el modo de renderizado predeterminado de Screen Space - Overlay es perfecto para usted.

4. El siguiente componente en FaderCanvas GameObject es Canvas Scaler . Esto se usa como una manera fácil de controlar los tamaños relativos de los elementos de la interfaz de usuario cuando se muestran en diferentes tamaños de pantalla. Su imagen se extenderá por toda la pantalla, por lo que no necesita preocuparse por su escala relativa. Haga clic en el icono de engranaje en la parte superior derecha del componente Escalar del lienzo para abrir el menú contextual. Seleccione Eliminar componente .



5. El componente final del FaderCanvas GameObject es el **Graphic Raycaster**. Esto se usa para detectar eventos de IU como clics. Determinará en qué elemento de la IU se hizo clic y enviará el evento a ese elemento para que el componente apropiado pueda reaccionar. El jugador no interactuará con la interfaz de usuario en su juego, por lo que este componente no es necesario. Haga clic en el icono de engranaje en la esquina superior derecha del componente Graphic Raycaster para abrir el menú contextual. Seleccione **Eliminar componente** .

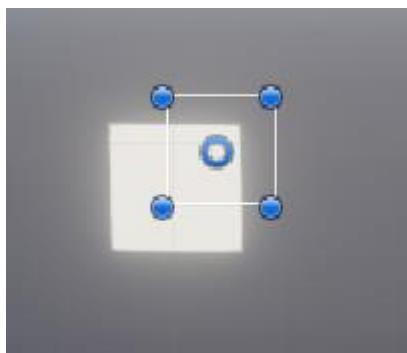
3. Estira la imagen

El siguiente paso es estirar la imagen en la pantalla:

1. En la Jerarquía, seleccione Image GameObject.
2. Seleccione la **herramienta de Rect** de la barra de herramientas o pulse **T** .



3. ¡ Espera! Algo gracioso está sucediendo aquí. ¿Por qué la imagen no está donde dice la herramienta Rect?

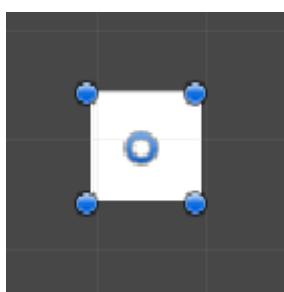


Esto se debe a que la ventana Escena está utilizando el procesamiento posterior que configuró en el tutorial anterior.

4. Haga clic en el botón **Efectos** en la ventana Escena para habilitar todos los efectos, luego haga clic nuevamente para deshabilitar todos los efectos.



Ahora su imagen debería mostrarse correctamente:



4. Explore el componente Rect Transform

Echemos un vistazo más de cerca al componente Rect Transform:

La posición de un GameObject 3D está representada por un único punto en el pivote de GameObject. La posición de este punto de pivote en un componente Transformar es relativa al padre del GameObject. Las Transformaciones Rect funcionan de manera similar, pero dado

que los elementos de la IU pueden representar un área, existen algunas diferencias. En lugar de que la posición de una Transformación Rect sea relativa a un único punto de pivote en su elemento primario, son relativos a un **área de su elemento primario**. Esta área del padre está representada por los **Anclas de una Transformación Rect**.

Es posible que haya visto un artilugio en la ventana de escena que parece una flor:

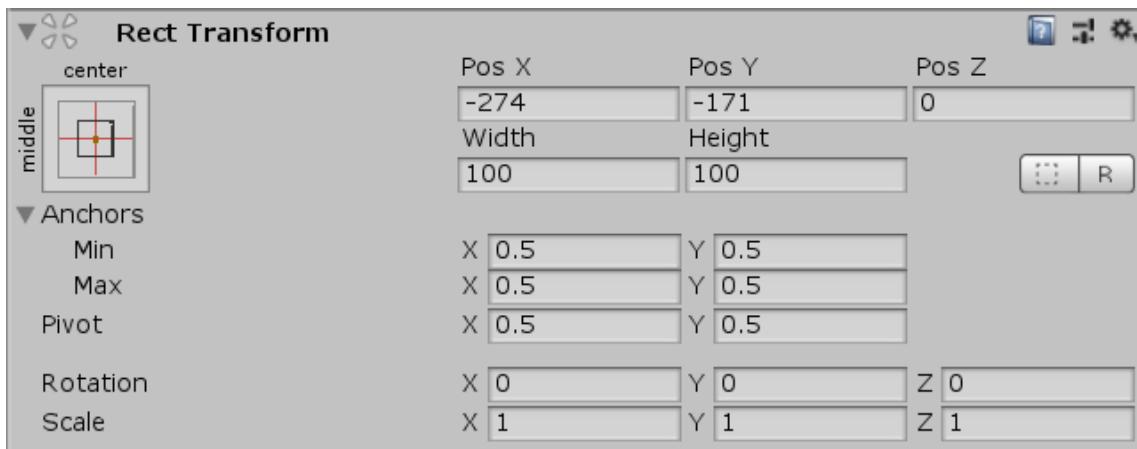


Esto es en realidad múltiples puntos todos juntos. Cada uno de estos puntos es un ancla. El rectángulo creado por los cuatro puntos de anclaje es una proporción del área total del padre.

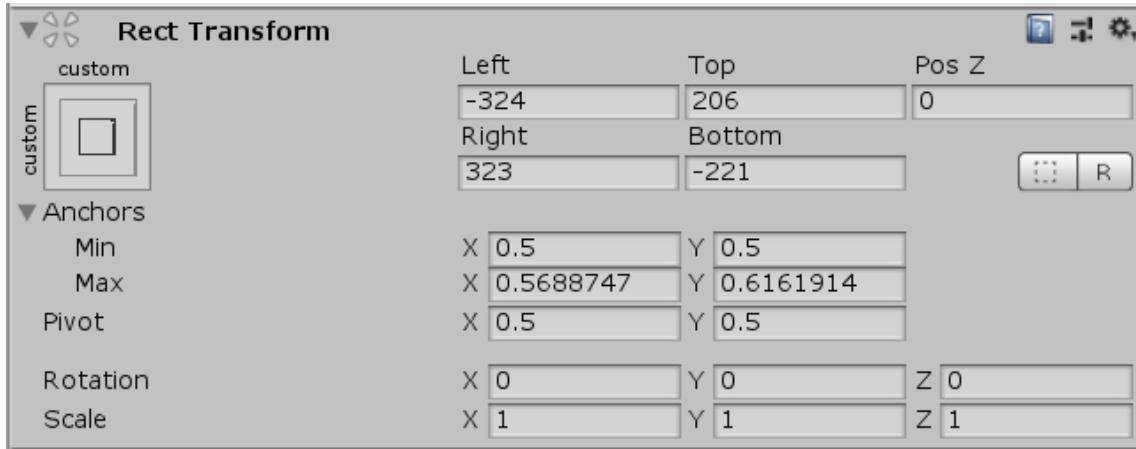


Tu imagen es hija de FaderCanvas, que llena toda la pantalla. Esto significa que la posición de su imagen es relativa a un área de toda la pantalla.

La posición de los elementos de la interfaz de usuario se mide en píxeles. Esto es muy importante, particularmente porque no todas las pantallas tienen el mismo número de píxeles. Esto es lo que hace que el sistema de anclaje sea tan poderoso: cuando las anclas están todas juntas y muestran un punto, la Transformación de rectificación muestra una posición en píxeles en la que el elemento de la IU está desplazado de ese punto.



Sin embargo, si los anclajes están separados, la Transformación Rect muestra un desplazamiento de píxeles desde los lados del área de anclaje:

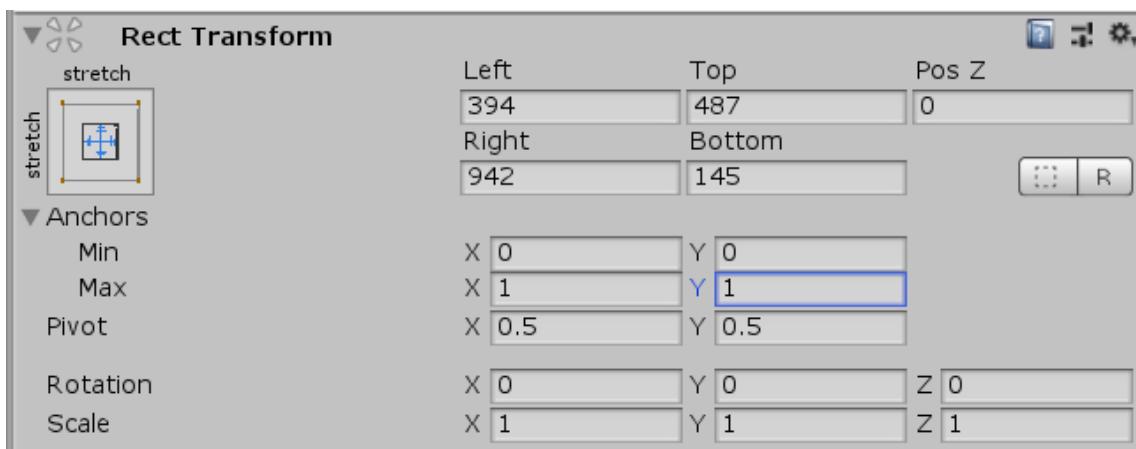


Entonces, ¿cómo se aplica esto a tu juego? Bueno, necesitas que toda la pantalla se desvanezca. Eso significa que su imagen debe cubrir toda la pantalla sin importar la forma o el tamaño de la pantalla. Para lograr esto, debe asegurarse de que el área de anclaje sea la pantalla completa y que no haya desplazamiento desde esa área.

5. Configure el componente Rect Transform

Para configurar el componente Rect Transform:

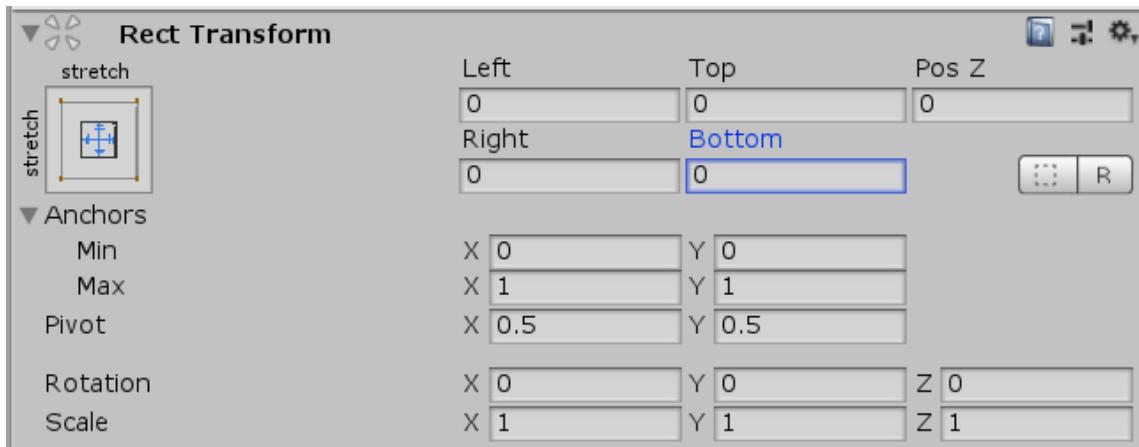
1. En la ventana Jerarquía, seleccione **Image** GameObject.
2. En el Inspector, busque el componente **Rect Transform**. Expanda la configuración de **Anclas**.
3. Establezca los **valores mínimos para x e y en 0**. Establezca los **valores máximos para x e y en 1**.



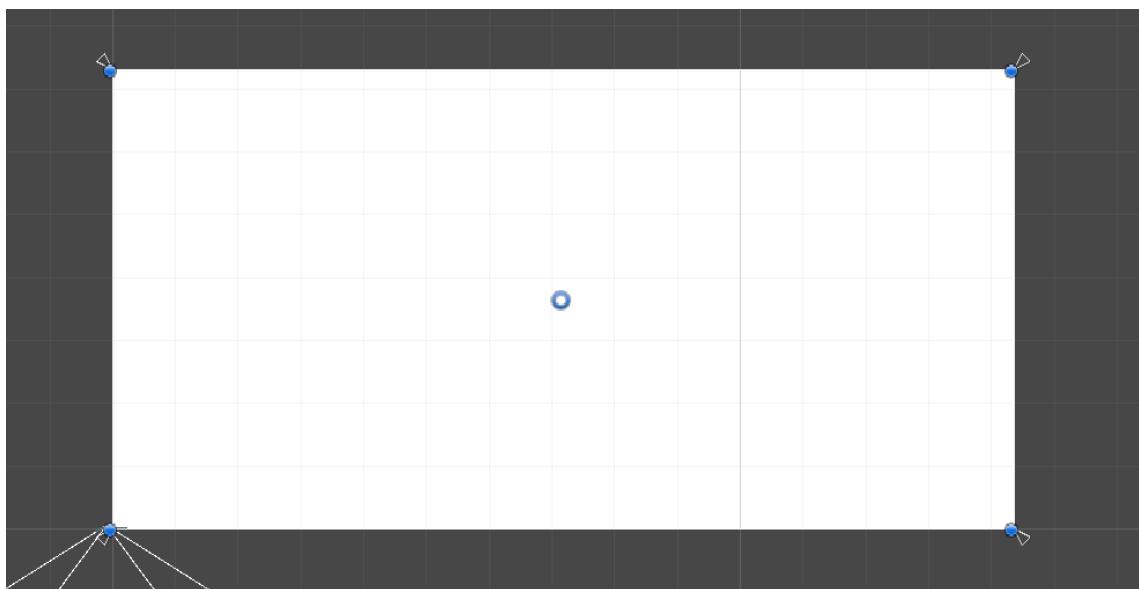
Recuerde que los Anchors son relativos a sus padres: 0 significa el extremo izquierdo o inferior de la pantalla, 1 significa el extremo derecho o superior de la pantalla. Ahora que ha

establecido los Anclajes, debería ver que la posición de su Imagen ha cambiado. La posición ahora aparece como Izquierda, Superior, Pos Z, Derecha e Inferior. Además de Pos Z (que puede ignorar), estas son la distancia del GameObject desde su área de anclaje en píxeles. Un valor negativo indica que el elemento está fuera del área de anclaje. Un valor de cero significa que no hay desplazamiento del área de anclaje. En el ejemplo anterior, el lado izquierdo de la imagen tiene 394 píxeles desde el lado izquierdo del área de anclaje; el lado derecho de la imagen está a 942 píxeles del lado derecho del área de anclaje; el lado superior de la imagen está a 487 píxeles del lado superior del área de anclaje y el lado inferior de la imagen está a 145 píxeles del fondo del área de anclaje.

- Dado que el área de anclaje de su imagen ahora es la pantalla completa, no se necesita ningún desplazamiento. En la transformación Rect de la imagen, establezca las propiedades **Izquierda , Superior , Derecha e Inferior en 0 .**

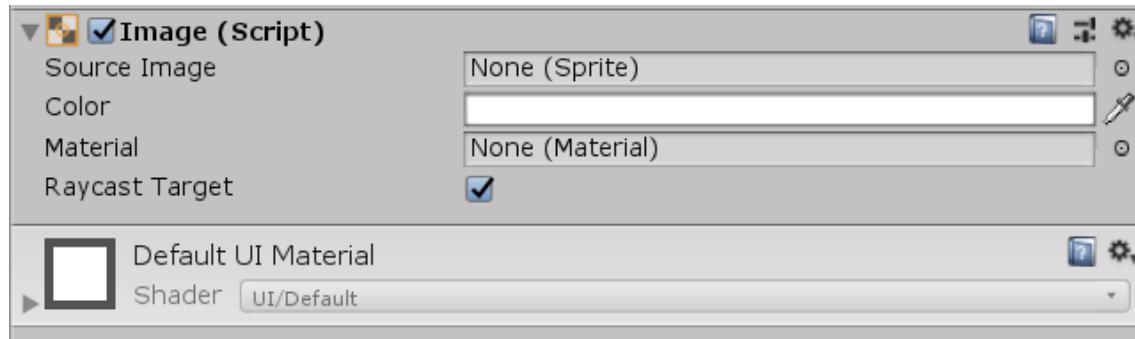


- La imagen ahora se extiende sobre la pantalla:



Esta imagen tiene el tamaño correcto ahora, pero no es el color correcto.

7. En el Inspector, busque el componente **Imagen** .



La primera propiedad se llama Imagen de origen. Esto le permite mostrar una imagen específica; si se deja en blanco, obtenga un rectángulo de color sólido. Esto se puede configurar con la propiedad **Color** .

8. Abra la ventana del selector de color. Establezca los canales RGB en **0** , dejando A al máximo (1 si el rango es de 0 a 1, o 255 si el rango es de 0 a 255). Esto establecerá el color en negro. A es el cuarto canal que compone Color: **Alpha** . El Alfa de un color es lo transparente que es. Cuanto más bajo sea el valor Alfa, más transparente será el GameObject. Ajustar el alfa de su imagen será la clave para que se desvanezca y desaparezca.

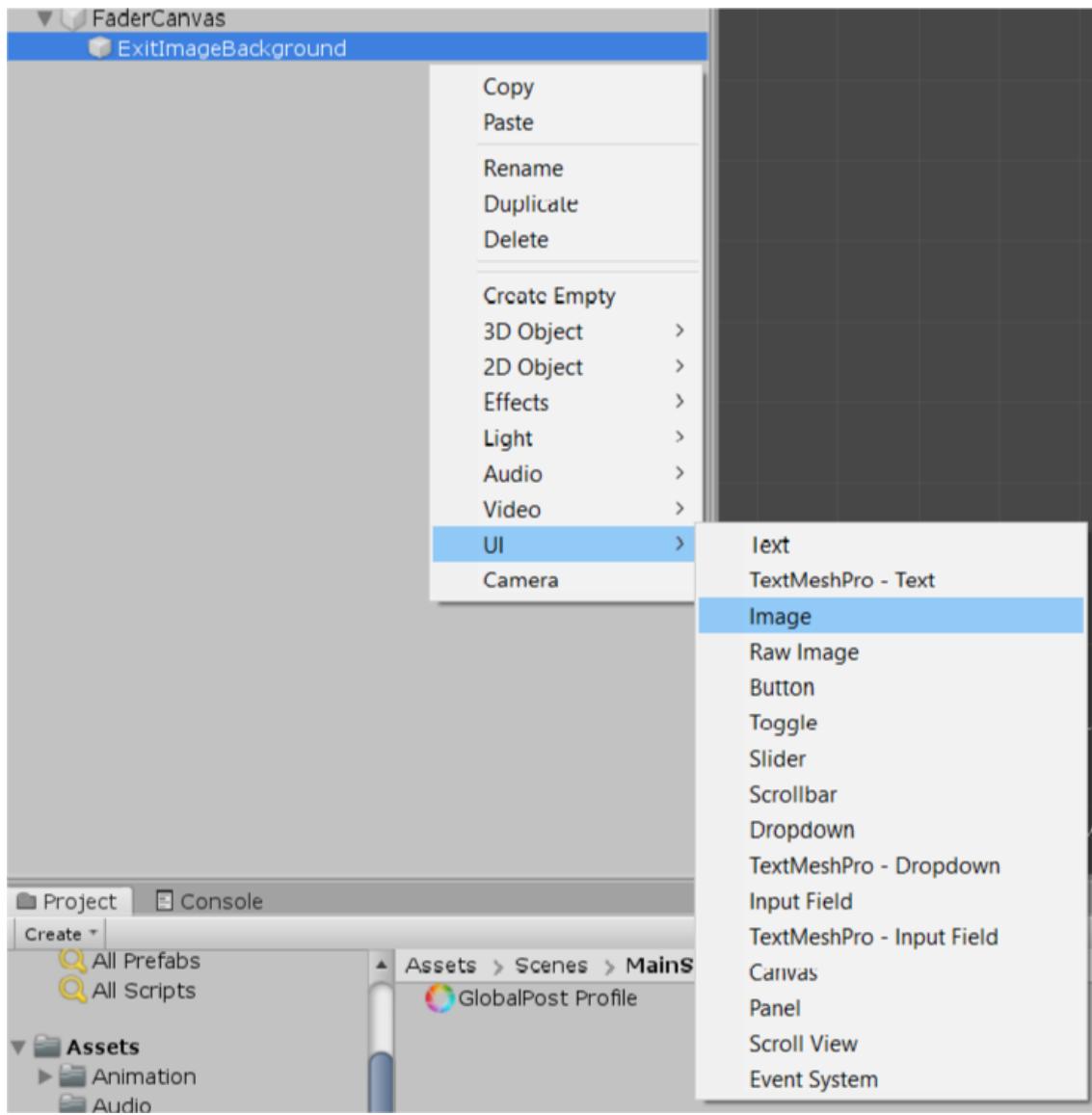
9. Haga clic en otra parte del Editor para cerrar la ventana del selector.

¡Ahora tienes una pantalla completamente negra!

6. Agregar una imagen de victoria

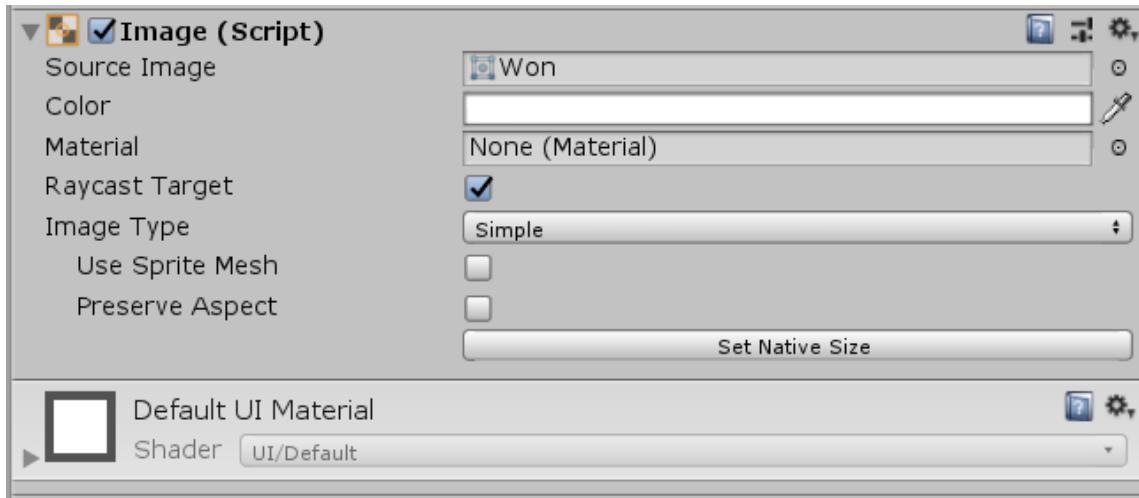
A continuación, agregará una imagen para mostrar en la parte superior de la pantalla negra:

1. En la Jerarquía, cambie el nombre de su Image GameObject actual a **ExitImageBackground** .
2. Haga clic derecho en **ExitImageBackground** GameObject. En el menú contextual que aparece, seleccione **UI > Imagen** .



Al hacer clic con el botón derecho en un GameObject para crear otro GameObject de esta manera, el GameObject recién creado se convierte en un elemento secundario del que se hizo clic. Debido a que este nuevo ExitImage GameObject es hijo del ExitImageBackground GameObject, se representa en la parte superior.

3. Cambie el nombre de esta nueva imagen GameObject a **ExitImage** .
4. Agreguemos una imagen. En el Inspector, busque el componente **Imagen** .
5. Haga clic en el botón de selección circular para la propiedad **Imagen de origen** . En el cuadro de diálogo, busque y seleccione la imagen llamada **Won** .
6. Ahora hay configuraciones adicionales disponibles en el componente Imagen:



Necesitará esta configuración para solucionar un problema importante: se verá mejor si la imagen llena la mayor parte de la pantalla posible, pero debe mantener la relación de aspecto correcta. Para ver este problema, cambiemos el tamaño de la Rect Transform como lo hicimos con el padre.

7. En el Inspector, busque el componente **RectTransform**. Expanda la configuración de **Anclas**
8. Establezca los **valores mínimos para x e y en 0**. Establezca los **valores máximos para x e y en 1**.
9. Establezca las propiedades Izquierda, Derecha, Superior e Inferior en **0**. ¡La imagen ahora se verá estirada!



10. En el componente Imagen, busque la propiedad **Tipo de imagen**. Active la casilla de verificación **Conservar aspecto(Preserve Aspect)**. Esto hace que la imagen sea lo más grande posible dentro de la Transformación Rect sin ser aplastada o estirada.



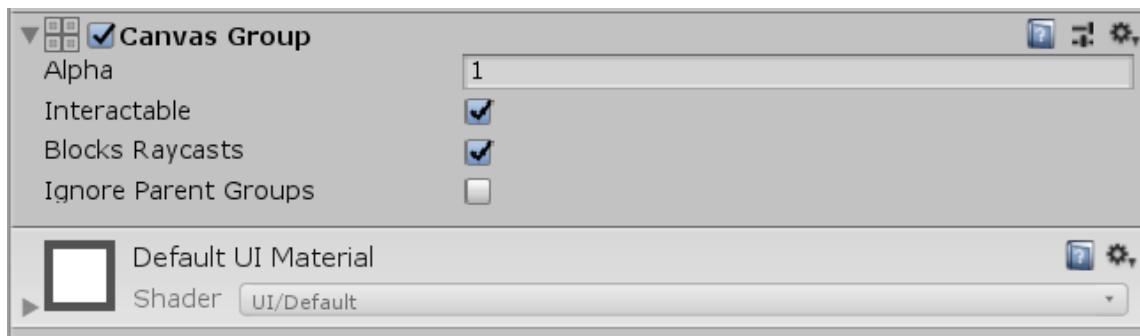
7. Agregar un componente de grupo de lienzo

A continuación, consideremos cómo hacer que estos elementos de la interfaz de usuario se desvanezcan cuando lo deseé. Ya sabe que puede ajustar el valor alfa de un componente de imagen para que desaparezca. Pero debido a que ahora hay dos imágenes, deberá cambiar los dos colores en lugar de cambiar solo un valor. Para ayudarlo a hacer esto, hay un componente llamado **Grupo Canvas**.

Un grupo de Canvas le permite controlar algunos aspectos de todos los elementos de la interfaz de usuario visibles en un GameObject y todos sus elementos secundarios.

Para agregar un componente de grupo de lienzo:

1. En la ventana Jerarquía, seleccione ExitImage Background GameObject.
2. En el Inspector, agregue un componente Grupo de lienzos. (**Canvas group**)



Cambia la propiedad **Alpha** a **0**.

3. Ahora que ha terminado de crear la interfaz de usuario, cambiemos la ventana de escena al nivel. Primero, **deshabilite el Modo 2D** en la ventana Escena.

4. En la ventana Jerarquía, seleccione JohnLemon GameObject. Con el cursor sobre la vista de escena, presione **F** para enfocar.

5. En la ventana Jerarquía, contraiga el FaderCanvas GameObject. Guarde la escena presionando Ctrl + S en Windows o CMD + S en macOS.

Eso está mejor, ahora puedes ver el nivel nuevamente. A continuación, necesita una forma de activar y controlar el desvanecimiento de la interfaz de usuario.

8. Crear un disparador de final de juego

La computadora necesita una forma de reconocer que JohnLemon ha salido de la casa embrujada, para que sepa cuándo cambiar el valor de la propiedad Alpha del grupo Canvas. Una técnica común para detectar cuándo un objeto físico (por ejemplo, el personaje JohnLemon) ha entrado en un área específica es usar un **disparador**.

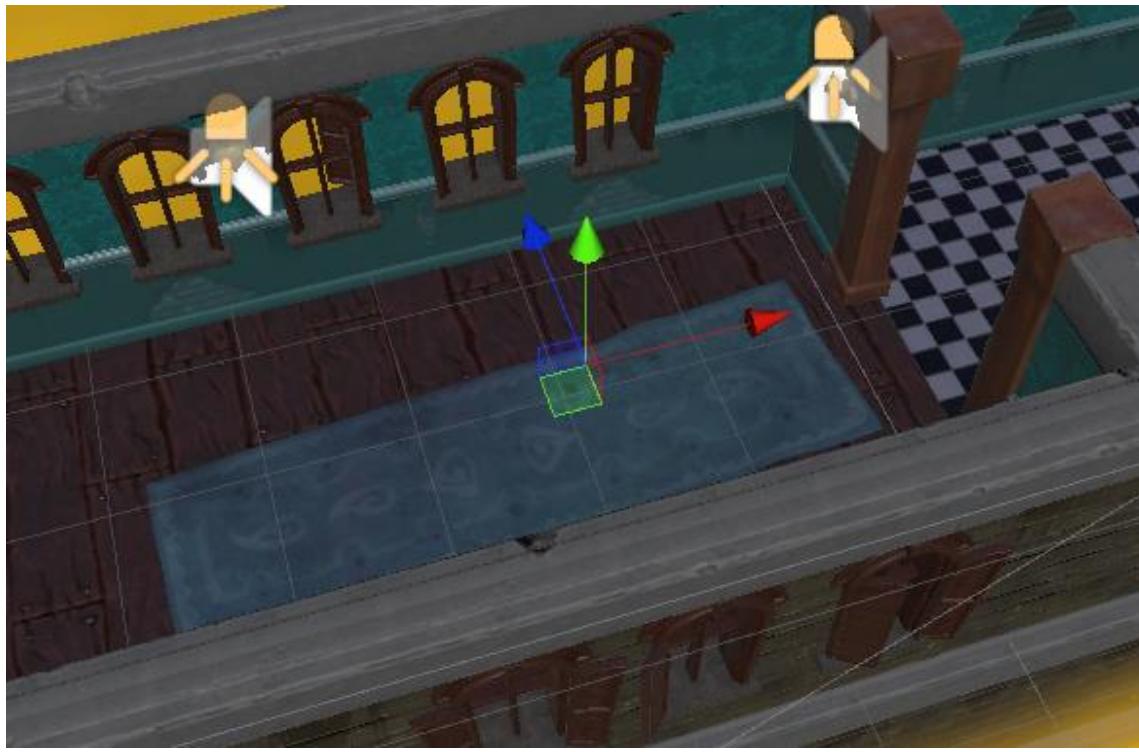
Los disparadores son colisionadores que no impiden el movimiento; en cambio, permiten que los objetos físicos los atraviesen libremente, pero informan el evento desencadenante para que puedan ocurrir otras acciones.

Primero, creamos un disparador:

1. En la ventana Jerarquía, haga clic en el menú Crear y seleccione **Create Empty**.

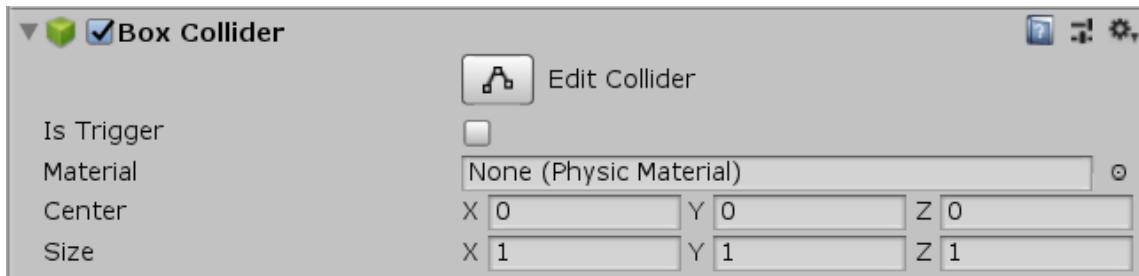
Cambie el nombre de este GameObject a **GameEnding**.

2. Establezca la posición de la transformación de GameEnding en (**18, 1, 1.5**). Esto está en el medio de la salida del nivel.



3. Ahora necesita agregar el disparador. En el Inspector, agregue un componente **Box Collider** al GameEnding GameObject. (¡Tenga cuidado de no agregar un Box Collider 2D, ya que no funcionará!)

4. Active la casilla de verificación **Is Trigger**. Esto cambia el Collider en un Trigger.



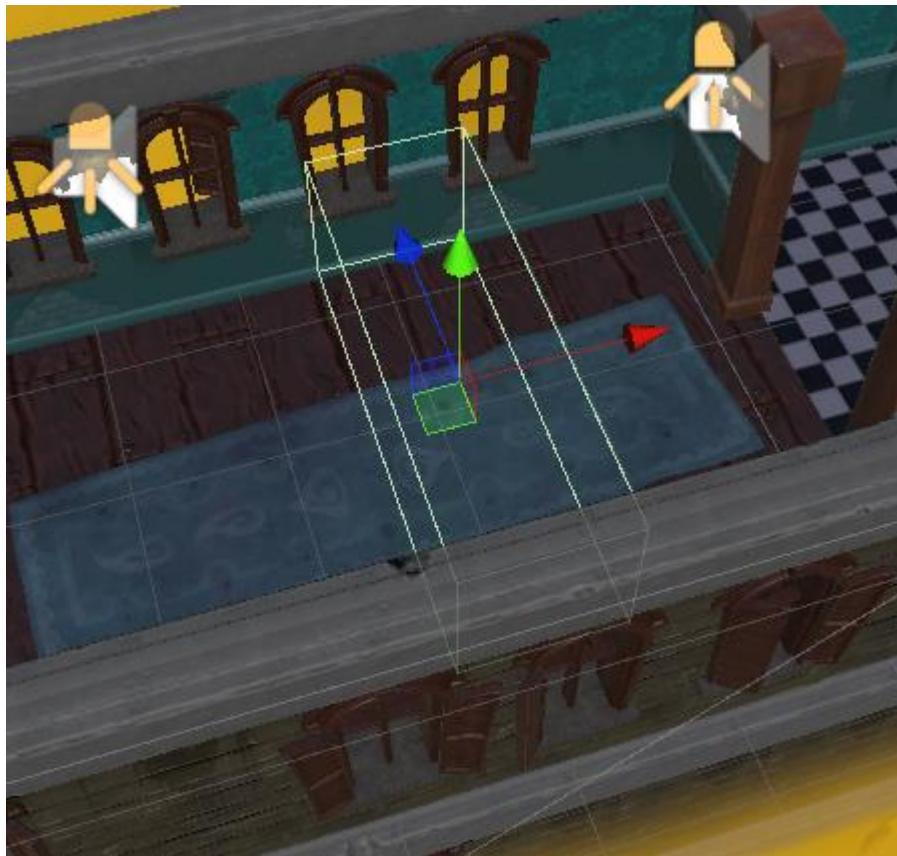
5. Luego, debe cambiar el tamaño del Disparador para que cubra la salida: JohnLemon tendrá que caminar hacia él para escapar. Hay dos formas de hacer esto: haga clic en el botón Editar colisionador y cambie el tamaño del colisionador de cuadros en la ventana de escena, o configure las propiedades Centro y Tamaño manualmente. Intentemos editar el disparador en la ventana de escena.

Haga clic en el botón **Editar Collider**; Aparecerán “manijas” a los lados del disparador en la ventana de escena.

6. Para cambiar cualquiera de las caras del Box Collider, haga clic y arrástrelo a la posición que desea. Necesitas el Trigger para llenar el corredor, de modo que JohnLemon no pueda salir sin moverse a través de él. Cuando haya terminado, haga clic en el botón **Editar Collider**

nuevamente. Cuando las asas han desaparecido de la ventana de escena, ya no está editando el Collider.

7. Alternativamente, puede establecer el Tamaño del Collider en (1 , 1 , 3.5).



Ahora ha completado el activador, pero deberá escribir un script para usarlo.

9. Crear un nuevo script

Para comenzar, creamos el script Asset:

1. En la ventana Proyecto, vaya a **Activo> Scripts** .
2. Vaya al menú **Crear** y seleccione **C # Script** . Nombre el script "**GameEnding**" . Recuerde que el nombre del activo debe ser exactamente el mismo que el nombre de la clase dentro del script. Si comete un error, elimine el activo y créelo nuevamente.
3. Arrastre el activo de secuencia de comandos desde la ventana Proyecto hasta el **GameOnding** GameObject en la ventana Jerarquía. Esto lo agregará como un componente.
4. Ahora puede comenzar a editar el script para darle alguna funcionalidad. Haga doble clic en el activo del script para abrirlo y editarlo.

10. Inicie la secuencia de comandos GameEnding

Para iniciar el script:

1. Elimine los métodos de Inicio y Actualización y sus comentarios. Su script limpio debería verse así:

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

public class GameEnding : MonoBehaviour
{

}
```

2. Pensemos en la información que necesita: Primero, necesita que la pantalla se desvanezca durante un cierto período de tiempo. Debe ser una **variable pública**, por lo que puede ajustarla desde el Inspector. También debe permitir números no enteros, por lo que la variable debe ser **flotante**.

Un valor predeterminado razonable para que ocurra el desvanecimiento es de 1 segundo, así que configurémoslo como predeterminado. Agregue la siguiente línea a su script, entre llaves:

```
public float fadeDuration = 1f;
```

3. A continuación, debe especificar que el desvanecimiento debe ocurrir cuando el personaje del jugador golpea el Disparador. Para asegurarte de que esto solo suceda cuando JohnLemon llegue al Trigger, necesitarás una referencia a ese GameObject. Nuevamente, esta debería ser una variable pública para que pueda ajustarla desde el Inspector. Agregue la siguiente línea debajo de la declaración de la variable fadeDuration:

```
public GameObject player;
```

Ahora tiene dos variables principales declaradas.

4. Uno de los trabajos esenciales de esta clase es detectar el GameObject controlado por el jugador. A continuación, usará otro método especial para MonoBehaviours llamado **OnTriggerEnter**. Agregue la siguiente definición de método debajo de las declaraciones de variables:

```
void OnTriggerEnter (Collider other)
{
}
```

5. Para asegurarse de que el final solo se desencadena cuando JohnLemon golpea el Box Collider, agregue la siguiente instrucción if al método OnTriggerEnter, entre las llaves:

```
if (other.gameObject == player)
{
}
```

Esto utiliza un nuevo operador llamado **operador de equivalencia**. Está representado por un doble igual y da como resultado un bool. El valor devuelto es verdadero si las cosas en ambos lados son idénticas, o falso si no lo son. El código anterior le indica a la computadora: "Si el otro GameObject del Collider (el que ingresó al Trigger) es equivalente a nuestra referencia al GameObject de JohnLemon, entonces haga lo que esté en el bloque de código".

6. Revisemos su script hasta ahora:

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

public class GameEnding : MonoBehaviour
{
    public float fadeDuration = 1f;
    public GameObject player;

    void OnTriggerEnter (Collider other)
    {
        if (other.gameObject == player)
        {

        }
    }
}
```

Recuerde, en C# no importa en qué orden se declaran los métodos en una clase; puede haber usado un orden ligeramente diferente, y eso está bien.

11. Agregar un método de actualización

Ahora debe agregar código para la instrucción if que inicia el desvanecimiento de la interfaz de usuario y luego abandona el juego cuando finaliza. Actualmente hay un problema: se llama a OnTriggerEnter solo una vez cuando los Colliders se superponen por primera vez. Necesita algo que se llama cada cuadro, por lo que puede cambiar gradualmente el alfa del Grupo de lienzo.

[En el tercer tutorial usaste el método de actualización](#), que se llama cada cuadro; puedes usar esto aquí:

1. Agregue un método de actualización como se muestra a continuación en OnTriggerEnter:

```
void Update ()  
{  
    [REDACTED]  
}
```

2. Necesita una forma de saber cuándo comenzar a desvanecer el grupo de lienzo. Dado que el grupo de lienzo debería estar desvaneciéndose o no, una **variable bool** es perfecta para esto.

Cree una variable bool llamada **m_IsPlayerAtExit** t de la siguiente manera entre las declaraciones de variables públicas y la definición OnTriggerEnter:

```
bool m_IsPlayerAtExit;
```

3. ¡Usemos el bool! Primero, primero debe configurarse. Dentro del bloque de código de la instrucción if en su método OnTriggerEnter, establezca m_IsPlayerAtExit en verdadero de la siguiente manera:

```
m_IsPlayerAtExit = true;
```

4. Ahora que ha configurado el bool en OnTriggerEnter, debe verificar si se ha configurado en Actualización. Agregue una instrucción if dentro de las llaves del método Update:

```
if(m_IsPlayerAtExit)  
{  
    [REDACTED]  
}
```

5. Su secuencia de comandos debería verse así actualmente:

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class GameEnding : MonoBehaviour  
{  
    public float fadeDuration = 1f;
```

```

public GameObject player;

bool m_IsPlayerAtExit;

void OnTriggerEnter (Collider other)
{
    if (other.gameObject == player)
    {
        m_IsPlayerAtExit = true;
    }
}

void Update ()
{
    if(m_IsPlayerAtExit)
    {

    }
}

```

Analicemos lo que ha codificado hasta ahora: Has agregado dos métodos, OnTriggerEnter y Update. Cuando se llama a OnTriggerEnter, las computadoras verifican si el Collider que ingresó al Trigger pertenece al personaje del jugador. Actualización se llama cada cuadro y se comprueba si el personaje del jugador está en la salida. Si el personaje del jugador está en la salida, entonces cae en el bloque de código de la declaración if; de lo contrario, no hace nada.

12. Escriba el bloque de código de declaración if

A continuación, debe escribir eso si el bloque de código de la declaración:

1. Cree un nuevo método debajo de Actualización llamado **EndLevel**. No necesita devolver nada y no necesita ningún parámetro, por lo que debería verse así:

```

void EndLevel ()
{
}

```

Esto ayudará a mantener las cosas ordenadas.

2. Dentro de la instrucción if en el método Update, agregue una llamada al método EndLevel:

```
EndLevel ();
```

3. El método EndLevel necesita desvanecer el grupo de lienzo y luego salir del juego. Anteriormente, cuando encontraba referencias a componentes, utilizaba el método GetComponent, pero esto solo funciona para componentes en el mismo GameObject que este script. Esta vez, creará una variable pública para el componente Grupo de lienzo, que se puede asignar en el Inspector. Agregue esta declaración de variable pública **debajo de la declaración de variable del jugador** :

```
public CanvasGroup exitBackgroundImageCanvasGroup;
```

4. También necesitas un temporizador para asegurarte de que el juego no termine antes de que el desvanecimiento haya terminado. Mientras declaramos variables, deberíamos considerar cualquier otra que podamos necesitar. Agregue la siguiente declaración de variable debajo del **bool m_IsPlayerAtExit** :

```
float m_Timer;
```

5. Luego, el método EndLevel necesita comenzar a contar el temporizador. Puede recordar por el script PlayerMovement que hay una forma de obtener cuánto tiempo ha pasado desde el último fotograma: usando Time.deltaTime. Puede configurar el temporizador igual a sí mismo más el deltaTime. Sin embargo, hay un atajo para decir exactamente eso usando el operador más-igual. Agregue la siguiente línea de código a su método EndLevel, dentro de las llaves:

```
m_Timer += Time.deltaTime;
```

Aquí, el operador más-igual es decir establecer lo que está a la izquierda para que sea igual a sí mismo más lo que esté a la derecha.

6. Es hora de configurar el Alfa de su Grupo de Canvas. El valor alfa debe ser 0 cuando el temporizador es 0 y 1 cuando el temporizador está hasta la duración de la atenuación. Para obtener este valor, puede dividir el temporizador por la duración. Agregue el siguiente código debajo de la línea de operador más-igual que acaba de agregar al método EndLevel:

```
exitBackgroundImageCanvasGroup.alpha = m_Timer / fadeDuration;
```

Ahora las imágenes se desvanecerán cuando JohnLemon llegue a la salida, pero aún tiene algunas cosas más que resolver antes de terminar.

7. Finalmente, el juego debe cerrarse cuando finalice el desvanecimiento. El desvanecimiento finalizará cuando el temporizador sea mayor que la duración. Agregue un nuevo operador debajo de la línea anterior que agregó:

```
if(m_Timer > fadeDuration)
{
}
```

El paréntesis angular derecho prueba si lo que está a su izquierda es **mayor que** lo que está a su derecha. Si es así, devuelve verdadero; de lo contrario, devuelve falso. El bloque de código de la instrucción if solo se ejecutará si el alfa es hasta 1, por lo que ahora todo lo que tenemos que hacer es agregar la siguiente línea al bloque de código para que el juego salga:

```
Application.Quit();
```

8. Hay algo importante que saber aquí. De hecho, este método dejará el juego, pero solo funciona para aplicaciones totalmente integradas. Actualmente, el juego es solo un proyecto que se juega en el Editor, por lo que aún no hará nada. Hará una compilación de su juego en el tutorial final de esta serie: cuando haya hecho esto, la línea de código funcionará correctamente. Hasta entonces, deberá salir del modo de reproducción como lo ha hecho antes.

9. En su estado actual, su secuencia de comandos hará que el juego se cierre tan pronto como las Imágenes hayan terminado de desvanecerse. Sería mucho mejor si las imágenes se desvanecieran, el jugador las viera por un breve tiempo y luego el juego se cerrara. Todo lo que necesita hacer para lograr esto es agregar algo más de tiempo en la última declaración if. Convertir esto en una variable significa que puede cambiar la duración de las imágenes que se muestran en el futuro, si lo desea. Debajo de la declaración de la variable fadeDuration (cerca de la parte superior de la clase), agregue otra declaración de variable de duración:

```
public float displayImageDuration = 1f;
```

10. Ahora todo lo que necesita hacer es agregar esta duración a la transición de fade en la instrucción if, cambiándola a:

```
if(m_Timer > fadeDuration + displayImageDuration)
{
    Application.Quit();
}
```

Ya terminaste! La secuencia de comandos completa debería ser similar a esto:

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

public class GameEnding : MonoBehaviour
{
    public float fadeDuration = 1f;
    public float displayImageDuration = 1f;
    public GameObject player;
    public CanvasGroup exitBackgroundImageCanvasGroup;
```

```

        bool m_IsPlayerAtExit;
        float m_Timer;

        void OnTriggerEnter (Collider other)
        {
            if (other.gameObject == player)
            {
                m_IsPlayerAtExit = true;
            }
        }

        void Update ()
        {
            if(m_IsPlayerAtExit)
            {
                EndLevel ();
            }
        }

        void EndLevel ()
        {
            m_Timer += Time.deltaTime;

            exitBackgroundImageCanvasGroup.alpha = m_Timer / fadeDuration;

            if(m_Timer > fadeDuration + displayImageDuration)
            {
                Application.Quit ();
            }
        }
    }
}

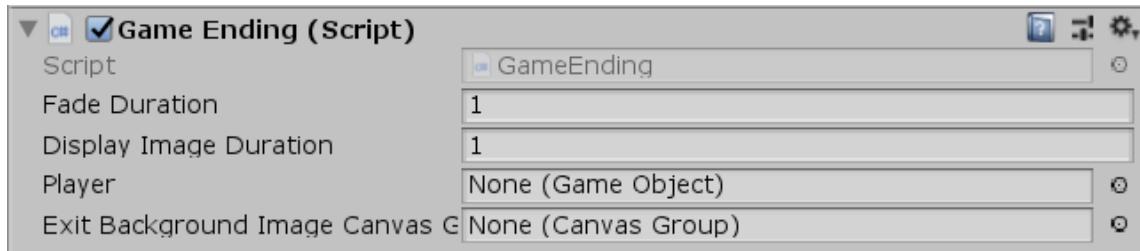
```

Asegúrese de guardar su secuencia de comandos, luego regrese al Editor de Unity.

13. Establezca las variables para su secuencia de comandos GameEnding

En su secuencia de comandos, creó una serie de variables que deben establecerse en el Editor. Esto facilita la personalización del juego y prueba los cambios fácilmente. Para establecer las variables:

1. En la Jerarquía, seleccione el **GameOnding** GameObject.



2. Arrastre el **JohnLemon** GameObject desde la ventana Jerarquía al script Final del juego en el Inspector. Esto asignará la **variable Player**.

3. En la Jerarquía, expanda las FaderCanvas. Arrastre **ExitImageBackground** GameObject al campo Exit Background Background Canvas Group del componente Game Ending. Unity encontrará automáticamente el componente correcto para asignar la **variable Exit Background Background Canvas Group**.

4. Guarde la escena.

¡Has terminado de crear un final para tu juego! Ingrese al modo de reproducción para probarlo. No olvide salir del modo de reproducción cuando haya terminado; el cierre automático no funcionará hasta que crees una versión de tu juego en el tutorial final.

14. Resumen

En este tutorial, creaste un sistema para terminar el juego. Para hacer esto, usó características de la interfaz de usuario, agregó un disparador y escribió otro script personalizado. ¡Ahora tienes un juego que funciona, pero una casa espeluznante no es nada sin criaturas extrañas que lo atormentan! A continuación, crearás algunos enemigos estáticos para poblar tu juego.

Enemigos, Parte 1: Observadores estáticos

Objetivo

Ahora ha pasado mucho tiempo desarrollando su juego, mejorando su comprensión de las secuencias de comandos en C# en el camino. Pero falta algo: ¿qué es un juego de sigilo sin enemigos que evadir? En este tutorial, usted:

- Crea un enemigo gárgola estático
- Escriba un script personalizado para que la Gárgola pueda encontrar a JohnLemon
- Configura el juego para reiniciar si JohnLemon es atrapado

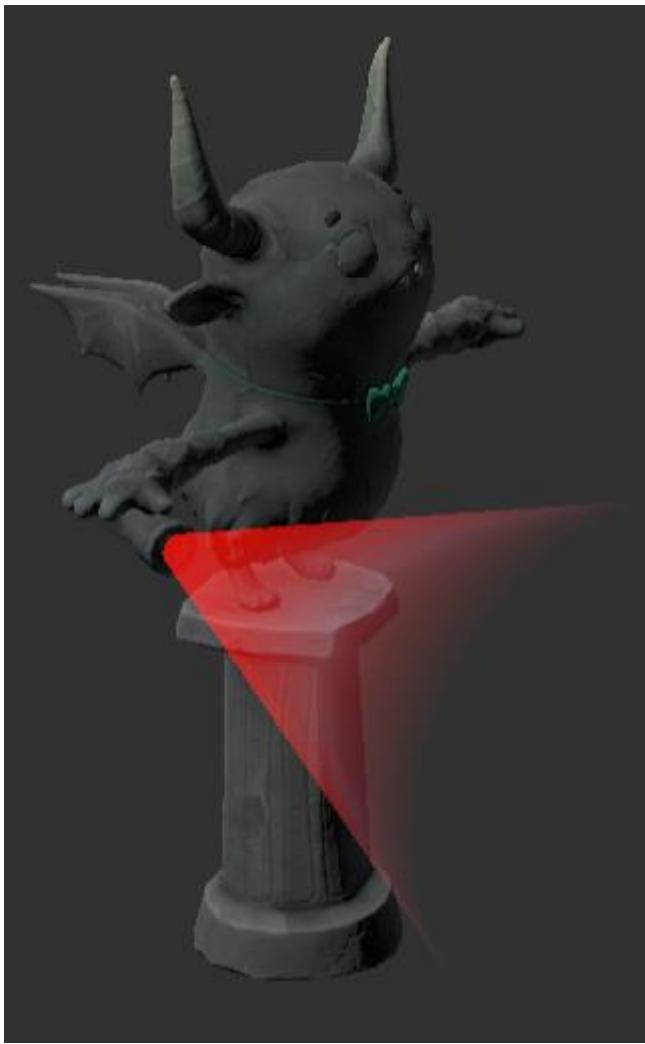
1. Configuración del prefabricado de gárgola

Tu juego ahora está tomando forma: JohnLemon puede moverse por un entorno con la cámara siguiéndolo, y cuando sale de la casa, el juego termina. Sin embargo, falta una cosa: ¡enemigos para hacer de eso un desafío!

En este tutorial, agregará un enemigo estático de gárgola a la casa embrujada.

El primer paso es traer el modelo Gargoyle a la escena, exactamente como lo hiciste con JohnLemon:

1. En la ventana Proyecto, navegue a la carpeta **Assets > Models** y busque el Activo **Gárgola**.



2. Arrastre el Activo desde la ventana Proyecto a la ventana Jerarquía, para crear una instancia del modelo.
3. Esta casa embrujada necesita múltiples gárgolas. Recuerde, un modelo es de solo lectura: para realizar ediciones deberá crear un Prefab. Arrastre el GameObject Gargoyle desde la ventana Jerarquía a la carpeta **Assets> Prefabs** en la ventana Proyecto. Cuando aparezca el cuadro de diálogo Crear prefabricado, seleccione **Original Prefab** .
4. Ahora que tiene un Prefab de Gargoyle, puede abrirlo para editarlo. ¡Esta vez usarás un atajo! En la Jerarquía, **haz clic en la flecha a la derecha del GameObject de Gargoyle** :



El Prefab se abrirá para editarlo.

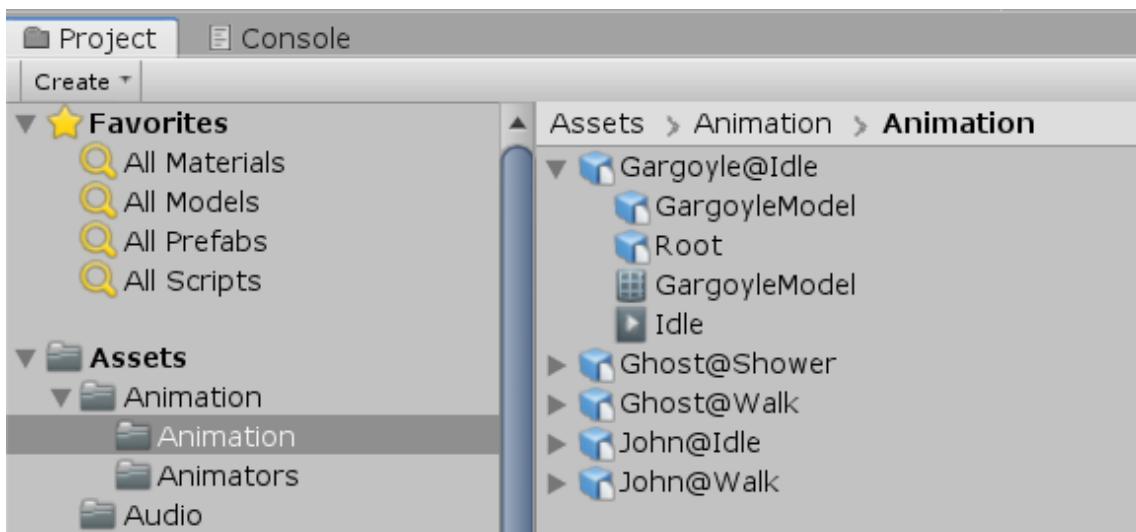
Ahora que ha agregado la Gárgola a su juego, es hora de configurarlo. Primero, ¡animémoslo!

2. Animar la gárgola

Al igual que JohnLemon, la Gárgola tiene un movimiento característico que le da personalidad y aumenta la atmósfera del juego. ¡Intentará atrapar al jugador!

La gárgola solo tiene una animación para reproducir, por lo que su controlador de animación será muy simple:

1. En la ventana Proyecto, vaya a la carpeta **Activos> Animación> Animadores** y haga clic derecho sobre ella.
2. En el menú contextual, seleccione **Crear> Controlador de animador**. Nombre del nuevo controlador de animación "**Gárgola**" .
3. Haga doble clic en **Gargoyle** para abrir la ventana Animator.
4. En la ventana Proyecto, navegue a **Activos> Animación> Animaciones** .
5. Expanda el activo de modelo **Gárgola @ inactivo** .



6. Arrastre la animación **Inactiva** desde la ventana Proyecto a la ventana Animador. Esto creará un estado de animación inactivo. El sencillo controlador de animación de la gárgola se ha completado, pero aún debe asignarse al componente animador.
7. En la ventana Proyecto, vaya a la carpeta **Activos> Animación> Animadores** .
8. En la ventana Jerarquía, seleccione el GameObject **Gargoyle** .

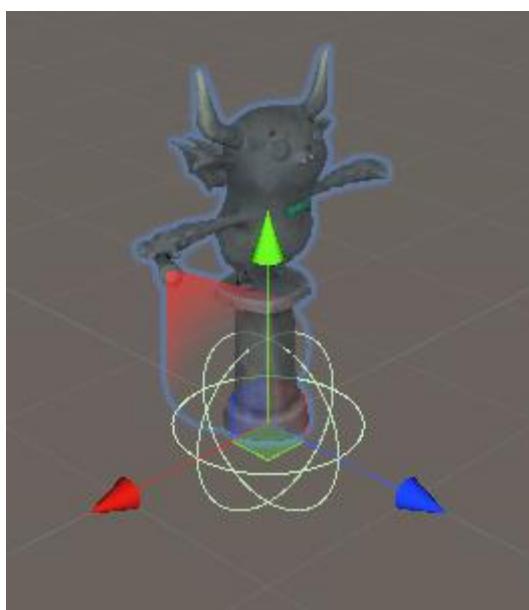
9. Arrastre el Gargoyle Animator Controller desde la ventana del Proyecto a la propiedad **Controller** del Gargoyle's Animator Component en el Inspector.
10. Guarde la escena, luego ingrese al modo de reproducción para ver cómo se ve su gárgola en acción. Asegúrese de salir del modo de reproducción cuando haya terminado.

3. Agregar un colisionador a la gárgola

Luego, debes asegurarte de que el jugador pueda toparse con Gargoyle y que pueda ver a JohnLemon con su haz de luz de la linterna; este es un trabajo para los Colliders.

Para agregar el Collider:

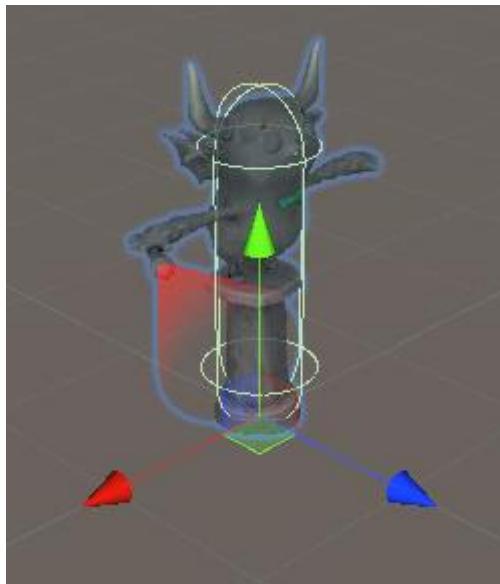
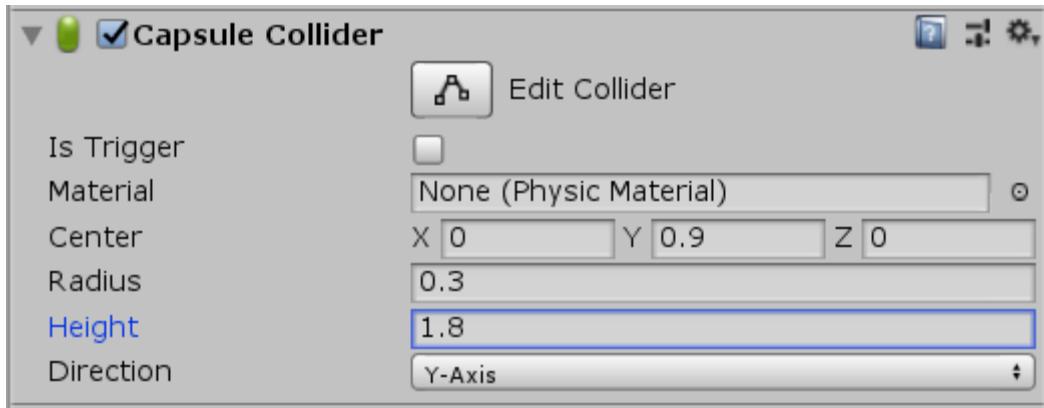
1. Asegúrese de que Unity Editor todavía esté en modo prefabricado. De lo contrario, puede usar el acceso directo que aprendió cuando creó el Garfayle Prefab.
2. En el Inspector, agregue un **componente Capsule Collider** al Gargoyle GameObject.



Un Capsule Collider tiene la forma correcta para este enemigo.

3. Ahora ajuste la configuración para que se ajuste mejor al modelo Gargoyle:

- Cambie la propiedad del **Centro** de colisionadores de cápsulas a (**0 , 0.9 , 0**)
- Cambie la propiedad **Radio** a **0.3**
- Cambie la propiedad **Altura** a **1.8**

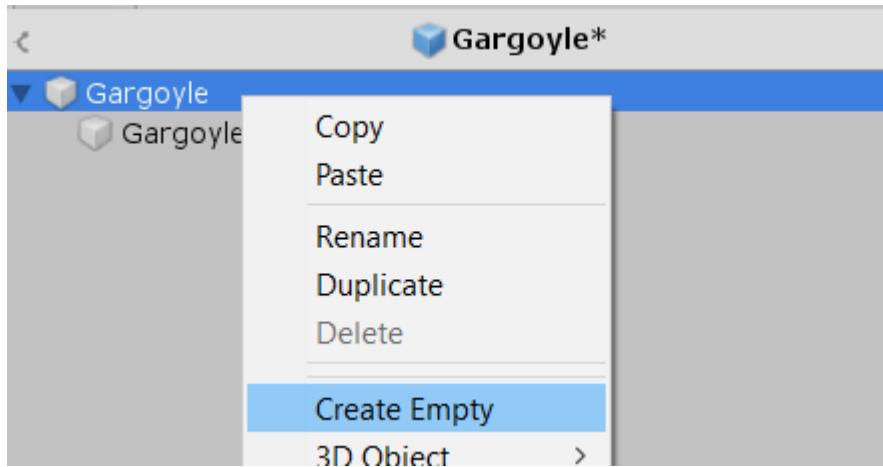


¡Eso es mucho mejor! Ahora el jugador puede toparse con la gárgola.

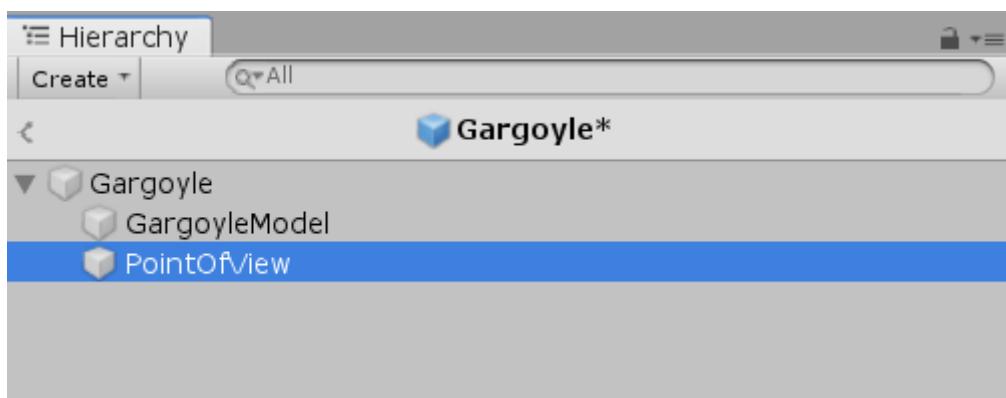
4. Crea un disparador para simular la línea de visión de la gárgola

La gárgola todavía necesita una forma de detectar a JohnLemon. Has utilizado la ubicación física del personaje para desencadenar eventos antes al crear el final de tu juego; Puedes hacer lo mismo aquí. Vas a escribir un script personalizado para asegurarte de que la Gárgola no pueda ver a través de las paredes, pero primero debes crear un Disparador. Para facilitar un poco el posicionamiento, creará otro GameObject como elemento secundario del Gargoyle GameObject y colocará el Trigger en él. Para crear el disparador:

1. En la ventana Jerarquía, haga clic con el botón derecho en Gargoyle GameObject y seleccione Create Empty.

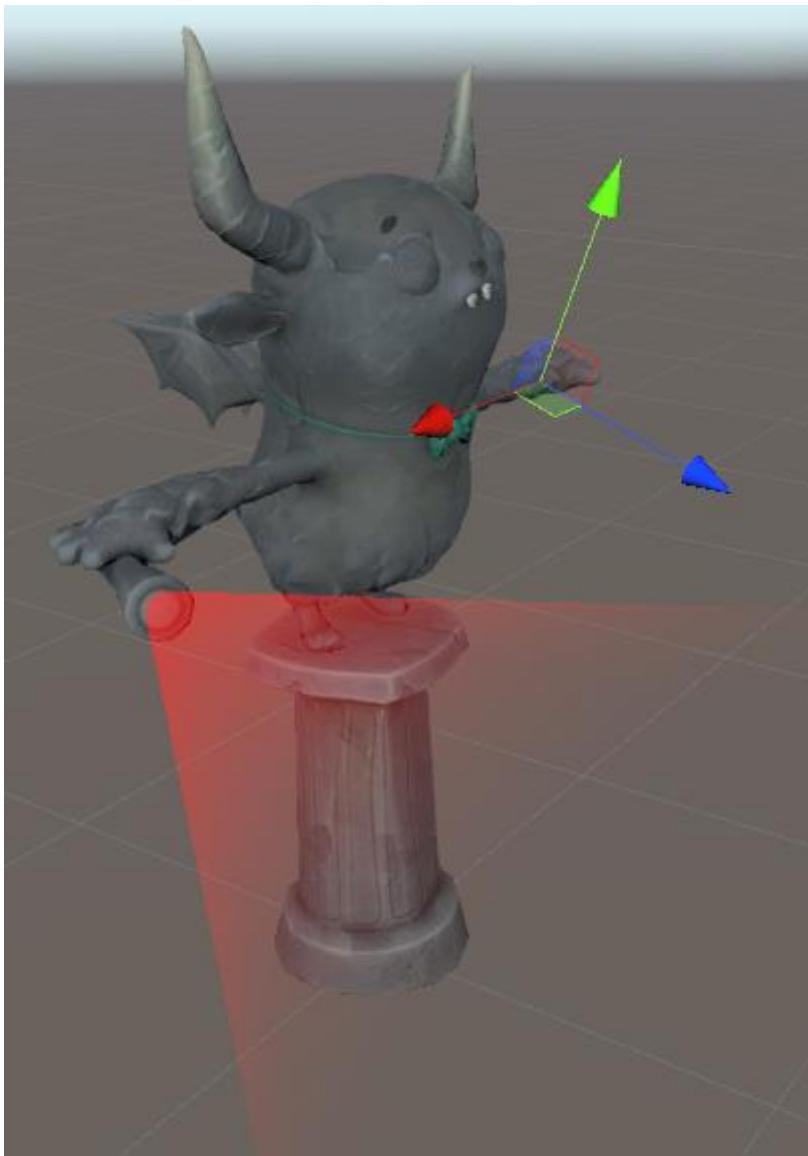


2. Este GameObject actuará como el punto de vista de la Gárgola en el mundo, así que cámbiele el nombre por "**PointOfView**" .



3. La animación Gargoyle lo muestra mirando por delante de su cara y ligeramente hacia abajo, por lo que debe reposicionar el PointOfView GameObject. En el Inspector, encuentre su componente **Transformar** .

- Cambie la propiedad **Posición** a (**0 , 1.4 , 0.4**)
- Cambie la propiedad de **rotación** a (**20 , 0 , 0**)



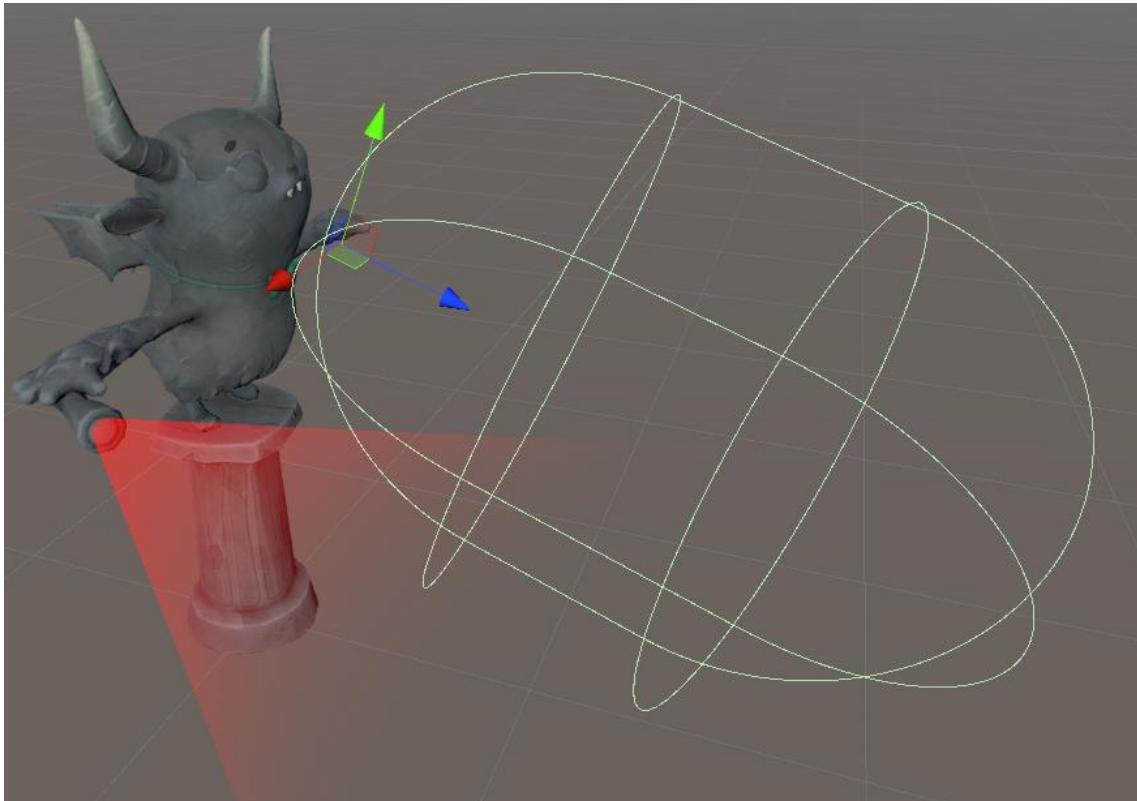
Nota: Si los controladores de transformación en la ventana de escena no apuntan ligeramente hacia abajo, podría deberse a que están configurados en Global en lugar de Local . Puede cambiar esto haciendo clic en la palanca de rotación de la manija en la barra de herramientas:



Ahora que ha configurado el punto de vista de Gargoyle, puede agregar y configurar el disparador.

4. En el Inspector, agregue un componente **Capsule Collider** .
 5. Active la casilla de verificación **Is Trigger** del componente .
 6. Vamos a ajustar la configuración del disparador.
- Cambie la propiedad del **Centro** de colisionadores de cápsulas a (**0 , 0 , 0.95**)

- Cambie la propiedad **Radio** a **0.7**
- Cambie la propiedad **Altura** a **2**
- Cambie la propiedad **Dirección** de Eje Y a Z



Has configurado el Collider para que coincida con el haz de la linterna de la gárgola, ¡perfecto!

5. Escriba un script de observador personalizado

Ahora que ha creado un Trigger, debe escribir un script para lo que sucede cuando JohnLemon entra en él.

Para crear el nuevo script:

1. En la ventana Proyecto, vaya a **Activos> Scripts** .
2. Haga clic derecho en la carpeta Scripts y seleccione **Crear> C # Script** . Nombre el nuevo Script "**Observer**". Este nombre describe lo que hará el Script: observar el personaje del jugador y hacer que el juego se reinicie si se detecta. También podrás reutilizar este script para otros enemigos, por lo que tiene sentido no vincular su nombre con la Gárgola.

3. Arrastre el Activo de script Observer desde la carpeta Scripts al **PointOfView** GameObject en la ventana Jerarquía para agregarlo como componente.
- 4) Haga doble clic en el activo del script para abrirlo y editarlo.

6. Agregue una clase para detectar el personaje del jugador

Comencemos su nuevo script agregando una clase para detectar el personaje del jugador:

1. Como lo hizo para el script GameEnding, elimine los métodos de Inicio y Actualización y sus comentarios. Su script limpio debería verse así:

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

public class Observer : MonoBehaviour
{
}
```

2. Primero, necesitas la clase Observador para detectar el personaje del jugador. Agregue la siguiente línea a su script, entre llaves:

```
public Transform player;
```

Esto es un poco diferente al enfoque que tomaste para GameEnding Trigger. Este script verificará la **Transformación** del personaje del jugador en lugar de su GameObject. Hará que sea más fácil acceder a la posición de JohnLemon y determinar si hay una línea de visión clara para él.

3. Anteriormente usó el método especial OnTriggerEnter para detectar el personaje del jugador y almacenó si el personaje estaba dentro del Trigger usando una variable bool. Eso puede ser reutilizado aquí. Agregue la siguiente línea directamente debajo de la declaración del jugador Transformar:

```
bool m_IsPlayerInRange;
```

4. Ahora agregue el método OnTriggerEnter directamente debajo de eso:

```
void OnTriggerEnter (Collider other)
{
}
```

5. Al igual que con el script GameEnding, es importante verificar que JohnLemon esté realmente dentro del rango cada vez que se llama a OnTriggerEnter. Agregue una instrucción if dentro de las llaves del método OnTriggerEnter para verificar que:

```
if(other.transform == player)
{
    m_IsPlayerInRange = true;
}
```

6) El personaje del jugador que ingresa a este Trigger puede no significar automáticamente el final del juego; por ejemplo, puede haber un muro en el camino. Esto significa que también es importante detectar cuándo JohnLemon deja el Disparador. Esto se puede hacer de manera simple con el método especial OnTriggerExit, que es lo opuesto a OnTriggerEnter. Copie y pegue el método OnTriggerEnter directamente debajo de sí mismo, luego ajuste Enter a Exit y true a false de la siguiente manera:

```
void OnTriggerExit (Collider other)
{
    if(other.transform == player)
    {
        m_IsPlayerInRange = false;
    }
}
```

7. Su script debería verse actualmente así:

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

public class Observer : MonoBehaviour
{
    public Transform player;

    bool m_IsPlayerInRange;

    void OnTriggerEnter (Collider other)
    {
        if (other.transform == player)
        {
            m_IsPlayerInRange = true;
```

```

        }
    }

    void OnTriggerExit (Collider other)
    {
        if (other.transform == player)
        {
            m_IsPlayerInRange = false;
        }
    }
}

```

7. Comprueba que la línea de visión del enemigo está despejada

En este juego, es importante verificar que la línea de visión del enemigo hacia JohnLemon esté despejada. De lo contrario, el final del juego podría activarse cuando en realidad hay un muro en el camino. Dado que la posición del personaje jugador podría cambiar en cualquier momento, esta verificación debe realizarse en cada cuadro. Continuemos su script:

1. Agregue un método de actualización inmediatamente debajo de OnTriggerExit, de la siguiente manera:

```

void Update ()
{
}

```

2. Solo tiene sentido comprobar la línea de visión cuando el personaje del jugador está realmente dentro del alcance. Agregue la siguiente instrucción if dentro de las llaves del método Update:

```

if(m_IsPlayerInRange)
{
}

```

Hasta ahora todo bien, pero ¿cómo puedes verificar la línea de visión? En Unity, es posible verificar si hay colisionadores a lo largo del camino de una línea que comienza desde un punto. Esta línea que comienza en un punto específico se llama **Ray**. La búsqueda de colisionadores a lo largo de este rayo se llama **Raycast**. Tu Ray necesita un **origen** y una **dirección**. El origen es solo la posición del PointOfView GameObject, pero calcular la dirección es un poco más complicado.

3. Agregue la siguiente línea de código dentro de IsPlayerInRange si las llaves de la declaración:

```
Vector3 direction = player.position - transform.position +  
Vector3.up;
```

Este código crea un nuevo Vector3 llamado **dirección**. De las matemáticas vectoriales, sabemos que un vector de A a B es B - A. Esto significa que la dirección desde el GameObject PointOfView a JohnLemon es la posición de JohnLemon menos la posición del GameObject PointOfView. Quizás recuerdes que la posición de JohnLemon está en el suelo, entre sus pies. Para asegurarte de que el observador pueda ver el centro de masa de JohnLemon, estás apuntando la dirección hacia arriba una unidad agregando Vector3.up. Vector3.up es un atajo para (0, 1, 0).

4. Ahora tiene la dirección, puede crear un Rayo. Agregue la siguiente línea de código debajo de la variable de dirección:

```
Ray ray = new Ray (transform.position, direction);
```

Esta línea incluye una palabra clave que no ha conocido antes: **nuevo**. Esta palabra clave se usa al crear una nueva instancia de algo llamando a un método especial llamado **constructor** del tipo . Llamar a un constructor siempre usa la siguiente sintaxis:

- La palabra clave **new**
- El tipo
- Paréntesis
- Parámetros para el constructor (en este caso, el origen y la dirección del Rayo)

5. Ahora que ha creado un Ray, puede realizar Raycast. Hay muchos métodos diferentes de Raycast en Unity, pero todos tienen dos cosas en común: necesitan definir el Rayo a lo largo del cual ocurre Raycast y restricciones sobre qué tipo de Colliders quieren detectar. El método Raycast que usará devuelve un valor bool que es verdadero cuando ha golpeado algo y falso cuando no ha golpeado nada. Debido a que devuelve un valor bool, es muy conveniente poner el método Raycast dentro de una instrucción if. El bloque de código de la instrucción if solo se ejecutará si Raycast ha golpeado algo. A continuación, donde se creó Ray, y aún dentro de la declaración if anterior que escribió, agregue el siguiente código:

```
if(Physics.Raycast(ray))  
{
```

```
    }
```

6. Ha definido un Rayo, y no está restringiendo los colisionadores que Raycast puede detectar. Las restricciones a los colisionadores tienden a funcionar más como filtros que a la identificación de colisionadores individuales. Lo que necesita es información sobre exactamente lo que recibe el Raycast, para que pueda verificar si este es el personaje del jugador. Pero hay un problema: la forma de obtener información de un método es con su retorno y este Raycast solo devuelve un bool. Sin embargo, afortunadamente hay un método Raycast muy similar que utiliza un **parámetro out**. Los parámetros de salida funcionan como parámetros normales, excepto que tienen la palabra clave escrita antes que ellos. Estos parámetros tienen sus valores alterados o establecidos por su método, para que puedan ser utilizados por lo que sea que lo llame. El parámetro out tiene un tipo llamado **RaycastHit**, y el método Raycast establece sus datos en información sobre cualquier golpe de Raycast. Entre la línea que crea la instrucción Ray y Raycast if, agregue el siguiente código:

```
RaycastHit raycastHit;
```

Esta línea define la variable RaycastHit.

7. Luego, debe cambiar el método Raycast para hacer uso de esta variable. Cambie la instrucción if con el método Raycast a lo siguiente:

```
if(Physics.Raycast(ray, out raycastHit))  
{  
    }
```

Este es un método de Raycast diferente, que funciona de manera muy similar pero utiliza el parámetro out para devolver información.

8. El script ahora puede identificar que el personaje del jugador está dentro del alcance, realizar un Raycast y saber si se ha golpeado algo. A continuación, debe verificar qué ha sido golpeado.

Dentro de la declaración if de Raycast, agregue el siguiente código:

```
if(raycastHit.collider.transform == player)  
{  
    }
```

9. Ahora necesitas terminar el juego. Para hacer esto, necesitará una referencia a la clase GameEnding. Entre el reproductor y las declaraciones de variables m_IsPlayerInRange hacia la parte superior del script, agregue el siguiente código:

```
public GameEnding gameEnding;
```

GameEnding es una clase como GameObject y Transform: cada uno tiene diferentes métodos y variables, pero hacer referencia a ellos funciona exactamente igual.

10. ¡Tu script de Observer está casi terminado! Repasemos lo que tienes hasta ahora:

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

public class Observer : MonoBehaviour
{
    public Transform player;
    public GameEnding gameEnding;

    bool m_IsPlayerInRange;

    void OnTriggerEnter (Collider other)
    {
        if (other.transform == player)
        {
            m_IsPlayerInRange = true;
        }
    }

    void OnTriggerExit (Collider other)
    {
        if (other.transform == player)
        {
            m_IsPlayerInRange = false;
        }
    }

    void Update ()
    {
        if (m_IsPlayerInRange)
        {
            Vector3 direction = player.position -
transform.position + Vector3.up;
```

```

        Ray ray = new Ray(transform.position, direction);
        RaycastHit raycastHit;

        if(Physics.Raycast(ray, out raycastHit))
        {
            if (raycastHit.collider.transform == player)
            {

            }
        }
    }
}

```

11. En este momento, esta clase GameEnding no tiene nada que pedir cuando el personaje del jugador ha sido atrapado por un enemigo. Necesita agregarle un poco más de funcionalidad.

Guarde el script de Observer y abra su script de GameEnding para editarlo. Es posible que pueda acceder a la secuencia de comandos GameEnding directamente desde su editor de código, o puede que tenga que volver a Unity Editor y abrirlo de la manera habitual.

8. Revise su secuencia de comandos GameEnding

Este juego necesita dos formas diferentes para finalizar el nivel: una si JohnLemon escapa y otra si es atrapado. Actualmente, solo su escape está incluido en el script GameEnding. En esta sección, revisarás tu script para permitir que los jugadores atrapados como enemigos reinicien el nivel en lugar de abandonar el juego. Antes de comenzar, revise su secuencia de comandos GameEnding:

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class GameEnding : MonoBehaviour
{
    public float fadeDuration = 1f;
    public float displayImageDuration = 1f;
    public GameObject player;
    public CanvasGroup exitBackgroundImageCanvasGroup;

    bool m_IsPlayerAtExit;
}

```

```

float m_Timer;

void OnTriggerEnter (Collider other)
{
    if (other.gameObject == player)
    {
        m_IsPlayerAtExit = true;
    }
}

void Update ()
{
    if(m_IsPlayerAtExit)
    {
        EndLevel ();
    }
}

void EndLevel ()
{
    m_Timer += Time.deltaTime;

    exitBackgroundImageCanvasGroup.alpha = m_Timer /
fadeDuration;

    if(m_Timer > fadeDuration + displayImageDuration)
    {
        Application.Quit ();
    }
}

```

9. Crea dos formas de terminar el nivel

Comencemos agregando las dos formas diferentes en que puede finalizar el nivel en su script:

1. Primero, agregue el siguiente código debajo de la **declaración de variable exitBackgroundImageCanvasGroup** :

```
public CanvasGroup caughtBackgroundImageCanvasGroup;
```

Esto creará otro CanvasGroup para las nuevas Imágenes que se mostrarán si JohnLemon ha sido capturado.

2. Agregue lo siguiente debajo de la **declaración de variable** **m_IsPlayerAtExit :**

```
bool m_IsPlayerCaught;
```

Esta variable verificará si JohnLemon ha sido capturado, de la misma manera que usted creó uno para verificar si ha llegado a la salida.

3. Ahora tiene dos nuevas variables, pero ¿cómo exactamente debe usarlas? El nivel debería terminar cuando JohnLemon es atrapado, pero de una manera diferente. Primero, agreguemos una **instrucción else-if** al método Update. Una declaración else-if se puede usar después de una declaración if, para continuar verificando cosas. Agregue el siguiente código debajo de la instrucción if en el método Update:

```
else if(m_IsPlayerCaught)
{
    EndLevel ();
}
```

Su método de actualización ahora debería verse así:

```
void Update ()
{
    if (m_IsPlayerAtExit)
    {
        EndLevel ();
    }
    else if (m_IsPlayerCaught)
    {
        EndLevel ();
    }
}
```

Este método ahora está instruyendo a la computadora: "Si m_IsPlayerAtExit es verdadero, llame a EndLevel. Si no es cierto, compruebe si m_IsPlayerCaught es verdadero; si es así, llame a EndLevel ". Esto necesita ser ajustado, ya que ambos métodos son actualmente iguales. Si m_IsPlayerAtExit es verdadero, debe desvanecerse en el exitBackgroundImageCanvasGroup. Si m_IsPlayerCaught es verdadero, entonces necesita desvanecerse en el capturadoBackgroundImageCanvasGroup .

4. Para solucionar esto, debe introducir un parámetro CanvasGroup en el método EndLevel, para que pueda cambiar el alfa de este nuevo parámetro. Cambie el método EndLevel de la siguiente manera:

```
void EndLevel (CanvasGroup imageCanvasGroup)
{
    m_Timer += Time.deltaTime;

    imageCanvasGroup.alpha = m_Timer / fadeDuration;

    if(m_Timer > fadeDuration + displayImageDuration)
    {
        Application.Quit ();
    }
}
```

En lugar de cambiar el alfa de exitBackgroundImageCanvasGroup, el script ahora cambiará el alfa de lo que se pase como parámetro.

5. Es posible que haya notado que su editor de código ahora le dice que hay un problema con las dos llamadas a EndLevel en nuestro método de actualización. Esto se debe a que su método EndLevel ahora requiere que pase un parámetro que falta. Para solucionar esto, debe pasar (asignar) el parámetro exitBackgroundImageCanvasGroup a la primera llamada, y pasar el catchBackgroundImageCanvasGroup a la segunda llamada. Ajuste su método de actualización de la siguiente manera:

```
void Update ()
{
    if (m_IsPlayerAtExit)
    {
        EndLevel (exitBackgroundImageCanvasGroup);
    }
    else if (m_IsPlayerCaught)
    {
        EndLevel (caughtBackgroundImageCanvasGroup);
    }
}
```

Ahora hay dos formas diferentes de salir del juego, pero hay algunos ajustes más que aún debes hacer.

10. Permitir al jugador reiniciar el nivel

Actualmente, al finalizar el nivel siempre se cierra el juego. ¡Sería bastante molesto si cada vez que un enemigo atrapara a JohnLemon, el jugador tuviera que volver a abrir el juego! Reiniciar el nivel creará una experiencia de jugador mucho mejor. Para agregar esta funcionalidad:

1. Si el jugador escapa del nivel, el juego aún debería salir, por lo que necesitará otro parámetro para el método EndLevel para ayudarlo a decidir qué hacer. Agregue un parámetro bool al método EndLevel de la siguiente manera:

```
void EndLevel (CanvasGroup imageCanvasGroup, bool doRestart)
```

2. Antes de comenzar a usar este nuevo parámetro, asegúrese de que las llamadas al método EndLevel sean correctas. Si el personaje del jugador ha llegado a la salida, el juego debería terminar. Esto debería pasar **falso** como el parámetro:

```
EndLevel (exitBackgroundImageCanvasGroup, false);
```

Si JohnLemon es atrapado, el juego debería reiniciarse. Esto debería pasar **verdadero** como el parámetro:

```
EndLevel (caughtBackgroundImageCanvasGroup, true);
```

3. Ahora necesita usar este parámetro. Actualmente, la declaración if en el método EndLevel solo cierra el juego. Debes ajustar esto, por lo que solo sucede si el juego no se reiniciará. Cambie el bloque de código de la instrucción if de la siguiente manera:

```
if (doRestart)
{
}

}
else
{
    Application.Quit ();
}
```

Esta es una declaración **más**. Funciona como la instrucción else-if que utilizó anteriormente, pero es incondicional. Mientras la declaración if sea falsa, se ejecutará el bloque de código else. Esto significa que si el parámetro doRestart es falso, el juego finalizará.

4. Si doRestart es verdadero, el nivel debe recargarse. El concepto de niveles en Unity está representado por escenas, como exploró anteriormente en este

tutorial. La forma más fácil de reiniciar una escena es cargarla nuevamente. La mayor parte de la funcionalidad para lidiar con escenas está en un **espacio de nombres** diferente , por lo que deberá agregar eso a su secuencia de comandos antes de continuar. Los espacios de nombres son una forma de organizar el código para que partes de él solo estén disponibles cuando se requieran. Por defecto, los scripts creados en Unity tienen tres espacios de nombres incluidos en la parte superior:

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

Debajo de los tres espacios de nombres, agregue otro a la lista:

```
using UnityEngine.SceneManagement;
```

5. Ahora tiene acceso a las diferentes clases y métodos que necesita para recargar la escena actual. Dentro del bloque de código de instrucción if en blanco que creó, agregue el siguiente código:

```
    SceneManager.LoadScene(0);
```

Esto está llamando a un método estático llamado **LoadScene** , de la clase SceneManager. Recuerde, los métodos estáticos son aquellos que no requieren una instancia de la clase para ser llamados. El parámetro para este método es el índice de compilación de una escena. Explorará los índices de compilación y la configuración de compilación con un poco más de detalle en el último tutorial de esta serie. Por ahora, solo necesita saber que en C # y en la mayoría de los lenguajes de programación, las colecciones de elementos están indexadas y que estos índices comienzan en 0 y no en 1. Puede pensar en qué tan lejos debe llegar una colección para llegar a un elemento específico: el primer elemento está inmediatamente allí, por lo que debe pasar cero elementos para llegar a él. Como solo tiene una escena, es la primera en su colección de escenas y también el índice 0.

6. Ya casi has terminado con este script; lo último que debe hacer es configurar **realmente m_IsPlayerCaught** . Esto ayudará a ilustrar la diferencia entre elementos públicos y privados en una clase. Hasta ahora, ha utilizado variables públicas que son visibles en el Inspector y otras variables que no son visibles allí. Las cosas en una clase que no están marcadas como públicas son por defecto privadas. La verdadera diferencia entre los dos es que a las cosas públicas se puede acceder desde fuera de la clase y las cosas privadas no. La variable m_IsPlayerCaught no está marcada como pública y, por lo tanto, es

privada . Esta es la variable que debe establecerse desde fuera de la clase para activar el reinicio del nivel. Su primera opción es hacer pública la variable, pero no es un buen hábito. No todas las clases necesitan saber si JohnLemon ha sido atrapado. En cambio, puede limitar el acceso a esta variable y crear un método público que lo establezca en verdadero. Al hacerlo de esta manera, otras clases pueden ver que JohnLemon ha sido atrapado, pero no que no lo hayan atrapado. Este método puede ir a cualquier parte de la clase, pero tiene más sentido ponerlo al lado del método OnTriggerEnter. De esa manera, los dos métodos que pueden desencadenar el final del nivel están uno al lado del otro.

Agregue el siguiente método debajo de OnTriggerEnter:

```
public void CaughtPlayer ()
{
    m_IsPlayerCaught = true;
}
```

7. ¡Has terminado de revisar tu script de GameEnding! La secuencia de comandos completa debería ser similar a esto:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameEnding : MonoBehaviour
{
    public float fadeDuration = 1f;
    public float displayImageDuration = 1f;
    public GameObject player;
    public CanvasGroup exitBackgroundImageCanvasGroup;
    public CanvasGroup caughtBackgroundImageCanvasGroup;

    bool m_IsPlayerAtExit;
    bool m_IsPlayerCaught;
    float m_Timer;

    void OnTriggerEnter (Collider other)
    {
        if (other.gameObject == player)
        {
            m_IsPlayerAtExit = true;
        }
    }
}
```

```

public void CaughtPlayer ()
{
    m_IsPlayerCaught = true;
}

void Update ()
{
    if (m_IsPlayerAtExit)
    {
        EndLevel (exitBackgroundImageCanvasGroup, false);
    }
    else if (m_IsPlayerCaught)
    {
        EndLevel (caughtBackgroundImageCanvasGroup, true);
    }
}

void EndLevel (CanvasGroup imageCanvasGroup, bool doRestart)
{
    m_Timer += Time.deltaTime;
    imageCanvasGroup.alpha = m_Timer / fadeDuration;

    if (m_Timer > fadeDuration + displayImageDuration)
    {
        if (doRestart)
        {
            SceneManager.LoadScene (0);
        }
        else
        {
            Application.Quit ();
        }
    }
}

```

¡Ya casi estás allí! Guarde la secuencia de comandos GameEnding y vuelva a abrir la secuencia de comandos de Observer para realizar algunos ajustes finales.

11. Completa tu Prefab de gárgola

Su script de observador ya puede:

- Identifica cuándo el personaje jugador está en el gatillo
- Use Raycast y sepa si ha golpeado a un Collider

- Identifica si ese Collider era el personaje jugador

Ahora que ha ajustado el script GameEnding, el script Observer tiene algo que llamar cuando su Raycast golpea el Collider del personaje del jugador. Hay algunos ajustes finales para hacer antes de que puedas probar a tu nuevo enemigo en el juego:

1. En la última **declaración if** del **método Update**, agregue el siguiente código:

```
gameEnding.CaughtPlayer();
```

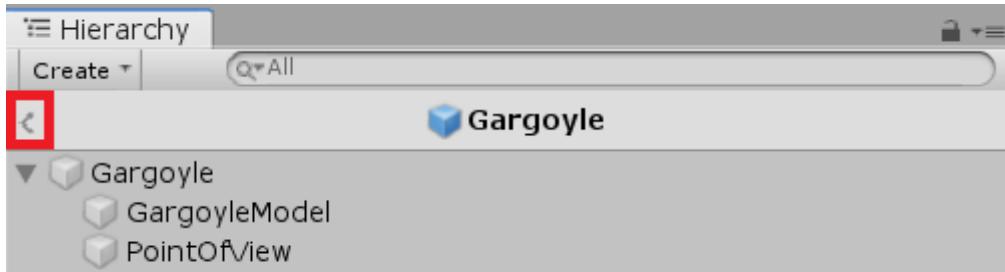
Puede agregar esto aquí porque el método CaughtPlayer es **público**. Si era privado, su editor de código informaría un error con esta línea.

2. Guarde el script del Observador y regrese a Unity.

3. ¡ Su prefabricada de gárgolas ya está terminada! Guarde el Prefab utilizando el botón Guardar.



4. En la Jerarquía, haga clic en la flecha hacia atrás para regresar a la Escena.



5. Puedes agregar más instancias prefabricadas de gárgola más tarde, pero pongamos la que tienes actualmente en la esquina de la sala de inicio. En el Inspector, encuentre su componente **Transformar**.

- Cambie la propiedad Posición a (-15.2 , 0 , 0.8)
- Cambie la propiedad de rotación a (0 , 135 , 0)

6. Ahora puede establecer las dos referencias que requiere el script Observer.

En la ventana Jerarquía, expanda el GameObject Gargoyle y seleccione el **GameObject PointOfView**.



7. Arrastre el objeto de juego JohnLemon desde la ventana Jerarquía al campo **Jugador** del script Observador en el Inspector. Esto asignará su transformación.

8. Arrastre GameOnding GameObject desde la ventana Jerarquía al campo **Game Ending** del script Observador en el Inspector.

9. A continuación, debe crear la interfaz de usuario para cuando finalice el nivel porque el jugador ha sido atrapado. Expanda el **FaderCanvas** GameObject en la ventana Jerarquía.



10. Haga clic con el botón derecho en **ExitImageBackground** GameObject y seleccione **Duplicar** en el menú contextual. El atajo para duplicar es Ctrl + D (Windows) o CMD + D (macOS).

11. Cambie el nombre de la nueva copia a **CaughtImageBackground**. En la ventana Jerarquía, expanda este GameObject para ver sus elementos secundarios.

12. Cambie el nombre del juego ExitImageObject a **CaughtImage**.

13. En la ventana del Inspector, debería ver que el componente Imagen todavía hace referencia al Won Sprite. Esta es la única configuración que necesita ajustar.

En el componente Imagen, haga clic en el botón de selección circular junto a la propiedad **Imagen de origen**. Cuando se abre el cuadro de diálogo, seleccione el Sprite llamado **Atrapado (Caugtht)**.

14. Finalmente, debe asignar una referencia al Grupo de lienzo de imagen de fondo captado en el script Final del juego. En la ventana Jerarquía, seleccione el **GameOnding** GameObject.

15. Arrastre el objeto de juego CaughtImageBackground desde la ventana Jerarquía al campo Grupo de lienzo de imagen de fondo capturado del componente Final del juego en el Inspector.

16. Guarda tu escena.

¡Tu gárgola está completa! Ahora puede ingresar al modo de reproducción para probarlo.

12. Resumen

En este tutorial, creaste un enemigo estático para tu juego. Usaste física y secuencias de comandos para asegurarte de que JohnLemon pudiera ser atrapado, y estableciste el nivel para reiniciar cuando lo está. Tu enemigo Gárgola es excelente, pero es estacionario: los enemigos que se mueven harán que el juego sea más interesante. En el próximo tutorial, crearás un enemigo Fantasma que puede deambular por un camino establecido y poblar todo el juego con enemigos.

Enemigos, Parte 2: Observadores dinámicos

Objetivo

Has creado un enemigo que John Lemon tiene que evitar, pero eso no parece ser un gran desafío. Es hora de crear otro enemigo que aumente el desafío para el jugador.

En este tutorial, usted:

- Crea un enemigo fantasma dinámico
- Escriba un script personalizado para que los fantasmas puedan patrullar la casa embrujada
- Llena tu juego de enemigos

Cuando hayas terminado el tutorial, tu juego estará lleno de enemigos, ¡y casi completo!

1. Configurar el fantasma prefab

En el tutorial anterior, creaste un enemigo Gárgola para la casa embrujada. Ahora vas a mejorar el desafío de tu nivel creando enemigos dinámicos (en movimiento). Esta vez, crearás un Fantasma para recorrer los pasillos de la casa en busca de JohnLemon.

El primer paso es usar el modelo Ghost para crear un Prefab: ahora debería estar bastante familiarizado con este proceso.

1. En la ventana Proyecto, abra la carpeta **Assets> Models> Characters** y busque el Activo **fantasma**.



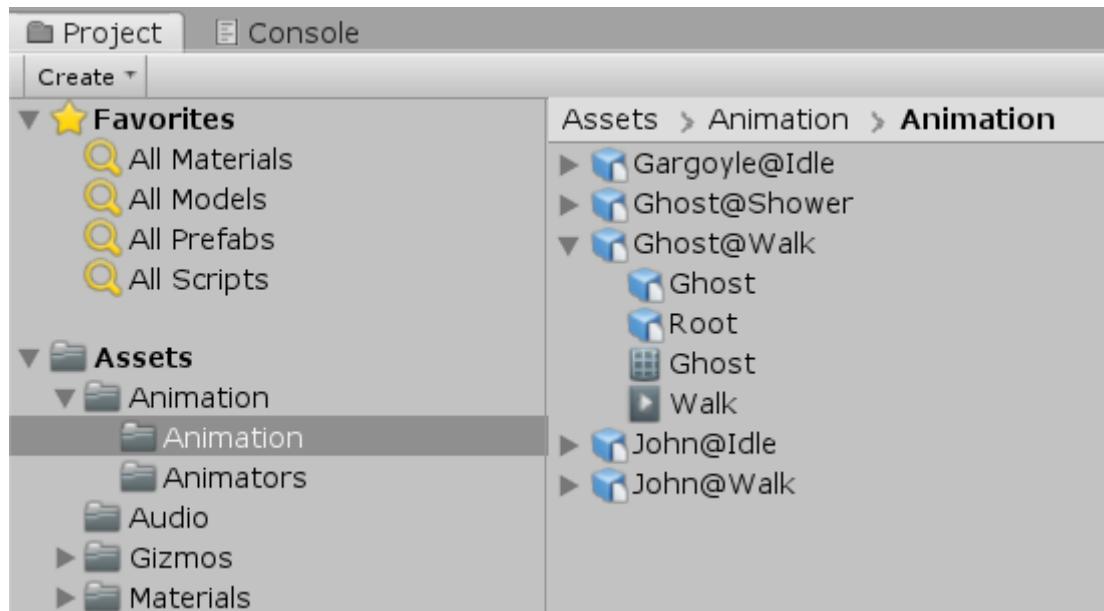
2. Arrastre el modelo Fantasma desde la ventana Proyecto a la ventana Jerarquía para crear una instancia del modelo.
3. Arrastre Ghost GameObject desde la ventana Jerarquía a la carpeta **Assets> Prefabs** en la ventana Proyecto. Cuando aparezca el cuadro de diálogo Crear prefabricado, seleccione **Original Prefab** .
4. Abra el Prefab para editar.

2. Animar al fantasma

El siguiente paso es animar a tu nuevo enemigo. El fantasma también tendrá un controlador de animación simple, ya que reproducirá una sola animación en bucle. Para animar al fantasma:

1. En la ventana Proyecto, vaya a la carpeta **Activos> Animación> Animadores** y haga clic derecho sobre ella.
2. En el menú contextual, seleccione **Crear> Controlador de animador**. Nombre el nuevo controlador de animación "**Fantasma**".

3. Haga doble clic en **Ghost** para abrir la ventana Animator.
4. En la ventana Proyecto, navegue a **Activos> Animación> Animaciones** .
5. Expanda el activo del modelo **Ghost @ Walk** .



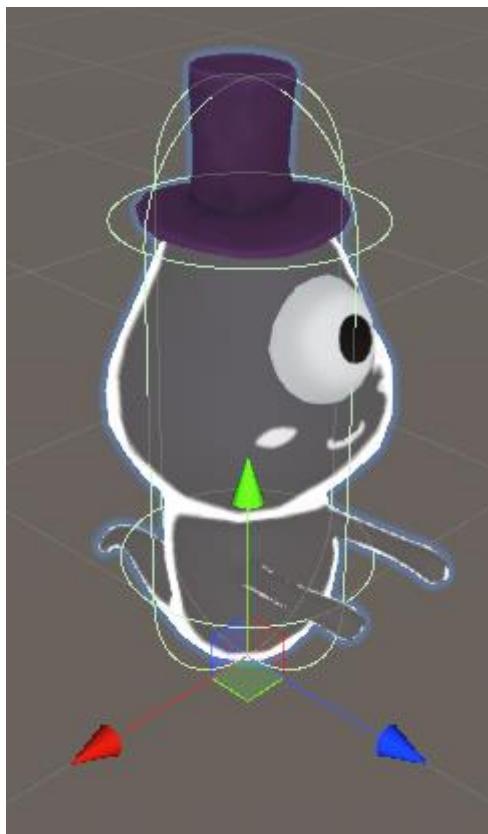
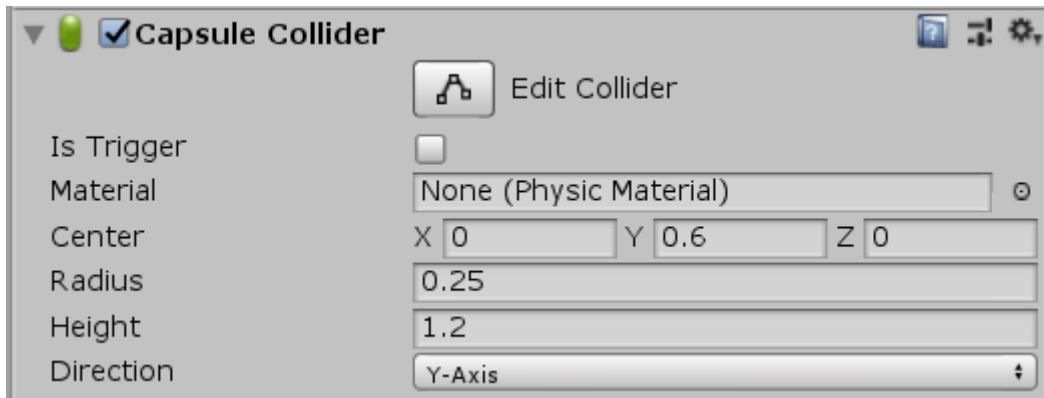
6. Arrastre la animación **Walk** desde la ventana Proyecto a la ventana Animador. El controlador de animación simple de Ghost está completo, pero aún debe asignarse al componente Animator.
7. En la ventana Proyecto, vaya a la carpeta **Activos> Animación> Animadores** .
8. En la ventana Jerarquía, seleccione **Ghost** GameObject.
9. Arrastre el controlador Ghost Animator desde la ventana del proyecto a la propiedad **Controller** del componente Ghost's Animator en el Inspector.
10. Guarde la escena.

3. Agregar un colisionador al fantasma

Al igual que JohnLemon y las Gárgolas (¡aunque tal vez sea un poco contradictorio!), Los Ghosts necesitan una presencia física en la escena. Esto significa que necesitarán un Collider. Para agregar el Collider:

1. Asegúrese de que Unity Editor todavía esté en modo Prefab. De lo contrario, puede usar el acceso directo que aprendió cuando creó el Garfayle Prefab.
2. En el Inspector, agregue un **componente Capsule Collider** al Ghost GameObject.
3. Ajustemos la configuración para que el Collider se ajuste mejor al modelo Ghost:

- Cambie la propiedad del **Centro** de colisionadores de cápsulas a (**0 , 0.6 , 0**)
- Cambie la propiedad **Radio** a **0.25**
- Cambie la propiedad **Altura** a **1.2**



¡Ahora el Collider encaja bien!

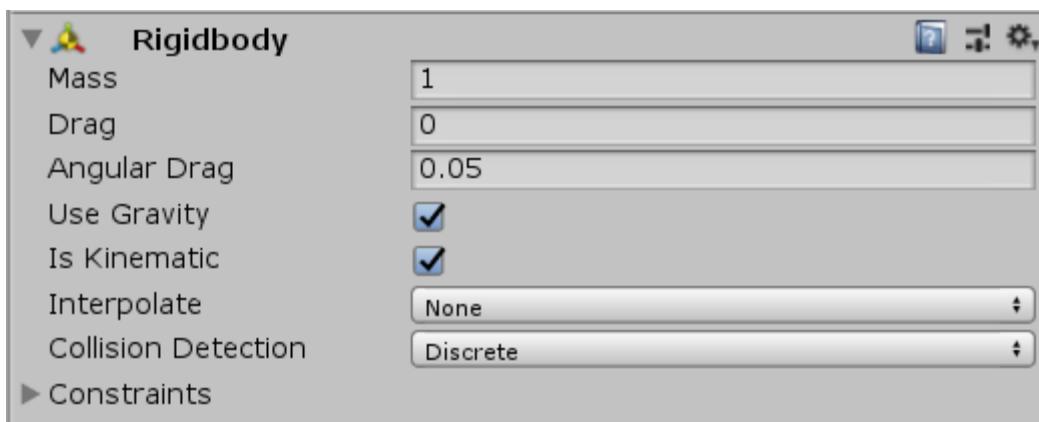
4. Agregue un componente Rigidbody al Ghost

GameObject

El fantasma también necesitará un cuerpo rígido, porque se está moviendo. Debe tener cuidado con la configuración para esto: una colisión con JohnLemon no debería causar ningún movimiento en el Fantasma.

Puede evitar el movimiento habilitando todas las Restricciones para la posición y la rotación, pero un enfoque más fácil es establecer el Cuerpo rígido como **cinemático**.

En el Inspector, busque el componente Ridigbody y **active** la casilla de verificación **Es cinemática** .



Un cuerpo rígido cinemático no puede verse afectado por fuerzas externas como colisiones, pero aún puede colisionar con otros GameObjects. La pala en el juego Breakout es un ejemplo perfecto de un cuerpo rígido cinemático: la pelota rebota en la pala, pero la pala no se ve afectada por el rebote.

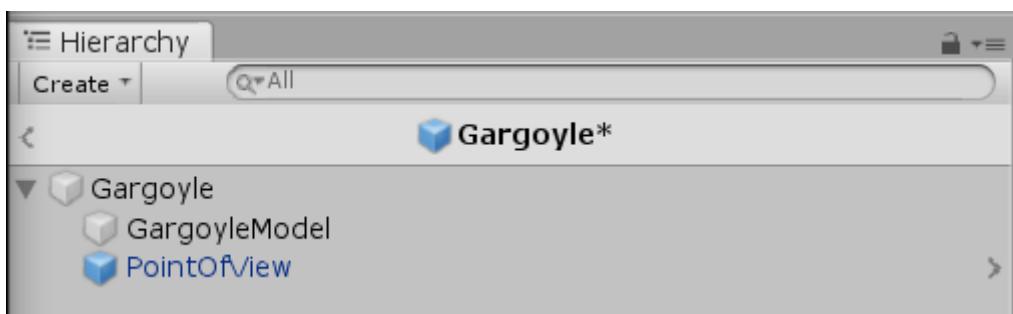
5. Haz del fantasma un observador

El Ghost Prefab ahora tiene todos los elementos básicos, pero actualmente no se moverá ni detectará a JohnLemon. Ya ha creado un GameObject PointOfView para el Prefab de Gargoyle que controla esto: ¡creemos un Prefab de eso, en lugar de repetir todo el trabajo! Para hacer del fantasma un observador:

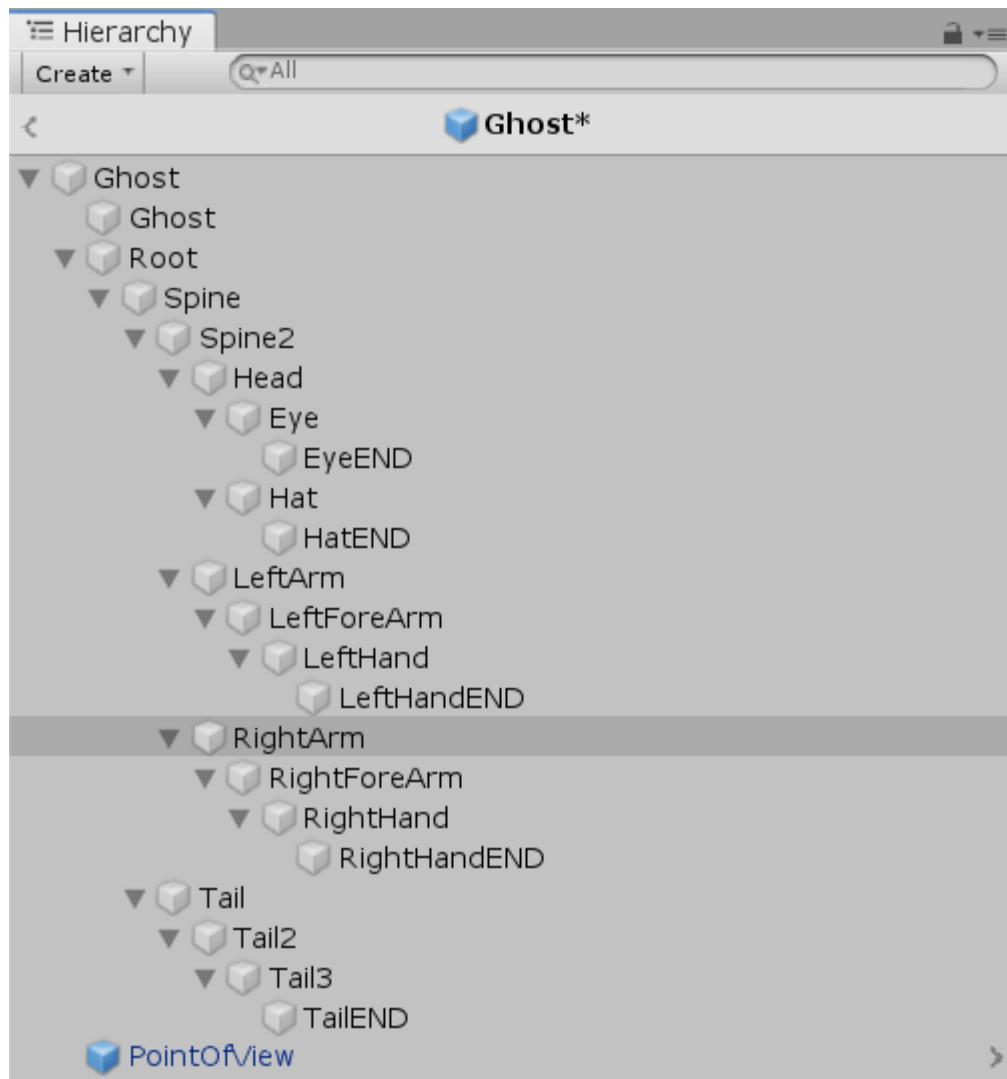
1. Primero, guarde sus cambios en Ghost Prefab.



2. En la ventana Proyecto, abra la carpeta **Assets> Prefabs** y seleccione el Prefab **Gargoyle** .
3. En la ventana del Inspector, haga clic en el botón **Abrir Prefab** .
4. En la Jerarquía, busque el **GameObject PointOfView** que creó como hijo de Gargoyle. Vas a convertir este GameObject en un Prefab, para que tanto los Prefabs de Gargoyle como Ghost puedan hacer referencia a él.
5. Arrastre el PointOfView GameObject desde la ventana Jerarquía a la carpeta **Assets> Prefabs** en la ventana Proyecto.
6. Debería ver que el GameObject PointOfView en la Jerarquía ahora está representado por un cubo azul en lugar de uno gris, y el nombre es azul en lugar de gris. ¡Esto se debe a que ahora es una instancia prefabricada!



7. Guarde la gárgola prefabricada.
8. En la ventana Proyecto, vaya a **Assets> Prebas** y seleccione el **Prefabricado fantasma**. Haga clic en el botón Abrir Prefab.
9. Arrastre el Prefab PointOfView desde la carpeta **Assets > Prefabs** al **Ghost GameObject** en la ventana Jerarquía.

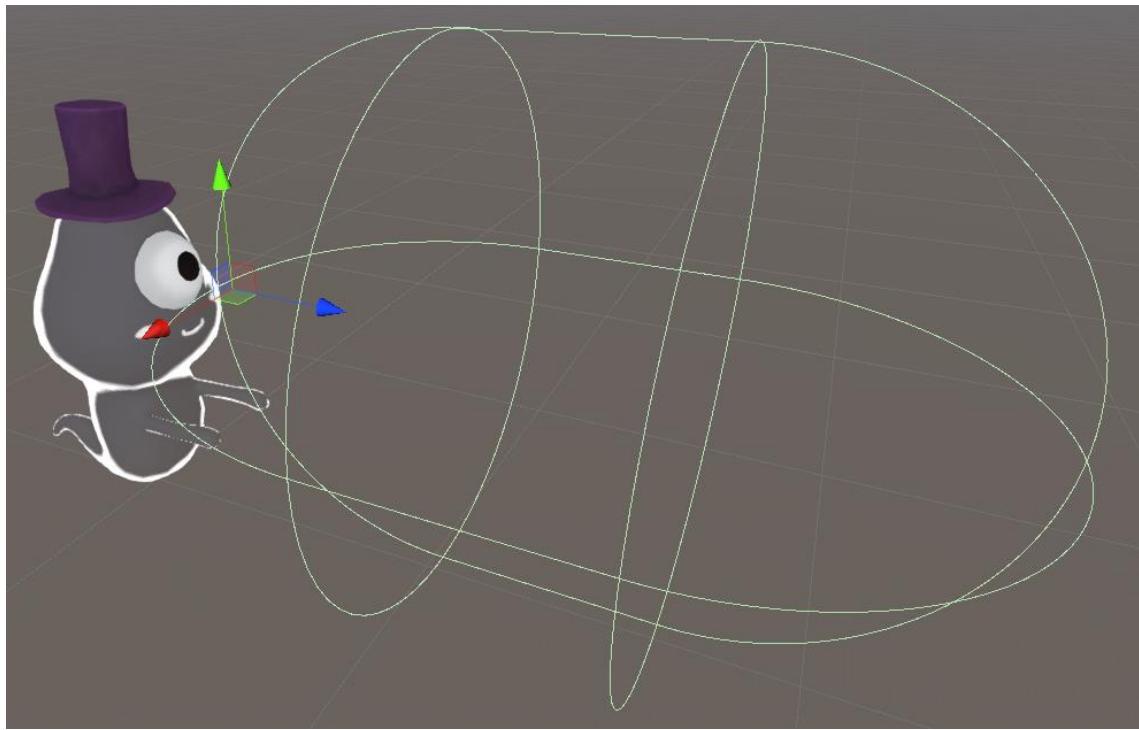


¡Ahora el Fantasma también tiene PointOfView y puede usar toda su funcionalidad! Sin embargo, hay un par de ajustes que hacer para que PointOfView funcione correctamente para este enemigo. La górgola es más alta que el fantasma y mira hacia abajo, mientras que el fantasma tendrá que mirar hacia adelante.

10. En la ventana Jerarquía, seleccione **PointOfView** GameObject.

11. En el Inspector, busque el componente Transformar.

- Cambie la propiedad **Posición** a (**0 , 0.75 , 0.4**)
- Cambie la propiedad de **rotación** a (**0 , 0 , 0**)



Ahora tu fantasma está configurado como observador. También puedes cambiar el Prefab PointOfView para actualizar a ambos enemigos en tu juego, si alguna vez necesitas hacer esto.

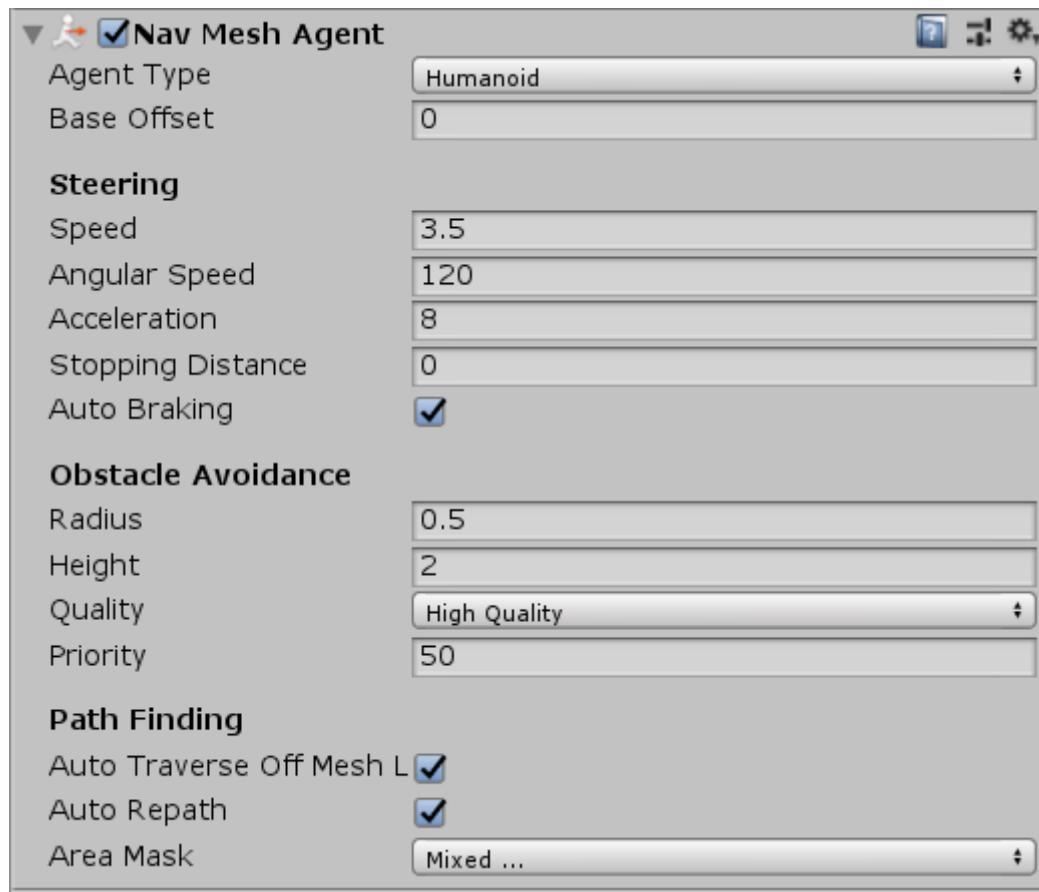
6. Configure un componente Nav Mesh Agent

Ahora estás listo para configurar el Fantasma para que pueda moverse por tu entorno de juego. Vas a usar dos componentes para hacer esto:

- Un agente de malla de navegación, que le permitirá al fantasma encontrar caminos alrededor de la malla de navegación que horneó en el [Tutorial de entorno](#)
- Un script que le dice al Nav Mesh Agent a dónde debe ir el fantasma

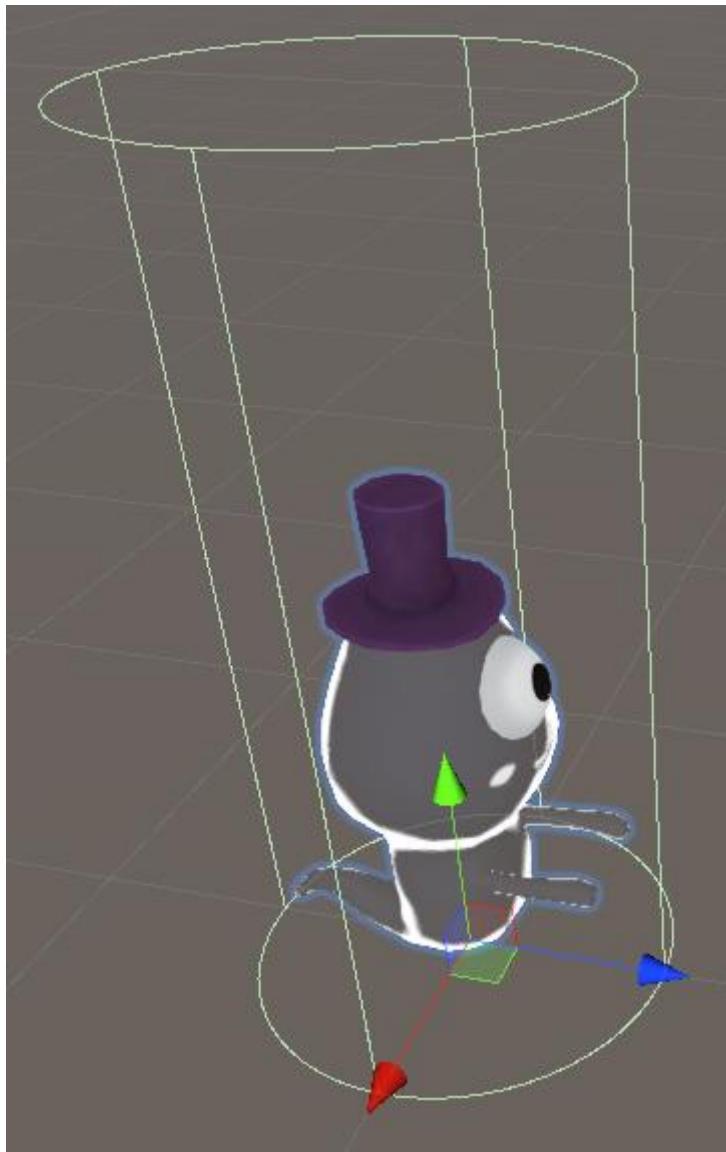
Comencemos creando un Nav Mesh Agent:

1. En el Inspector, agregue un componente **Nav Mesh Agent** al Ghost GameObject.



En su mayoría, podrá usar la configuración predeterminada para este componente, pero hay algunas que debe ajustar.

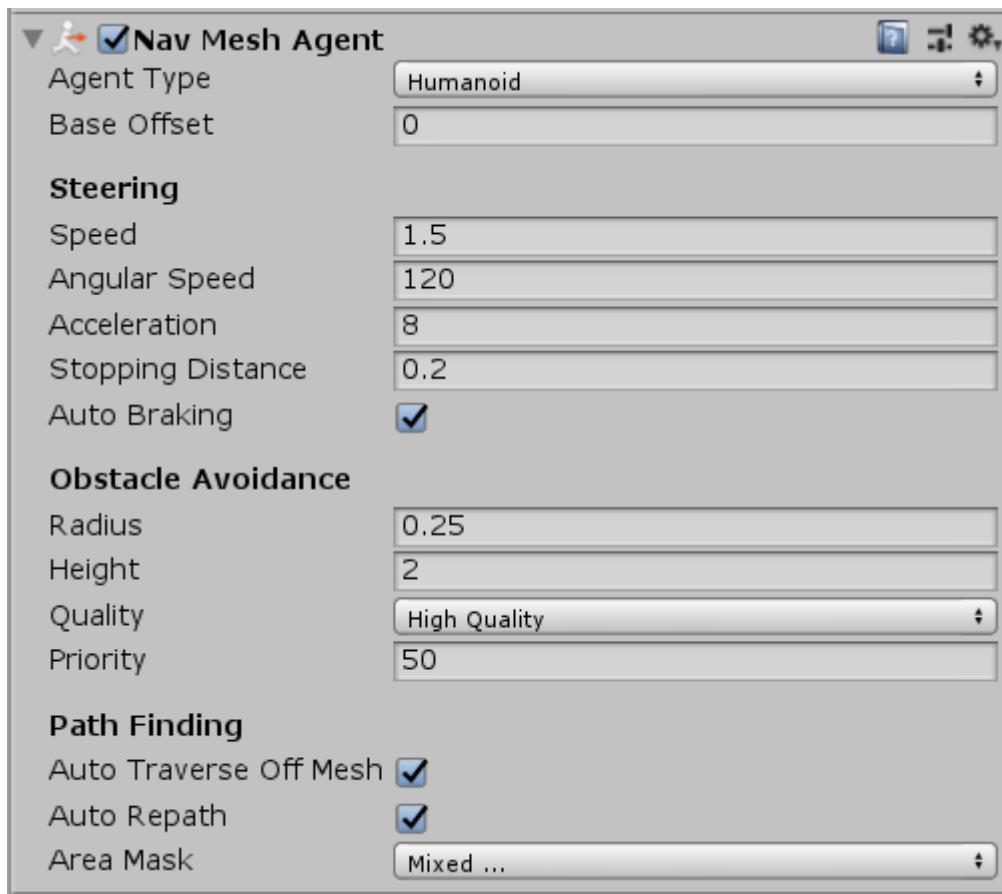
2. En la vista de escena, debería ver una representación cilíndrica del Nav Mesh Agent alrededor del fantasma.



Esto es un poco grande. En el Inspector, cambie la propiedad **Radius** del componente Nav Mesh Agent a **0.25**. La altura del agente también es bastante más alta que la del Fantasma, pero como el entorno no tiene techo, esto no importará.

3. Cambie la propiedad **Speed** del componente Nav Mesh Agent a **1.5**. La velocidad predeterminada de 3.5 metros por segundo es bastante rápida para un fantasma, y hará que tu juego sea demasiado difícil.

4. Cambie la propiedad **Stopping distance** del componente Nav Mesh Agent a **0.2**. No necesita demasiada precisión en lo cerca que están los puntos de referencia de los Ghosts, por lo que puede aumentar la distancia que el Agente de malla de navegación puede estar desde su destino y aún así se considera llegado.



5. Guarde sus cambios.

El Nav Mesh Agent ahora está configurado, pero no hará nada por sí mismo. Necesita tener su destino establecido, y para hacer esto necesita escribir un script.

7. Crear un nuevo script WaypointPatrol

Los Ghosts en tu juego se moverán patrullando en bucle a través de una colección de puntos de referencia, por lo que tiene sentido llamar a este script WaypointPatrol. No olvide guardar sus cambios a medida que avanza. Para crear el nuevo script:

En la ventana Proyecto, vaya a **Activos > Scripts**.

1. Haga clic derecho en la carpeta Scripts y seleccione **Crear > Script C #**. Nombre el nuevo script "**WaypointPatrol**".
2. En la Jerarquía, seleccione el Ghost GameObject.
3. Arrastre el script WaypointPatrol recién creado desde la ventana Proyecto a la ventana Inspector para agregarlo como un fantasma como componente.
4. Haga doble clic en el script WaypointPatrol para abrirlo y editarlo.

8. Establezca el destino del agente de malla de navegación

Antes de comenzar a editar el script, pensemos brevemente en lo que debe hacer. Debe establecer el destino del Nav Mesh Agent tanto cuando la escena se carga por primera vez como cuando el fantasma ha llegado a su destino. Para saber cuándo ha llegado a su destino, deberá verificar cada cuadro. Comencemos a editar su nuevo script:

1. Necesitará los métodos de Inicio y Actualización en este script, por lo que no tiene sentido eliminarlos como lo hizo anteriormente. En cambio, simplemente elimine los **comentarios** sobre los métodos de Inicio y Actualización. El script ahora debería verse así:

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

public class WaypointPatrol : MonoBehaviour
{
    void Start ()
    {

    }

    void Update ()
    {

}
}
```

2. Tendrá que decirle al agente Nav Mesh qué destino debe tener, por lo que necesitará una referencia. Para realizar secuencias de comandos con Nav Mesh Agent, debe incluir su espacio de nombres. En la parte superior del script con las otras directivas que usan, agregue el siguiente código:

```
using UnityEngine.AI;
```

La inclusión del espacio de nombres AI le dará acceso a la **clase NavMeshAgent**.

3. Sobre el método de Inicio, agregue lo siguiente:

```
public NavMeshAgent navMeshAgent;
```

Esta referencia pública al componente le permitirá asignar la referencia del Agente de malla de navegación en la ventana del Inspector.

4. A continuación, debe establecer los puntos de referencia que el Fantasma debe patrullar. El destino de un Agente de malla de navegación es un Vector3, una posición

en el espacio mundial. Pero si creó los puntos de referencia Vector3, tendrá que establecer manualmente todas las posiciones y esperar que los números que utilice sean precisos. En cambio, tiene sentido tener una colección de GameObjects vacíos y usar sus posiciones como puntos de referencia. Estos GameObjects se pueden mover por la escena mucho más fácilmente, haciendo que cualquier cambio que desee sea mucho más simple. Sin embargo, en lugar de tener referencias a una colección de GameObjects, solo puede tener referencias a sus componentes Transform.

Simplemente podría tener varias variables de Transformación públicas y luego configurar cada una por separado en la ventana del Inspector, pero esto no permite mucha flexibilidad en cuántos puntos de ruta puede tener cada Fantasma. En su lugar, puede usar algo llamado **matriz**. Una matriz es una colección básica de variables que existen juntas. Se definen utilizando corchetes. Agregue la siguiente línea justo debajo de la declaración de variable navMeshAgent

```
public Transform[] waypoints;
```

Esta línea de código declara una variable pública llamada **waypoints**, que es una matriz de transformaciones.

5. A continuación, agregue una línea a su método de Inicio para establecer el destino inicial del Nav Mesh Agent:

```
navMeshAgent.SetDestination(waypoints[0].position);
```

9. ¿Cómo funcionan las matrices?

Exploraremos esto para entender un poco más sobre cómo funcionan las matrices: Su línea de código está llamando al método SetDestination de su componente Nav Mesh Agent. Toma un Vector3 como parámetro, por lo que está utilizando la propiedad de posición del primer waypoint Transformar en la matriz.

Las cosas individuales que forman una matriz se llaman **elementos**. Se accede a los elementos individuales de una matriz utilizando su índice entre corchetes. Piense en el índice como la cantidad de elementos que necesita omitir desde el comienzo de la matriz para llegar al elemento que desea. No necesita omitir ningún elemento para acceder al primer elemento, por lo que su índice es **0**.

10. Agregar destinos adicionales

Sigamos agregando a su script:

1. Tu fantasma necesita pasar al siguiente punto de ruta cuando llegue al último que había establecido como destino. La forma más fácil de rastrear qué punto de ruta es el siguiente es

almacenando el índice actual de la matriz de puntos de ruta. Debajo de la declaración de matriz de puntos de referencia, agregue el siguiente código:

```
int m_CurrentWaypointIndex;
```

2. A continuación, pasemos al método Actualizar para que pueda usar este índice. En el método de actualización, debe realizar una comprobación: desea saber si Nav Mesh Agent ha llegado a su destino. Una manera fácil de verificar esto es ver si la distancia restante al destino es menor que la distancia de detención que configuró anteriormente en la ventana del Inspector. Agregue la siguiente instrucción if al método Update:

```
if(navMeshAgent.remainingDistance <  
navMeshAgent.stoppingDistance)  
{  
    [REDACTED]  
}
```

3. Ahora necesita actualizar el índice actual, y luego usarlo para configurar el destino del Nav Mesh Agent. Para hacer esto, usará un nuevo operador llamado **operador restante** que está representado por el carácter de porcentaje: %. Agregue el siguiente código dentro de la instrucción if:

```
m_CurrentWaypointIndex = (m_CurrentWaypointIndex + 1) %  
waypoints.Length;
```

El operador restante toma lo que está a su izquierda y lo divide por lo que está a su derecha, luego devuelve el resto. Por ejemplo, 3% 4 devolvería 3 (porque 4 entra en 3 cero veces con 3 sobrantes). 5% 4 devolvería 1 (porque 4 entra en 5 una vez con 1 sobrante). Su código dice: "Agregue uno al índice actual, pero si ese incremento pone el índice igual al número de elementos en la matriz de puntos de referencia, en su lugar, configúrelo en cero". el resto al dividir cualquier número por sí mismo es cero.

4. Ahora que ha incrementado el índice (y lo ha vuelto a poner en cero cuando sea necesario), debe usarlo. Agregue lo siguiente debajo del incremento del índice:

```
navMeshAgent.SetDestination  
(waypoints[m_CurrentWaypointIndex].position);
```

Esto es exactamente lo que hizo en el método de Inicio, excepto que en lugar de usar cero como índice, está usando cualquier punto de referencia que el Fantasma esté haciendo actualmente como índice.

5. ¡ Eso es! Su script completo debería verse así:

```
using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
using UnityEngine.AI;
```

```

public class WaypointPatrol : MonoBehaviour
{
    public NavMeshAgent navMeshAgent;
    public Transform[] waypoints;

    int m_CurrentWaypointIndex;

    void Start ()
    {
        navMeshAgent.SetDestination (waypoints[0].position);
    }

    void Update ()
    {
        if(navMeshAgent.remainingDistance < navMeshAgent.stoppingDistance)
        {
            m_CurrentWaypointIndex = (m_CurrentWaypointIndex + 1) % waypoints.Length;
            navMeshAgent.SetDestination(waypoints[m_CurrentWaypointIndex].position);
        }
    }
}

```

6. Guarde el script y regrese al Unity Editor.

11. Asigne la referencia del agente de malla de navegación al fantasma prefabricado

Debería poder ver en la ventana del Inspector que necesita asignar la referencia del Agente de malla de navegación y algunas Transformaciones para los puntos de referencia. La referencia del Agente de malla de navegación será la misma para cada instancia de Prefab, por lo que puede asignarla ahora. Sin embargo, los puntos de ruta serán diferentes para cada Fantasma y, por lo tanto, no deberían ser parte del prefab, no solo eso, sino que los puntos de ruta serán parte de la Escena, por lo que Ghost Prefab no puede hacer referencia a ellos.

1. En la ventana Jerarquía, seleccione el **Ghost** GameObject.
2. Arrastre el nombre del componente **Nav Mesh Agent** en la ventana del Inspector hasta el campo Nav Mesh Agent en el script Waypoint Patrol. Esto asignará la referencia de Nav Mesh Agent.
3. Guarde el Ghost Prefab y regrese a la escena.

4. Verifiquemos que el script de Ghost's Observer tenga las referencias que necesita. En la Jerarquía, expanda el GameObject fantasma y seleccione el **GameObject** secundario **PointOfView**.
5. Arrastre el JohnLemon GameObject desde la ventana Jerarquía al campo **Jugador** del script Observador para asignar su Transformación. 6. Luego, haz clic en el botón de selección circular y asigna el campo GameEnding. Como solo hay un Componente GameEnding en la escena, solo habrá una opción para que selecciones.

¡Tus enemigos ahora están muy cerca de completarse! Solo necesita agregar más Fantasmas y asignar sus puntos de referencia, luego agregar un par de Gárgolas estáticas más.

12. Coloca fantasmas en tu escena

¡Ahora estás listo para agregar enemigos a tu escena! Primero, crea cuatro fantasmas duplicados y llena el nivel con enemigos en movimiento:

1. En la ventana Jerarquía, contraiga y luego seleccione **Ghost** GameObject. Duplique el GameObject presionando Ctrl + D (Windows) o CMD + D (macOS) hasta que tenga cuatro copias de Ghost.
2. El primer fantasma debe moverse hacia adelante y hacia atrás cerca de la habitación en la que comienza JohnLemon. En el Inspector, establezca la posición del fantasma llamado **ghost** en (-5.3 , 0 , -3.1).
3. El segundo Fantasma se moverá de un lado a otro por un largo corredor, de modo que JohnLemon tendrá que meterse en las habitaciones laterales para pasar. Establezca la posición del Fantasma llamado **ghost (1)** en (1.5 , 0 , 4).
4. El tercer fantasma estará dando vueltas alrededor de la mesa en uno de los comedores. Establezca la posición del Fantasma llamado **ghost (2)** en (3.2 , 0 , 6.5).
5. El fantasma final se moverá en el dormitorio cerca de la salida, por lo que JohnLemon quedará atrapado si va por el camino equivocado. Establezca la posición del Fantasma llamado **ghost (3)** en (7.4 , 0 , -3).

Ahora que tiene sus fantasmas posicionados, necesita hacer y colocar sus puntos de referencia.

13. Crear y posicionar los puntos de referencia

fantasma

Los puntos de referencia solo deben ser GameObjects vacíos, ya que solo está utilizando sus componentes Transform y todos los GameObjects tienen una Transform. Para ser lo más eficiente posible, creará todos los puntos de referencia y luego los posicionará y asignará. Para configurar y asignar los waypoints:

1. En la ventana Jerarquía, haga clic en el botón Crear y seleccione **Create Empty**.
2. Cambie el nombre del GameObject vacío recién creado a "**Waypoint**".
3. Duplique el Objeto de juego Waypoint nueve veces para crear diez puntos de ruta en total: desde Waypoint hasta Waypoint (9).
4. En la Jerarquía, seleccione el **Ghost** GameObject.
5. En el Inspector, desplácese hacia abajo hasta que el **componente Waypoint Patrol** sea visible.
6. Arrastre el **Waypoint** GameObject desde la ventana Jerarquía al nombre del campo **Waypoints** para agregarlo a la matriz.
7. Arrastre el **WayObject (1)** GameObject sobre el nombre del campo **Waypoints** para agregarlo también a la matriz.
8. Su primer fantasma tiene dos puntos de referencia en su matriz, pero necesita posicionarlos. El fantasma se moverá hacia adelante y hacia atrás a través de la sala de partida.
 - En el Inspector, establezca la posición de **Waypoint** en (**-5.3 , 0 , 6.7**).
 - Establezca la posición del **Waypoint (1)** en (**-5.5 , 0 , -4.5**).
9. El segundo fantasma se moverá hacia arriba y hacia abajo por el pasillo con habitaciones laterales. Seleccione el GameObject **Ghost (1)** y asigne **Waypoint (2)** y **Waypoint (3)** a su matriz de Waypoints.
 - Establezca la posición del **Waypoint (2)** en (**1.2 , 0 , 7.7**).
 - Establezca la posición del **Waypoint (3)** en (**0.9 , 0 , -3.5**).
10. El tercer fantasma estará dando vueltas alrededor de la mesa en el comedor y necesitará más puntos de referencia.

Seleccione el **Ghost** (2) **GameObject** y asigne **Waypoint** (4) , **Waypoint** (5) , **Waypoint** (6) y **Waypoint** (7) a su matriz de Waypoints.

- Establezca la posición del **Waypoint** (4) en **(3.2, 0, 5.6)** .
- Establezca la posición del **Waypoint** (5) en **(3.2, 0, 12.3)** .
- Establezca la posición del **Waypoint** (6) en **(6.5, 0, 12.3)** .
- Establezca la posición del **Waypoint** (7) en **(6.5, 0, 5.6)** .

11. El fantasma final está en la habitación inferior derecha. Seleccione el **GameObject Ghost** (3) y asigne **Waypoint** (8) y **Waypoint** (9) a su matriz de Waypoints.

- Establezca la posición del **Waypoint** (8) en **(3.2, 0, -5)** .
- Establezca la posición del punto de **Waypoint** (9) en **(7.4, 0, -2)**

Eso es todo, ¡tus fantasmas están terminados!

14. Coloca las gárgolas en tu escena

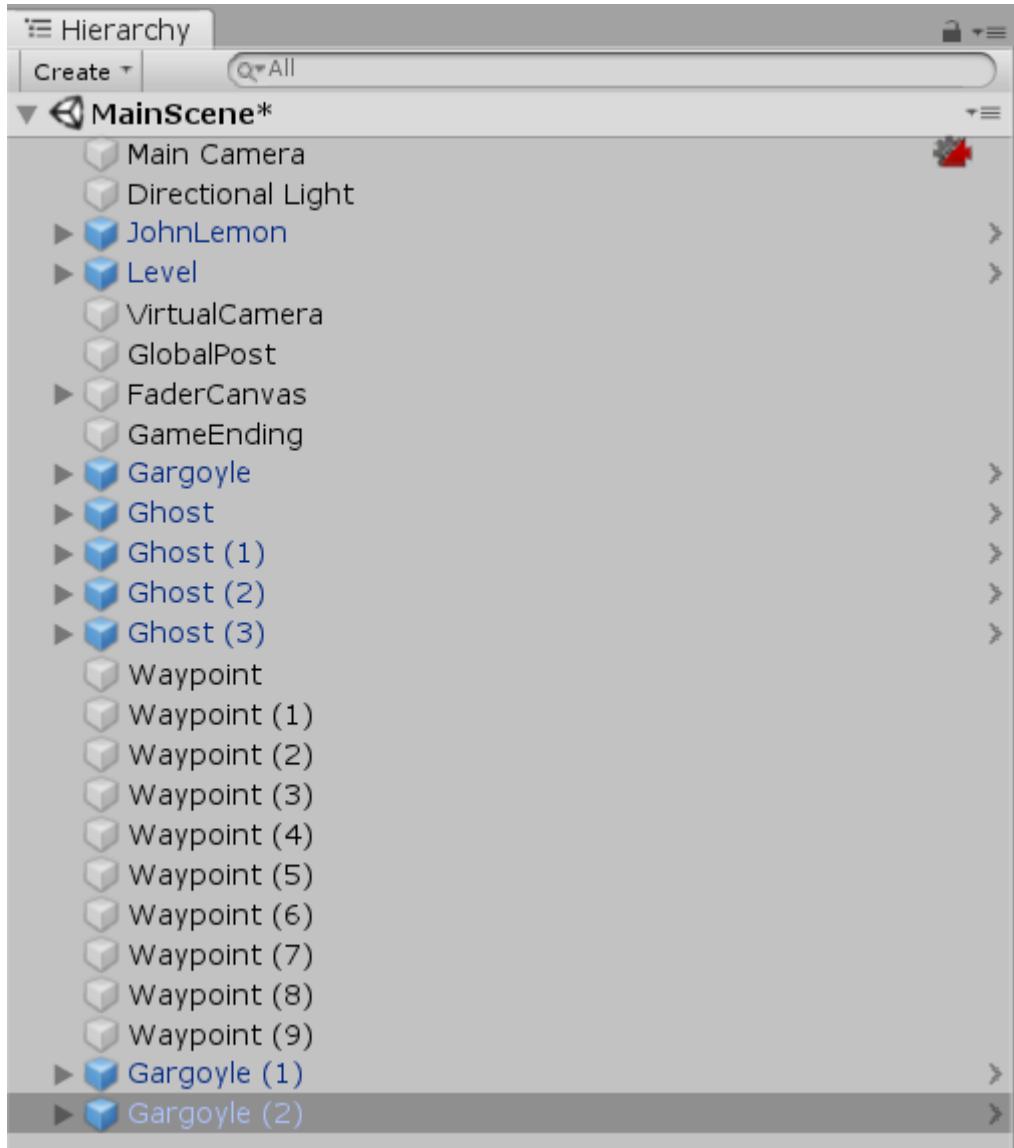
Luego, coloca tres gárgolas en tu nivel:

1. Duplicar el **GameObject Gargoyle** dos veces, para hacer tres Gárgolas en total.
 2. Ya has posicionado la primera gárgola. El segundo debe ir al final del largo corredor, por lo que JohnLemon no puede ir muy lejos.
- Establezca la posición de **Gárgola** (1) en **(-2.6, 0, -8.5)** .
 - Establezca la rotación de **Gárgola** (1) en **(0, 30, 0)** .
3. La tercera gárgola debe ir en la esquina del corredor al lado del comedor, por lo que JohnLemon queda atrapado si va por el camino equivocado. Dado que será la misma esquina que la primera Gárgola, no necesitará cambiar la rotación, pero aún tendrá que ajustar la posición.
- Establezca la posición de **Gárgola** (2) en **(-4.8, 0, 10.6)** .

Ahora su nivel está poblado con una variedad de enemigos para hacer que la fuga de JohnLemon sea más desafiante.

15. Limpiar la jerarquía

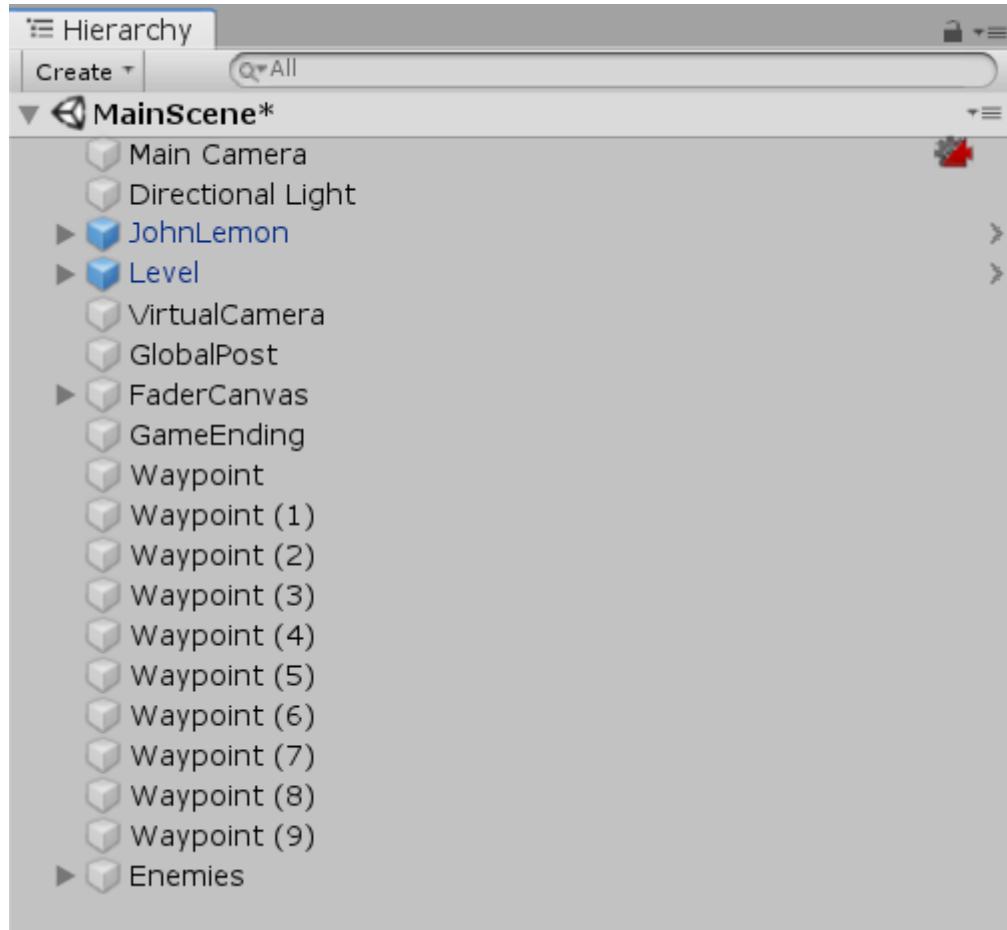
Hay una última cosa que debes hacer antes de que hayas terminado por completo con los enemigos y sus puntos de referencia: ordenarlos. La ventana Jerarquía se ve un poco desordenada:



La mejor solución es crear algunos GameObjects vacíos que puedan actuar como padres de los enemigos y puntos de referencia. Esto le permitirá expandirlos o contraerlos según sea necesario, manteniendo la ventana de la Jerarquía ordenada y ordenada. Para ordenar la Jerarquía:

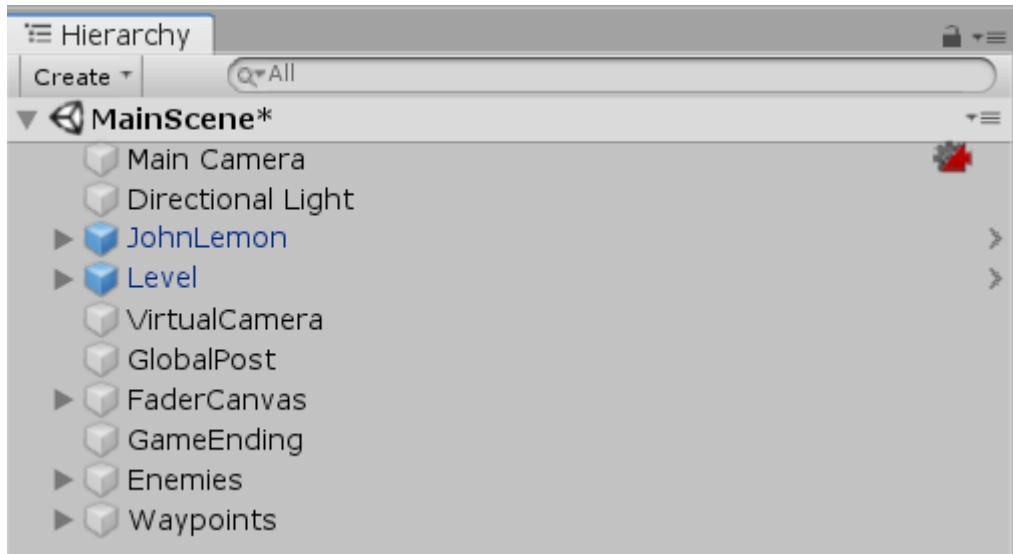
1. En la ventana Jerarquía, cree un GameObject vacío. Cambie el nombre a "**Enemigos**".
2. En el Inspector, establece la posición de los **Enemigos** en **(0, 0, 0)**. Esto actuará como padre de todos los GameObjects de Ghost y Gargoyle.

- 3.** Mantenga presionada la tecla Ctrl (Windows) o la tecla CMD (macOS) y en la ventana Jerarquía, haga clic en cada uno de los Fantasmas y Gárgolas. Cuando estén todos seleccionados, arrástrelos al **Enemigos** GameObject para que sea su padre.



Ahora puedes expandir y colapsar a los enemigos cuando quieras. A continuación, debe hacer lo mismo para los waypoints.

- 4.** En la ventana Jerarquía, cree un GameObject vacío. Cámbiele el nombre a "**Waypoints**".
- 5.** En el Inspector, configure la posición de los **Waypoints** en **(0, 0, 0)**. Esto actuará como padre de todos los GameObjects de Ghost y Gargoyle.
- 6.** Mantenga presionada la tecla Ctrl (Windows) o la tecla CMD (macOS) y en la ventana Jerarquía, haga clic en cada uno de los Waypoints. Cuando estén todos seleccionados, arrástrelos al **Waypoints** GameObject para que sea su padre.



¡Eso es todo, has ordenado a tus enemigos!

Guarde la escena e ingrese al modo de reproducción para probarla. Cuando haya terminado, asegúrese de salir del modo de reproducción.

16. Resumen

En este tutorial, terminamos de crear enemigos para tu juego. ¡Está casi completo! Sin embargo, es un poco silencioso: hay un zumbido proveniente de las luces, pero sería bueno tener más sonido para mejorar la atmósfera. En el próximo tutorial, agregará más audio a su juego.

Audio

1. Audio

Ahora llenaste tu nivel de enemigos para hacer que el escape de JohnLemon fuera un desafío, pero cuando probaste el juego fue un poco silencioso. En este tutorial, agregará los toques finales a su juego con una gama de audio espeluznante para mejorar la atmósfera.

Hay dos tipos diferentes de audio que usará en este tutorial:

- **Non-diegetic sound. Sonido no diegético**, que no tiene una fuente identificable (por ejemplo, una banda sonora)
- **Diegetic sound. Sonido diegético**, que tiene una fuente identificable (por ejemplo, el sonido de una pistola disparando)

¡Comencemos agregando audio no diegético a su juego!

2. Una cartilla sobre audio en la unidad

Antes de comenzar a agregar audio a su juego, exploremos rápidamente cómo funciona el audio en Unity. Hay tres partes principales: clips de audio, fuentes de audio y el oyente de audio.

- **Los clips de audio** son activos como MP3 que contienen todos los datos específicos de un sonido en particular.
- **Las fuentes (Source) audio** son componentes que actúan como el origen de un sonido en el mundo del juego. La mayoría de las cosas que producen sonido en un juego deben tener un componente de Audio source, para que el sonido tenga una ubicación.
- **Audio Listener** es un componente único en una escena que funciona como los oídos virtuales del reproductor (de la misma manera que el componente de la cámara funciona como los ojos virtuales del reproductor). Por defecto, el componente de escucha de audio está en la cámara principal.

Estas diferentes partes generalmente funcionan juntas de la siguiente manera: Audio source reproduce un clip de audio, y si el oyente de audio está lo suficientemente cerca de la fuente de audio, entonces se escucha el sonido.

La **combinación espacial** de una fuente de audio en particular determina si suena como si viniera de un punto particular en el mundo del juego, o si es igualmente fuerte, sin importar la distancia entre la fuente y el listener de audio.

Estás comenzando con audio no diegético. Dado que este sonido no tiene un origen, su **mezcla espacial** se establece en **2D**. Cuando la combinación espacial se configura como puramente 2D, la distancia entre la fuente de audio y el listener de audio no afecta el volumen. Se llama 2D porque aún puede desplazarse hacia la izquierda y hacia la derecha usando la configuración de Stereo Pan. Si la combinación espacial de una fuente de audio está configurada en 3D, el volumen variará con la distancia al listener de audio. Los valores entre 2D y 3D varían la intensidad de este efecto.

3. ¿Qué es el audio no diegético?

El audio no diegético (audio sin una fuente identificable o identifiable) puede tener un gran impacto en su juego y la experiencia de los jugadores. Piensa en los juegos que has disfrutado jugando y cómo usan estos sonidos para crear una atmósfera particular: ¿cómo te hacen sentir? ¿Qué hay de ellos es particularmente efectivo? Vas a agregar tres sonidos no dietéticos a tu juego:

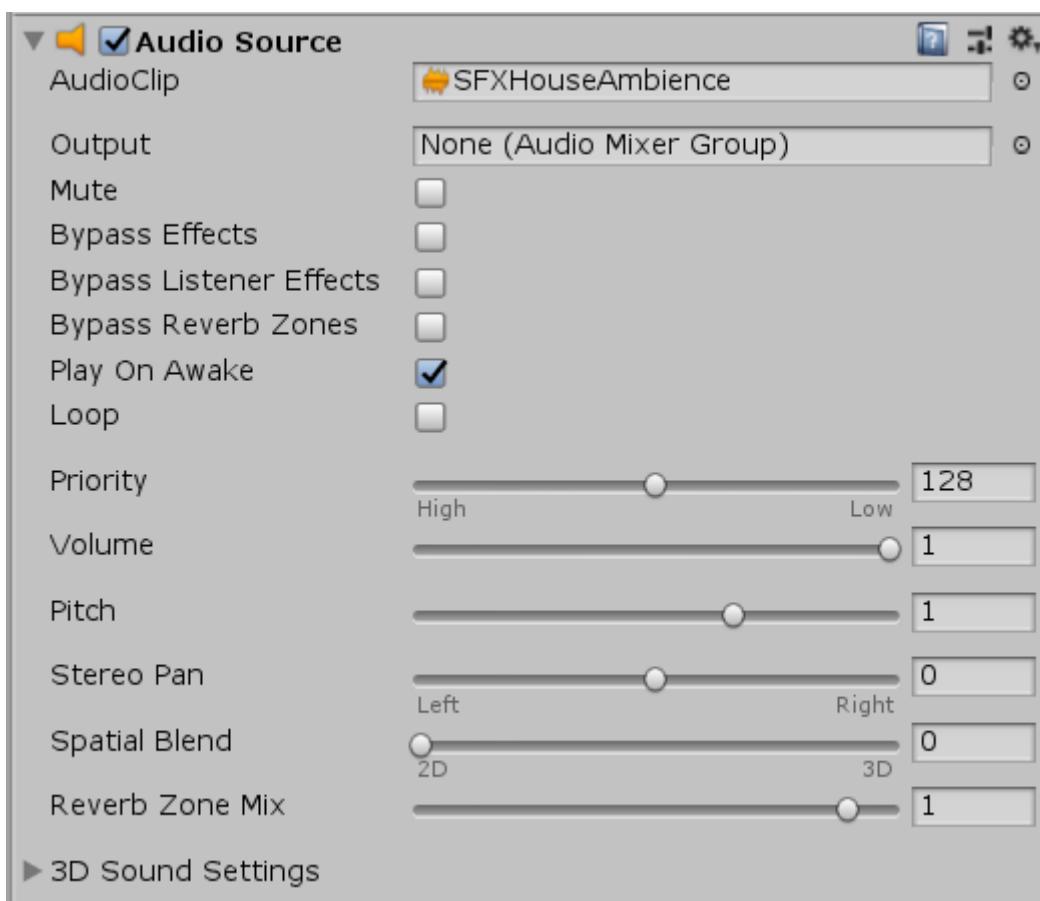
- Una pista ambiental que se repetirá y dará una atmósfera general a la casa embrujada
- Un efecto de sonido que suena cuando JohnLemon es atrapado
- Un efecto de sonido que suena cuando JohnLemon logra escapar

4. Crea fuentes de audio para tu juego

Primero, crea algunas fuentes de audio para tu juego:

1. En la ventana Jerarquía, haga clic en el menú Crear y seleccione **Crear GameObject vacío**. Cambie el nombre del objeto de juego "**Ambiente**".
2. En el Inspector, establezca la posición de Ambiente en **(0, 0, 0)**. La posición del GameObject no importa técnicamente, ya que el volumen del audio será el mismo donde sea que esté. Sin embargo, siempre es útil mantener sus GameObjects organizados posicionalmente en caso de que esto sea importante más adelante.

3. A continuación, debe agregar un componente de fuente de audio. Normalmente, haría esto con el botón Agregar componente, pero cuando quiera asignar un Clip de audio también hay un atajo que puede usar. En la ventana Proyecto, vaya a **Activos > Audio**. Arrastre el **clip de audio SFXHouseAmbience** de la ventana del proyecto a la ventana del **inspector** para crear un componente de fuente de audio y asignarlo automáticamente como el clip de audio.



4. La propiedad **Spatial Blend** ya se ha configurado completamente en 2D, por lo que no necesita cambiarla. **Play On Awake** también se ha habilitado de forma predeterminada. Esto significa que tan pronto como comience el nivel, el audio comenzará a reproducirse.

5. Sin embargo, de forma predeterminada se detendrá después de una sola ejecución. Active la casilla de verificación **Loop** para recorrer la pista ambiental de su juego.

6. ¡ Ahora tu casa embrujada tiene un sonido de fondo espeluznante! Ingrese al modo de reproducción y compruebe que el audio se reproduce correctamente.

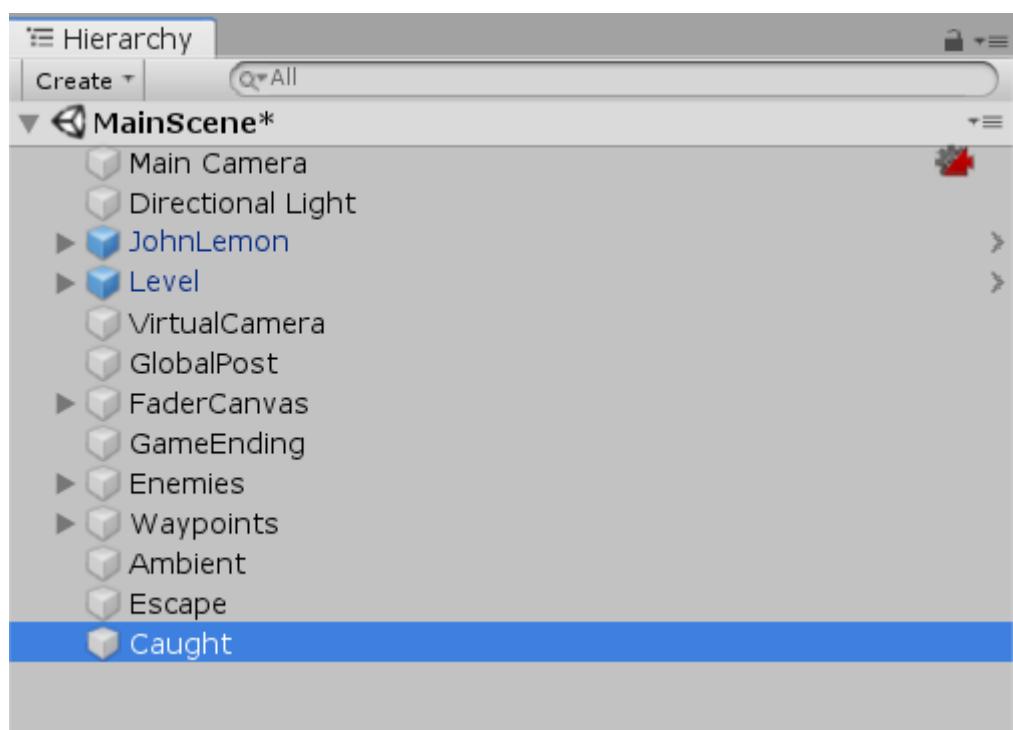
7. Debería escuchar el audio, pero es un poco alto en comparación con los otros sonidos que agregará en este tutorial. Salga del modo de reproducción para que pueda ajustar esto.

8. En el Inspector, establezca la propiedad **Volumen** del Componente de fuente de audio en **0.5**.

5. Duplica tu fuente de audio

En lugar de repetir este proceso para los dos efectos de sonido finales del juego, seamos más eficientes y dupliquemos los GameObjects y luego cambiemos la configuración según corresponda.

1. En la ventana Jerarquía, seleccione Ambient GameObject. Duplícalo dos veces con el atajo Ctrl + D (Windows) o CMD + D en (macOS). Cambie el nombre de la primera copia "**Escape**" y la segunda "**Atrapado**".



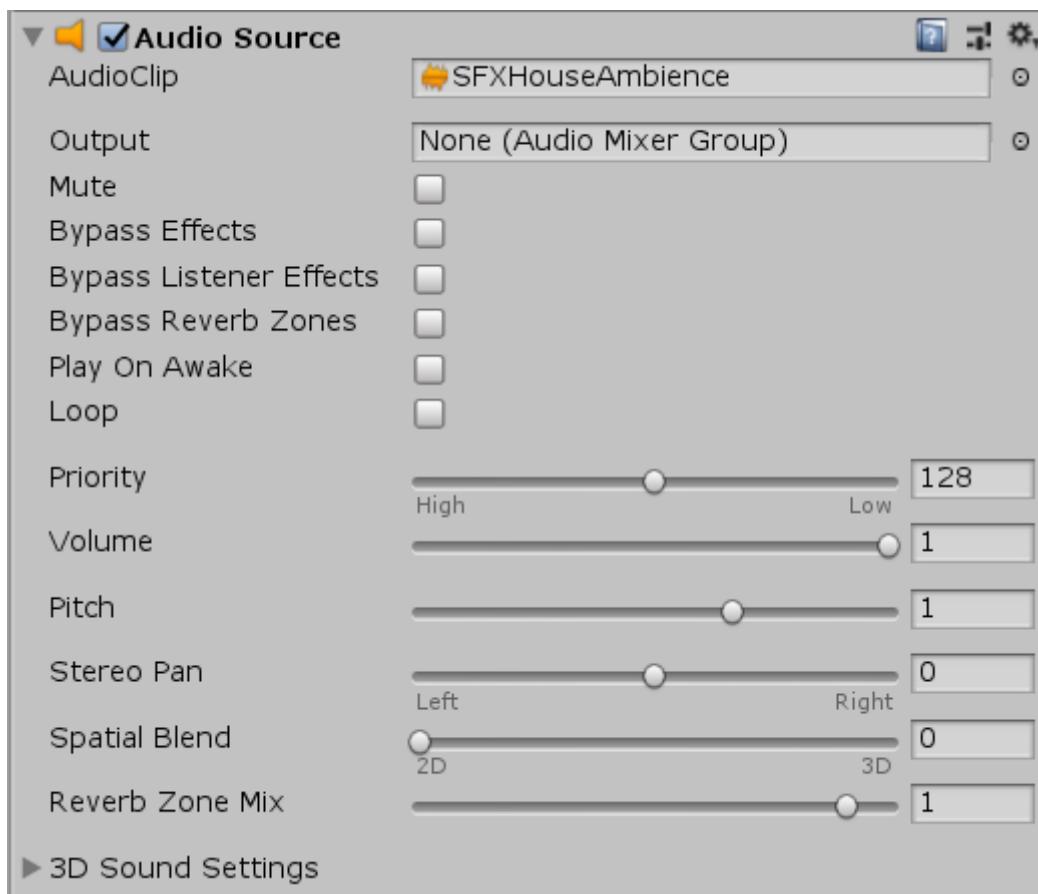
2. La única diferencia entre estos dos efectos de sonido será el AudioClip utilizado; su configuración es exactamente la misma. En la ventana Jerarquía, presione Ctrl (Windows) o CMD (macOS) y haga clic en los objetos de juego **Escape** and **Caught**. Esto le permite editarlos juntos.

3. Estos ruidos no deben reproducirse de inmediato o en bucle. También necesitan ser escuchados sobre los otros sonidos en el juego, por lo que deben

ser más fuertes que el audio ambiental. Con los objetos de juego Escape y Caught seleccionados:

- Deshabilite la casilla de verificación **Jugar al despertar**
- Desactiva la casilla de verificación **Loop**

Establezca el valor de **Volumen** en **1**



Ahora que tiene sus ajustes comunes ajustados, puede configurar el **AudioClip** para cada uno.

4. Seleccione el **Escape** GameObject y configure el **AudioClip** para su Componente de fuente de audio en **SFXWin**, ya sea arrastrándolo desde la ventana Proyecto o utilizando el botón de selección circular.

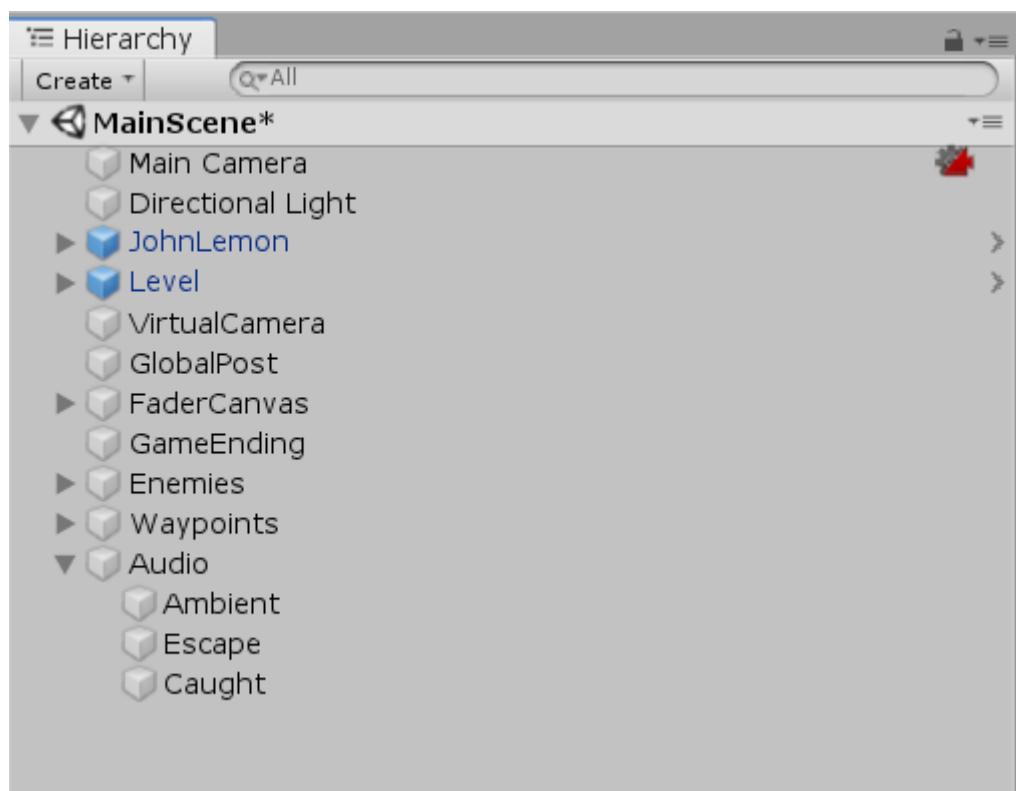
5. Seleccione la **Atrapados** GameObject y establecer el **AudioClip** por su componente fuente de audio a **SFXGameOver**, o bien arrastrándolo desde la ventana de proyecto o utilizando el botón de selección círculo.

Ahora ha configurado sus fuentes de audio no diegéticas, pero antes de continuar con la implementación de los sonidos Escape and Caught vamos a ordenar las fuentes de audio bajo un solo parent.

6. Ordena tus fuentes de audio

Para ordenar sus fuentes de audio:

1. En la ventana Jerarquía, cree un GameObject vacío. Cambie el nombre a "Audio".
2. Seleccione los objetos de juego Ambient, Escape y Caught en la ventana Jerarquía, manteniendo presionada la tecla Ctrl (Windows) o CMD (macOS) y haciendo clic en cada uno de ellos.
3. Arrastre los GameObjects seleccionados al Audio GameObject, para que sea su padre.



7. Regrese a su secuencia de comandos

GameEnding

Ahora que has ordenado, pasemos a implementar los sonidos finales del juego en el script GameEnding.

En la ventana Proyecto, abra la carpeta **Activos> Scripts** y haga doble clic en el script GameEnding para abrirlo y editarlo.

La última vez que terminaste el script GameEnding, se veía así:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameEnding : MonoBehaviour
{
    public float fadeDuration = 1f;
    public float displayImageDuration = 1f;
    public GameObject player;
    public CanvasGroup exitBackgroundImageCanvasGroup;
    public CanvasGroup caughtBackgroundImageCanvasGroup;

    bool m_IsPlayerAtExit;
    bool m_IsPlayerCaught;
    float m_Timer;

    void OnTriggerEnter (Collider other)
    {
        if (other.gameObject == player)
        {
            m_IsPlayerAtExit = true;
        }
    }

    public void CaughtPlayer ()
    {
        m_IsPlayerCaught = true;
    }

    void Update ()
    {
        if (m_IsPlayerAtExit)
        {
```

```

        EndLevel (exitBackgroundImageCanvasGroup, false);
    }
    else if (_IsPlayerCaught)
    {
        EndLevel (caughtBackgroundImageCanvasGroup, true);
    }
}

void EndLevel (CanvasGroup imageCanvasGroup, bool doRestart)
{
    _Timer += Time.deltaTime;
    imageCanvasGroup.alpha = _Timer / fadeDuration;

    if (_Timer > fadeDuration + displayImageDuration)
    {
        if (doRestart)
        {
            SceneManager.LoadScene (0);
        }
        else
        {
            Application.Quit ();
        }
    }
}

```

8. Actualice su secuencia de comandos

GameEnding para reproducir audio

Debe agregar la funcionalidad de reproducir una fuente de audio cuando se llama al método EndLevel. Hay dos fuentes de audio diferentes que pueden reproducirse, por lo que necesitará una referencia para cada una. Recuerde: el método EndLevel se llama cada cuadro en Actualización. No desea que las fuentes de audio sigan reproduciéndose, por lo que necesitará una forma de detenerlo después de la primera vez. Comencemos agregando sus referencias de fuente de audio:

1. Debajo de la declaración de variable `exitBackgroundImageCanvasGroup`, agregue el siguiente código:

```
public AudioSource exitAudio;
```

2. Debajo de la declaración de variable

`catchBackgroundImageCanvasGroup`, agregue lo siguiente:

```
public AudioSource caughtAudio;
```

3. A continuación, debe crear una variable para asegurarse de que el audio solo se reproduce una vez. Una variable bool será falsa de manera predeterminada, cuando desee reproducir el audio, puede verificar que sea falso y reproducir el audio, configurándolo en verdadero una vez que lo haya hecho. Debajo de la **declaración de variable m_Timer**, agregue lo siguiente:

```
bool m_HasAudioPlayed;
```

9. Ajuste su método para usar las nuevas variables

Ahora usemos estas nuevas variables:

1. Dado que el juego debería reproducir una fuente de audio diferente dependiendo de si JohnLemon fue atrapado o escapó, debe agregar otro parámetro al método EndLevel. Cambie la firma del método EndLevel a lo siguiente:

```
void EndLevel (CanvasGroup imageCanvasGroup, bool doRestart,  
AudioAudio sourceSource)
```

2. Sus llamadas a EndLevel ya no reciben los parámetros correctos, ahora también necesitan AudioSource. Cambie la primera llamada al método EndLevel a lo siguiente:

```
EndLevel (exitBackgroundImageCanvasGroup, false, exitAudio);
```

3. Cambie la segunda llamada al método EndLevel por lo siguiente:

```
EndLevel (caughtBackgroundImageCanvasGroup, true, caughtAudio);
```

Las llamadas al método son correctas nuevamente, pero actualmente su método EndLevel no hace nada con su nuevo parámetro.

10. Asegúrese de que el audio solo se reproduzca una vez

El audio debe reproducirse independientemente de cuán lejos esté el temporizador, por lo que tiene sentido tener el código de audio al comienzo del método EndLevel. También solo desea que se reproduzca el audio si aún no se ha reproducido, por lo que debe colocarlo dentro de una instrucción if que lo verifique.

1. Agregue lo siguiente al inicio del método EndLevel:

```
if(!m_HasAudioPlayed)
{
}
```

El signo de exclamación significa negar todo lo que está a la derecha. Eso significa que el código dentro de la instrucción if solo se ejecutará si el audio no se ha reproducido.

2. Lo primero que debe hacer dentro de esto si la declaración es reproducir el audio. Para reproducir el clip de audio asignado a una fuente de audio, llame al método de reproducción de la fuente de audio de la siguiente manera:

```
audioSource.Play();
```

4. Debido a que solo desea que el audio se reproduzca una vez, ahora debe establecer el valor de m_HasAudioPlayed en verdadero. Esto significará que el código en la instrucción if no se llama nuevamente. Dentro de la declaración if debajo de la llamada al método Play, agregue lo siguiente:

```
m_HasAudioPlayed = true;
```

5. Ahora ha terminado de editar el script GameEnding. Debe tener un aspecto como este:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class GameEnding : MonoBehaviour
{
    public float fadeDuration = 1f;
    public float displayImageDuration = 1f;
    public GameObject player;
    public CanvasGroup exitBackgroundImageCanvasGroup;
    public AudioSource exitAudio;
    public CanvasGroup caughtBackgroundImageCanvasGroup;
    public AudioSource caughtAudio;

    bool m_IsPlayerAtExit;
    bool m_IsPlayerCaught;
    float m_Timer;
    bool m_HasAudioPlayed;

    void OnTriggerEnter (Collider other)
    {
        if (other.gameObject == player)
```

```

        {
            m_IsPlayerAtExit = true;
        }
    }

    public void CaughtPlayer ()
    {
        m_IsPlayerCaught = true;
    }

    void Update ()
    {
        if (m_IsPlayerAtExit)
        {
            EndLevel (exitBackgroundImageCanvasGroup, false,
exitAudio);
        }
        else if (m_IsPlayerCaught)
        {
            EndLevel (caughtBackgroundImageCanvasGroup, true,
caughtAudio);
        }
    }

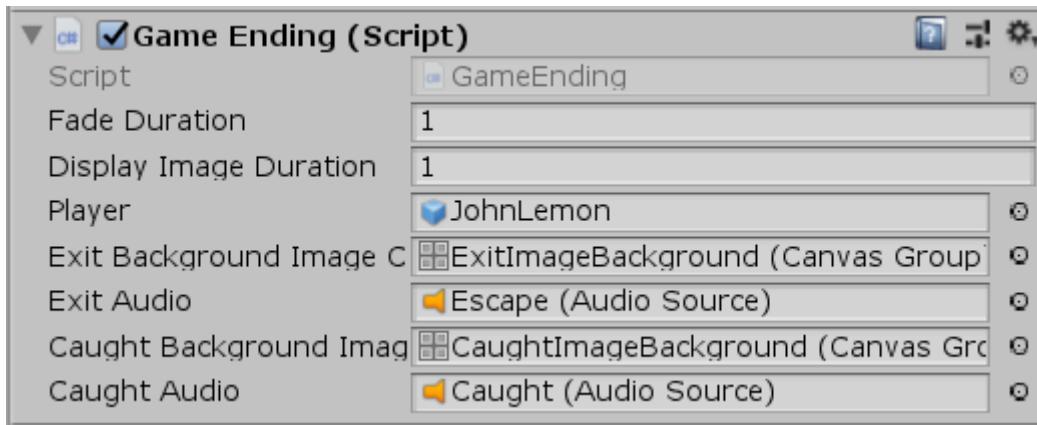
    void EndLevel (CanvasGroup imageCanvasGroup, bool doRestart,
AudioAudio sourceSource)
    {
        if (!m_HasAudioPlayed)
        {
            audioSource.Play();
            m_HasAudioPlayed = true;
        }

        m_Timer += Time.deltaTime;
        imageCanvasGroup.alpha = m_Timer / fadeDuration;

        if (m_Timer > fadeDuration + displayImageDuration)
        {
            if (doRestart)
            {
                SceneManager.LoadScene (0);
            }
            else
            {
                Application.Quit ();
            }
        }
    }
}

```

6. Guarde el script y regrese a Unity.
7. El paso final es establecer las referencias que agregó al script GameEnding. En la ventana Jerarquía, seleccione el **GameOnding** GameObject.
8. En el Inspector, use el botón de selección circular para establecer el campo **Salir de audio en Escape** y el campo **Audio** capturado en **Atrapado**.



Eso es todo, ¡has terminado el audio no diegético!

11. Agregar audio de pasos

Ahora necesita implementar el audio donde puede identificar la fuente. Comencemos con los pasos:

1. En la ventana Jerarquía, seleccione JohnLemon GameObject.
2. En el Inspector, agregue un componente **AudioSource**.
3. Utilice el botón de selección para ajustar el círculo **AudioClip** propiedad a **SFXFootstepsLooping**.
4. A pesar de que se trata de un sonido diegético, aún debe usar una Mezcla espacial de 2D completo para que el volumen no varíe a medida que JohnLemon se mueve: la configuración predeterminada está bien.
5. No desea que el audio se reproduzca tan pronto como comience la escena cuando JohnLemon comience a estar parado. **Deshabilite** la casilla de verificación **Reproducir al despertar**.
6. Desea que el sonido siga reproduciéndose hasta que le pida que se detenga, por lo que debe repetirse. **Active** la casilla de verificación **Loop**.

Tal como lo hizo antes, ahora necesita editar un script para poder reproducir el audio cuando lo desee.

12. Regrese a su secuencia de comandos PlayerMovement

Para habilitar el audio de los pasos, va a editar su secuencia de comandos PlayerMovement. Haga doble clic en la propiedad **Script** del componente PlayerMovement para abrir el script PlayerMovement para editarlo.

La última vez que terminó el script PlayerMovement, se veía así:

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

public class PlayerMovement : MonoBehaviour
{
    public float turnSpeed = 20f;

    Animator m_Animator;
    Rigidbody m_Rigidbody;
    Vector3 m_Movement;
    Quaternion m_Rotation = Quaternion.identity;

    void Start ()
    {
        m_Animator = GetComponent<Animator> ();
        m_Rigidbody = GetComponent<Rigidbody> ();
    }

    void FixedUpdate ()
    {
        float horizontal = Input.GetAxis ("Horizontal");
        float vertical = Input.GetAxis ("Vertical");

        m_Movement.Set(horizontal, 0f, vertical);
        m_Movement.Normalize ();

        bool hasHorizontalInput = !Mathf.Approximately
(horizontal, 0f);
        bool hasVerticalInput = !Mathf.Approximately (vertical,
0f);
        bool isWalking = hasHorizontalInput || hasVerticalInput;
        m_Animator.SetBool ("IsWalking", isWalking);
    }
}
```

```

        Vector3 desiredForward = Vector3.RotateTowards
(transform.forward, m_Movement, turnSpeed * Time.deltaTime, 0f);
        m_Rotation = Quaternion.LookRotation (desiredForward);
    }

    void OnAnimatorMove ()
{
    m_Rigidbody.MovePosition (m_Rigidbody.position +
m_Movement * m_Animator.deltaPosition.magnitude);
    m_Rigidbody.MoveRotation (m_Rotation);
}
}

```

13. Actualice su script PlayerMovement para reproducir audio

Ahora necesita editar su script, por lo que los pasos se reproducen cuando JohnLemon está caminando y se detienen cuando está quieto.

Lo primero que necesitará es una referencia al Componente de fuente de audio que acaba de agregar a JohnLemon

1. Debajo de la **declaración de variable m_Rigidbody**, agregue el siguiente código:

```
 AudioSource m

```

Como esta no es una variable pública, no podrá asignarla en la ventana del Inspector. En su lugar, deberá asignarlo en código como lo hizo con los componentes Animator y Rigidbody.

2. En el método de Inicio, después de que las **variables m_Animator y m_Rigidbody** tengan asignadas sus referencias, agregue el siguiente código:

```
m

```

¡Eso es! A continuación, debe usar esta referencia para el componente Fuente de audio.

14. Ajuste su método para usar las nuevas variables

Afortunadamente, ya tiene una variable creada en el método **FixedUpdate** llamada **isWalking** . Esto funcionará perfectamente para tocar y detener el sonido de los pasos:

1. Debajo de la llamada al método **SetBool** en el componente **Animator** , agregue la siguiente instrucción if-else:

```
if(isWalking)
{
}
else
{
}

}
```

Ahora puede llamar a Play en la fuente de audio si **isWalking** es verdadero, y detener en la fuente de audio si es falso.

2. No desea seguir llamando a Reproducir cada cuadro, solo si la Fuente de audio no se está reproduciendo. Para asegurarse de que no llame a todos los cuadros, agregue la siguiente instrucción if dentro de la instrucción if que acaba de agregar:

```
if(!m
```

3. Ahora agregue la llamada al método **Play** a la instrucción if:

```
m
```

4. Para detener la reproducción de la fuente de audio, agregue el siguiente código a la instrucción else:

```
m
```

5. Comprueba que el script de **PlayerMovement** completado tenga este aspecto:

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

public class PlayerMovement : MonoBehaviour
{
    public float turnSpeed = 20f;

    Animator m_Animator;
    Rigidbody m_Rigidbody;
    AudioSource m
```

```

    Vector3 m_Movement;
    Quaternion m_Rotation = Quaternion.identity;

    void Start ()
    {
        m_Animator = GetComponent<Animator> ();
        m_Rigidbody = GetComponent<Rigidbody> ();
        m_AudioSource = GetComponent< AudioSource > ();
    }

    void FixedUpdate ()
    {
        float horizontal = Input.GetAxis ("Horizontal");
        float vertical = Input.GetAxis ("Vertical");

        m_Movement.Set(horizontal, 0f, vertical);
        m_Movement.Normalize ();

        bool hasHorizontalInput = !Mathf.Approximately
(horizontal, 0f);
        bool hasVerticalInput = !Mathf.Approximately (vertical,
0f);
        bool isWalking = hasHorizontalInput || hasVerticalInput;
        m_Animator.SetBool ("IsWalking", isWalking);

        if (isWalking)
        {
            if (!m_AudioSource.isPlaying)
            {
                m_AudioSource.Play();
            }
        }
        else
        {
            m_AudioSource.Stop ();
        }
    }

    Vector3 desiredForward = Vector3.RotateTowards
(transform.forward, m_Movement, turnSpeed * Time.deltaTime, 0f);
    m_Rotation = Quaternion.LookRotation (desiredForward);
}

void OnAnimatorMove ()
{
    m_Rigidbody.MovePosition (m_Rigidbody.position +
m_Movement * m_Animator.deltaPosition.magnitude);
    m_Rigidbody.MoveRotation (m_Rotation);
}
}

```

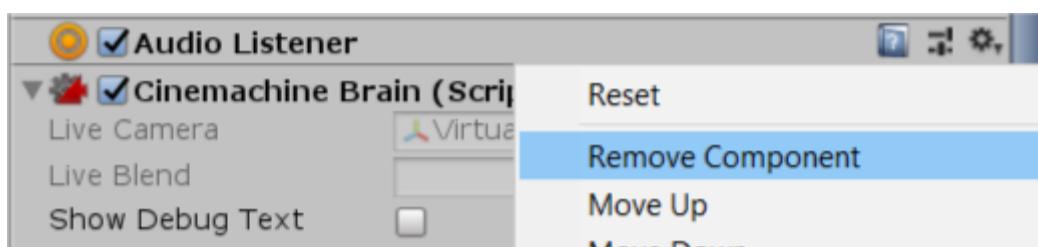
6. Guarde el script y regrese a Unity.

15. Mueva el listener de audio a JohnLemon

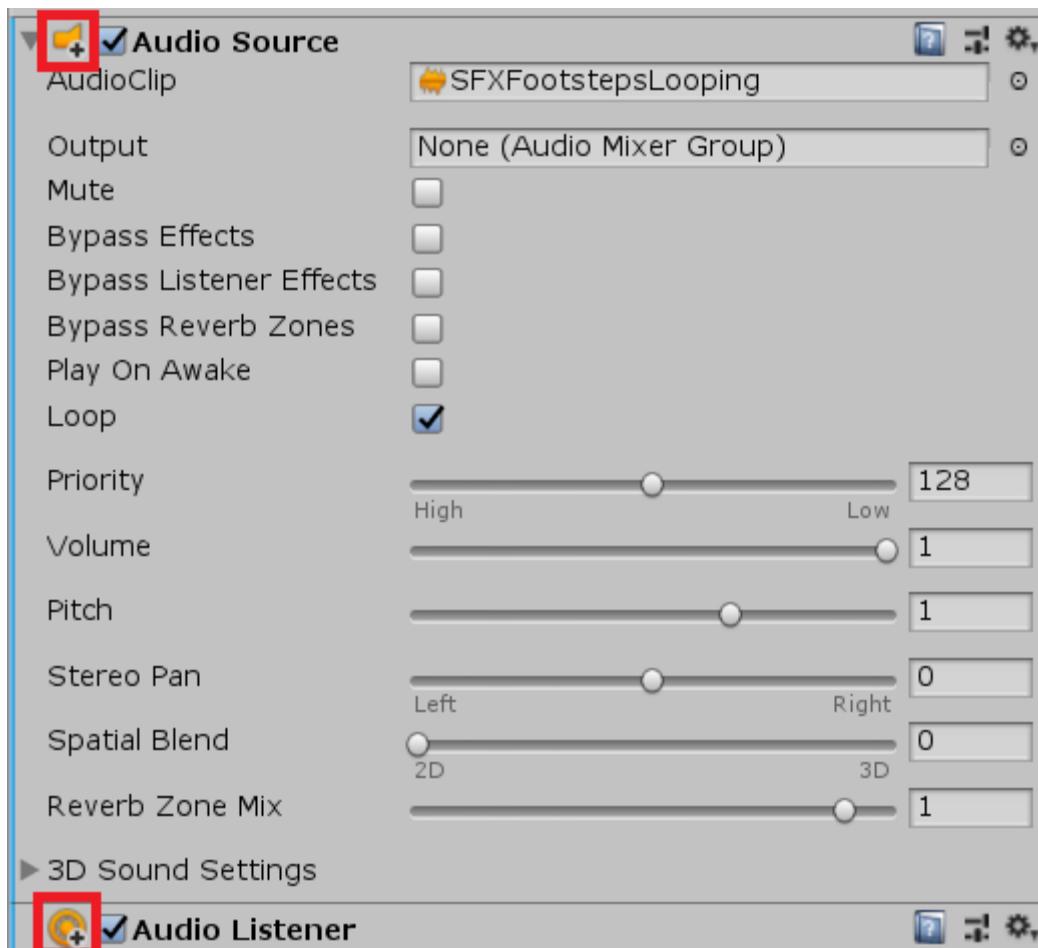
La última pieza de audio para tu juego son los sonidos espeluznantes que los Ghosts harán a medida que pasan. Esto no solo desarrollará la atmósfera, sino que también ayudará al jugador a identificar qué tan cerca está el peligro. Un desafío es que los Ghosts no se harán más fuertes a medida que se acerquen a JohnLemon, se harán más fuertes a medida que se acerquen a la cámara. Esto se debe a que el escucha de audio está en la cámara principal de forma predeterminada. El primer paso es cambiar esto.

Para mover el componente Audio Listener al JohnLemon GameObject:

1. En la Jerarquía, seleccione el Objeto de juego de la **cámara** principal.
2. En el Inspector, haga clic en el menú contextual del componente Audio Listener y seleccione **Eliminar componente**.



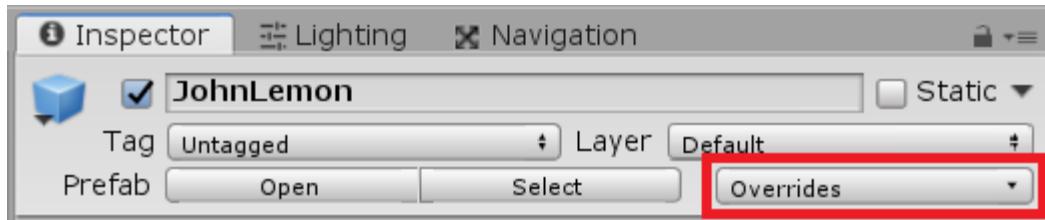
3. En la Jerarquía, seleccione el objeto de **juego** JohnLemon.
4. En el Inspector, agregue un componente de **escucha de audio** a JohnLemon.



16. Actualice el prefabricado JohnLemon

Los cambios que ha realizado en JohnLemon en este tutorial se han realizado en una instancia del prefabricado JohnLemon. Los nuevos componentes que ha agregado tienen un + sobre su ícono, para mostrar que se han agregado en la instancia Prefab. Si tuviera que crear una instancia de otro JohnLemon, estos componentes de audio no estarían allí. Puede solucionar esto aplicando estos cambios al prefabricado JohnLemon:

1. En la Jerarquía, seleccione el JohnOemon GameObject.
2. En el Inspector, se encuentran las **anulaciones** desplegable menú en la parte superior de la ventana.



El menú desplegable Anulaciones le muestra todas las formas en que la instancia de JohnLemon es diferente del prefab de JohnLemon.

3. Haga clic en el menú desplegable para ver los cambios.



Tiene dos componentes agregados: una fuente de audio y un listener de audio. Hagamos que estos cambios formen parte de JohnLemon Prefab.

4. Haga clic en el botón Aplicar todo en la parte inferior del menú desplegable Anulaciones.

17. Agregar una fuente de audio a los fantasmas

Ahora que JohnLemon Prefab tiene un listener de audio, todo el audio de Ghosts tendrá un volumen relativo a la posición de JohnLemon. Esto hará que sea mucho más fácil para el jugador identificar qué tan cerca están. Ahora agreguemos audio a los Ghosts:

1. En la ventana Jerarquía, expanda **Enemies** GameObject.
2. Seleccione uno de los **Ghost** GameObjects. Use el atajo de flecha junto a su nombre para abrir el Prefab para editar.

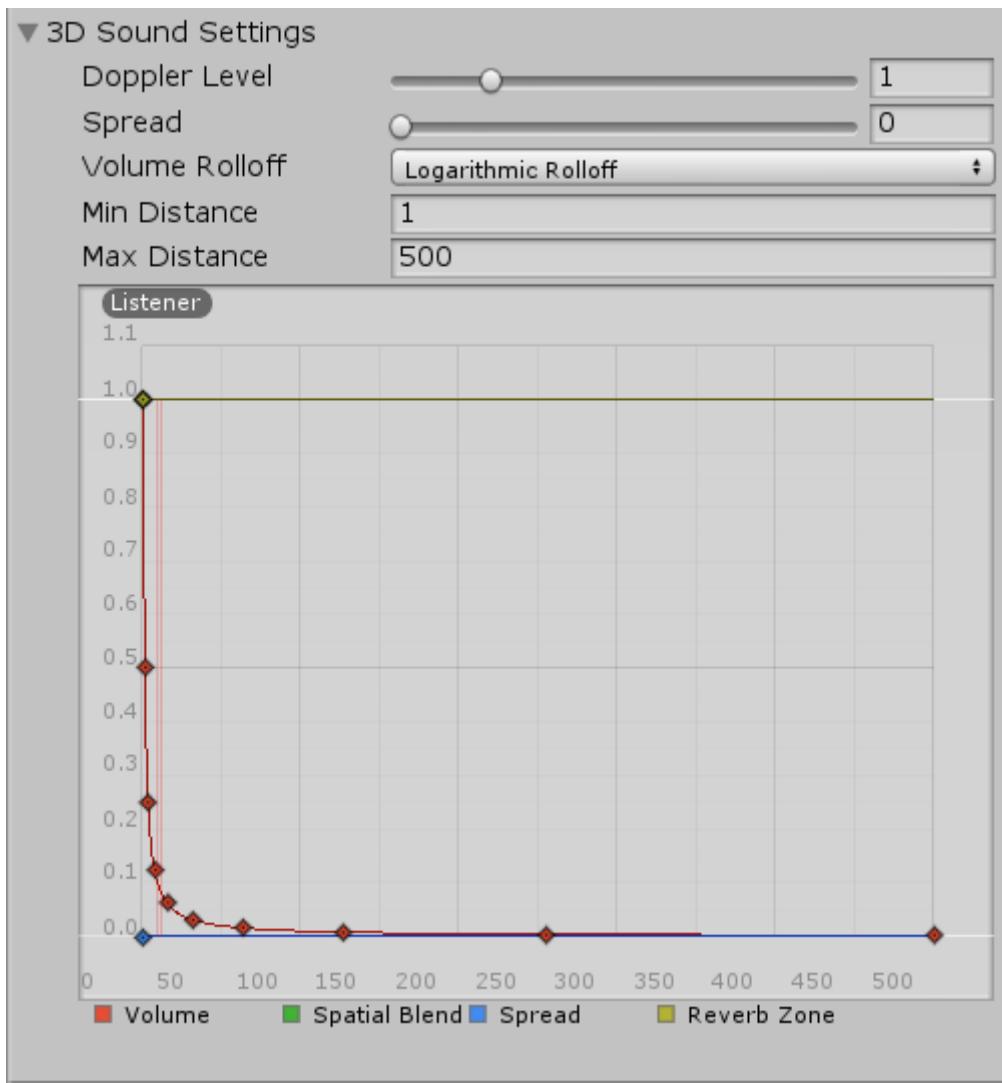
3. En la ventana Proyecto, expanda la carpeta **Activos> Audio** y busque el **Clip de audio SFXGhostMove** .
 4. Arrastre el **clip de audio SFXGhostMove** desde la ventana Proyecto al **Ghost GameObject** en la ventana Jerarquía.
 5. En la ventana Inspector, busque el componente Fuente de audio.
 6. Para evitar que el nivel de volumen sea abrumador y asegúrese de que varía según la distancia de JohnLemon:

 - **Habilite** la casilla de verificación Propiedad de **loop**
 - Establezca la propiedad **Volumen** en **0.4**
 - Establezca la **Mezcla espacial (Spatial blend)** en **1** , para que sea completamente 3D
- ¡Eso es lo básico ordenado!

18. Ajuste las funciones de configuración de sonido 3D

Ajustemos un par de funciones avanzadas para que sus Ghosts suenen realmente bien:

1. En el componente Fuente de audio , expanda la sección **Configuración de sonido 3D** .



La configuración de sonido 3D controla cómo varía el audio con la distancia desde el listener de audio.

2. Cambie la propiedad **Distancia máxima** a **10**. Establecer la Distancia máxima en 10 significará que cuando un Fantasma esté a 10 metros de distancia, el jugador podrá escucharlo, pero en voz muy baja.
3. La forma en que el volumen cambia con la distancia es controlada por la **reducción de volumen**. Actualmente, esto está configurado en Logarithmic Rolloff, que funciona bien para distancias más largas. Dado que su Distancia máxima ahora es solo 10, cambie la Reducción de volumen a **Reducción personalizada**.
4. La curva personalizada predeterminada es la correcta para este juego, por lo que no hay necesidad de ajustar esto.

Sin embargo, hay una última cosa que solucionar antes de que haya terminado.

19. Corrige la dirección del efecto de sonido

fantasma

Pones el componente de escucha de audio en JohnLemon, pero hay un pequeño problema: cuando JohnLemon gira, el escucha de audio gira con él. Esto significa que cuando JohnLemon mira hacia la pantalla, los ojos virtuales del jugador (la cámara) y los oídos virtuales del jugador (el listener de audio) estarán orientados en diferentes direcciones. Debido a esto, los Ghosts sonarán como si estuvieran en el lado opuesto. Usemos una combinación de dos propiedades para hacer que el sonido del Fantasma parezca sin dirección pero aún más fuerte a medida que los Fantasmas se acercan:

1. El clip de audio **SFXGhostMove** se ha configurado para reproducir mono en lugar de estéreo. Esto significa que el sonido es idéntico a través de los canales izquierdo y derecho. (Si desea ver la configuración de importación, búscala en **Activos > Audio** y seleccione el clip para ver su configuración).
2. En la Fuente de audio para Ghost Prefab, busque la **Configuración de sonido 3D**. La propiedad Spread controla el rango en grados de los que parece que proviene un sonido.
3. Establezca la propiedad **Spread** en **180**. Esto significa que la mitad del sonido vendrá de cada canal, y dado que estos canales son iguales, el audio parecerá sin dirección.

▼ Audio Source

AudioClip	SFXGhostMove
Output	None (Audio Mixer Group)
Mute	<input type="checkbox"/>
Bypass Effects	<input type="checkbox"/>
Bypass Listener Effects	<input type="checkbox"/>
Bypass Reverb Zones	<input type="checkbox"/>
Play On Awake	<input checked="" type="checkbox"/>
Loop	<input checked="" type="checkbox"/>
Priority	128
Volume	0.4
Pitch	1
Stereo Pan	0
Spatial Blend	1
Reverb Zone Mix	1

▼ 3D Sound Settings

Doppler Level	1
Spread	180
Volume Rolloff	Custom Rolloff
Min Distance	Controlled by curve
Max Distance	10

Listener

Volume	Spatial Blend	Spread	Reverb Zone
--------	---------------	--------	-------------

4. Guarde el Ghost Prefab y regrese a la escena.

5. Guarde la escena.

¡Tu juego ya está terminado! ¡Felicitaciones! Asegúrate de probar tu juego para ver cómo será para los jugadores.

20. Resumen

En este tutorial, exploró los conceptos básicos de audio en Unity agregando una banda sonora ambiental y efectos de sonido a su juego. ¡Tu juego ya está completo! Hay una cosa más que hacer: en el tutorial final, crearás una versión del juego para que puedas distribuirlo.

Construir, ejecutar, distribuir

Ahora has visto cada paso que debes dar para crear un juego en Unity, desde crear tu primer GameObject hasta escribir scripts personalizados.

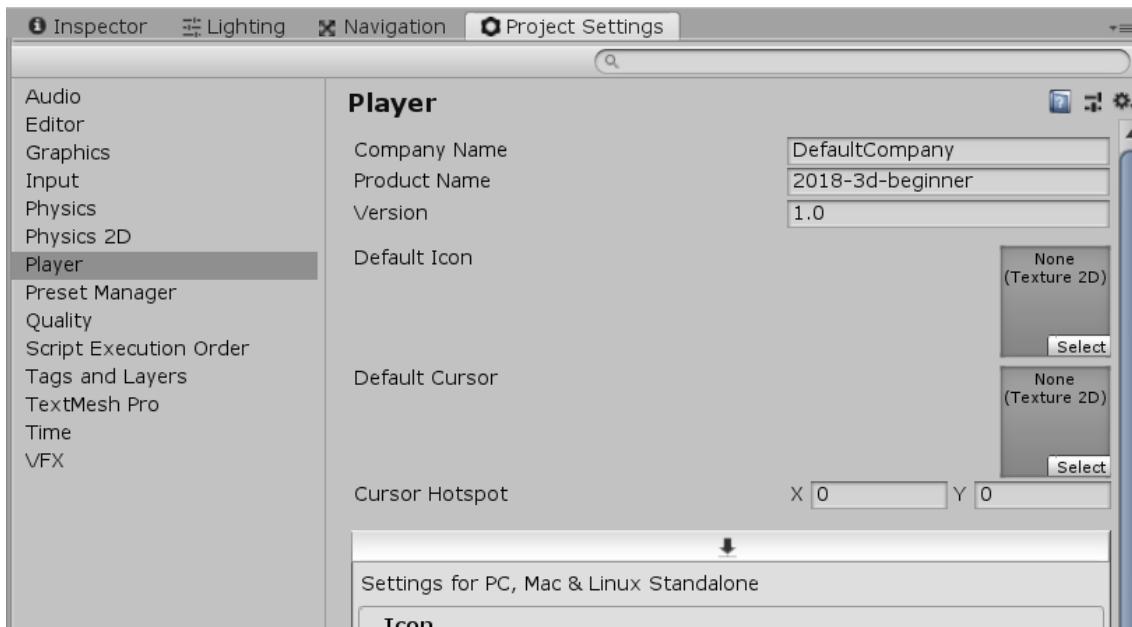
Ahora que su juego está completo, debe **compilarlo** para crear una aplicación independiente que pueda cargar en una tienda digital. Esto significa que las personas podrán jugar el juego sin tener que instalar el Editor de la Unidad y descargar todos sus Activos a su máquina.

1. Ajuste la configuración básica del reproductor

La aplicación que creas desde el Editor para distribuir tu juego a los usuarios se llama **Player**.

Antes de crear el reproductor, ajustemos algunas configuraciones del reproductor:

1. En la barra de menú superior, vaya a **Edición> Configuración del proyecto** y seleccione **Reproductor**. Aquí es donde puedes configurar parte de la información sobre tu juego.



2. El **nombre de la empresa** se utiliza para crear las carpetas donde se almacenarán los archivos creados para los juegos creados. Puede cambiar esto si lo desea, pero eso no es necesario para que el proceso de compilación funcione.

3. El **nombre del producto** es el nombre de tu juego. Cambia este nombre a "**John Hamon's Haunted Jaunt**". Si desea obtener más información sobre la Configuración general del reproductor, puede encontrar más información en la documentación.

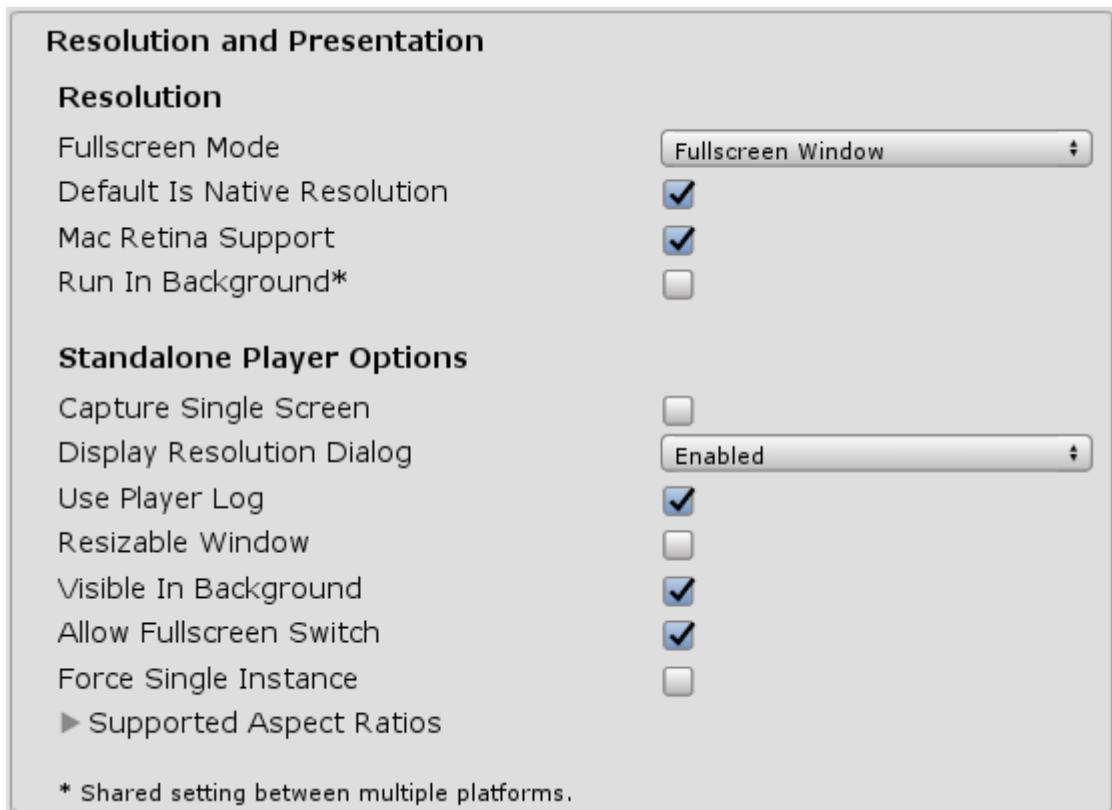
4. El botón de flecha hacia abajo en la parte superior de la siguiente sección aquí significa que todas las configuraciones en esta sección son para **plataformas independientes de PC, Mac y Linux**. Si instaló soporte para otras plataformas al instalar Unity, verá varios botones en esa barra de herramientas:



Se puede hacer clic en cada sección para expandirla y mostrar su configuración. Aquí solo destacaremos un par, pero puede encontrar una explicación de cada configuración en la documentación sobre la [configuración del reproductor para plataformas independientes](#).

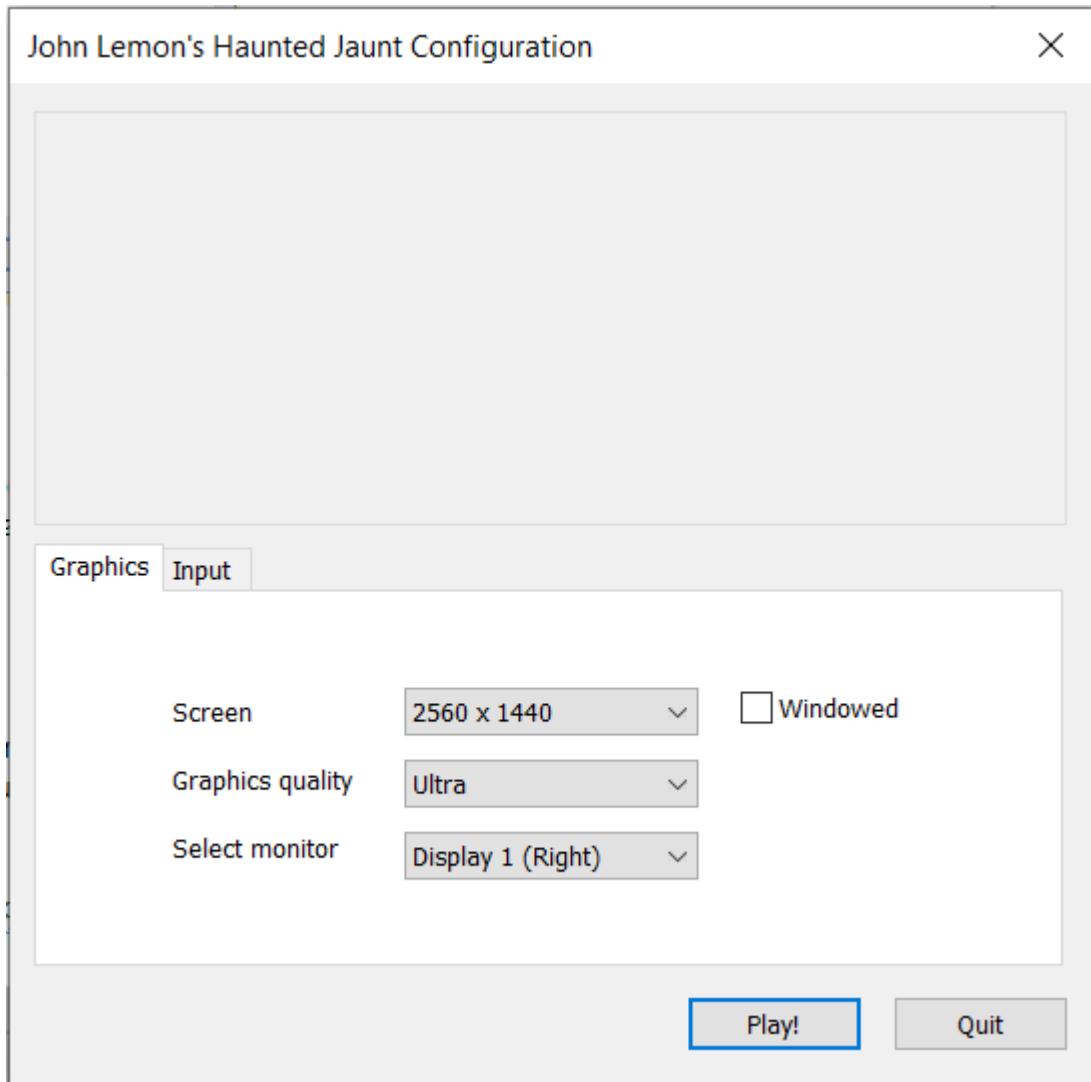
2. Ajuste la configuración de resolución y presentación

Vamos a ajustar cómo se mostrará tu juego a los jugadores: 1. Haga clic en la sección **Resolución y Presentación** para abrirlo.



2. La **resolución** le permite definir el modo predeterminado en el que comienza el juego. La configuración **Ejecutar en segundo plano** determina si su juego continúa ejecutándose si la ventana / aplicación no tiene el foco. Por ejemplo, cuando esa opción está desactivada y la persona que juega el juego abre un navegador web y navega por la web mientras juega, el juego se detendrá hasta que regrese al juego.

3. En la subsección **Opciones de reproductor independiente**, asegúrese de que el **cuadro de diálogo Resolución de pantalla** esté configurado como Activado. Esto le permite mostrar una ventana cuando el usuario inicia el juego que le permite seleccionar la resolución:



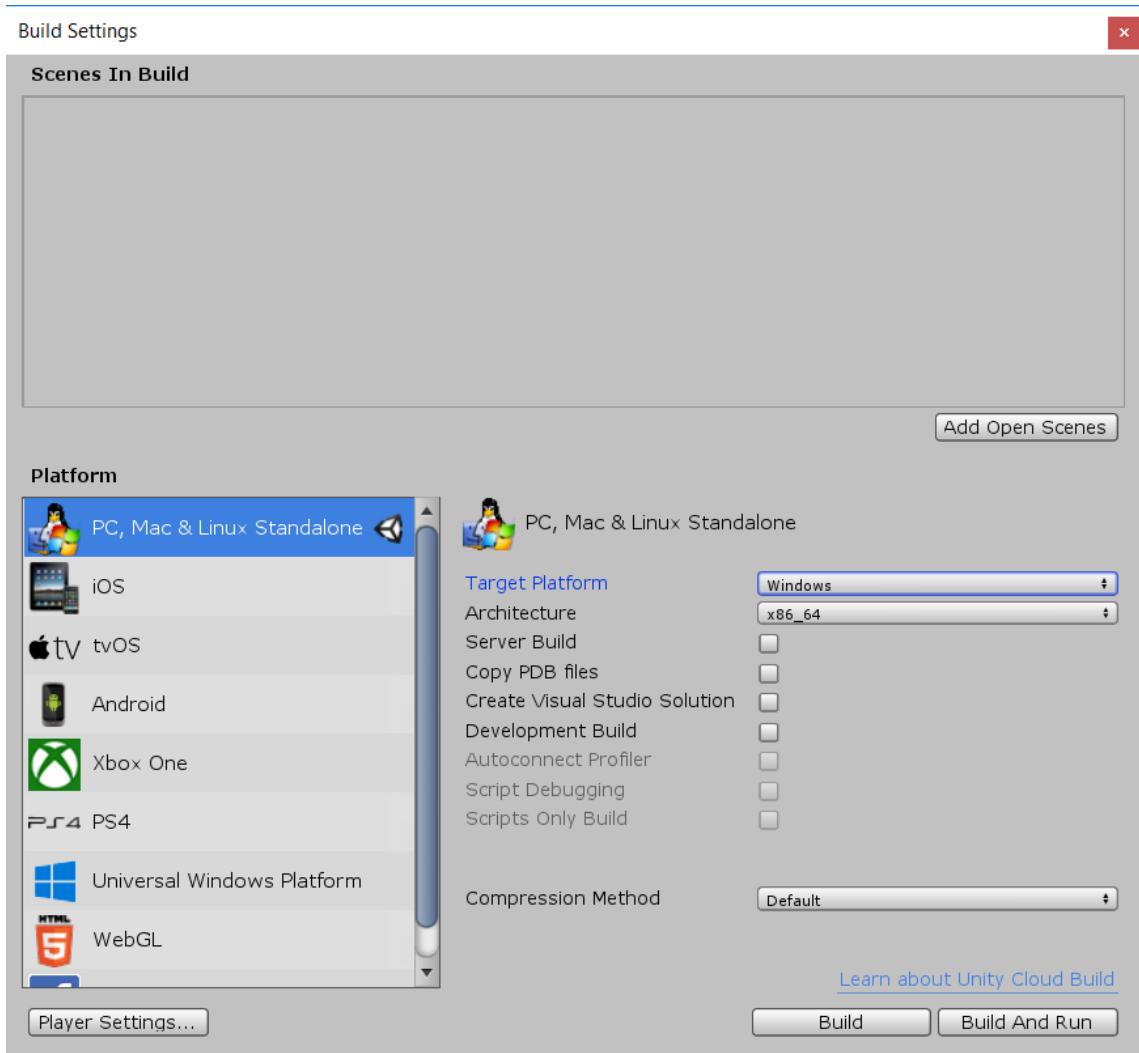
4. En la sección Splash Images, puede cambiar la imagen que se muestra en la parte superior del cuadro de diálogo anterior (**Banner de diálogo de configuración de la aplicación**) o el logotipo que se muestra cuando comienza el juego (**Logos**).

No necesitamos ajustar más configuraciones, ahora estás listo para construir el juego.

3. Construye tu juego

Ahora que has configurado los ajustes del jugador, ¡es hora de construir tu juego! Esto toma todos los activos (por ejemplo, los scripts, las imágenes y el sonido) y los empaqueta en un formato optimizado para su distribución.

Para construir su aplicación en Unity: **1.** En el menú superior, seleccione **Archivo>Configuración de compilación**.



2. La sección **Escenas en construcción** en la parte superior enumera todas las escenas que se incluirán en tu juego. Puede tener escenas en su proyecto que solo use para probar características o para depurar, por lo que Unity necesita saber cuáles incluir en el producto final. Si su MainScene aún está abierta, haga clic en **Agregar escenas abiertas** para agregarla a la lista. Alternativamente, puede arrastrar y soltar sus escenas desde la ventana del proyecto a las escenas en la sección de compilación de la ventana de configuración de compilación. **3.** La sección **Plataforma** en la parte inferior izquierda te permite elegir en qué plataformas quieres que se ejecute tu juego. Por defecto, el Editor solo es compatible con la plataforma en la que está instalado. Para instalar más plataformas:

- Abra el Unity Hub
- Haga clic **instala**
- Haga clic en ... junto a la versión relevante de Unity
- Haga clic en **Agregar componente** y seleccione las plataformas que desea instalar

4. En la parte inferior derecha hay configuraciones relacionadas con la Plataforma seleccionada actual. Estas configuraciones son principalmente para depuración o compilaciones especiales, por lo que puede ignorarlas por ahora.

5. Por ahora, ejecutemos una compilación para la plataforma en la que ha instalado el Editor.

Haga clic en **Compilar** para activar una compilación.

6. Unity abrirá un explorador de archivos y le pedirá que seleccione una carpeta para almacenar su juego creado. El explorador de archivos debe mostrar la carpeta que contiene su Proyecto (que contiene las carpetas Activos, Configuración de proyectos y Biblioteca).

Cree una nueva carpeta llamada **Build** y seleccione esa carpeta.

4. ¿Qué produce Unity?

¡Unity ahora construirá tu juego! No puede hacer nada más con el Editor de Unity durante este tiempo, porque es:

- Comprimir y empacar todos sus activos
- Detectar e ignorar los que no se utilizan.
- Compilar sus scripts en una forma optimizada

Cuando finaliza, Unity abre la carpeta en la que se creó el Proyecto y encontrará el ejecutable creado para su plataforma (.exe en Windows, .app en Mac OSX, etc.). ¡Ejecuta el ejecutable para probar tu juego!

5. Resumen

¡Has terminado este proyecto y has creado una versión de Haunted Jaunt de John Lemon! Ahora puedes enviar todos los archivos de esa carpeta de compilación a tus amigos para que puedan probar tu juego.

Ahora has creado un juego completo en Unity. Pero su viaje hacia el desarrollo de juegos recién comienza. También puede desarrollar su comprensión explorando en el [manual](#) y la [referencia](#) del [script](#) para obtener más información sobre cualquier cosa que hayamos introducido en este Proyecto.